

Programare declarativă¹

Map, Filter, Fold

Traian Florin Șerbănuță

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

28 octombrie 2016

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

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)
```

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
```

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
```

Concatenare

Soluție recursivă

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

Soluție folosind **foldr**

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

Map în acțiune

Varianta 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:[])

Map în acțiune

Varianta 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:[])
```

Map în acțiune

Varianta 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

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
```

Currying

Adunarea numerelor

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

`add x y = x + y`

`add 3 4`

`=`

`3 + 4`

`=`

`7`

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.

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 (++) []
```


Funcții anonime

Simplificăm definiția

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

Simplificare incorectă

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

Simplificăm definiția

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

Simplificare incorectă

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

Simplificare corectă

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

Funcții anonime / Lambda Calcul

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

Lambda Calcul

- Introdus de logicianul Alonzo Church (1903–1995) pentru dezvoltarea unei teorii a calculabilității
- În Haskell, \backslash e folosit în locul simbolului λ
- Matematic scriem $\lambda x. x * x$ în loc de $\backslash x \rightarrow x * x$
 $\lambda x. x > 0$ în loc de $\backslash x \rightarrow x > 0$

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
```

Explicație pentru Currying folosind λ -expresii

```

(\x -> \y -> x + y) 3 4
=
((\x -> (\y -> x + y)) 3) 4
=
(let x = 3 in \y -> x + y) 4
=
(\y -> 3 + y) 4
=
let y = 4 in 3 + y
=
3 + 4
=
7

```

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:

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

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

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))
```

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)
```

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;
}
```