

Programare declarativă¹

Map, Filter, Fold

Traian Florin Șerbănuță
Ioana Leuștean

Departamentul de Informatică, FMI, UB
traian.serbanuta@fmi.unibuc.ro
ioana@fmi.unibuc.ro

27 octombrie 2017

¹bazat pe cursul Informatics 1: Functional Programming de la University of Edinburgh

Programarea funcțională

Funcțiile sunt cetățeni de ordinul I.

- funcțiile sunt valori
- funcțiile pot fi transmise ca argumente altor funcții
- funcțiile pot fi întoarse ca valori

Funcții de nivel înalt

sunt funcțiile care primesc ca argumente alte funcții.

Programarea funcțională

- funcțiile sunt valori

```
Prelude> let x = head
```

```
Prelude> x [1,2]
```

```
1
```

Programarea funcțională

- funcțiile sunt valori

```
Prelude> let x = head
```

```
Prelude> x [1,2]
```

```
1
```

- funcțiile pot fi transmise ca argumente altor funcții

```
Prelude> map head ["higher", "order", "function"]  
"hof"
```

Programarea funcțională

- funcțiile sunt valori

```
Prelude> let x = head
```

```
Prelude> x [1,2]
```

```
1
```

- funcțiile pot fi transmise ca argumente altor funcții

```
Prelude> map head ["higher", "order", "function"]  
"hof"
```

- funcțiile pot fi întoarse ca valori

```
Prelude> :t flip
```

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

Programarea funcțională

- funcțiile sunt valori

```
Prelude> let x = head
```

```
Prelude> x [1,2]
```

```
1
```

- funcțiile pot fi transmise ca argumente altor funcții

```
Prelude> map head ["higher", "order", "function"]  
"hof"
```

- funcțiile pot fi întoarse ca valori

```
Prelude> :t flip
```

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

```
Prelude> let f = flip (:)
```

```
Prelude> let (<:>) = flip (:)
```

```
Prelude> 1:[2,3] == [2,3] <:> 1
```

```
True
```

Programarea funcțională

Prelucarea listelor se poate face folosind funcții de nivel înalt.

- Transformarea fiecărui element al unei liste se poate face folosind funcția **map**.
- Selecția elementelor unei liste se poate face folosind funcția **filter**.
- Combinarea elementelor unei liste se poate face folosind funcția **foldr**.

Map (Transformarea fiecărui element dintr-o listă)

Pătrate

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

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

Soluție descriptivă

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

Soluție recursivă

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

Coduri ASCII

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

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

Soluție descriptivă

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

Soluție recursivă

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

Funcția **map**

Definiție

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
```

Map în acțiune

Varianta descriptivă

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

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

```
squares [1,2,3]
= map sqr [1,2,3]
```

Map în acțiune

Varianta descriptivă

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

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

```
squares [1,2,3]
= map sqr [1,2,3]
= [ sqr x | x <- [1,2,3]]
```

Map în acțiune

Varianta descriptivă

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

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

```
squares [1,2,3]
= map sqr [1,2,3]
= [ sqr x | x <- [1,2,3]]
= [ sqr 1 ] ++ [ sqr 2 ] ++ [ sqr 3 ]
= [ 1, 4, 9 ]
```

Map în acțiune

Varianta recursivă

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

```
squares [1,2,3]
```

```
= map sqr [1,2,3]
```

```
= map sqr (1:2:3:[])
```

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


Map în acțiune

Varianta recursivă

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

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

```
squares [1,2,3]
```

```
= map sqr [1,2,3]
```

```
= map sqr (1:2:3:[])
```

```
= sqr 1 : map sqr (2:3:[])
```

Map în acțiune

Varianta recursivă

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

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

```
squares [1,2,3]
= map sqr [1,2,3]
= map sqr (1:2:3:[])
= sqr 1 : map sqr (2:3:[])
= sqr 1 : sqr 2: map sqr (3:[])
= sqr 1 : sqr 2: sqr 3: map sqr []
= sqr 1 : sqr 2: sqr 3: []
= [ 1, 4, 9 ]
```

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
```

Filter — Selectarea elementelor dintr-o listă

Selectarea elementelor pozitive dintr-o listă

```
*Main> positives [1,-2,3]  
[1,3]
```

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
```

Selectarea cifrelor dintr-un șir de caractere

```
*Main> digits "a2c3"  
"23"
```

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
```

Funcția **filter**

Definiție

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 — Positive

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 — Cifre

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
```

Fold — Agregarea elementelor dintr-o listă

Suma

Definiți o funcție care dată fiind o listă de numere întregi calculează suma elementelor din listă.

```
*Main> sum [1,2,3,4]  
10
```

Soluție recursivă

```
sum :: [Int] -> Int  
sum []      = 0  
sum (x:xs) = x + sum xs
```

Produs

Definiți o funcție care dată fiind o listă de numere întregi calculează produsul elementelor din listă.

```
*Main> product [1,2,3,4]  
24
```

Soluție recursivă

```
product :: [Int] -> Int  
product []      = 1  
product (x:xs) = x * sum xs
```

Concatenare

Definiți o funcție care concatenează o listă de liste.

```
*Main> concat [[1,2,3],[4,5]]  
[1,2,3,4,5]
```

```
*Main> concat ["con","ca","te","na","re"]  
"concatenare"
```

Soluție recursivă

```
concat :: [[a]] -> [a]  
concat []      = []  
concat (xs:xss) = xs ++ concat xss
```

Funcția foldr

Definiție

Date fiind o funcție de actualizare a valorii calculate cu un element curent, o valoare inițială, și o listă, calculați valoare obținută prin aplicarea repetată a funcției de actualizare fiecărui element din listă.

Soluție recursivă

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr f i [] = i

foldr f i (x:xs) = f x (**foldr** i xs)

Funcția foldr

Definiție

Date fiind o funcție de actualizare a valorii calculate cu un element curent, o valoare inițială, și o listă, calculați valoare obținută prin aplicarea repetată a funcției de actualizare fiecărui element din listă.

Soluție recursivă

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

Soluție recursivă cu operator infix

```
foldr :: (a -> b -> b) -> b -> [a] -> b  
foldr op i [] = i  
foldr op i (x:xs) = x 'op' (foldr i xs)
```

Funcția foldr

Definiție

Date fiind o funcție de actualizare a valorii calculate cu un element curent, o valoare inițială, și o listă, calculați valoare obținută prin aplicarea repetată a funcției de actualizare fiecărui element din listă.

Soluție recursivă

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr f i [] = i

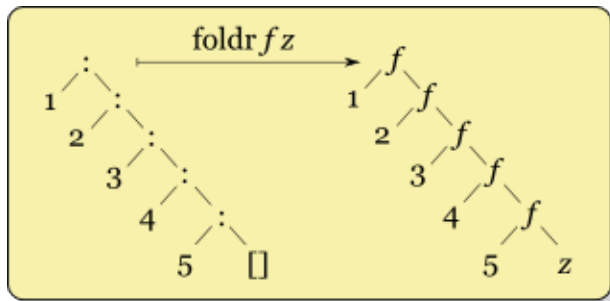
foldr f i (x:xs) = f x (**foldr** i xs)

Funcția foldr

foldr :: (Int -> b -> b) -> b -> [Int] -> b

f :: Int -> b -> b

z :: b



[https://en.wikipedia.org/wiki/Fold_\(higher-order_function\)](https://en.wikipedia.org/wiki/Fold_(higher-order_function))

Suma

Soluție recursivă

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
```

Soluție folosind **foldr**

```
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

foldr în acțiune

Varianța recursivă

sum :: [Int] -> Int

sum xs = **foldr** (+) 0 xs

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr _ i [] = i

foldr op i (x:xs) = x 'op' (**foldr** i xs)

sum [1,2]

= foldr (+) 0 [1,2]

= foldr (+) 0 (1:2:[])

foldr în acțiune

Varianța recursivă

```
sum :: [Int] -> Int
```

```
sum xs = foldr (+) 0 xs
```

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

```
foldr _ i [] = i
```

```
foldr op i (x:xs) = x 'op' (foldr i xs)
```

```
sum [1,2]
```

```
= foldr (+) 0 [1,2]
```

```
= foldr (+) 0 (1:2:[])
```

```
= 1 + foldr (+) 0 (2:[])
```

foldr în acțiune

Varianța recursivă

sum :: [Int] -> Int

sum xs = **foldr** (+) 0 xs

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr _ i [] = i

foldr op i (x:xs) = x 'op' (**foldr** i xs)

sum [1,2]

= foldr (+) 0 [1,2]

= foldr (+) 0 (1:2:[])

= 1 + foldr (+) 0 (2:[])

= 1 + 2 + 0

= 3

Produs

Soluție recursivă

```
product :: [Int] -> Int  
product [] = 1  
product (x:xs) = x * product xs
```

Soluție folosind **foldr**

```
product :: [Int] -> Int  
product xs = foldr (*) 1 xs
```

Concatenare

Soluție recursivă

```
concat :: [[a]] -> [a]
concat []      = []
concat (xs:xss) = xs ++ concat xss
```

Soluție folosind **foldr**

```
concat :: [[a]] -> [a]
concat xs = foldr (++) [] xs
```

foldr pe liste infinite

```
Prelude> let li = (:[])
```

```
Prelude> li 1  
[1]
```

```
Prelude> let infLL = map li [1..]
```

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

```
Prelude> let infL = foldr (++) [] infLL  
[1,2,3,4,5]
```

```
infL = foldr (++) [] (map li [1 ..])
```

Putem defini **infL** folosind numai **foldr**?

mai mult despre **foldr**

```
infl = foldr aux [] [1..]  
  where  
    aux x xs = (li x)++xs
```

mai mult despre **foldr**

```
infl = foldr aux [] [1..]  
  where  
    aux x xs = (li x)++xs
```

Funcția **map** poate fi definită cu **foldr**

```
map f xs = foldr aux [] xs  
  where  
    aux x xs = (f x) : xs
```

Map, Filter, Fold — combine

Suma pătratelor numerelor pozitive

```
f :: [Int] -> Int
f xs = sum (squares (positives xs))
```

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

```
f :: [Int] -> Int
f [] = 0
f (x:xs) | x > 0 = (x*x) + f xs
          | otherwise = f xs
```

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

Map/Filter/Fold combine

Problemă

Aflați lungimea celui mai lung cuvânt care începe cu litera 'c' dintr-o listă dată.

```

strs = ["cezara", "petru", "claudia", "", "virgil"];
maxLengthFn = foldr max 0 .
               map length .
               filter testC
  where testC ('c':_) = True
        testC _      = False
maxLength = maxLengthFn strs

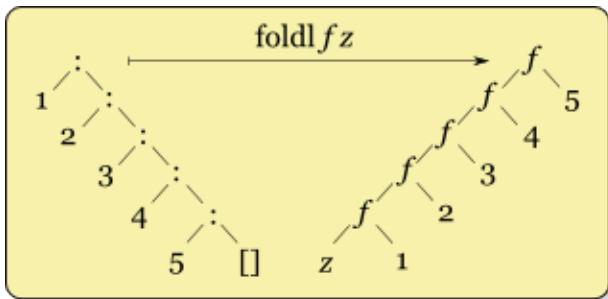
```

Funcția **foldl**

foldl :: (b -> a -> b) -> b -> [a] -> b

foldl op i [] = i

foldl op i (x:xs) = (**foldl** op (i 'op' x) xs)



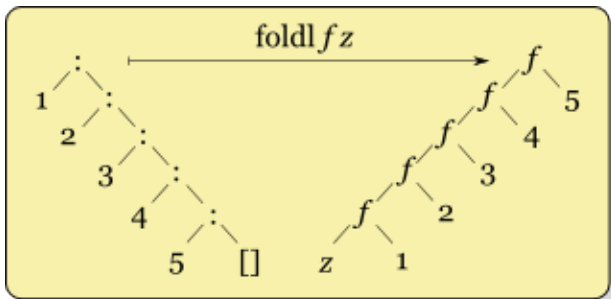
[https://en.wikipedia.org/wiki/Fold_\(higher-order_function\)](https://en.wikipedia.org/wiki/Fold_(higher-order_function))

Funcția **foldl**

foldl :: (b -> a -> b) -> b -> [a] -> b

foldl op i [] = i

foldl op i (x:xs) = (**foldl** op (i 'op' x) xs



[https://en.wikipedia.org/wiki/Fold_\(higher-order_function\)](https://en.wikipedia.org/wiki/Fold_(higher-order_function))

Atenție! **foldl** nu poate fi folosită pe liste infinite!

Funcția **foldl**

reverse :: [a] -> [a]

reverse [] = []

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

Funcția **foldl**

reverse :: [a] -> [a]

reverse [] = []

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

- definiția cu **foldr**

reverse ys = **foldr** (\x xs -> xs ++ [x]) [] ys

- definiția cu **foldl**

reverse ys = **foldl** (\xs x -> x:xs) [] ys

Currying

Exemplu: adunarea numerelor

```
add' :: (Int, Int) -> Int
```

```
add' (x, y) = x + y
```

```
Prelude> add' (3, 4)
```

```
7
```

```
add = curry add'
```

```
Prelude> :t add
```

```
add :: Int -> Int -> Int
```

```
Prelude> add 3 4
```

```
7
```

Exemplu: adunarea numerelor

`add :: Int -> (Int -> Int)`

`(add x) y = x + y`

`(add 3) 4`

`= 3 + 4`

`= 7`

Currying

A funcție cu două argumente este de fapt o funcție de primul argument care întoarce o funcție de al doilea argument.

Exemplu: adunarea numerelor

```
add :: Int -> (Int -> Int)
(add x) y = x + y
```

```
(add 3) 4
= 3 + 4
= 7
```

Currying

A funcție cu două argumente este de fapt o funcție de primul argument care întoarce o funcție de al doilea argument.

- aplicarea funcțiilor este asociativă la stânga
- operatorul \rightarrow este asociativ la dreapta

Currying

A funcție cu două argumente este de fapt o funcție de primul argument care întoarce o funcție de al doilea argument.

```
add :: Int -> (Int -> Int)
```

```
add x = g
```

```
  where
```

```
    g y = x + y
```

```
(add 3) 4
```

```
=
```

```
g 4
```

```
  where
```

```
    g y = 3 + y
```

```
=
```

```
3 + 4
```

```
=
```

```
7
```

Currying

Haskell Curry (1900–1982)

```
add :: Int -> (Int -> Int)
add x y = x + y
```

este echivalent (semantic) cu

```
add :: Int -> (Int -> Int)
add x = g
  where
    g y = x + y
```

De asemeni,

```
add 3 4
```

este echivalent (semantic) cu

```
(add 3) 4
```

Aplicații Currying — Stilul funcțional

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

```
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

este echivalent (semantic) cu

```
foldr :: (a -> a -> a) -> a -> ([a] -> a)
foldr f a []      = a
foldr f a (x:xs) = f x (foldr f a xs)
sum :: [Int] -> Int
sum = foldr (+) 0
```


Aplicații Currying — Stilul funcțional

Suma, Produs, Concatenare

```
sum :: [Int] -> Int  
sum = foldr (+) 0
```

```
product :: [Int] -> Int  
product = foldr (*) 1
```

```
concat :: [[a]] -> [a]  
concat = foldr (++) []
```

```
idl :: [a] -> [a]  
idl = foldr (:) []
```

Funcții anonime

Funcții anonime

Funcții anonime = lambda expresii

\<pattern> -> expresie

Prelude> (\x -> x+ 1) 3

4

inc = \x -> x + 1

add = \x y -> x+ y

Funcții anonime

Funcții anonime = lambda expresii

\<pattern> -> expresie

Prelude> (\x -> x+ 1) 3

4

inc = \x -> x + 1

add = \x y -> x+ y

prod = \ (x,y)-> x*y

head2 = \ (x:y:l) -> (x,y)

Funcții anonime

Funcții anonime = lambda expresii

\<pattern> -> expresie

Prelude> (\x -> x+ 1) 3

4

inc = \x -> x + 1

add = \x y -> x+ y

prod = \ (x,y)-> x*y

head2 = \ (x:y:l) -> (x,y)

aplic2 = \f -> f . f

Prelude> aplic2 sqrt 16

2.0

Funcții anonime

Funcții anonime = lambda expresii

\<pattern> -> expresie

Prelude> (\x -> x+ 1) 3

4

inc = \x -> x + 1

add = \x y -> x+ y

prod = \ (x,y)-> x*y

head2 = \ (x:y:l) -> (x,y)

aplic2 = \f -> f . f

Prelude> aplic2 sqrt 16

2.0

comb f g = \ x y -> g (f x) (f y)

Prelude> (comb head (<)) "abc" "def"

True

Simplificăm definiția

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

Simplificare incorectă

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

Simplificăm definiția

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

Simplificare incorectă

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

Simplificare corectă

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


Simplificăm definiția

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

Aceeasi definitie folosind funcții anonime:

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

Explicație pentru Currying folosind λ -expresii

$$\begin{aligned}
 & (\lambda x \rightarrow \lambda y \rightarrow x + y) \ 3 \ 4 \\
 = & \\
 & ((\lambda x \rightarrow (\lambda y \rightarrow x + y)) \ 3) \ 4 \\
 = & \\
 & (\text{let } x = 3 \text{ in } \lambda y \rightarrow x + y) \ 4 \\
 = & \\
 & (\lambda y \rightarrow 3 + y) \ 4 \\
 = & \\
 & \text{let } y = 4 \text{ in } 3 + y \\
 = & \\
 & 3 + 4 \\
 = & \\
 & 7
 \end{aligned}$$

Funcții anonime / Lambda Calcul

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

Lambda Calcul

- Introdus de logicianul Alonzo Church (1903–1995) pentru dezvoltarea unei teorii a calculabilității
- În Haskell, `\` e folosit în locul simbolului λ
- Matematic scriem
 $\lambda x. x * x$ în loc de `\ x -> x * x`

Evaluarea λ -expresiilor

β -reducție

Formula generală pentru evaluarea aplicării λ -expresiilor este prin substituirea argumentului formal cu argumentul actual în corpul funcției:

$$(\lambda x.N) M \xrightarrow{\beta} M[N/x]$$

β -reducția poate fi descrisă de următoarea identitate Haskell:

$(\lambda x . n) m == \mathbf{let} \ x = m \ \mathbf{in} \ n$

Evaluarea λ -expresiilor

```
(\x -> x > 0) 3
=
let x = 3 in x > 0
=
3 > 0
=
True
```

```
(\x -> x * x) 3
=
let x = 3 in x * x
=
3 * 3
=
9
```

Exemple: **foldr** și funcții anonime

```
map f xs = foldr aux [] xs
  where
    aux x xs = (f x) : xs
```

Exemple: **foldr** și funcții anonime

```
map f xs = foldr aux [] xs
      where
        aux x xs = (f x) : xs
```

Cu λ -expresii

```
map f xs = foldr (\x xs -> (f x):xs) [] xs
```

Exemple: **foldr** și funcții anonime

```
map f xs = foldr aux [] xs
      where
          aux x xs = (f x) : xs
```

Cu λ -expresii

```
map f xs = foldr (\x xs -> (f x):xs) [] xs
length xs = foldr (\x n -> n+1) 0 xs
```


Exemple: **foldr** și funcții anonime

```
map f xs = foldr aux [] xs
      where
        aux x xs = (f x) : xs
```

Cu λ -expresii

```
map f xs = foldr (\x xs -> (f x):xs) [] xs
length xs = foldr (\x n -> n+1) 0 xs
filter p xs = foldr (\x xs ->
                      if (p x) then (x:xs) else xs) [] xs
```

Exemple: **foldr** și funcții anonime

```
map f xs = foldr aux [] xs
      where
          aux x xs = (f x) : xs
```

Cu λ -expresii

```
map f xs = foldr (\x xs -> (f x):xs) [] xs
length xs = foldr (\x n -> n+1) 0 xs
filter p xs = foldr (\x xs ->
                      if (p x) then (x:xs) else xs) [] xs
```

Aplicații Currying — Stilul funcțional

```
map f      = foldr (\x xs -> (f x):xs) []
length    = foldr (\x n -> n+1) 0
filter p = foldr (\x xs -> if (p x) then (x:xs) else xs) []
```

Secțiuni (Tăieturi)

Secțiuni

- (> 0) e forma scurtă a lui $(\lambda x \rightarrow x > 0)$
- $(2 *)$ e forma scurtă a lui $(\lambda x \rightarrow 2 * x)$
- $(+ 1)$ e forma scurtă a lui $(\lambda x \rightarrow x + 1)$
- $(2 ^)$ e forma scurtă a lui $(\lambda x \rightarrow 2 ^ x)$
- $(^ 2)$ e forma scurtă a lui $(\lambda x \rightarrow x ^ 2)$

Secțiuni

- (> 0) e forma scurtă a lui $(\lambda x \rightarrow x > 0)$
- $(2 *)$ e forma scurtă a lui $(\lambda x \rightarrow 2 * x)$
- $(+ 1)$ e forma scurtă a lui $(\lambda x \rightarrow x + 1)$
- $(2 ^)$ e forma scurtă a lui $(\lambda x \rightarrow 2 ^ x)$
- $(^ 2)$ e forma scurtă a lui $(\lambda x \rightarrow x ^ 2)$
- $('op' 2)$ e forma scurtă a lui $(\lambda x \rightarrow x 'op' 2)$
- $(2 'op')$ e forma scurtă a lui $(\lambda x \rightarrow 2 'op' x)$

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

Secțiuni

Secțiunile sunt afectate de **asociativitatea** și **precedența** operatorilor.

```
Prelude> :t (+ 3 * 4)
(+ 3 * 4) :: Num a => a -> a
```

```
Prelude> :t (* 3 + 4) -- + are precedenta mai mica decat *
error
```

```
Prelude> :t (* 3 * 4) -- * este asociativa la stanga
error
```

```
Prelude> :t (3 * 4 *)
(3 * 4 *) :: Num a => a -> a
```

Secțiuni — Exemplu

```
f :: [Int] -> [Int]
f xs = map sqr [x | x<- xs, x>0]
      where
        sqr x = x^2
```

Folosind λ -expresii

```
f xs = map (\ x -> x * x) [x | x<- xs, (\ x -> x > 0) x]
```

Folosind secțiuni

```
f xs = map (^2) [x | x<- xs, (>0) x])
```

Secțiuni — Exemplu

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

```
x <*> y = x * x + y
```

```
functions = map (<*>) [0..]
```

Ce tip are **functions**?

Secțiuni — Exemplu

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

```
x <*> y = x * x + y
```

```
functions = map (<*>) [0..]
```

Ce tip are **functions**?

```
functions :: [Int -> Int]
```

```
functions = [(0 <*>), (1 <*>), (2 <*> ), ...]
```

```
Prelude> (functions !! 50) 10
2510
```

Compunerea funcțiilor

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)
```

Operatorul . — stilul funcțional

Definiție cu parametru explicit

```
f :: [Int] -> Int
f xs = foldr (+) 0 (map ( ^ 2) (filter ( > 0) xs))
```

Definiție compozițională

```
f :: [Int] -> Int
f = foldr (+) 0 . map ( ^ 2) . filter ( > 0)
```

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 \$

<http://learnyouahaskell.com/higher-order-functions>

Operatorul (\$) este asociativ la dreapta.

```
sum (filter (> 10) (map (*2) [2..10]))
```

se poate scrie

```
sum $ filter (> 10) $ map (*2) [2..10].
```

Operatorul \$

<http://learnyouahaskell.com/higher-order-functions>

Operatorul (\$) este asociativ la dreapta.

```
sum (filter (> 10) (map (*2) [2..10]))
```

se poate scrie

```
sum $ filter (> 10) $ map (*2) [2..10].
```

Exemplu folosind secțiuni:

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

```
[7.0,30.0,9.0,1.7320508075688772]
```