Programare funcțională

Procesarea fluxurilor de date. Evaluare leneșă.

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

27 octombrie 2020

1. Funcții de ordin înalt: map și filter
2. Funcții de ordin înalt: foldr și foldl
3. Proprietatea de universalitate a funcției **foldr**
4. Generarea funcțiilor cu **foldr**
5. Evaluarea leneșă. Liste infinite

Funcții de ordin înalt: map și filter

Funcțiile sunt valori

Funcțiile sunt valori!

```
Prelude> ap n f = if (n<=0) then id else (f . (ap (n-1) f))

Prelude> ap 3 (\x -> x*x) 4
65536

Prelude> ap 3 (\ (x, y) -> (x*x, y+y)) (4,5)
(65536,40)
```

Funcțiile sunt valori

Funcțiile sunt valori!

```
Prelude> ap n f = if (n<=0) then id else (f . (ap (n-1) f))

Prelude> ap 3 (\x -> x*x) 4
65536

Prelude> ap 3 (\ (x, y) -> (x*x, y+y)) (4,5)
(65536,40)
```

Observați folosirea funcțiilor anonime (λ-expresii)!

Funcțiile sunt valori

Funcțiile sunt valori!

```
Prelude> ap n f = if (n<=0) then id else (f . (ap (n-1) f))

Prelude> ap 3 (\x -> x*x) 4
65536

Prelude> ap 3 (\ (x, y) -> (x*x, y+y)) (4,5)
(65536,40)
```

Observați folosirea funcțiilor anonime (λ-expresii)!

```
Prelude> :t ap
ap :: (Eq t, Num t) => t -> (b -> b) -> b -> b

Prelude> :t ap 5
ap 5 :: (b -> b) -> b -> b
```

Funcțiile sunt valori

```
Prelude> ap n f = if (n<=0) then id else (f . (ap (n-1) f))

Prelude> :t ap
ap :: (Eq t, Num t) => t -> (b -> b) -> b -> b

Prelude> g = ap 2 (\x -> x*x)
Prelude> g 3 == ap 2 (\x -> x*x) 3
True
```

Funcțiile sunt valori

```
Prelude> ap n f = if (n<=0) then id else (f . (ap (n-1) f))

Prelude> :t ap
ap :: (Eq t, Num t) => t -> (b -> b) -> b -> b

Prelude> g = ap 2 (\x -> x*x)
Prelude> g 3 == ap 2 (\x -> x*x) 3
True

Prelude> g  == ap 2 (\x -> x*x)
error
```

Funcțiile nu pot fi comparate!

Din nou **map**

map :: (a -> b) -> [a] -> [b]

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

$$
\begin{array}{ccccccccc}
x_1 & : & x_2 & : & \cdots & : & x_n & : & [\,] \\
\downarrow & & \downarrow & & & & \downarrow & & \\
f(x_1) & : & f(x_2) & : & \cdots & : & f(x_n) & : & [\,]
\end{array}
$$

Din nou **map**

map :: (a -> b) -> [a] -> [b]

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

$$\begin{array}{ccccccccc} x_1 & : & x_2 & : & \cdots & : & x_n & : & [\,] \\ \downarrow & & \downarrow & & & & \downarrow & & \\ f(x_1) & : & f(x_2) & : & \cdots & : & f(x_n) & : & [\,] \end{array}$$

Problemă

Scrieți o funcție care scrie un șir de caractere cu litere mari.

Din nou **map**

map :: (a -> b) -> [a] -> [b]

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

$$\begin{array}{ccccccccc} x_1 & : & x_2 & : & \cdots & : & x_n & : & [\,] \\ \downarrow & & \downarrow & & & & \downarrow & & \\ f(x_1) & : & f(x_2) & : & \cdots & : & f(x_n) & : & [\,] \end{array}$$

Problemă

Scrieți o funcție care scrie un șir de caractere cu litere mari.

```
scrieLitereMari s = map toUpper s
```

Din nou **map**

map :: (a -> b) -> [a] -> [b]

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

$$\begin{array}{ccccccccc} x_1 & : & x_2 & : & \cdots & : & x_n & : & [\,] \\ \downarrow & & \downarrow & & & & \downarrow & & \\ f(x_1) & : & f(x_2) & : & \cdots & : & f(x_n) & : & [\,] \end{array}$$

Problemă

Scrieți o funcție care scrie un șir de caractere cu litere mari.

```
scrieLitereMari s = map toUpper s
```

```
Prelude Data.Char> :t toUpper
toUpper :: Char -> Char
```

```
Prelude Data.Char> map toUpper "abac"
"ABAC"
```

Din nou **filter**

filter :: (a -> Bool) -> [a] -> [a]

```
filter prop xs  = [x | x <- xs, prop x]
```

```
Prelude> filter (>= 2) [1,3,4]
[3,4]
Prelude> filter (\ (x,y)-> x+y >= 10) [(1,4),(2,7), (3,10)]
[(3,10)]
```

Din nou **filter**

filter :: (a -> Bool) -> [a] -> [a]

```
filter prop xs  = [x | x <- xs, prop x]
```

```
Prelude> filter (>= 2) [1,3,4]
[3,4]
Prelude> filter (\ (x,y)-> x+y >= 10) [(1,4),(2,7), (3,10)]
[(3,10)]
```

Problemă

Scrieți o funcție care scrie selectează dintr-o listă de cuvinte pe cele care încep cu literă mare.

Din nou **filter**

filter :: (a -> Bool) -> [a] -> [a]

```
filter prop xs  = [x | x <- xs, prop x]
```

```
Prelude> filter (>= 2) [1,3,4]
[3,4]
Prelude> filter (\ (x,y)-> x+y >= 10) [(1,4),(2,7), (3,10)]
[(3,10)]
```

Problemă

Scrieți o funcție care scrie selectează dintr-o listă de cuvinte pe cele care încep cu literă mare.

```
incepeLM xs = filter (\x -> isUpper (head x))  xs
```

```
Prelude Data.Char> incepeLM ["carte","Ana","minge","Petre"]
["Ana","Petre"]
```

map și filter

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

Secvență Collatz: $c_1, c_2, \ldots, c_n$ (numere naturale)

$$x_{n+1} = \begin{cases} x_n/2 & \text{dacă } x_n \text{ este par} \\ 3x_n + 1 & \text{dacă } x_n \text{ este impar} \end{cases}$$

Exemplu: 22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1

map și filter

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

Secvență Collatz: $c_1, c_2, \ldots, c_n$ (numere naturale)

$$x_{n+1} = \begin{cases} x_n/2 & \text{dacă } x_n \text{ este par} \\ 3x_n + 1 & \text{dacă } x_n \text{ este impar} \end{cases}$$

Exemplu: 22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1

Conjectura lui Collatz:

orice secvență Collatz se termină cu 1

map și filter

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

Secvență Collatz: $c_1, c_2, \ldots, c_n$ (numere naturale)

$$x_{n+1} = \begin{cases} x_n/2 & \text{dacă } x_n \text{ este par} \\ 3x_n + 1 & \text{dacă } x_n \text{ este impar} \end{cases}$$

Exemplu: 22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1

Conjectura lui Collatz:

orice secvență Collatz se termină cu 1

Problemă

1. Scrieți o funcție care calculează secvența lui Collatz care începe cu n.
2. Determinați secvențele Collatz de lungime ≤ 15 care încep cu un număr din intervalul $[1, 100]$

Secvență Collatz

Secvență Collatz: $c_1, c_2, \ldots, c_n$ (numere naturale)

$$x_{n+1} = \begin{cases} x_n/2 & \text{dacă } x_n \text{ este par} \\ 3x_n + 1 & \text{dacă } x_n \text{ este impar} \end{cases}$$

Problemă

1. Scrieți o funcție care calculează secvența lui Collatz care începe cu n

Secvență Collatz

Secvență Collatz: $c_1, c_2, \ldots, c_n$ (numere naturale)

$$x_{n+1} = \begin{cases} x_n/2 & \text{dacă } x_n \text{ este par} \\ 3x_n + 1 & \text{dacă } x_n \text{ este impar} \end{cases}$$

Problemă

1. Scrieți o funcție care calculează secvența lui Collatz care începe cu n

```
collatz n
  | n == 1 = []
  | n  > 1 = n : collatz (next n)
  where next x | even x     = x `div` 2
               | otherwise = 3 * x + 1
```

map și filter

Problemă

1. Scrieți o funcție care calculează secvența lui Collatz care începe cu *n*.

```
collatz n
  | n == 1 = []
  | n  > 1 = n : collatz (next n)
  where next x | even x      = x `div` 2
               | otherwise = 3 * x + 1)
```

map și filter

Problemă

1. Scrieți o funcție care calculează secvența lui Collatz care începe cu *n*.

```
collatz n
  | n == 1 = []
  | n  > 1 = n : collatz (next n)
  where next x | even x      = x `div` 2
               | otherwise = 3 * x + 1)
```

2. Determinați secvențele Collatz de lungime ≤ 5 care încep cu un număr din intervalul $[1, 100]$.

map și filter

Problemă

1. Scrieți o funcție care calculează secvența lui Collatz care începe cu *n*.

```
collatz n
  | n == 1 = []
  | n  > 1 = n : collatz (next n)
  where next x | even x     = x `div` 2
               | otherwise = 3 * x + 1)
```

2. Determinați secvențele Collatz de lungime ≤ 5 care încep cu un număr din intervalul $[1, 100]$.

```
Prelude> filter (\x -> length x <= 5) (map collatz [1..100])
```

map și filter

Problemă

1. Scrieți o funcție care calculează secvența lui Collatz care începe cu *n*.

```
collatz n
  | n == 1 = []
  | n  > 1 = n : collatz (next n)
  where next x | even x      = x `div` 2
               | otherwise = 3 * x + 1)
```

2. Determinați secvențele Collatz de lungime ≤ 5 care încep cu un număr din intervalul $[1, 100]$.

```
Prelude> filter (\x -> length x <= 5) (map collatz [1..100])

[[1],[2,1],[4,2,1],[8,4,2,1],[16,8,4,2,1]]
```

Funcții de ordin înalt: foldr și foldl

Funcții de ordin înalt

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.

Funcții de ordin înalt

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) ...)))
```

Funcții de ordin înalt

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
```

foldr și foldl

Definiție

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.

foldr și foldl

Definiție

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
foldr f i []      = i
```

foldr și foldl

Definiție

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
foldr f i []     = i
foldr f i (x:xs) = f x (foldr f i xs)
```

foldr și foldl

Definiție

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
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
```

Filtrare, transformare, agregare

Problemă

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

Filtrare, transformare, agregare

Problemă

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

```
maxLengthFn xs = foldr max 0 (map length (filter test xs))
                 where test = \x -> head x =='c'
```

Filtrare, transformare, agregare

Problemă

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

Definiția compozițională:

```
maxLengthFn = foldr max 0 .
              map length .
              filter (\x -> head x =='c')
```

Proprietatea de universalitate a funcției **foldr**

Proprietatea de universalitate

Observație

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

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

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

Demonstrație:

$\Rightarrow$ Înlocuind $g = foldr\ f\ i$ se obține definiția lui **foldr**

$\Leftarrow$ Prin inducție dupa lungimea listei.

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

$$\begin{array}{lcl} g\,[] & = & i \\ g\,(x : xs) & = & f\ x\ (g\ xs) \end{array} \quad \Leftrightarrow \quad g = \mathit{foldr}\ f\ i$$

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.

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
```

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**

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

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:
$\mathbf{sum}[x_1, \ldots, x_n] = (x_1 + (x_2 + \ldots (x_n + 0) \ldots)$

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)
```

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 $[x_1, \ldots, x_n]$ $0 = (\ldots(0 + x_1) + x_2) + \ldots x_n)$

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 $[x_1, \ldots, x_n]\ 0 = (\ldots(0 + x_1) + x_2) + \ldots x_n)$

Definim suml cu **foldr**

- Obervăm că

```
suml :: [Int] -> (Int -> Int)
```

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

Definirea suml cu foldr

Proprietatea de universalitate

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

Definirea suml cu foldr

Proprietatea de universalitate

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

Observăm că

```
suml [] = id    -- suml [] n = n
```

Definirea suml cu foldr

Proprietatea de universalitate

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

Observăm că

```
suml [] = id    -- suml [] n = n
```

Vrem să găsim *f* astfel încât

$$suml\ (x : xs) = f\ x\ (suml\ xs)$$

deoarece, din proprietatea de universalitate, va rezulta că

$$suml = foldr\ f\ id$$

Definirea suml cu foldr

```
suml :: [Int] -> (Int -> Int)
```

$$
\begin{array}{lcll}
suml\ (x : xs) & = & f\ x\ (suml\ xs) & (\text{vrem}) \\
suml\ (x : xs)\ n & = & f\ x\ (suml\ xs)\ n & (\text{vrem}) \\
suml\ xs\ (n + x) & = & f\ x\ (suml\ xs)\ n & (\text{def } suml)
\end{array}
$$

Definirea suml cu foldr

```
suml :: [Int] -> (Int -> Int)
```

$$\begin{array}{lcll} suml\ (x:xs) & = & f\ x\ (suml\ xs) & \text{(vrem)} \\ suml\ (x:xs)\ n & = & f\ x\ (suml\ xs)\ n & \text{(vrem)} \\ suml\ xs\ (n+x) & = & f\ x\ (suml\ xs)\ n & \text{(def } suml\text{)} \end{array}$$

Notăm $u = suml\ xs$ și obținem

$$u\ (n+x) \quad = \quad f\ x\ u\ n$$

Definirea suml cu foldr

```
suml :: [Int] -> (Int -> Int)
```

$$
\begin{array}{lcll}
suml\ (x : xs) & = & f\ x\ (suml\ xs) & \text{(vrem)} \\
suml\ (x : xs)\ n & = & f\ x\ (suml\ xs)\ n & \text{(vrem)} \\
suml\ xs\ (n + x) & = & f\ x\ (suml\ xs)\ n & \text{(def } suml\text{)}
\end{array}
$$

Notăm $u = suml\ xs$ și obținem

$$
u\ (n + x) \quad = \quad f\ x\ u\ n
$$

Soluție

```
f = \ x u n -> u (n+x)

suml = foldr (\ x u -> f x u) id

suml = foldr (\x u -> (\n -> u (n+x))) id

suml = foldr (\x u n -> u (n+x)) id
       -- tipurile sunt determinate corespunzator
```

Definirea sum cu foldr

```haskell
sum :: [Int] -> Int
sum xs = foldr (\ x u n -> u (n+x)) id xs 0
-- sum xs = suml xs 0
```

Definirea sum cu foldr

```
sum :: [Int] -> Int

sum xs = foldr (\ x u n -> u (n+x)) id xs 0

-- sum xs = suml xs 0
```

```
Prelude> sum xs = foldr (\ x u -> \ n -> u (n+x)) id xs 0
Prelude> sum [1,2,3]
6
```

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**

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

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**

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

În definiția de mai sus elementele sunt procesate de la stânga la dreapta:
rev$[x_1, \ldots, x_n] = (...(([] <:> x_1) <:> x_2).....) <:> x_n$

Problemă

Scrieți o definiție a funcției rev folosind **foldr**.

Inversarea elementelor unei liste

rev cu acumulator

```
rev :: [a] -> [a]
rev xs = revl xs []
  where
    revl [] l        = l
    revl (x:xs) rxs = revl xs (x:rxs)
```

În definiția de mai sus elementele sunt procesate de la stânga la dreapta, fiind mutate în argumentul auxiliar:

Inversarea elementelor unei liste

rev cu acumulator

```
rev :: [a] -> [a]
rev xs = revl xs []
  where
    revl [] l        = l
    revl (x:xs) rxs = revl xs (x:rxs)
```

În definiția de mai sus elementele sunt procesate de la stânga la dreapta, fiind mutate în argumentul auxiliar:

Definim suml cu **foldr**

- Obervăm că

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

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

Definirea revl cu foldr

Proprietatea de universalitate

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

Definirea revl cu foldr

Proprietatea de universalitate

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

Observăm că

```
revl [] = id    -- revl [] l = l
```

Definirea revl cu foldr

Proprietatea de universalitate

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

Observăm că

```
revl [] = id    -- revl [] l = l
```

Vrem să găsim *f* astfel încât

$$revl\ (x : xs) = f\ x\ (revl\ xs)$$

deoarece, din proprietatea de universalitate, va rezulta că

$$revl = foldr\ f\ id$$

Definirea revl cu foldr

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

$$
\begin{array}{lcll}
revl\ (x : xs) & = & f\ x\ (revl\ xs) & \text{(vrem)} \\
revl\ (x : xs)\ xs' & = & f\ x\ (revl\ xs)\ xs' & \text{(vrem)} \\
revl\ xs\ (x : xs') & = & f\ x\ (revl\ xs)\ xs' & \text{(def } revl)
\end{array}
$$

Definirea revl cu foldr

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

$$\begin{array}{lcll} revl\ (x:xs) & = & f\ x\ (revl\ xs) & \text{(vrem)} \\ revl\ (x:xs)\ xs' & = & f\ x\ (revl\ xs)\ xs' & \text{(vrem)} \\ revl\ xs\ (x:xs') & = & f\ x\ (revl\ xs)\ xs' & (\text{def } revl) \end{array}$$

Notăm $u = revl\ xs$ și obținem

$$u\ (x:xs') \quad = \quad f\ x\ u\ xs'$$

Definirea revl cu foldr

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

$$
\begin{array}{lcll}
revl\ (x : xs) & = & f\ x\ (revl\ xs) & \text{(vrem)} \\
revl\ (x : xs)\ xs' & = & f\ x\ (revl\ xs)\ xs' & \text{(vrem)} \\
revl\ xs\ (x : xs') & = & f\ x\ (revl\ xs)\ xs' & (\text{def } revl)
\end{array}
$$

Notăm $u = revl\ xs$ și obținem

$$u\ (x : xs') \quad = \quad f\ x\ u\ xs'$$

Soluție

```
f = \ x u xs' -> u (x:xs')

revl = foldr (\ x u -> f x u) id

revl = foldr (\x u -> (\xs' -> u (x:xs'))) id

revl = foldr (\x u n -> u (x:xs')) id
      -- tipurile sunt determinate corespunzator
```

Definirea rev cu foldr

```haskell
rev :: [a] -> [a]
rev xs = foldr (\ x u xs' -> u (x:xs')) id xs []
-- rev xs = revl xs []
```

Definirea rev cu foldr

```haskell
rev :: [a] -> [a]

rev xs = foldr (\ x u xs' -> u (x:xs')) id xs []

-- rev xs = revl xs []
```

```
Prelude> rev xs = foldr (\ x u xs' -> u (x:xs')) id xs []
Prelude> rev [1,2,3]
[3,2,1]
```

foldl

Definiție

Funcția *foldl*

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

foldl

Definiție

Funcția *foldl*

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

```haskell
foldl h i xs = foldl' h xs i
               where
                 foldl' h [] i = i
                 foldl' h (x:xs) i = foldl' h xs (h i x)
```

foldl

Definiție

Funcția *foldl*

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

```
foldl h i xs = foldl' h xs i
               where
                 foldl' h [] i = i
                 foldl' h (x:xs) i = foldl' h xs (h i x)
```

```
foldl' :: (b -> a -> b) -> [a] -> b -> b
foldl' h :: [a] -> (b ->b)
foldl' h xs :: b -> b
```

foldl' cu **foldr**

Observăm că

```
foldl' h [] = id    -- suml [] n = n
```

foldl’ cu **foldr**

Observăm că

```
foldl' h [] = id    -- suml [] n = n
```

Vrem să găsim *f* astfel încât

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

deoarece, din proprietatea de universalitate, va rezulta că

$$foldl'\ h = foldr\ f\ id$$

foldl cu **foldr**

Soluție

```
h :: b -> a -> b

foldl' h = foldr f id

f = \ x u -> \ y -> 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
```

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 = (:[])  -- sing x = [x]

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

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

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

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 = (:[])  -- sing x = [x]

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

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

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

Ce se întâmplă?

Evaluarea leneșă. Liste infinite

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.

Evaluare leneșă. Liste infinite

- Putem folosi funcțile **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.

Evaluare leneșă: lista numerelor prime

Vă amintiți din primul curs:

```
primes = sieve [2..]
sieve (p:ps) = p : sieve [ x | x <- ps, mod x p /= 0 ]
```

Evaluare leneșă: lista numerelor prime

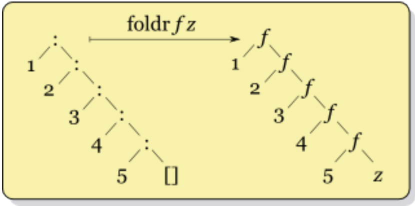
Vă amintiți din primul curs:

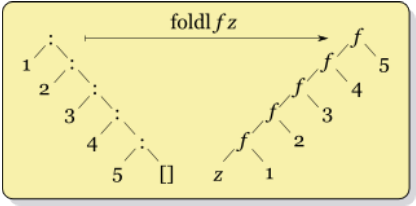
```
primes = sieve [2..]
sieve (p:ps) = p : sieve [ x | x <- ps, mod x p /= 0 ]
```

Intuitiv, evaluarea leneșă funcționează astfel:

```
sieve [2..] -->

2 : sieve [ x | x <- [3..], mod x 2 /= 0 ] -->

2 : sieve (3:[ x | x <- [4..], mod x 2 /= 0 ])  -->

2 : 3 : sieve ([ y | y <- [x | x <- [4..], mod x 2 /= 0 ],
                                          mod y 3 /= 0 ])

--> ...
```

foldr și foldl





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 calculeaz\u a la infinit
```

Evaluare leneșă. Liste infinite

- Intuitiv, evaluarea leneșă funcționează astfel:

```
foldr (++) [] (map sing [1..]) -->

(++) [1] (foldr (++) [] (map sing [2..]) -->

(++) [1] ((++) [2] (foldr (++) [] (map sing [3..]))) -->

(++) [1] ((++) [2] ((++) [3] (foldr (++) []
                                    (map sing [4..])))) -->
                                        ...
```

Evaluare leneșă. Liste infinite

- Intuitiv, evaluarea leneșă funcționează astfel:

```
foldr (++) [] (map sing [1..]) -->

(++) [1] (foldr (++) [] (map sing [2..]) -->

(++) [1] ((++) [2] (foldr (++) [] (map sing [3..])) -->

(++) [1] ((++) [2] ((++) [3] (foldr (++) []
                                  (map sing [4..])))) -->
                                        ...
```

- În momentul în care apelăm **take** 3 forțăm evaluarea.

Evaluare leneșă. Liste infinite

- Intuitiv, evaluarea leneșă funcționează astfel:

```
foldr (++) [] (map (:[]) [1..]) -->
(++) [1] (foldr (++) [] (map (:[]) [2..]) -->
(++) [1] ((++) [2] (foldr (++) [] (map (:[]) [3..])) -->
```

- În momentul în care apelăm **take** n forțăm evaluarea.

Evaluare leneșă. Liste infinite

- Intuitiv, evaluarea leneșă funcționează astfel:

```
foldr (++) [] (map (:[]) [1..]) -->
(++) [1] (foldr (++) [] (map (:[])[2..]) -->
(++) [1] ((++) [2] (foldr (++) [] (map (:[]) [3..])) -->
```

- În momentul în care apelăm **take** n forțăm evaluarea.
- Deoarece (++) este liniară în primul argument:

```
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

primii n termeni ai expresiei

```
(++) [1] ((++) [2] (foldr (++) [] (map (:[]) [3..])))
```

pot fi determinați fără a calcula toată lista

```
1: ((++) [2] (foldr (++) [] (map (:[]) [3..])) -->
1: 2 : ((++) [3] (foldr (++) [] (map (:[]) [4..])) -->
```

Evaluare leneșă. Liste infinite

- Intuitiv, evaluarea leneșă funcționează astfel:

```
foldl (++) [] (map (:[]) [1..]) -->
foldl (++) [] (1: map (:[]) [2..]) -->
foldl (++) ((++) [1] []) (map (:[]) [2..]) -->
foldl (++) ((++) [1] []) (2: map (:[]) [3..]) -->
foldl (++) ((++) ((++) [1] []) [2]) (map (:[]) [3..]) -->
```

Evaluare leneșă. Liste infinite

- Intuitiv, evaluarea leneșă funcționează astfel:

```
foldl (++) [] (map (:[]) [1..]) -->
foldl (++) [] (1: map (:[]) [2..]) -->
foldl (++) ((++) [1] []) (map (:[]) [2..]) -->
foldl (++) ((++) [1] []) (2: map (:[]) [3..]) -->
foldl (++) ((++) ((++) [1] []) [2]) (map (:[]) [3..]) -->
```

- În cazul lui **foldl** se expresia care calculează rezultatul final trebuie definită complet, ceea ce nu este posibil în cazul listelor infinite.

Pe săptămâna viitoare!