

Programare funcțională

Funcții de ordin înalt. Procesarea fluxurilor de date.

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

20 octombrie 2020

Cuprins

- 1 Operatori. Secțiuni
 - Operatori
 - Secțiuni
- 2 Procesarea fluxurilor de date: Map, Filter, Fold
 - Transformarea fiecărui element dintr-o listă
 - Exemple
 - Funcția map
 - Selectarea elementelor dintr-o listă
 - Exemple
 - Funcția filter
 - Agregarea elementelor dintr-o listă
 - Exemple
 - Funcția foldr
 - Map, Filter, Fold — combinate
 - Map/Filter/Fold în alte limbaje

Cuprins

- 1 Operatori. Secțiuni
 - Operatori
 - Secțiuni
- 2 Procesarea fluxurilor de date: Map, Filter, Fold
 - Transformarea fiecărui element dintr-o listă
 - Exemple
 - Funcția map
 - Selectarea elementelor dintr-o listă
 - Exemple
 - Funcția filter
 - Agregarea elementelor dintr-o listă
 - Exemple
 - Funcția foldr
 - Map, Filter, Fold — combinate
 - Map/Filter/Fold în alte limbaje

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

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`

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
```

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 ' '

$\text{mod } 5 \ 2 == 5 \text{ 'mod' } 2$

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
```

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
```

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'

Asociativitate

Operatorul - asociativ la stanga

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

--

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

Asociativitate

Operatorul - asociativ la stanga

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

--

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

Operatorul : asociativ la dreapta

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

Operatorul - asociativ la stanga

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

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

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

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

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 ++?

Asocativitate

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

Cuprins

- 1 Operatori. Secțiuni
 - Operatori
 - **Secțiuni**
- 2 Procesarea fluxurilor de date: Map, Filter, Fold
 - Transformarea fiecărui element dintr-o listă
 - Exemple
 - Funcția map
 - Selectarea elementelor dintr-o listă
 - Exemple
 - Funcția filter
 - Agregarea elementelor dintr-o listă
 - Exemple
 - Funcția foldr
 - Map, Filter, Fold — combinate
 - Map/Filter/Fold în alte limbaje

Secțiuni ("operator sections")

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

Aplicarea parțială

Fie $f : A \times B \rightarrow C$ o funcție. În mod uzual scriem $f(a, b) = c$ unde $a \in A$, $b \in B$ și $c \in C$.

Pentru $a \in A$ și $b \in B$ (arbitrare, fixate) definim

$f_a : B \rightarrow C$, $f_a(b) = c$ dacă și numai dacă $f(a, b) = c$,

$f^b : A \rightarrow C$, $f^b(a) = c$ dacă și numai dacă $f(a, b) = c$.

Funcțiile f_a și f_b se obțin prin aplicarea parțială a funcției f .

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 ++)`

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 (++ " world!")
```

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

```
Prelude> (++ " world!") "Hello" -- atentie la  
    paranteze  
"Hello world!"
```

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 (++ " world!")
```

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

```
Prelude> (++ " world!") "Hello" -- atentie la  
paranteze
```

```
"Hello world!"
```

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

error

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 (++ " world!")
```

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

```
Prelude> (++ " world!") "Hello" -- atentie la  
paranteze
```

```
"Hello world!"
```

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

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
```

2

Secțiuni

- Secțiunile operatorului (:

Secțiuni

- Secțiunile operatorului (:)

```
Prelude> (2 :) [1,2]
```

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

```
Prelude> (: [1,2]) 3
```

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

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

Secțiunile sunt definite prin lambda expresii:

- ('op' 2) e forma scurtă a lui ($\lambda x \rightarrow x \text{ 'op' } 2$)
- (2 'op') e forma scurtă a lui ($\lambda x \rightarrow 2 \text{ 'op' } x$)

Exemple

- (> 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 ($\lambda x \rightarrow x ^ 2$)

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> :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
```

Cuprins

- 1 Operatori. Secțiuni
 - Operatori
 - Secțiuni
- 2 Procesarea fluxurilor de date: Map, Filter, Fold
 - Transformarea fiecărui element dintr-o listă
 - Exemple
 - Funcția map
 - Selectarea elementelor dintr-o listă
 - Exemple
 - Funcția filter
 - Agregarea elementelor dintr-o listă
 - Exemple
 - Funcția foldr
 - Map, Filter, Fold — combinate
 - Map/Filter/Fold în alte limbaje

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
```

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
```

Cuprins

- 1 Operatori. Secțiuni
 - Operatori
 - Secțiuni
- 2 **Procesarea fluxurilor de date: Map, Filter, Fold**
 - Transformarea fiecărui element dintr-o listă
 - Exemple
 - Funcția map
 - **Selectarea elementelor dintr-o listă**
 - Exemple
 - Funcția filter
 - Agregarea elementelor dintr-o listă
 - Exemple
 - Funcția foldr
 - Map, Filter, Fold — combinate
 - Map/Filter/Fold în alte limbaje

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
```

Cuprins

- 1 Operatori. Secțiuni
 - Operatori
 - Secțiuni
- 2 **Procesarea fluxurilor de date: Map, Filter, Fold**
 - Transformarea fiecărui element dintr-o listă
 - Exemple
 - Funcția map
 - Selectarea elementelor dintr-o listă
 - Exemple
 - Funcția filter
 - **Agregarea elementelor dintr-o listă**
 - Exemple
 - Funcția foldr
 - Map, Filter, Fold — combinate
 - Map/Filter/Fold în alte limbaje

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

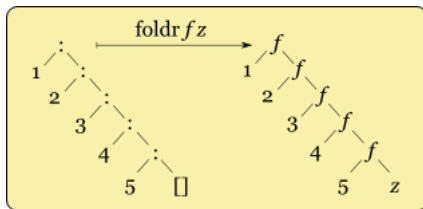
```
concat :: [[a]] -> [a]  
concat []      = []  
concat (xs:xss) = xs ++ concat xss
```

Funcția **foldr**

Definiție

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

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



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
```

Exemplu

```
foldr (+) 0 [1, 2, 3] == 1 + (2 + (3 + 0))
```

Produs

Soluție recursivă

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

Soluție folosind **foldr**

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

Exemplu

```
foldr (*) 1 [1, 2, 3] == 1 * (2 * (3 * 1))
```

Concatenare

Soluție recursivă

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

Soluție folosind **foldr**

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

Exemplu

```
foldr (++) [] ["Ana ", "are ", "mere."]
  == "Ana " ++ ("are " ++ ("mere." ++ []))
```

Cuprins

- 1 Operatori. Secțiuni
 - Operatori
 - Secțiuni
- 2 Procesarea fluxurilor de date: Map, Filter, Fold
 - Transformarea fiecărui element dintr-o listă
 - Exemple
 - Funcția map
 - Selectarea elementelor dintr-o listă
 - Exemple
 - Funcția filter
 - Agregarea elementelor dintr-o listă
 - Exemple
 - Funcția foldr
 - **Map, Filter, Fold — combinate**
 - Map/Filter/Fold în alte limbaje

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
```

Foldr cu secțiuni — Exemplu

Folosind λ -expresii

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

Folosind secțiuni

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

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)
```

Cuprins

- 1 Operatori. Secțiuni
 - Operatori
 - Secțiuni
- 2 Procesarea fluxurilor de date: Map, Filter, Fold
 - Transformarea fiecărui element dintr-o listă
 - Exemple
 - Funcția map
 - Selectarea elementelor dintr-o listă
 - Exemple
 - Funcția filter
 - Agregarea elementelor dintr-o listă
 - Exemple
 - Funcția foldr
 - Map, Filter, Fold — combinate
 - Map/Filter/Fold în alte limbaje

Map/Filter/Reduce în Haskell

Problemă

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

Map/Filter/Reduce în Haskell

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
```

Map/Filter/Reduce în Python

<http://www.python-course.eu/lambda.php>

```
strs = ["cezara", "petru", "claudia", "", "virgil"];  
def maxLengthFn(strs):  
    return reduce(max,  
                  map(len,  
                        filter(lambda s: len(s) > 0 and s[0] == 'c',  
                              strs)));  
maxLength = maxLengthFn(strs);  
print(maxLength);
```

Map/Filter/Reduce în Javascript

<http://crypto.net/~joepie91/blog/2015/05/04/>

[functional-programming-in-javascript-map-filter-reduce/](#)

```
var strs = ["cezara", "petru", "claudia", "", "virgil"];
var maxLength = strs
    .filter(function(s){ return s[0]== 'c'; })
    .map(function(s){ return s.length; })
    .reduce(function(a,b){ return Math.max(a,b); })
```

Map/Filter/Reduce în PHP

<http://eddmann.com/posts/mapping-filtering-and-reducing-in-php/>

```
$strs = array("cezara", "petru", "claudia", "", "virgil");  
$max_length = array_reduce(  
    array_map(  
        "strlen",  
        array_filter(  
            $strs,  
            function($s){ return isset($s[0]) && $s[0]== 'c' ;}) ,  
        "max",  
        0);  
echo $max_length;
```

Map/Filter/Reduce în Java 8

<http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>

```
package edu.unibuc.fmi;  
import java.util.*;  
  
public class Main {  
    public static void main(String[] args) {  
        List<String> myList = Arrays.asList(  
            "cezara", "petru", "claudia", "", "virgil");  
        int l =  
            myList  
                .stream()  
                .filter(s -> s.startsWith("c"))  
                .map(String::length)  
                .reduce(0, Integer::max);  
        System.out.println(l);  
    }  
}
```

Map/Filter/Reduce în C++11

https://meetingcpp.com/tl_files/mcpp/slides/12/FunctionalProgrammingInC++11.pdf

```
#include <algorithm>
#include <string>
#include <iostream>
using namespace std;
int main() {
    vector<string> strs {"cezara", "petru", "claudia", "", "virgi"};
    strs.erase(remove_if(strs.begin(), strs.end(),
        [](string x){ return x[0] != 'c'; } ),
        strs.end());
    vector<int> lengths;
    transform(strs.begin(), strs.end(), back_inserter(lengths),
        [](string x) { return x.length(); });
    int max_length = accumulate(lengths.begin(), lengths.end(),
        0, [](int a, int b){ return a > b ? a : b; });
    cout << max_length;
}
```