

Programare declarativă¹

Introducere în programarea funcțională folosind Haskell

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

Sintaxă

- Comentarii

```
-- comentariu pe o linie
{- comentariu pe
   mai multe
   linii -}
```

- Identificatori

- siruri formate din litere, cifre, caracterele `_` si `'` (single quote)
- incep cu o litera

- Haskell este case sensitive

- identificatorii pentru variabile incep cu litera mica
- identificatorii pentru constructori incep cu litera mare

```
let double x = 2 * x
data Point a = Pt a a
```

Sinatxa

Blocuri si indentare

Blocurile sunt delimitate prin indentare.

```
fact n = if n == 0 then 1
        else n * fact(n-1)
```

```
trei = let
        a = 1
        b = 2
      in (a + b)
```

- echivalent, putem scrie

```
trei = let a = 1; b = 2 in (a + b)
```

Module

Program in Haskell

Un program in Haskell este o colectie de module.

- modulele contin declaratii de functii, tipuri si clase
- modulele sunt scrise in fisiere; un fisier contine un singur modul, numele fisierului coincide cu numele modulului si incepe cu litera mare

```
module MyDouble where
```

```
double :: Integer -> Integer
double x = x + x
```

- modulele pot fi importate

```
import MyDouble
```

Variabile

= reprezintă o legătură (binding)

În Haskell, variabilele sunt imuabile

- dacă fișierul test.hs conține

```
x=1
x=2
```

```
Prelude> :l test.hs
```

```
test.hs:2:1: error:
  Multiple declarations of 'x'
  Declared at: test.hs:1:1
               test.hs:2:1
```

```
2 | x=2
  | ^
```

Legarea variabilelor

let .. in ...

crează scop local

- dacă fișierul testlet.hs conține

```
x=1
z= let x=3 in x
```

```
Prelude> :l testlet.hs
[1 of 1] Compiling Main
Ok, 1 module loaded.
```

```
*Main> z
3
*Main> x
1
```

Legarea variabilelor

- let .. in ... crează scop local

```
x = let
      z = 5
      g u = z + u
    in let
      z = 7
    in (g 0 + z)  -- x=12
```

```
x= let z=5; g u = z+u in let z=7 in g 0 -- x=5
```

- ... where ... crează scop local

```
f x = (g x) + (g x) + z
      where g x = 2*x
            z = x-1
```

Legarea variabilelor

- let .. in ... este o expresie

```
x = [ let y =8 in y, 9]  -- x=[8,9]
```

- where este o clauză

```
x = [y where y =8, 9]  -- error: parse error ...
```

Clauza **where** poate fi folosită pentru a defini funcții și expresii **case**.

<pre>h x x == 0 = 0 x == 1 = y + 1 x == 2 = y * y otherwise = y where y = x*x</pre>	<pre>f x = case x of 0 -> 0 1 -> y + 1 2 -> y * y otherwise -> y where y = x*x</pre>
---	--

Expresii și funcții

Signatura unei funcții

`fact :: Integer -> Integer`

- Definiții folosind `if`

```
fact n = if n == 0 then 1
        else n * fact(n-1)
```

- Definiții folosind ecuații

```
fact 0 = 1
fact n = n * fact(n-1)
```

- Definiții folosind cazuri

```
fact n
| n == 0    = 1
| otherwise = n * fact(n-1)
```

Șabloane (patterns)

- `x:y = [1,2,3] -- x=1 si y =[2,3]`

Observati ca : este constructorul pentru liste

- `(u,v)=('a' ,[(1, 'a') ,(2, 'b')])` -- `u='a'`,
-- `v=[(1, 'a') ,(2, 'b')]`
`(_,_)` este un tip compus

- Definitii folosind `case...of`

`selectie :: (Integer, String) -> String`

```
selectie (x,s) = case (x,s) of
    (0,_) -> s
    (1, z:zs) -> zs
    (1, []) -> []
    (_,_) -> (s ++ s)
```

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]
progresieInfinita = [3,7..] -- [3,7,11,15,19,...]
```

- Definiții prin selecție

```
pare :: [Integer] -> [Integer]
pare xs = [x | x<-xs, even x]

pozitiiPare :: [Integer] -> [Integer]
pozitiiPare xs = [i | (i,x) <- [1..] 'zip' xs, even x]
```

Sistemul tipurilor

"There are three interesting aspects to types in Haskell: they are strong, they are static, and they can be automatically inferred."

<http://book.realworldhaskell.org/read/types-and-functions.html>

tare garanteaza absenta anumitor erori

static tipul fiecari valori este calculat la compilare

dedus automat compilatorul deduce automat tipul fiecărei expresii

```
Prelude> :t [( 'a' ,1 ,"abc" )]
[( 'a' ,1 ,"abc" )] :: Num b => [(Char, b, [Char])]
```

Sistemul tipurilor

Tipurile de baza

Int Integer Float Double Bool Char String

- tipuri compuse: tupluri si liste

```
Prelude> :t [( 'a',1,"abc")]
[( 'a',1,"abc")] :: Num b => [(Char, b, [Char])]
Prelude> :t ["ana", "ion"]
["ana", "ion"] :: [[Char]]
```

- tipuri noi definite de utilizator

```
data RGB = Rosu|Verde|Albastru
data Point a = Pt a a    -- tip parametrizat
                        -- a este variabila de tip
```

Tipuri. Clase de tipuri. Variabile de tip. Signaturi de tip

```
Prelude> :t 'a'
'a' :: Char
Prelude> :t "ana"
"ana" :: [Char]
Prelude> :t 1
1 :: Num a => a
Prelude> :t [1,2,3]
[1,2,3] :: Num t => [t]
Prelude> :t 3.5
3.5 :: Fractional a => a
Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude> :t (+3)
(+3) :: Num a => a -> a
Prelude> :t (3+)
(3+) :: Num a => a -> a
```

Expresii ca valori

Funcțiile — „cetățeni de rangul I”

- Funcțiile sunt valori care pot fi luate ca argument sau întoarse ca rezultat

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = \ x y -> f y x
-- sau alternativ folosind matching
flip f x y = f y x
-- sau flip ca valoare de tip functie
flip = \ f x y -> f y x
-- Currying
flip = \f -> \x -> \y -> f y x
```

- Aplicare parțială a funcțiilor

```
inumatateste :: Integral a => a -> a
inumatateste = ('div' 2)
```

Funcții de ordin înalt

map, filter, foldl, foldr

```
Prelude> map (*3) [1,3,4]
[3,9,12]
Prelude> filter (>=2) [1,3,4]
[3,4]
Prelude> foldr (*) 1 [1,3,4]
12
Prelude> foldl (flip (:)) [] [1,3,4]
[4,3,1]
```

Compunere si aplicare

```
Prelude> map (*3) ( filter (<=3) [1,3,4])
[3,9]
Prelude> map (*3) . filter (<=3) [1,3,4]
[3,9]
```

Lenevire

- 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]
```

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

- Liste infinite (fluxuri de date)

```
ones = [1, 1..]
```

```
zeros = [0, 0..]
```

```
both = zip ones zeros
```

```
short = take 5 both -- [(1,0), (1,0), (1,0), (1,0), (1,0)]
```

Interacțiuni cu mediul extern

- Monade
- Acțiuni
- Secvențiere

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 ()
() :: ()
```

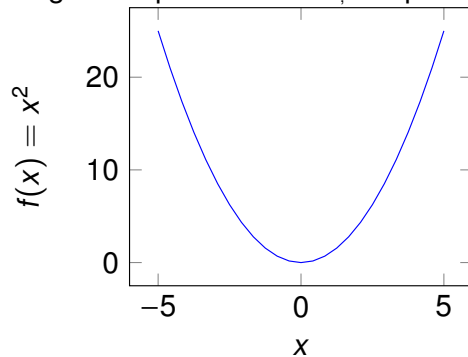
- **Liste**

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

Funcții



Ce e o funcție?

- 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

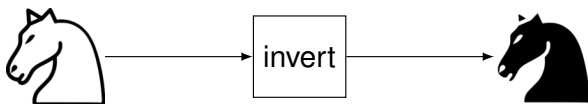
- Integer: 4, 0, -5
- Float: 3.14
- Char: 'a'
- Bool: True, False
- String: "abc"
- Tuplu: (1,2)
- Lista: [1..100], [1..]
- Picture: , 

Tipuri de funcții și aplicarea lor

`invert :: Picture -> Picture`

`knight :: Picture`

`invert knight`



Compunerea funcțiilor

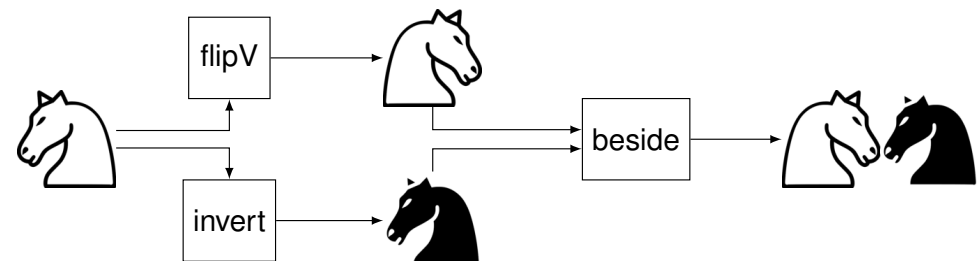
`beside :: Picture -> Picture -> Picture`

`flipV :: Picture -> Picture`

`invert :: Picture -> Picture`

`knight :: Picture`

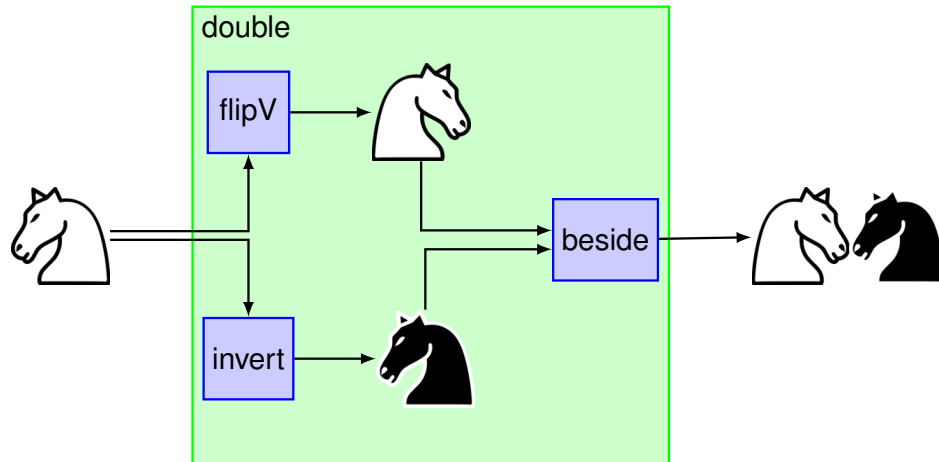
`beside (flipV knight) (invert knight)`



Definirea unei funcții noi

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

```
double knight
```



Liste

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)

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**) Construieste o listă nouă având primul argument ca prim element și continuând cu al doilea argument ca restul listei.

++ (**append**) Construieste 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

```
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) | x <- [x1,...,xn], 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]
```

```
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 [] = []
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 : []))
=
1 : (4 : (9 : [])) = [1,4,9]
```

{x ↦ 1, xs ↦ 2 : (3 : [])}
{x ↦ 2, xs ↦ 3 : []}
{x ↦ 3, xs ↦ []}

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]
=
oddsRec (1 : (2 : (3 : [])))
=
{ x ↦ 1, xs ↦ 2 : (3 : []) }; odd 1 = True
1 : oddsRec (2 : (3 : []))
=
{ x ↦ 2, xs ↦ 3 : [] }; odd 2 = False
1 : oddsRec (3 : [])
=
{ x ↦ 3, xs ↦ [] }; odd 3 = True
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 [] = 0
suma (x:xs) = x + suma xs

suma [1,2,3]
=
suma (1 : (2 : (3 : [])))
=
{ x ↦ 1, xs ↦ 2 : (3 : []) }
1 + suma (2 : (3 : []))
=
{ x ↦ 2, xs ↦ 3 : [] }
1 + (2 + suma (3 : []))
=
{ x ↦ 3, xs ↦ [] }
1 + (2 + (3 + suma []))
=
1 + (2 + (3 + 0)) = 6

```

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
 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

{x ↦ 1, xs ↦ 2 : (3 : [])}

{x ↦ 2, xs ↦ 3 : []}

{x ↦ 3, xs ↦ []}

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] =
 sumSqOddRec (1 : (2 : (3 : [])))
 = {x ↦ 1, xs ↦ 2 : (3 : [])}; odd 1 = True
 1 * 1 + sumSqOddRec (2 : (3 : []))
 = {x ↦ 2, xs ↦ 3 : []}; odd 2 = False
 1 * 1 + sumSqOddRec (3 : [])
 = {x ↦ 3, xs ↦ []}; odd 3 = True
 1 * 1 + (3 * 3 + sumSqOddRec [])
 =
 1 * 1 + (3 * 3 + 0) = 10

Programare declarativă¹

Operatori, Funcții (din nou), Recursie (din nou)

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

Operatori în formă infixă

Operatorii sunt funcții

Operatorii în Haskell

- sunt definiți folosind numai "simboluri" (ex: `*!*`)
- au două argumente
- sunt apelați folosind notația infix

- Operatori predefiniți

`(||) :: Bool -> Bool -> Bool`

`True || True = True`

`False || True = True`

`True || False = True`

`False || False = False`

- Operatori definiți de utilizator

`(&&) :: Bool -> Bool -> Bool -- atentie la paranteze`

`True && b = b`

`False && _ = False`

Funcțiile sunt operatori

Operatori aritmetici

Prelude> mod 5 2

1

Prelude> 5 'mod' 2

1

`divide :: Int -> Int -> Bool`

`x 'divide' y = y 'mod' x == 0`

`apartine :: Int -> [Int] -> Bool`

`x 'apartine' [] = False`

`x 'apartine' (y:xs) = x == y || (x 'apartine' xs)`

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'

Declararea precedenței și a modului de grupare

infix, infixl, infixr

infixl 6 <+>

(<+>) :: Int -> Int -> Int

x <+> y = x + y + 1

***Main> 1 <+> 2 * 3 <+> 4**

13

infix 4 'egal'

egal :: Float -> Float -> Bool

x 'egal' y = abs(x - y) <= 0.001

***Main> 1 / 32 'egal' 1 / 33**

True

De ce?

De ce este operatorul - asociativ la stanga?

$5 - 2 - 1 == (5 - 2) - 1$ **--** $\neq 5 - (2 - 1)$

De ce este operatorul : asociativ la dreapta?

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

De ce este operatorul ++ asociativ la dreapta?

(++) :: [a] -> [a] -> [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
- vrem ca lungimea primului argument să fie cât mai mică

Secțiuni ("operator sections")

Secțiunile operatorului binar **op** sunt **(op e)** și **(e op)**.

- secțiunile lui **||** sunt **(|| e)** și **(e ||)**

Prelude> :t (|| True)

(|| True) :: Bool -> Bool

Prelude> (|| True) False *-- atentie la paranteze*

True

Prelude> || True False

error

- secțiunile lui **<+>** sunt **(<+> e)** și **(e <+>)**

Prelude> :t (<+> 3)

(<+> 3) :: Int -> Int

Prelude> (<+> 3) 4

8

Secțiuni

Secțiunile sunt afectate de **asociativitatea** și **precedența** operatorilor.

Prelude> :t (+ 3 * 4)

(+ 3 * 4) :: Num a => a -> a

Prelude> :t (* 3 + 4) *-- + are precedenta mai mica decat **

error

Prelude> :t (* 3 * 4) *-- * este asociativa la stanga*

error

Prelude> :t (3 * 4 *)

(3 * 4 *) :: Num a => a -> a

Funcții anonime și secțiuni

Funcții anonime = lambda expresii

\<pattern> -> expresie

```
Prelude> (\x -> x+ 1) 3
```

```
4
```

```
Prelude> inc = \x -> x + 1
```

```
Prelude> add = \x y -> x+ y
```

```
Prelude> aplic = \ (f ,x) -> f x
```

Secțiunile sunt definite prin lambda expresii:

```
(x+) = \y -> x+y
```

```
(+ y) = \x -> x+y
```

Funcții(din nou)

Definirea funcțiilor folosind șabloane ("patterns")

Ce este greșit?

```
wfact 0 =1
```

```
wfact (succ n) = (succ n) * (wfact n)
```

```
Prelude> :t succ
```

```
succ :: Enum a => a -> a
```

succ nu este constructor!

Forma corectă

```
fact 0 =1
```

```
fact n = n * fact (n -1)
```

Definirea funcțiilor folosind șabloane ("patterns")

Ce este greșit?

```
wlen [] = 0
```

```
wlen [x] = 1
```

```
wlen (xs ++ ys) = (wlen xs) ++ (wlen ys)
```

++ nu este constructor!

Forma corectă

```
length [] = 0
```

```
length (x:xs) = 1 + (length xs)
```


Definirea funcțiilor folosind șabloane ("patterns")

<https://www.haskell.org/tutorial/patterns.html>

`take :: Int -> [a] -> [a]`

```
Prelude> take 3 [1,2,3,4,5,6]
[1,2,3]
```

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

- șabloanele se definesc folosind constructori
- se face potrivirea între parametrii actuali ai funcției și șabloane
- ordinea de scriere a ecuațiilor este importantă

Definirea funcțiilor folosind șabloane ("patterns")

<https://www.haskell.org/tutorial/patterns.html>

Ordinea de scriere a ecuațiilor este importantă!

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

```
take1 _ [] = []
take1 0 _ = []
take1 n (x:xs) = x : take1 (n-1) xs
```

```
*Main> take 0 undefined      *Main> take1 undefined []
[]                             []
*Main> take1 0 undefined     *Main> take undefined []
Exception: Prelude.undefined  Exception: Prelude.undefined
```

Definirea funcțiilor folosind șabloane ("patterns")

<https://www.haskell.org/tutorial/patterns.html>

Ordinea de scriere a ecuațiilor este importantă!

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

```
*Main> take 0 undefined
[]
*Main> take undefined []
Exception: Prelude.undefined
```

Care este explicația?

- potrivirea dintre `_` și `undefined` nu forțează evaluarea
- potrivirea dintre `0` și `undefined` forțează evaluarea

Definirea funcțiilor

error și undefined

```
Prelude> :t error
error :: [Char] -> a
Prelude> :t undefined
undefined :: a
```

```
fact 0 = 1
fact n = if (n >= 1)
         then n * fact (n - 1)
         else undefined      -- error "nu se calculeaza"
```

Definirea funcțiilor

tratarea cazurilor de eroare

<http://book.realworldhaskell.org/read/functional-programming.html>

Definirea funcției **head**

- folosind **length**

```
myHead xs = if length xs > 0
             then head xs
             else undefined
```

- folosind **null**

```
myHead xs = if not (null xs)
             then head xs
             else undefined
```

Care variantă este mai bună?

Varianta cu **null**, pentru a calcula **length** trebuie parcursă toată lista!

Definirea funcțiilor

Gărzi

```
fact 0 = 1
fact n
  | (n >= 1) = n * fact (n - 1)
  | otherwise = undefined  -- otherwise == True
```

Ordinea gărzilor are importanță.

```
tanar n
  | (n >= 60) = "nu asa de tanar"
  | (n >= 40) = "tanar"
  | (n >= 18) = "foarte tanar"
  | (n >= 14) = "adolescent"
  | (n > 0)  = "copil"
  | otherwise = undefined
```

Recursie (din nou)

Generarea [m..n]

```
Prelude> [3..7]
[3,4,5,6,7]
Prelude> enumFromTo 3 7
[3,4,5,6,7]
```

[m..n] este o notație pentru **enumFromTo** m n

```
enumFromTo :: Integer -> Integer -> [Integer]
enumFromTo m n | m > n      = []
                | otherwise = m : enumFromTo (m + 1) n
```

Generarea [m..]

[m..] este o notăție pentru **enumFrom** m

```
enumFrom :: Integer -> [Integer]
enumFrom m = m : enumFrom (m + 1)
```

Exemplu de rulare

```
enumFrom 4
= 4 : enumFrom 5
= 4 : 5 : enumFrom 6
= 4 : 5 : 6 : enumFrom 7
= 4 : 5 : 6 : 7 : enumFrom 8
= .....
```

Generarea [m..n]

```
Prelude> [3..7]
[3,4,5,6,7]
Prelude> enumFromTo 3 7
[3,4,5,6,7]
```

[m..n] este o notăție pentru **enumFromTo** m n

```
enumFromTo :: Integer -> Integer -> [Integer]
enumFromTo m n | m > n      = []
                | otherwise = m : enumFromTo (m + 1) n
```

Generarea [m..]

[m..] este o notăție pentru **enumFrom** m

```
enumFrom :: Integer -> [Integer]
enumFrom m = m : enumFrom (m + 1)
```

Exemplu de rulare

```
enumFrom 4
= 4 : enumFrom 5
= 4 : 5 : enumFrom 6
= 4 : 5 : 6 : enumFrom 7
= 4 : 5 : 6 : 7 : enumFrom 8
= .....
```

Zip

Zip împerechează (în ordine, câte două) elementele a două liste

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys = []
zip xs [] = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

Exemplu de rulare

```
zip [0,1,2] "abc"
= (0,'a') : zip [1,2] "bc"
= (0,'a') : ((1,'b') : zip [2] "c")
= (0,'a') : ((1,'b') : ((2,'c') : zip [] ""))
= (0,'a') : ((1,'b') : ((2,'c') : []))
= [(0,'a'),(1,'b'),(2,'c')]
```

Zip cu liste infinite

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys = []
zip xs [] = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

Exemplu de rulare (leneșă)

```
zip [0..] "abc"
= zip (0:[1..]) "abc"
= zip (0:[1..]) ('a':"bc")
= (0,'a') : zip [1..] "bc"
= (0,'a') : ((1,'b') : zip [2..] "c")
= (0,'a') : ((1,'b') : ((2,'c') : zip [3..] ""))
= (0,'a') : ((1,'b') : ((2,'c') : zip (3:[4..]) ""))
= (0,'a') : ((1,'b') : ((2,'c') : []))
= [(0,'a'), (1,'b'), (2,'c')]
```

Produs scalar

Pentru doi vectori \bar{a} și \bar{b} de aceeași lungime, produsul scalar este $\sum_i a_i * b_i$

```
dot :: Num a => [a] -> [a] -> a
dot xs ys = sum [x * y | (x,y) <- xs 'zip' ys]
```

Exemplu de rulare

```
[1,2,3] 'dot' [4,5,6]
= sum [x * y | (x,y) <- [1,2,3] 'zip' [4,5,6]]
= sum [x * y | (x,y) <- [(1,4),(2,5),(3,6)]]
= sum [1*4, 2*5, 3*6]
= sum [4,10,18]
= 720
```

Search

search caută toate pozițiile dintr-o listă pe care apare un element dat.

```
search :: Eq a => [a] -> a -> [Int]
search xs x = [i | (i,y) <- [0..] 'zip' xs, y == x]
```

Exemplu de rulare

```
search "abac" 'a'
= [i | (i,y) <- [0..] 'zip' "abac", y == 'a']
= [i | (i,y) <- [(0,'a'),(1,'b'),(2,'a'),(3,'c')], y == 'a']
= [0 | 'a' == 'a'] ++ [1 | 'b' == 'a'] ++ [2 | 'a' == 'a'] ++
  [3 | 'c' == 'a']
= [0,2]
```

Search

search caută toate pozițiile dintr-o listă pe care apare un element dat.

```
search :: Eq a => [a] -> a -> [Int]
search xs x = [i | (i,y) <- [0..] 'zip' xs, y == x]
```

Exemplu de rulare

```
search "abac" 'a'
= [i | (i,y) <- [0..] 'zip' "abac", y == 'a']
= [i | (i,y) <- [(0,'a'),(1,'b'),(2,'a'),(3,'c')], y == 'a']
= [0 | 'a' == 'a'] ++ [1 | 'b' == 'a'] ++ [2 | 'a' == 'a'] ++
  [3 | 'c' == 'a']
= [0,2]
```

Programare declarativă¹

Map, Filter, Fold

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

Programarea funcțională

Funcțiile sunt cetățeni de ordinul I.

- funcțiile sunt valori
- funcțiile pot fi transmise ca argumente altor funcții
- funcțiile pot fi întoarse ca rezultate

Funcții de nivel înalt

sunt funcțiile care primesc ca argumente alte funcții.

Programarea funcțională

- funcțiile sunt valori

```
Prelude> let x = head
Prelude> x [1,2]
1
```
- funcțiile pot fi transmise ca argumente altor funcții

```
Prelude> map head ["higher", "order", "function"]
"hof"
```
- funcțiile pot fi întoarse ca valori

```
Prelude> :t flip
flip :: (a -> b -> c) -> b -> a -> c

Prelude> let f = flip (:)
Prelude> let (<:>) = flip (:)
Prelude> 1:[2,3] == [2,3] <:> 1
True
```

Programarea funcțională

Prelucarea listelor se poate face folosind funcții de nivel înalt.

- Transformarea fiecărui element al unei liste se poate face folosind funcția **map**.
- Selecția elementelor unei liste se poate face folosind funcția **filter**.
- Combinarea elementelor unei liste se poate face folosind funcția **foldr**.

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]
= [ 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

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 — Pozitive

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

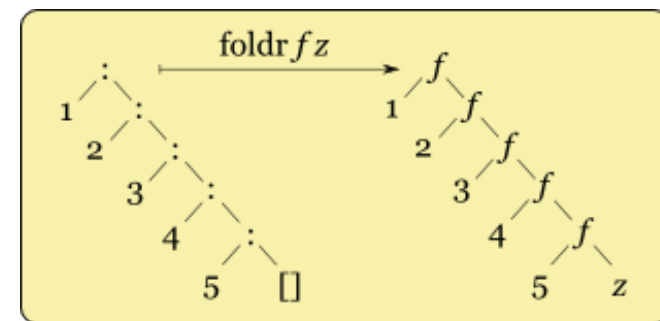
```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f i []      = i
foldr f i (x:xs) = f x (foldr f i xs)
```

Soluție recursivă cu operator infix

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op i []      = i
foldr op i (x:xs) = x `op` (foldr op i xs)
```

Funcția foldr

```
foldr :: (Int -> b -> b) -> b -> [Int] -> b
f :: Int -> b -> b
z :: b
```



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

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
```

foldr 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
```

Produs

Soluție recursivă

```
product :: [Int] -> Int
product [] = 1
product (x:xs) = x * product 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 :: [[a]] -> [a]
concat xs = foldr (++) [] xs
```

foldr pe liste infinite

```
Prelude> let li = (:[])
```

```
Prelude> li 1
[1]
```

```
Prelude> let infLL = map li [1..]
```

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

```
Prelude> let infL = foldr (++) [] infLL
```

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

```
infL = foldr (++) [] (map li [1..])
```

Putem defini **infL** folosind numai **foldr**?

mai mult despre foldr

```
infL = foldr aux [] [1..]
  where
    aux x xs = (li x)++xs
```

Funcția **map** poate fi definită cu **foldr**

```
map f xs = foldr aux [] xs
  where
    aux x xs = (f x) : xs
```

Map, Filter, Fold — combinate

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
```

Map/Filter/Fold combine

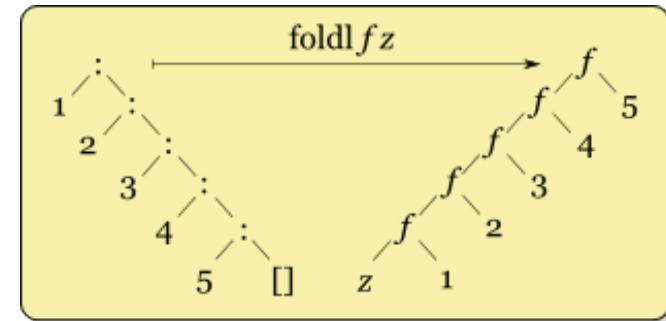
Problemă

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
```

Funcția foldl

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl op i []      = i
foldl op i (x:xs) = foldl op (i `op` x) xs
```



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

Atenție! **foldl** nu poate fi folosită pe liste infinite!

Currying

Currying

Currying

Exemplu: adunarea numerelor

```
add' :: (Int, Int) -> Int
add' (x, y) = x + y
```

```
Prelude> add' (3, 4)
7
```

```
add = curry add'
```

```
Prelude> :t add
add :: Int -> Int -> Int
```

```
Prelude> add 3 4
7
```

Exemplu: 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.

- aplicarea funcțiilor este asociativă la stânga
- operatorul \rightarrow este asociativ la dreapta

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 asemenea,

```
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 (++) []
```

```
idl :: [a] -> [a]
idl = foldr (:) []
```

Funcții anonime

Funcții anonime

Funcții anonime = lambda expresii

\<pattern> -> expresie

```
Prelude> (\x -> x + 1) 3
4
```

```
inc = \x -> x + 1
add = \x y -> x + y
```

```
prod = \ (x,y) -> x * y
head2 = \ (x:y:l) -> (x,y)
```

```
aplic2 = \f -> f . f
Prelude> aplic2 sqrt 16
2.0
```

```
comb f g = \ x y -> g (f x) (f y)
Prelude> (comb head (<)) "abc" "def"
True
```

Simplificăm definiția

```
f :: [Int] -> [Int]
f xs = map sqr x
      where
        sqr x = x * x
```

Simplificare incorectă

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

Simplificare corectă

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

Simplificăm definiția

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

Aceeasi definitie folosind funcții anonime:

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

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
```

Funcții anonime / Lambda Calcul

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

Lambda Calcul

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

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:

```
(\ x . n) m == let x = m in n
```


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
```

Exemple: **foldr**, **foldl** și funcții anonime

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

- definiția cu **foldr**

```
reverse ys = foldr (\x xs -> xs ++ [x]) [] ys
```

- definiția cu **foldl**

```
reverse ys = foldl (\xs x -> x:xs) [] ys
```

Exemple: **foldr** și funcții anonime

```
map f xs = foldr aux [] xs
  where
    aux x xs = (f x) : xs
```

Cu λ -expresii

```
map f xs = foldr (\x xs -> (f x):xs) [] xs
length xs = foldr (\x n -> n+1) 0 xs
filter p xs = foldr (\x xs ->
  if (p x) then (x:xs) else xs) [] xs
```

Aplicații Currying — Stilul funcțional

```
map f      = foldr (\x xs -> (f x):xs) []
length     = foldr (\x n -> n+1) 0
filter p   = foldr (\x xs -> if (p x) then (x:xs) else xs) []
```

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

Secțiunile operatorului binar **op** sunt (**op e**) și (**e op**).

Secțiuni

Secțiunile sunt afectate de **asociativitatea** și **precedența** operatorilor.

```
Prelude> :t (+ 3 * 4)
(+ 3 * 4) :: Num a => a -> a
```

```
Prelude> :t (* 3 + 4) -- + are precedenta mai mica decat *
error
```

```
Prelude> :t (* 3 * 4) -- * este asociativa la stanga
error
```

```
Prelude> :t (3 * 4 *)
(3 * 4 *) :: Num a => a -> a
```

Secțiuni — Exemplu

```
f :: [Int] -> [Int]
f xs = map sqr [x | x <- xs, x > 0]
  where
    sqr x = x^2
```

Folosind λ -expresii

```
f xs = map (\x -> x * x) [x | x <- xs, (\x -> x > 0) x]
```

Folosind secțiuni

```
f xs = map (^2) [x | x <- xs, (>0) x])
```

Secțiuni — Exemplu

```
(<*>) :: Int -> Int -> Int
x <*> y = x * x + y
```

```
functions = map (<*>) [0..]
```

Ce tip are **functions**?

```
functions :: [Int -> Int]
```

```
functions = [(0 <*>), (1 <*>), (2 <*>), ...]
```

```
Prelude> (functions !! 50) 10
2510
```

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)
```

Operatorul \$

Operatorul (\$) are precedența 0.

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

```
Prelude> sqrt 3 + 4 +9
14.732050807568877
Prelude> sqrt (3 + 4 +9)
4.0
Prelude> sqrt $ 3 + 4 +9
4.0
```

Operatorul \$

<http://learnyouahaskell.com/higher-order-functions>

Operatorul (\$) este asociativ la dreapta.

```
sum (filter (> 10) (map (*2) [2..10]))
```

se poate scrie

```
sum $ filter (> 10) $ map (*2) [2..10].
```

Exemplu folosind secțiuni:

```
Prelude> map ($ 3) [(4+), (10*), (^2), sqrt]
```

```
[7.0,30.0,9.0,1.7320508075688772]
```

Map/Filter/Reduce în Javascript

<http://crypto.net/~joepie91/blog/2015/05/04/functional-programming-in-javascript-map-filter-reduce/>

Problemă

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

```
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 alte limbaje

- Phyton
<http://www.python-course.eu/lambda.php>
- PHP
<http://eddmann.com/posts/mapping-filtering-and-reducing-in-php/>
- Java 8
<http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>
- C++ 11
<http://www.grimm-jaud.de/images/stories/pdfs/FunctionalProgrammingInC++11.pdf>

Programare declarativă

Proprietatea de universalitate a funcției **foldr**¹

Traian Florin Șerbănuță
Ioana Leuștean

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

¹bazat pe [Graham Hutton, A tutorial on the universality and expressiveness of fold, J. of Functional Programming, 9 \(4\): 355-372, 1999](#)

foldr și foldl

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.

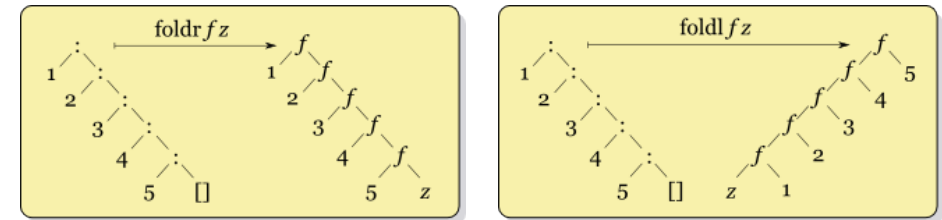
Funcția foldr

```
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
```

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!
- foldl nu poate fi folosită pe liste infinite!

În continuare, listele sunt considerate finite.

foldr - proprietatea de universalitate

Proprietatea de universalitate

Observație

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

Teoremă

Fie g o funcție care procesează liste finite. Atunci

$$\begin{aligned} g [] &= i \\ g (x : xs) &= f x (g xs) \end{aligned} \Leftrightarrow g = \text{foldr } f \ i$$

Demonstrație:

⇒ Înlocuind $g = \text{foldr } f \ i$ se obține definiția lui **foldr**

⇐ Prin inducție după lungimea listei.

Teorema determină condiții necesare și suficiente pentru ca o funcție g care procesează liste să poată fi definită folosind **foldr**.

Generarea funcțiilor cu foldr

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> foldr (.) **id** [(+1), (^2)] 3

10

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

Suma

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

Soluție cu foldr

sum = **foldr** (+) 0

În definiția de mai sus elementele sunt procesate de la dreapta la stânga:
sum[x_1, \dots, x_n] = ($x_1 + (x_2 + \dots (x_n + 0) \dots$)

Problemă

Scrieți o definiție a sumei folosind **foldr** astfel încât elementele să fie procesate de la stânga la dreapta.

Suma

sum cu acumulator

sum :: [Int] → Int

sum xs = suml xs 0

where

suml [] n = n

suml (x:xs) n = suml xs (n+x)

În definiția de mai sus elementele sunt procesate de la stânga la dreapta:
 suml [x_1, \dots, x_n] 0 = ($\dots (0 + x_1) + x_2) + \dots x_n$)

Definim suml cu foldr

- Observăm că

suml :: [Int] → (Int → Int)

- Definim suml cu **foldr** aplicând proprietatea de universalitate.

Definirea suml cu foldr

Proprietatea de universalitate

$$\begin{aligned} g [] &= i \\ g (x : xs) &= f x (g xs) \quad \Leftrightarrow \quad g = \text{foldr } f \ i \end{aligned}$$

Observăm că

$$\text{suml } [] = \text{id} \quad \text{-- } \text{suml } [] \ n = n$$

Vrem să găsim f astfel încât

$$\text{suml } (x : xs) = f \ x \ (\text{suml } xs)$$

deoarece, din proprietatea de universalitate, va rezulta că

$$\text{suml} = \text{foldr } f \ \text{id}$$

Definirea suml cu foldr

$\text{suml} :: [\text{Int}] \rightarrow (\text{Int} \rightarrow \text{Int})$

$$\begin{aligned} \text{suml } (x : xs) &= f \ x \ (\text{suml } xs) \\ \text{suml } (x : xs) \ n &= f \ x \ (\text{suml } xs) \ n \\ \text{suml } xs \ (n + x) &= f \ x \ (\text{suml } xs) \ n \end{aligned}$$

Notăm $u = \text{suml } xs$ și obținem

$$\begin{aligned} u \ (n + x) &= f \ x \ u \ n \\ u \ (n + x) &= (f \ x \ u) \ n \end{aligned}$$

Rezultă că $f = \lambda x u. (\lambda n. u(n + x))$

Soluție

$$\begin{aligned} f &= \lambda x u \rightarrow \lambda n \rightarrow u \ (n+x) \\ \text{suml} &= \text{foldr } (\lambda x u \rightarrow \lambda n \rightarrow u \ (n+x)) \ \text{id} \end{aligned}$$

Definirea sum cu foldr

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

$\text{sum } xs = \text{foldr } (\lambda x u \rightarrow \lambda n \rightarrow u \ (n+x)) \ \text{id} \ xs \ 0$

$\text{-- } \text{sum } xs = \text{suml } xs \ 0$

Prelude> sum xs = **foldr** ($\lambda x u \rightarrow \lambda n \rightarrow u \ (n+x)$) **id** xs 0

Prelude> sum [1,2,3]

6

foldl

Definiție

Funcția foldl

$\text{foldl} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
 $\text{foldl } h \ i \ [] = i$
 $\text{foldl } h \ i \ (x:xs) = \text{foldl } h \ (h \ i \ x) \ xs$

$\text{foldl } h \ i \ xs = \text{foldl}' \ h \ xs \ i$
where
 $\text{foldl}' \ h \ [] \ i = i$
 $\text{foldl}' \ h \ (x:xs) \ i = \text{foldl}' \ h \ xs \ (h \ i \ x)$

$\text{foldl}' :: (b \rightarrow a \rightarrow b) \rightarrow [a] \rightarrow b \rightarrow b$
 $\text{foldl}' \ h :: [a] \rightarrow (b \rightarrow b)$
 $\text{foldl}' \ h \ xs :: b \rightarrow b$

foldl' cu foldr

Observăm că

foldl' h [] = id $\text{suml [] } n = n$

Vrem să găsim f astfel încât

$$\text{foldl'} h (x : xs) = f x (\text{foldl'} h xs)$$

deoarece, din proprietatea de universalitate, va rezulta că

$$\text{foldl'} h = \text{foldr } f \text{ id}$$

foldl cu foldr

Soluție

$h :: b \rightarrow a \rightarrow b$

foldl' h = foldr f id

$f = \lambda x u \rightarrow \lambda y \rightarrow u (h y x)$

foldl h i xs = foldl' h xs i

foldl h i xs = foldr (\ x u -> \ y -> u (h y x)) id xs i

foldl cu foldr

```
Prelude> let myfoldl h i xs =  
           foldr (\x u -> \y -> u (h y x)) id xs i
```

```
Prelude> myfoldl (+) 0 [1,2,3]  
6
```

```
Prelude> let sing = (:[])
```

```
Prelude> take 3 (foldr (++) [] (map sing [1..]))  
[1,2,3]
```

```
Prelude> take 3 (myfoldl (++) [] (map sing [1..]))  
Interrupted.
```