

# Programare funcțională

Introducere în programarea funcțională folosind Haskell  
C04

---

Claudia Chiriță

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

**Fold:**

**agregarea elementelor dintr-o listă**

---

## Exemplu - Suma

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

### Soluție recursivă

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

```
Prelude> sum [1,2,3,4]
10
```

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

### Soluție recursivă

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

```
Prelude> product [1,2,3,4]
```

24

## Exemplu - Concatenare

Definiți o funcție care concatenează o listă de liste.

**Soluție recursivă**

```
concat :: [[a]] -> [a]
```

```
concat [] = []
```

```
concat (xs:xss) = xs ++ concat xss
```

```
Prelude> concat [[1,2,3],[4,5]]
```

```
[1,2,3,4,5]
```

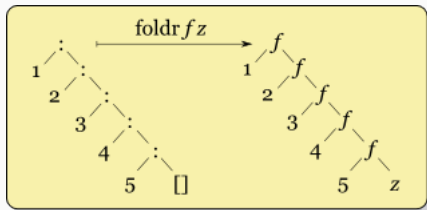
```
Prelude> concat ["con","ca","te","na","re"]
```

```
"concatenare"
```

# Funcția foldr

**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.



# Funcția foldr

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

## Soluție recursivă

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

**foldr** f i [] = i

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

## Exemplu — Suma

### Soluție recursivă

**sum** :: [Int] -> Int

**sum** [] = 0

**sum** (x:xs) = x + **sum** xs

### Soluție folosind foldr

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

**sum** :: [Int] -> Int

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

### Exemplu

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



## Exemplu — Produs

### Soluție recursivă

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

### Soluție folosind foldr

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

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

### Exemplu

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

## Exemplu — Concatenare

### Soluție recursivă

**concat** :: [[a]] -> [a]

**concat** [] = []

**concat** (xs:xss) = xs ++ **concat** xss

### Soluție folosind foldr

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

**concat** :: [Int] -> Int

**concat** xs = **foldr** (++) [] xs

### Exemplu

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

Quiz time!



<https://tinyurl.com/PF2023-C04-Quiz3>

## Exemplu – 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
```

## Exemplu – Suma pătratelor numerelor pozitive

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

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

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

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

## Exemplu - Compunerea funcțiilor

În definiția lui **foldr**

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

b poate fi tipul unei funcții.

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

`compose` = **foldr** (.) **id**

**Prelude**> `compose [(+1), (^2)] 3`

10

*-- functia (foldr (.) id [(+1), (^2)]) aplicata lui 3*

## foldr și foldl

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.

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

**foldr** op z [a1, a2, a3, ..., an] =  
a1 `op` (a2 `op` (a3 `op` (... (an `op` z) ...)))

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

**foldl** op z [a1, a2, a3, ..., an] =  
( ... (((z `op` a1) `op` a2) `op` a3) ...) `op` an

## Funcția **foldr**

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

**foldr** f i [] = i

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

## Funcția **foldl**

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

**foldl** h i [] = i

**foldl** h i (x:xs) = **foldl** h (h i x) xs



## Suma elementelor dintr-o listă

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

### Soluție cu foldr

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

Cu **foldr**, elementele sunt procesate de la dreapta la stânga:

$$\text{sum } [x_1, \dots, x_n] = (x_1 + (x_2 + \dots (x_n + 0) \dots))$$

## Suma elementelor dintr-o listă

Soluție în care elementele sunt procesate de la stânga la dreapta.

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

Elementele sunt procesate de la stânga la dreapta:

$$\text{suml } [x_1, \dots, x_n] 0 = (\dots (0 + x_1) + x_2) + \dots x_n$$

**Soluție cu foldl**

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

## Inversarea elementelor unei liste

Definiți o funcție care dată fiind o listă de elemente, calculează lista în care elementele sunt scrise în ordine inversă.

### Soluție cu foldl

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

```
rev = foldl (<:>) []
      where (<:>) = flip (:)
```

Elementele sunt procesate de la stânga la dreapta:

**rev**  $[x_1, \dots, x_n] = (\dots(([] \text{<:>} x_1) \text{<:>} x_2) \dots) \text{<:>} x_n$

## Evaluare leneșă. Liste infinite

Putem folosi funcțiile **map** și **filter** pe liste infinite:

```
Prelude> inf = map (+10) [1..] -- inf nu este evaluat
```

```
Prelude> take 3 inf
```

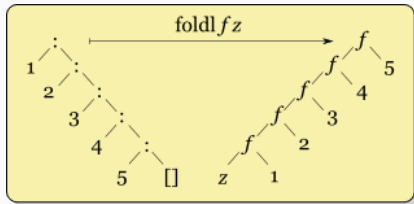
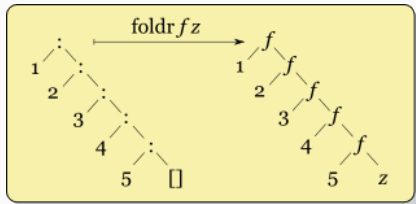
```
[11,12,13]
```

### Limbajul Haskell folosește implicit evaluarea leneșă

- expresiile sunt evaluate numai când este nevoie de valoarea lor
- expresiile nu sunt evaluate total, elementele care nu sunt folosite rămân neevaluate
- o expresie este evaluată o singură dată.

În exemplul de mai sus, este acceptată definiția lui `inf`, fără a fi evaluată. Când expresia `take 3 inf` este evaluată, numai primele 3 elemente ale lui `inf` sunt calculate, restul rămânând neevaluate.

# foldr și foldl



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

- **foldr** poate fi folosită pe liste infinite (în anumite cazuri)
- **foldl** nu poate fi folosită pe liste infinite niciodată

## foldr și foldl

- **foldr** poate fi folosită pe liste infinite (în anumite cazuri),
- **foldl** **nu** poate fi folosită pe liste infinite niciodată.

```
Prelude> foldr (*) 0 [1..]
```

\*\*\* Exception: stack overflow

```
Prelude> take 3 $ foldr (\x xs -> (x+1):xs) [] [1..]  
[2,3,4]
```

*-- foldr a functionat pe o lista infinita*

```
Prelude> take 3 $ foldl (\xs x -> (x+1):xs) [] [1..]  
-- expresia se calculeaza la infinit
```

Quiz time!



<https://tinyurl.com/PF2023-C05-Q1>

**Pe săptămâna viitoare!**