

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C03

Claudia Chiriță

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

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 folosind șabloane

flip f x y = f y x

- definiția cu lambda expresii

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!



<https://tinyurl.com/PF2023-C03-Quiz1>

Currying

Currying

Currying este procedeul prin care o funcție cu mai multe argumente este transformată într-o funcție care are un singur argument și întoarce o altă funcție.

- În Haskell toate funcțiile sunt în forma **curry**, deci au un singur argument.
- Operatorul \rightarrow pe tipuri este asociativ la dreapta, adică tipul $a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n$ îl gândim ca $a_1 \rightarrow (a_2 \rightarrow \dots (a_{n-1} \rightarrow a_n) \dots)$.
- Aplicarea funcțiilor este asociativă la stânga, adică expresia $f\ x_1 \dots x_n$ o gândim ca $(\dots ((f\ x_1)\ x_2) \dots x_n)$.

Funcțiile curry/uncurry și mulțimi

Prelude> :t curry

curry :: ((a, b) -> c) -> a -> b -> c

Prelude> :t uncurry

uncurry :: (a -> b -> c) -> (a, b) -> c

Exemplu:

f :: (Int , String) -> String

f (n,s) = take n s

Prelude> let cf = curry f

Prelude> :t cf

cf :: Int -> String -> String

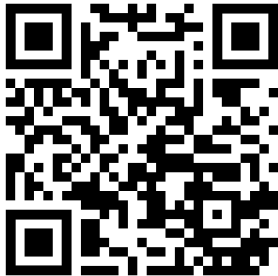
Prelude> f(1,"abc")

"a"

Prelude> cf 1 "abc"

"a"

Quiz time!



<https://tinyurl.com/PF2023-C03-Quiz2>

Procesarea fluxurilor de date: Map, Filter, Fold

Map:

transformarea fiecărui element dintr-o listă

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
```

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 xs = [ f x | x <- xs]
```

```
Prelude> map ($ 3) [(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!



<https://tinyurl.com/PF2023-C04-Quiz1>

Filter:

selectarea elementelor dintr-o listă

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"
```


Funcția filter

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!



<https://tinyurl.com/PF2023-C04-Quiz2>

Fold:

agregarea elementelor dintr-o listă

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]
```

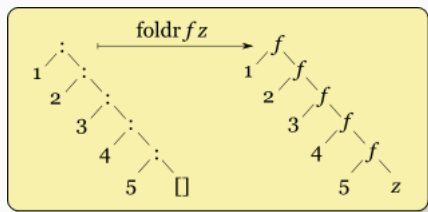
```
Prelude> concat ["con","ca","te","na","re"]
```

```
"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." ++ []))

Pe săptămâna viitoare!