

Programare declarativă¹

Tipuri de date, liste, funcții, recursie

Traian Florin Șerbănuță
Ioana Leuștean

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

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

Tipuri de date

Tipuri de date

- **Integer:** 4, 0, -5

```
Prelude> 4 + 3
```

```
Prelude> (+) 4 3
```

```
Prelude> mod 4 3
```

```
Prelude> 4 `mod` 3
```

- **Float:** 3.14

```
Prelude> truncate 3.14
```

```
Prelude> sqrt 4
```

```
Prelude> let x = 4 :: Int
```

```
Prelude> sqrt (fromIntegral x)
```

- **Char:** 'a','A', '\n'

```
Prelude> :m + Data.Char
```

```
Prelude> chr 65
```

```
Prelude> ord 'A'
```

```
Prelude> toUpper 'a'
```

```
Prelude> digitToInt '4'
```

Tipuri de date

- **Bool**: True, False

data Bool = True | False

Prelude> True && False || True
Prelude> not True

Prelude> 1 /= 2
Prelude> 1 == 2

- **String**: "prog\ndec"

type String = [Char] — *sinonim pentru tip*

Prelude> "aa"++"bb"
"aabb"
Prelude> "aabb" !! 2
'b'

Prelude> lines "prog\ndec"
["prog","dec"]
Prelude> words "pr og\nde cl"
["pr","og","de","cl"]

Tipuri de date compuse

- Tupluri - secvențe de de tipuri deja existente

```
Prelude> :t (1 :: Int , 'a' , "ab")
(1 :: Int , 'a' , "ab") :: (Int , Char , [Char])
Prelude> fst (1 , 'a')
Prelude> snd (1 , 'a')
```

- Tipul `unit`

```
Prelude> :t ()
() :: ()
```

- Liste

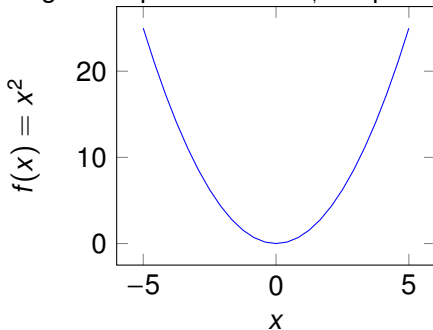
```
Prelude> [1,2,3] == 1:2:3:[]
True
```

Funcții

Ce e o funcție?

Ce e o funcție?

- DEX(online): Mărime variabilă care depinde de una sau de mai multe mărimi variabile independente
- O rețetă pentru a obține ieșiri din intrări: „Ridică un număr la pătrat”
- O relație între intrări și ieșiri $\{(1, 1), (2, 4), (3, 9), (4, 16), \dots\}$
- O ecuație algebrică $f(x) = x^2$
- Un grafic reprezentând ieșirile pentru intrările posibile





Tipuri de date

pentru intrări/ieșiri ale funcțiilor

- Integer: 4, 0, -5
- Float: 3.14
- Char: 'a'
- Bool: True, False
- String: "abc"
- Tuplu: (1,2)
- Lista: [1..100], [1..]

Tipuri de date

pentru intrări/ieșiri ale funcțiilor

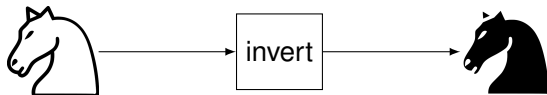
- Integer: 4, 0, -5
- Float: 3.14
- Char: 'a'
- Bool: True, False
- String: "abc"
- Tuplu: (1,2)
- Lista: [1..100], [1..]
- Picture: , 

Tipuri de funcții și aplicarea lor

`invert :: Picture -> Picture`

`knight :: Picture`

`invert knight`



Compunerea funcțiilor

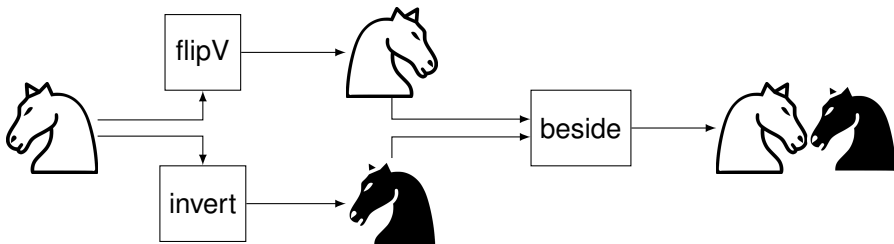
`beside :: Picture -> Picture -> Picture`

`flipV :: Picture -> Picture`

`invert :: Picture -> Picture`

`knight :: Picture`

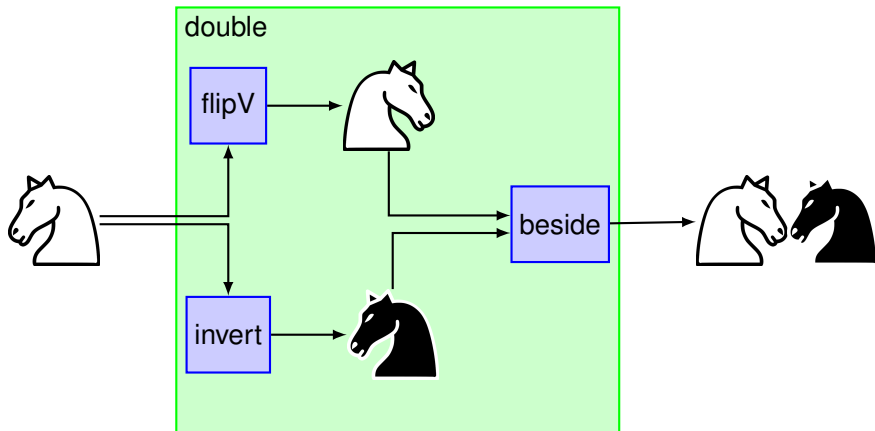
`beside (flipV knight) (invert knight)`



Definirea unei funcții noi

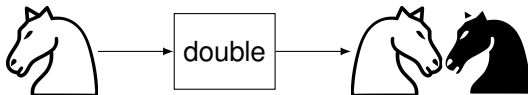
```
double :: Picture -> Picture  
double p = beside (flipV p) (invert p)
```

```
double knight
```



Definirea unei funcții noi

```
double :: Picture -> Picture  
double p = beside (flipV p) (invert p)  
  
double knight
```



Terminologie

Prototipul funcției

`double` :: Picture -> Picture

- Numele funcției
- Signatura funcției

Definiția funcției

`double p` = beside (flipV p) (invert p)

- numele funcției
- parametrul formal
- corpul funcției

Aplicarea funcției

`double knight`

- numele funcției
- parametrul actual (argumentul)

Liste

Operatorii : și ++

Mod de folosire

```
Prelude> :t (:)  
(:) :: a -> [a] -> [a]
```

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

```
Prelude> :t "bcd"  
"bc" :: [Char]
```

```
Prelude> 'a' : "bcd"  
"abcd"
```

```
Prelude> :t (++)  
(++) :: [a] -> [a] -> [a]
```

```
Prelude> [1] ++ [2,3]  
[1,2,3]
```

```
Prelude> [1,2] ++ [3]  
[1,2,3]
```

```
Prelude> "a" ++ "bcd"  
"abcd"
```

```
Prelude> "ab" ++ "cd"  
"abcd"
```

: (**cons**) Construiește o listă nouă având primul argument ca prim element și continuând cu al doilea argument ca restul listei.

++ (**append**) Construiește o listă nouă obținută prin alipirea celor două liste argument

[**Char**] Șirurile de caractere (**String**) sunt liste de caractere (**Char**)

Operatorii : și ++

Erori de începător

```
Prelude> :t (:)  
(:) :: a -> [a] -> [a]
```

```
Prelude> :t (++)  
(++) :: [a] -> [a] -> [a]
```

```
Prelude> [1,2] : 3  
-- eroare de tipuri
```

```
Prelude> 1 ++ [2,3]  
-- eroare de tipuri
```

```
Prelude> [1] : [2,3]  
-- eroare de tipuri
```

```
Prelude> "ab" : 'c'  
-- eroare de tipuri
```

```
Prelude> 'a' ++ "bc"  
-- eroare de tipuri
```

```
Prelude> "a" : "bc"  
-- eroare de tipuri
```

Liste

Definiție

Observație

Orice listă poate fi scrisă folosind doar constructorul `(:)` și lista vidă `[]`

- $[1,2,3] == 1 : (2 : (3 : [])) == 1 : 2 : 3 : []$
- $"abcd" == ['a','b','c','d'] == 'a' : ('b' : ('c' : ('d' : []))) == 'a' : 'b' : 'c' : 'd' : []$

Definiție recursivă

O listă este

- vidă, notată `[]`; sau
- compusă, notată `x:xs`, dintr-un element `x` numit **capul listei (head)** și o listă `xs` numită **coada listei (tail)**.

Definiții de liste

- Intervale și progresii

<code>interval = ['c'..'e']</code>	--	<code>['c', 'd', 'e']</code>
<code>progresie = [20,17..1]</code>	--	<code>[20,17,14,11,8,5,2]</code>
<code>progresie' = [2.0,2.5..4.0]</code>	--	<code>[2.0,2.5,3.0,3.5,4.0]</code>

Definiții de liste

- Intervale și progresii

```
interval = ['c'..'e']      -- ['c', 'd', 'e']
progresie = [20,17..1]     -- [20,17,14,11,8,5,2]
progresie' = [2.0,2.5..4.0] -- [2.0,2.5,3.0,3.5,4.0]
```

- Definiții prin selecție (comprehensiune)

$[E(x) \mid x \leftarrow [x_1, \dots, x_n], P(x)]$

Prelude> let xs = [0..10]

Prelude> [x | x<-xs, even x]

[0,2,4,6,8,10]

Definiții de liste

- Intervale și progresii

```
interval = ['c'..'e']           -- ['c', 'd', 'e']
progresie = [20,17..1]          -- [20,17,14,11,8,5,2]
progresie' = [2.0,2.5..4.0]     -- [2.0,2.5,3.0,3.5,4.0]
```

- Definiții prin selecție (comprehensiune)

$[E(x) \mid x \leftarrow [x_1, \dots, x_n], P(x)]$

```
Prelude> let xs = [0..10]
```

```
Prelude> [x | x<-xs, even x]
```

```
[0,2,4,6,8,10]
```

```
Prelude> let xs = [0..6]
```

```
Prelude> [(x,y) | x<-xs, y<-xs, x+y == 10]
```

```
[(4,6),(5,5),(6,4)]
```

Definiții de liste

- Intervale și progresii

```
interval = ['c'..'e']           -- ['c', 'd', 'e']
progresie = [20,17..1]          -- [20,17,14,11,8,5,2]
progresie' = [2.0,2.5..4.0]     -- [2.0,2.5,3.0,3.5,4.0]
```

- Definiții prin selecție (comprehensiune)

$[E(x) \mid x \leftarrow [x_1, \dots, x_n], P(x)]$

Prelude> let xs = [0..10]

Prelude> [x | x<-xs, even x]

[0,2,4,6,8,10]

Prelude> let xs = [0..6]

Prelude> [(x,y) | x<-xs, y<-xs, x+y == 10]

[(4,6),(5,5),(6,4)]

Prelude> [(i,j) | i<-[1..3], let k=i*i, j<-[1..k]]

Observați folosirea lui **let** pentru declarații locale!

Procesarea listelor

```
Prelude> head [1,2,3]
```

```
1
```

```
Prelude> tail [1,2,3]
```

```
[2,3]
```

```
Prelude> null [1,2,3]
```

```
False
```

```
Prelude> null []
```

```
True
```

Evaluare leneșă

Argumentele sunt evaluate doar cand e necesar si doar cat e necesar

```
Prelude> head []
```

```
*** Exception: Prelude.head: empty list
```

```
Prelude> let x = head []
```

```
Prelude> let f a = 5
```

```
Prelude> f x
```

```
5
```

```
Prelude> head [1,head [],3]
```

```
1
```

```
Prelude> head [head [],3]
```


Procesarea listelor

Evaluare leneșă

Se pot defini liste infinite (fluxuri de date)

```
Prelude> let natural = [0,...]  
Prelude> take 5 natural  
[0,1,2,3,4]
```

Procesarea listelor

Evaluare leneșă

Se pot defini liste infinite (fluxuri de date)

```
Prelude> let natural = [0,..]
```

```
Prelude> take 5 natural  
[0,1,2,3,4]
```

```
Prelude> let evenNat = [0,2..] -- progresie infinită
```

```
Prelude> take 7 evenNat  
[0,2,4,6,8,10,12]
```

Procesarea listelor

Evaluare leneșă

Se pot defini liste infinite (fluxuri de date)

```
Prelude> let natural = [0,..]
```

```
Prelude> take 5 natural  
[0,1,2,3,4]
```

```
Prelude> let evenNat = [0,2..] -- progresie infinită a
```

```
Prelude> take 7 evenNat  
[0,2,4,6,8,10,12]
```

```
Prelude> let ones = [1,1..]
```

```
Prelude> let zeros = [0,0..]
```

```
Prelude> let both = zip ones zeros
```

```
Prelude> take 5 both  
[(1,0),(1,0),(1,0),(1,0),(1,0)]
```

Funcții și recursie

Funcții și recursie - Probleme

- Transformarea fiecărui element dintr-o listă
- Selectarea elementelor dintr-o listă
- Agregarea elementelor dintr-o listă
- Mapare, filtrare și agregare deodată

Transformarea fiecărui element dintr-o listă

Problemă și abordare

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

Soluție descriptivă

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

Soluție recursivă

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

Variante recursive

Ecuational (pattern matching)

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

Condițional (cu operatori de legare)

```
squaresCond :: [Int] -> [Int]
squaresCond ys =
  if null ys then []
  else let
    x = head ys
    xs = tail ys
  in
    x*x : squaresCond xs
```

Recursia în acțiune

`squaresRec :: [Int] -> [Int]`

`squaresRec [1,2,3]`

`squaresRec [] = []`

`squaresRec (x:xs) = x*x : squaresRec xs`

Recursia în acțiune

`squaresRec :: [Int] -> [Int]`

`squaresRec [] = []`

`squaresRec (x:xs) = x*x : squaresRec xs`

`squaresRec [1,2,3]`

`=`

`squaresRec (1 : (2 : (3 : [])))`

Recursia în acțiune

```

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

squaresRec [1,2,3]
=
squaresRec (1 : (2 : (3 : [])))
=
1 * 1 : squaresRec (2 : (3 : []))

```

$\{x \mapsto 1, xs \mapsto 2 : (3 : [])\}$

Recursia în acțiune

```

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

squaresRec [1,2,3]
=
squaresRec (1 : (2 : (3 : [])))
=
1 * 1 : squaresRec (2 : (3 : []))
=
1 * 1 : (2 * 2 : squaresRec (3 : []))

```

{x ↦ 2, xs ↦ 3 : []}

Recursia în acțiune

```

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

squaresRec [1,2,3]
=
squaresRec (1 : (2 : (3 : [])))
=
1 * 1 : squaresRec (2 : (3 : []))
=
1 * 1 : (2 * 2 : squaresRec (3 : []))
=
1 * 1 : (2 * 2 : (3 * 3 : squaresRec []))

```

{x ↦ 3, xs ↦ []}

Recursia în acțiune

```

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

squaresRec [1,2,3]
=
squaresRec (1 : (2 : (3 : [])))
=
1 * 1 : squaresRec (2 : (3 : []))
=
1 * 1 : (2 * 2 : squaresRec (3 : []))
=
1 * 1 : (2 * 2 : ( 3 * 3 : squaresRec []))
=
1 * 1 : (2 * 2 : ( 3 * 3 : []))

```

Recursia în acțiune

$\text{squaresRec} :: [\text{Int}] \rightarrow [\text{Int}]$
 $\text{squaresRec []} = []$
 $\text{squaresRec (x:xs)} = x*x : \text{squaresRec xs}$

$\text{squaresRec [1,2,3]}$
 $=$
 $\text{squaresRec (1 : (2 : (3 : [])))}$
 $=$
 $1 * 1 : \text{squaresRec (2 : (3 : []))}$
 $=$
 $1 * 1 : (2 * 2 : \text{squaresRec (3 : [])})$
 $=$
 $1 * 1 : (2 * 2 : (3 * 3 : \text{squaresRec []}))$
 $=$
 $1 * 1 : (2 * 2 : (3 * 3 : []))$
 $=$
 $1 : (4 : (9 : []))$

Recursia în acțiune

$\text{squaresRec} :: [\text{Int}] \rightarrow [\text{Int}]$
 $\text{squaresRec []} = []$
 $\text{squaresRec (x:xs)} = x*x : \text{squaresRec xs}$

$\text{squaresRec [1,2,3]}$
 $=$
 $\text{squaresRec (1 : (2 : (3 : [])))}$
 $=$
 $1 * 1 : \text{squaresRec (2 : (3 : []))}$
 $=$
 $1 * 1 : (2 * 2 : \text{squaresRec (3 : [])})$
 $=$
 $1 * 1 : (2 * 2 : (3 * 3 : \text{squaresRec []}))$
 $=$
 $1 * 1 : (2 * 2 : (3 * 3 : []))$
 $=$
 $1 : (4 : (9 : [])) = [1,4,9]$

Selectarea elementelor dintr-o listă

Problemă și abordare

Definiți o funcție care dată fiind o listă de numere întregi selectează doar elementele impare din listă.

Soluție descriptivă

```
odds :: [Int] -> [Int]
odds xs = [ x | x <- xs, odd x ]
```

Soluție recursivă

```
oddsRec :: [Int] -> [Int]
oddsRec [] = []
oddsRec (x:xs) | odd x = x : oddsRec xs
                | otherwise = oddsRec xs
```


Variante recursive

Ecuational (pattern matching)

```
oddsRec :: [Int] -> [Int]
oddsRec [] = []
oddsRec (x:xs) | odd x      = x : oddsRec xs
                | otherwise = oddsRec xs
```

Condițional (cu operatori de legare)

```
oddsCond :: [Int] -> [Int]
oddsCond ys =
  if null ys then []
  else let
    x  = head ys
    xs = tail ys
  in
    if odd x then x : oddsCond xs
    else oddsCond xs
```

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]    oddsRec []           = []  
oddsRec (x:xs) | odd x      = x : oddsRec xs  
               | otherwise = oddsRec xs
```

oddsRec [1,2,3]

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
oddsRec [] = []
oddsRec (x:xs) | odd x = x : oddsRec xs
                | otherwise = oddsRec xs
```

```
oddsRec [1,2,3]
=
oddsRec (1 : (2 : (3 : [])))
```

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]    oddsRec []                = []
oddsRec (x:xs) | odd x      = x : oddsRec xs
                | otherwise = oddsRec xs
```

```
oddsRec [1,2,3]
=
oddsRec (1 : (2 : (3 : [])))
=
1 : oddsRec (2 : (3 : []))
```

$\{x \mapsto 1, xs \mapsto 2 : (3 : [])\}; \text{odd } 1 = \text{True}$

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]    oddsRec []                = []
oddsRec (x:xs) | odd x      = x : oddsRec xs
                | otherwise = oddsRec xs
```

```
oddsRec [1,2,3]
=
oddsRec (1 : (2 : (3 : [])))
=
1 : oddsRec (2 : (3 : []))
=
1 : oddsRec (3 : [])
```

$\{x \mapsto 2, xs \mapsto 3 : []\}; \text{odd } 2 = \text{False}$

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
oddsRec [] = []
oddsRec (x:xs) | odd x = x : oddsRec xs
                | otherwise = oddsRec xs
```

```
oddsRec [1,2,3]
=
oddsRec (1 : (2 : (3 : [])))
=
1 : oddsRec (2 : (3 : []))
=
1 : oddsRec (3 : [])
=
1 : (3 : oddsRec [])
```

$\{x \mapsto 3, xs \mapsto []\}; \text{odd } 3 = \text{True}$

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]  oddsRec []           = []
oddsRec (x:xs) | odd x    = x : oddsRec xs
                | otherwise = oddsRec xs
```

```
oddsRec [1,2,3]
=
oddsRec (1 : (2 : (3 : [])))
=
1 : oddsRec (2 : (3 : []))
=
1 : oddsRec (3 : [])
=
1 : (3 : oddsRec [])
=
1 : (3 : [])
```

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
oddsRec [] = []
oddsRec (x:xs) | odd x    = x : oddsRec xs
                | otherwise = oddsRec xs
```

```
oddsRec [1,2,3]
=
oddsRec (1 : (2 : (3 : [])))
=
1 : oddsRec (2 : (3 : []))
=
1 : oddsRec (3 : [])
=
1 : ( 3 : oddsRec [])
=
1 : ( 3 : [] ) = [1,3]
```


Agregarea elementelor dintr-o listă

Problemă și abordare

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

Soluție recursivă

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

Recursia în acțiune

`suma :: [Int] -> Int`

`suma [1,2,3]`

`suma [] = 0`

`suma (x:xs) = x + suma xs`

Recursia în acțiune

$\text{suma} :: [\text{Int}] \rightarrow \text{Int}$

$\text{suma } [1,2,3]$

$=$

$\text{suma } (1 : (2 : (3 : [])))$

$\text{suma } [] = 0$

$\text{suma } (x:xs) = x + \text{suma } xs$

Recursia în acțiune

`suma :: [Int] -> Int`

`suma [1,2,3]`

`=`

`suma (1 : (2 : (3 : [])))`

`=`

`1 + suma (2 : (3 : []))`

`suma [] = 0`

`suma (x:xs) = x + suma xs`

$\{x \mapsto 1, xs \mapsto 2 : (3 : [])\}$

Recursia în acțiune

`suma :: [Int] -> Int`

`suma [1,2,3]`

`=`

`suma (1 : (2 : (3 : [])))`

`=`

`1 + suma (2 : (3 : []))`

`=`

`1 + (2 + suma (3 : []))`

`suma [] = 0`

`suma (x:xs) = x + suma xs`

$\{x \mapsto 2, xs \mapsto 3 : []\}$

Recursia în acțiune

$\text{suma} :: [\text{Int}] \rightarrow \text{Int}$

$\text{suma } [1,2,3]$

$=$

$\text{suma } (1 : (2 : (3 : [])))$

$=$

$1 + \text{suma } (2 : (3 : []))$

$=$

$1 + (2 + \text{suma } (3 : []))$

$=$

$1 + (2 + (3 + \text{suma } []))$

$\text{suma } [] = 0$

$\text{suma } (x:xs) = x + \text{suma } xs$

$\{x \mapsto 3, xs \mapsto []\}$

Recursia în acțiune

`suma :: [Int] -> Int`

`suma [1,2,3]`

`=`

`suma (1 : (2 : (3 : [])))`

`=`

`1 + suma (2 : (3 : []))`

`=`

`1 + (2 + suma (3 : []))`

`=`

`1 + (2 + (3 + suma []))`

`=`

`1 + (2 + (3 + 0))`

`suma [] = 0`

`suma (x:xs) = x + suma xs`

Recursia în acțiune

$\text{suma} :: [\text{Int}] \rightarrow \text{Int}$

$\text{suma } [1,2,3]$

$=$

$\text{suma } (1 : (2 : (3 : [])))$

$=$

$1 + \text{suma } (2 : (3 : []))$

$=$

$1 + (2 + \text{suma } (3 : []))$

$=$

$1 + (2 + (3 + \text{suma } []))$

$=$

$1 + (2 + (3 + 0)) = 6$

$\text{suma } [] = 0$

$\text{suma } (x:xs) = x + \text{suma } xs$

Problemă și abordare

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

Soluție recursivă

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

Recursia în acțiune

`produs :: [Int] -> Int`

`produs [1,2,3]`

`produs [] = 1`

`produs (x:xs) = x * produs xs`

Recursia în acțiune

`produs :: [Int] -> Int`

`produs [1,2,3]`

`=`

`produs (1 : (2 : (3 : [])))`

`produs [] = 1`

`produs (x:xs) = x * produs xs`

Recursia în acțiune

`produs :: [Int] -> Int`

`produs [1,2,3]`

`=`

`produs (1 : (2 : (3 : [])))`

`=`

`1 * produs (2 : (3 : []))`

`produs [] = 1`

`produs (x:xs) = x * produs xs`

$\{x \mapsto 1, xs \mapsto 2 : (3 : [])\}$

Recursia în acțiune

produs :: [Int] -> Int

produs [1,2,3]

=

produs (1 : (2 : (3 : [])))

=

1 * produs (2 : (3 : []))

=

1 * (2 * produs (3 : []))

produs [] = 1

produs (x:xs) = x * produs xs

$\{x \mapsto 2, xs \mapsto 3 : []\}$

Recursia în acțiune

`produs :: [Int] -> Int`

`produs [1,2,3]`

`=`

`produs (1 : (2 : (3 : [])))`

`=`

`1 * produs (2 : (3 : []))`

`=`

`1 * (2 * produs (3 : []))`

`=`

`1 * (2 * (3 * produs []))`

`produs [] = 1`

`produs (x:xs) = x * produs xs`

$\{x \mapsto 3, xs \mapsto []\}$

Recursia în acțiune

`produs :: [Int] -> Int`

`produs [1,2,3]`

`=`

`produs (1 : (2 : (3 : [])))`

`=`

`1 * produs (2 : (3 : []))`

`=`

`1 * (2 * produs (3 : []))`

`=`

`1 * (2 * (3 * produs []))`

`=`

`1 * (2 * (3 * 1))`

`produs [] = 1`

`produs (x:xs) = x * produs xs`

Recursia în acțiune

`produs :: [Int] -> Int`

`produs [] = 1`
`produs (x:xs) = x * produs xs`

`produs [1,2,3]`
`=`
`produs (1 : (2 : (3 : [])))`
`=`
`1 * produs (2 : (3 : []))`
`=`
`1 * (2 * produs (3 : []))`
`=`
`1 * (2 * (3 * produs []))`
`=`
`1 * (2 * (3 * 1)) = 6`

Mapare, filtrare și agregare deodată

Problemă și abordare

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

Soluție descriptivă

```
sumSqOdd :: [Int] -> Int
sumSqOdd xs = sum [ x * x | x <- xs, odd x ]
```

Soluție recursivă

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

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
```

```
sumSqOddRec [] = 0
```

```
sumSqOddRec (x:xs) | odd x    = x*x + sumSqOddRec xs  
                   | otherwise = sumSqOddRec xs
```

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

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
```

```
sumSqOddRec [] = 0
```

```
sumSqOddRec (x:xs) | odd x    = x*x + sumSqOddRec xs  
                   | otherwise = sumSqOddRec xs
```

```
sumSqOddRec [1,2,3] =
```

```
sumSqOddRec (1 : (2 : (3 : [])))
```

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
```

```
sumSqOddRec [] = 0
```

```
sumSqOddRec (x:xs) | odd x = x*x + sumSqOddRec xs
                   | otherwise = sumSqOddRec xs
```

```
sumSqOddRec [1,2,3] =
```

```
sumSqOddRec (1 : (2 : (3 : [])))
```

```
=
```

```
1 * 1 + sumSqOddRec (2 : (3 : []))
```

$\{x \mapsto 1, xs \mapsto 2 : (3 : [])\}; \text{odd } 1 = \text{True}$

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
```

```
sumSqOddRec [] = 0
```

```
sumSqOddRec (x:xs) | odd x = x*x + sumSqOddRec xs
                   | otherwise = sumSqOddRec xs
```

```
sumSqOddRec [1,2,3] =
```

```
sumSqOddRec (1 : (2 : (3 : [])))
```

```
=
```

```
1 * 1 + sumSqOddRec (2 : (3 : []))
```

```
=
```

```
1 * 1 + sumSqOddRec (3 : [])
```

```
{x ↦ 2, xs ↦ 3 : []}; odd 2 = False
```

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
```

```
sumSqOddRec [] = 0
```

```
sumSqOddRec (x:xs) | odd x = x*x + sumSqOddRec xs
                   | otherwise = sumSqOddRec xs
```

```
sumSqOddRec [1,2,3] =
```

```
sumSqOddRec (1 : (2 : (3 : [])))
```

```
=
```

```
1 * 1 + sumSqOddRec (2 : (3 : []))
```

```
=
```

```
1 * 1 + sumSqOddRec (3 : [])
```

```
=
```

```
1 * 1 + ( 3 * 3 + sumSqOddRec [])
```

$\{x \mapsto 3, xs \mapsto []\}; \text{odd } 3 = \text{True}$

Recursia în acțiune

`oddsRec :: [Int] -> [Int]`

`sumSqOddRec [] = 0`

`sumSqOddRec (x:xs) | odd x = x*x + sumSqOddRec xs`
`| otherwise = sumSqOddRec xs`

`sumSqOddRec [1,2,3] =`

`sumSqOddRec (1 : (2 : (3 : [])))`

`=`

`1 * 1 + sumSqOddRec (2 : (3 : []))`

`=`

`1 * 1 + sumSqOddRec (3 : [])`

`=`

`1 * 1 + (3 * 3 + sumSqOddRec [])`

`=`

`1 * 1 + (3 * 3 + 0)`

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
```

```
sumSqOddRec [] = 0
```

```
sumSqOddRec (x:xs) | odd x    = x*x + sumSqOddRec xs
                   | otherwise = sumSqOddRec xs
```

```
sumSqOddRec [1,2,3] =
```

```
sumSqOddRec (1 : (2 : (3 : [])))
```

```
=
```

```
1 * 1 + sumSqOddRec (2 : (3 : []))
```

```
=
```

```
1 * 1 + sumSqOddRec (3 : [])
```

```
=
```

```
1 * 1 + ( 3 * 3 + sumSqOddRec [])
```

```
=
```

```
1 * 1 + ( 3 * 3 + 0 ) = 10
```