# Programare funcțională

## Evaluare lenesă

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

3 noiembrie 2020

# Evaluare lenesă. 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 lenesă

- 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 lenesă. 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 lenesă**

- 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 lenesă: lista numerelor prime

Vă amintiți din primul curs:

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

# Evaluare lenesă: 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 lenesă 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

## 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
```

# 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

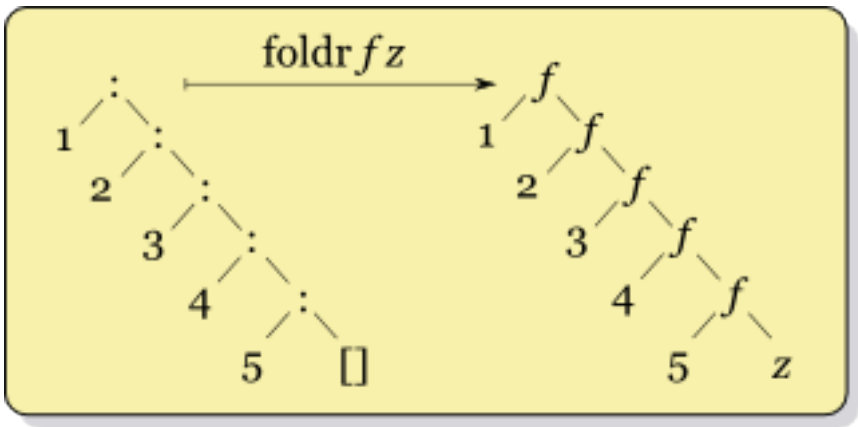
## foldr și foldl

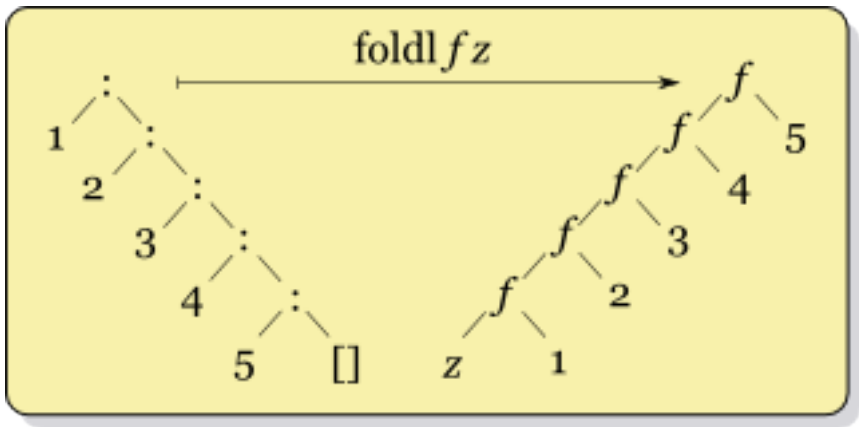
Date fiind o funcție de actualizare a valorii calculate cu un element curent, o valoare inițială, și o listă, calculați valoare obținută prin aplicarea repetată a funcției de actualizare fiecărui element din listă.

```
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




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 lenesă. Liste infinite

- Intuitiv, evaluarea lenesă 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 lenesă. Liste infinite

- Intuitiv, evaluarea lenesă 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 lenesă. Liste infinite

- Intuitiv, evaluarea lenesă funcționează astfel:

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

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

# Evaluare lenesă. Liste infinite

- Intuitiv, evaluarea lenesă 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 lenesă. Liste infinite

- Intuitiv, evaluarea lenesă funcționează astfel:

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

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

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

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

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

# Evaluare lenesă. Liste infinite

- Intuitiv, evaluarea lenesă 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.

# Optimizarea recursiei folosind evaluarea lenesă

- Să presupunem că vrem să optimizăm generarea șirului Fibonacci

```
f   :: Int -> Integer
f 0 = 0
f 1 = 1
f n = f (n-2) + f (n-1)
```

- o variantă folosind evaluarea lenesă

```
next (a : b : t) = (a + b) : next (b : t)
fibs = 0 : 1 : next fibs
```

# Optimizarea recursiei - evaluare lenesă

- o variantă folosind evaluarea lenesă

```
next (a : b : t) = (a + b) : next (b : t)
fibs = 0 : 1 : next fibs
```

Intuitiv, evaluarea lenesă funcționează astfel:

# Optimizarea recursiei - evaluare lenesă

- o variantă folosind evaluarea lenesă

```
next (a : b : t) = (a + b) : next (b : t)
fibs = 0 : 1 : next fibs
```

Intuitiv, evaluarea lenesă funcționează astfel:

```
fibs --> 0 : 1 : next fibs -->

0 : 1 : next ( 0 : 1 : t) -->          [fibs --> 0 : 1 : t]
0 : 1 : 1 : next (1 : t)  -->          [t --> 1 : next (1 : t)
    ]
0 : 1 : 1 : next (1: 1: t') -->        [t--> 1 : t']
0 : 1 : 1 : 2 : next(1:t')  -->        [t'--> 2 : next (1:t')]
0 : 1 : 1 : 2: next(1:2:t'')  -->      [t'--> 2 : t'']
0 : 1 : 1 : 2 : 3 : next(2:t'')-->     [t''-->3 : next(2:t'')]
```

# Optimizarea recursiei

## Memoizare

- Să presupunem că vrem să optimizăm generarea șirului Fibonacci

```
f   :: Int -> Integer
f 0 = 0
f 1 = 1
f n = f (n-2) + f (n-1)
```

prin reținerea și accesarea directă a valorilor anterior calculate (*memoizare*).

# Optimizarea recursiei

## Memoizare

- Să presupunem că vrem să optimizăm generarea șirului Fibonacci

```
f   :: Int -> Integer
f 0 = 0
f 1 = 1
f n = f (n-2) + f (n-1)
```

  prin reținerea și accesarea directă a valorilor anterior calculate (*memoizare*).

- Haskell este un limbaj *stateless*, nu avem posibilitatea de a reține valorile într-un vector, așa cum am face într-un limbaj imperativ.

  Cum procedăm?

# Optimizarea recursiei

- Să presupunem că vrem să optimizăm generarea șirului Fibonacci

```
f :: Int -> Integer
f 0 = 0
f 1 = 1
f n = f (n-2) + f (n-1)
```

  prin reținerea valorilor anterioare.
- În Haskell putem reține valorile generate de o funcție într-o listă folosind funcția **map**

```
genf :: Int -> Integer
genf n = (map f [1..]) !! n
```

  Obervați că:
  - folosim *evaluarea lenesă* pentru a construi lista *tuturor* numerelor
  - accesăm elementul n din lista

# Optimizarea recursiei

- Să presupunem că vrem să optimizăm generarea șirului Fibonacci

```
f :: Int -> Integer
f 0 = 0
f 1 = 1
f n = f (n-2) + f (n-1)
```

  prin reținerea valorilor anterioare.

- În Haskell putem reține valorile generate de o funcție într-o listă folosind funcția **map**

```
genf :: Int -> Integer
genf n = (map f [1..]) !! n
```

  Obervați că:

  - folosim *evaluarea lenește* pentru a construi lista *tuturor* numerelor
  - accesăm elementul n din lista

  Nu am rezolvat problema optimizării,
  dar am găsit o modalitate de a construi lista valorilor.

# Optimizarea recursiei

- Deoarece știm cum să construim lista de valori, putem elimina apelul recursiv cu accesarea elementelor listei:

```
f :: Int -> Integer
f 0 = 0
f 1 = 1
f n = genf (n-2) + genf (n-1)  -- am inlocuit
                               -- apelul recursiv

genf :: Int -> Integer
genf n = (map f [1..]) !! n
```

- Credeți că am rezolvat problema?

# Optimizarea recursiei

- Deoarece știm cum să construim lista de valori, putem elimina apelul recursiv cu accesarea elementelor listei:

```
f :: Int -> Integer
f 0 = 0
f 1 = 1
f n = genf (n-2) + genf (n-1)  -- am inlocuit
                               -- apelul recursiv

genf :: Int -> Integer
genf n = (map f [1..]) !! n
```

- Credeți că am rezolvat problema? Nu, deoarece listele care calculează rezultatele în genf (n−2) și genf (n−1) sunt diferite. Fiecare apel al lui f creaza liste noi, de fapt complexitatea crește.

# Optimizarea recursiei

- Deoarece știm cum să construim lista de valori, putem elimina apelul recursiv cu accesarea elementelor listei:

```
f :: Int -> Integer
f 0 = 0
f 1 = 1
f n = genf (n-2) + genf (n-1)  -- am inlocuit
                               -- apelul recursiv

genf :: Int -> Integer
genf n = (map f [1..]) !! n
```

- Credeți că am rezolvat problema? Nu, deoarece listele care calculează rezultatele în genf (n−2) și genf (n−1) sunt diferite. Fiecare apel al lui f creaza liste noi, de fapt complexitatea crește.

  Trebuie să găsim o soluție în care să folosim o singură listă.

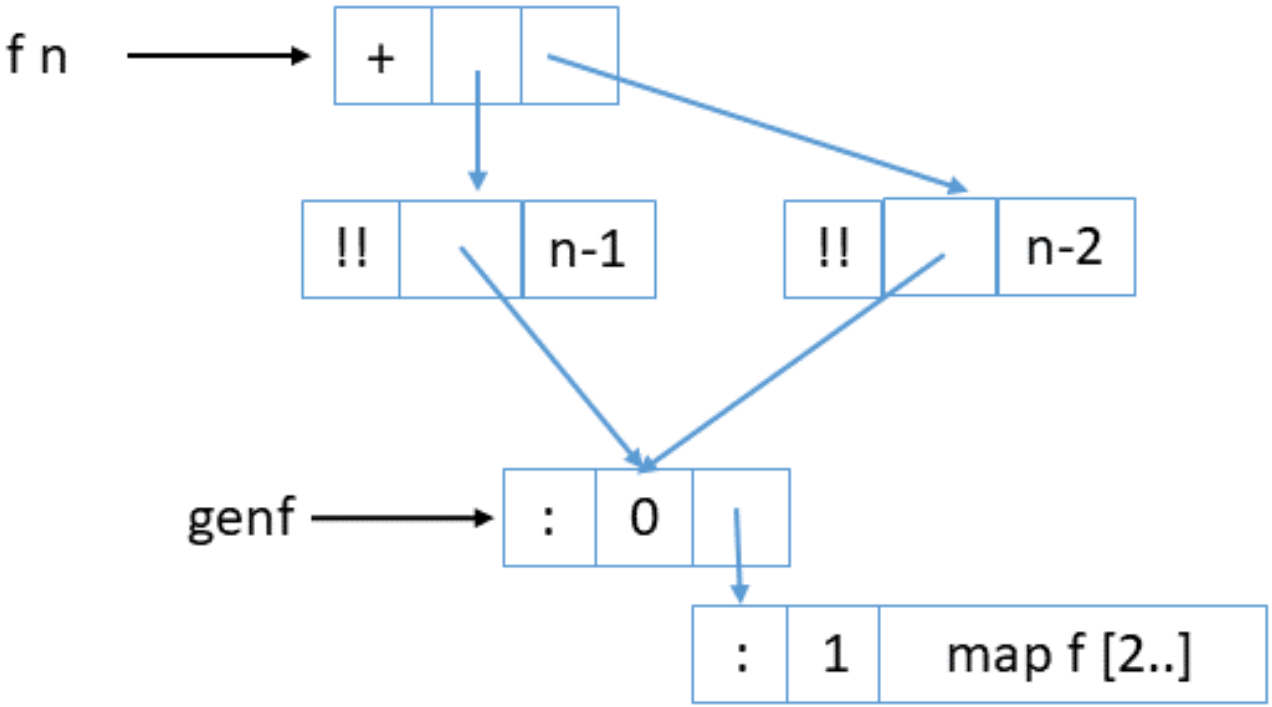
# Optimizarea recursiei: memoizare

O soluția corectă:

```
f :: Int -> Integer
f 0 = 0
f 1 = 1
f n = (genf !! (n-2)) + (genf !! (n-1))


genf = map f [0..]

*Main> f 200
280571172992510140037611932413038677189525
(0.01 secs, 206,448 bytes)
```

# Optimizarea recursiei: memoizare

In Haskell expresiile sunt reprezentate sub forma unor grafuri:




Elementele lui genf sunt evaluate o singură dată!