Haskell: Introducere

- Data publicării: 28.03.2021
- Data ultimei modificări: 28.03.2021

Obiective

Scopul acestui laborator este acomodarea cu primele noțiuni din Haskell și observarea incipientă a diferențelor dintre acesta și Racket.

Aspectele urmărite sunt:

- diferențele principale dintre cele 2 limbaje
- definirea funcțiilor în Haskell
- pattern matching
- tipuri
- gărzi
- funcționale
- domenii de vizibilitate ale definițiilor: top-level și local

Introducere

În jurul anilor 1990 un comitet de cercetători în limbaje de programare (Simon Marlow, Simon Peyton Jones, Philip Wadler etc) au creat un limbaj nou care a ajuns să fie standardul de-facto în cercetarea din domeniul programării funcționale. Inspirat dintr-o varietate de limbaje – Miranda, ML, Scheme, APL, FP – limbajul a influențat la rândul lui majoritatea limbajelor de programare cunoscute.

Haskell este un limbaj funcțional **pur**. Spre deosebire de limbajele imperative (sau Racket unde există *set!*), în Haskell aproape toate funcțiile sunt pure. Funcțiile impure sunt marcate diferit prin intermediul sistemului de tipuri.

În plus față de Racket, Haskell are **tipare statică** (și tipuri **polimorfice**). Fiecare expresie are un tip și este sarcina programatorului să efectueze conversiile necesare între tipuri dacă este necesar. De cele mai multe ori, informația despre ce face o funcție se găsește integral în tipul acesteia și numele ei. Astfel, se pune accentul pe *ce* face funcția, nu *cum* efectuează ea operațiile cerute.

Deși tipurile există, programatorul nu este obligat să depună efort în a le scrie în program. Haskell deține inferență de tipuri puternică. Exceptând cazurile în care se folosesc concepte avansate, programatorul poate lucra fără a scrie o singură semnătură de funcție (deși nu e recomandat pentru că se pierde o parte din documentația funcției).

În plus, tipurile ajută programatorul în procesul de scriere a codului transformându-l într-un exercițiu de rezolvare a unui puzzle: pur și simplu trebuiesc potrivite tipurile folosind funcții existente sau scriind alte funcții. Pentru a căuta funcțiile ce respectă o semnătură se poate folosi Hoogle [http://www.haskell.org/hoogle/], un motor de căutare similar Google dar doar pentru funcții Haskell.

Deosebirea fundamentală față de alte limbaje este **evaluarea leneșă**. Funcțiile nu vor fi evaluate și expresiile nu vor fi reduse până în momentul în care valoarea lor este necesară. Programatorul poate astfel lucra cu date infinite, extrăgând din ele strictul necesar pentru a obține soluția. Ca dezavantaj, analiza performanței unui cod Haskell este puțin mai dificilă, dar există instrumente auxiliare (dezvoltate în Haskell).

În cele ce urmează vom parcurge fiecare dintre aceste particularități, accentuând diferențele față de Racket. Pe scurt, o **paralelă** între cele două limbaje arată în felul urmator:

Limbaj	Evaluare	Tipare	Efecte laterale
Racket	Aplicativă	Dinamică	Da
Haskell	Leneșă	Statică	Nu

Laboratoarele de Haskell din cadrul cursului se vor axa pe evidențierea acestor diferențe și a beneficiilor care rezultă din ele. Pentru noțiuni avansate despre limbaj, consultați primele 2 resurse din bibliografie (sau Reddit, Stack Overflow, Planet Haskell, etc.)

Un exemplu pentru diferența de evaluare dintre Racket și Haskell:

```
(define (square x) (* x x))
```

```
square x = x * x
```

Evaluarea aplicativă din Racket va forța evaluarea parametrului x și apoi va apela funcția square:

```
(square (+ 1 2))
(square 3)
(* 3 3)
9
```

Evaluarea leneșă din Haskell va evalua parametrul x la cerere în cadrul apelului funcției square:

```
square (1 + 2)
(1 + 2) * (1 + 2)
3 * (1 + 2)
```

```
3 * 3
9
```

GHC. GHCi

Există o varietate de compilatoare și interpretoare pentru Haskell. În momentul de față, limbajul și evoluția lui sunt strâns legate de eforturile dezvoltatorilor de la Glasgow. GHC, *The Glorious Glasgow Haskell Compilation System*, este și compilatorul pe care-l vom folosi pentru cursul de PP.

Pentru a avea o experiență bună cu acest limbaj, recomandarea este să vă instalați Haskell Stack [https://docs.haskellstack.org/en/stable/install_and_upgrade/] (citiți instrucțiunile de aici: Limbaje [https://ocw.cs.pub.ro/courses/pp/22/limbaje]). Față de compilator și de suita minimală de pachete, Haskell Stack aduce în plus o suită de biblioteci utile pentru dezvoltarea unor aplicații reale.

Codul Haskell poate fi atât compilat cât și interpretat. Pentru interpretare vom folosi **ghci**, iar pentru compilare vom folosi **ghc**. Fișierele de cod Haskell au în mod normal extensia `.hs`, dar se poate folosi și `.lhs` pentru variantele de **Literate Haskell** (programare ca o poveste - comentariile ocupă majoritatea textului în timp ce secvențele de cod sunt puține - este formatul preferat pentru publicarea de articole despre Haskell pe bloguri, oricine poate copia textul articolului într-un fișier și îl poate compila și rula apoi).

În cadrul laboratorului, recomandăm folosirea **ghci** în modul următor:

- se salvează definițiile de funcții într-un fișier cu extensia `.hs` (ex.: lab6-ex.hs)
- se deschide consola sistemului (ex.: cmd/powershell pe Win) in directorul cu fisierul `.hs`
- se rulează `stack exec ghci nume.hs` pentru a-l încărca în interpretor
- se rulează în interpretor apelurile de funcții de test necesare, verificarea de tipuri, etc
- dacă se dorește **editarea fișierului** se poate face într-un terminal separat (recomandat) sau folosind `:e` (`:edit`) din `ghci`
- ATENTIE! după editarea fișierului, pentru a reîncărca definițiile se folosește `:re` (`:reload`)
- dacă se dorește încărcarea altor module în interpretor (pentru testare), se poate folosi `:m +Nume.Modul`
- pentru a verifica tipul unor expresii se folosește `:t expresie`
- pentru a ieși din interpretor se folosește `:q` (sau EOF - ^Z pe Linux/OSX, ^D pe Windows)

Modulul `Prelude` conține funcții des folosite și este inclus implicit în orice fișier (deși poate fi exclus la nevoie, consultați bibliografia).

În cadrul unor programe reale va trebui să compilați sursele. Pentru aceasta va trebui să aveți un modul `Main.hs` care să conțină o funcție `main`. Modulul poate importa alte module prin `import Nume.Modul`. Pentru temă, nu va trebui să efectuați acești pași, scheletul de cod ce va fi oferit va conține partea de interacțiune cu mediul exterior. 😃

Comentarii

În Haskell, avem comentarii pe un singur rând, folosind `--` (2 de - legați) sau comentarii pe mai multe rânduri, încadrate de `{-` și `-}`. În plus, există comentarii pentru realizarea de documentație dar nu vom insista pe ele aici.

Funcții

O funcție anonimă în **Racket**

```
(lambda (x y) (+ x y))
```

poate fi tradusă imediat în Haskell utilizând tot o *abstracție lambda*

```
\x y -> x + y
```

Sintaxa Haskell este ceva mai apropriată de cea a **calculului Lambda**: `\` anunță parametrii (separați prin spații), iar `->` precizează corpul funcției.

În Racket, puteam da nume funcției folosind legarea dinamică prin intermediul lui `define` sau legarea statică prin intermediul lui `let`. Exemplele următoare ilustrează cele 2 cazuri, împreună cu apelul funcției.

```
(define f (lambda (x y) (+ x y)))
(f 2 3)
```

```
(let ((f (lambda (x y) (+ x y))))
  (f 2 3))
```

În Haskell, funcția echivalentă ar fi

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

sau

```
f x y = x + y
```

Și apelul

```
f 2 3
```

Observați că se renunță la paranteze. Unul dintre principiile dezvoltării limbajului a fost oferirea de construcții cât mai simple pentru lucrurile folosite cel mai des. Fiind vorba de programarea funcțională, aplicarea de funcții trebuia făcută cât mai simplu posibil.

Tot din același considerent, operatorii în Haskell sunt scriși în forma **infixată**. Restul apelurilor de funcții sunt în forma prefixată, exact ca în Racket. Totuși, se pot folosi ambele forme: pentru a folosi un operator prefixat se folosesc paranteze, în timp ce pentru a infixa o funcție se folosesc *back-quotes* (`).

```
2 + 3 == (+) 2 3
elem 2 [1,2,3] == 2 `elem` [1, 2, 3]
```

Tot pentru simplitate, Haskell permite folosirea unor secțiuni – elemente de zăhărel sintactic care se vor traduce în funcții anonime:

```
(2 +) == \x -> 2 + x
(+ 2) == \x -> x + 2
(- 2) == -2
(2 -) == \x -> 2 - x
```

Atenție: În construcția `(- x)` operatorul - este unar, nu binar (este echivalentul funcției `negate`). Dacă doriți să aplicați pațial la dreapta operatorul de scădere, utilizați funcția `subtract`, ca în expresia `(\subtract` 2)`.

Tipuri de bază

În această secțiune vom prezenta tipurile existente în limbajul Haskell. Veți observa că limbajul este mult mai bogat în tipuri decât Racket. Programatorul își va putea defini alte tipuri proprii dacă dorește.

Pentru a putea vedea tipul unei expresii în `ghci` folosiți `:t expresie`.

```
> :t 'a'
'a' :: Char
```

Operatorul `::` separă o expresie de tipul acesteia.

```
> :t (42::Int)
(42::Int) :: Int
```

Numere, caractere, siruri, booleeni

Următorii literali sunt valizi in **Haskell** (dupa operatorul `::` sunt precizate tipurile acestora):

```
5 :: Int
'H' :: Char
"Hello" :: String -- sau [Char] -- lista de Char
True :: Bool
False :: Bool
```

Observați că există tipul caracter și tipul șir de caractere. Tipul `String` este de fapt un sinonim pentru tipul `[Char]` - tipul listă de caractere. Astfel, operațiile pe liste vor funcționa și pe șiruri.

Tipurile numerice sunt puțin diferite față de alte limbaje de programare cunoscute:

```
> :t 42
42 :: Num a => a
```

În exemplul de mai sus, `a` este o variabilă de tip (stă pentru orice fel de tip) restricționată (prin folosirea `=>`) la toate tipurile numerice (`Num a`).

Important de reținut este faptul că există **2** tipuri întregi: `Int` și `Integer`. Primul este finit, determinat de arhitectură, în timp ce al doilea este infinit, putând ajunge oricât de mare.

Liste

O listă în Haskell se construiește similar ca în Racket, prin intermediul celor doi constructori:

```
[] -- lista vidă
(:) -- operatorul de adăugare la începutul listei - cons
```

Pentru simplitate, o construcție de forma

```
1:3:5:[]
```

se poate scrie și

```
[1, 3, 5]
```

sau

```
[1, 3 .. 6]
```

sau

```
[1, 3 .. 5]
```

Echivalente pentru `car` și `cdr` din Racket sunt funcțiile `head` și `tail`:

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

Operatorul de **concatenare** este `++`:

```
> [1, 2, 3] ++ [4, 5]
[1, 2, 3, 4, 5]
```

Perechi

În Haskell, elementele unei liste **sunt de același tip**. Dacă dorim construcții cu elemente de tipuri diferite vom folosi tuplurile

```
(3, "Ana") :: (Int, String)
(3, True, "Ana") :: (Int, Bool, String)
```

Haskell oferă funcții pentru extragerea componentelor din perechi

```
> fst (3, "Ana")
3
> snd (3, "Ana")
"Ana"
```

Se ofera de asemenea și funcțiile `zip` și `unzip`, **doar** pentru perechi, dar există și `zip3` și `unzip3` pentru triplete.

Pentru tuplurile cu mai mult de 2 elemente este sarcina programatorului să definească funcțiile folosite.

Definirea funcțiilor

Să considerăm că vrem să scriem o funcție pentru factorialul unui număr. Prima implementare ar folosi sintaxa `if`:

```
factorial_if x = if x < 1 then 1 else x * factorial_if (x - 1)
```

Este obligatoriu ca `if` să conțină **ambele** ramuri și ca acestea să fie **de același tip**.

Mai frumos, putem scrie funcția de mai sus folosind **gărzi**:

```
factorial_guards x
    | x < 1 = 1
    | otherwise = x * factorial_guards (x - 1)
```

Observați indentarea: orice linie care face parte din aceeași expresie ca cea de deasupra trebuie să înceapă la exact aceeași indentare. Orice linie care este o subexpresie a expresiei de mai sus (`then` sau `else` în cazul `if`, fiecare gardă în parte, etc.) trebuie să fie indentată mai spre dreapta.

Construcția `otherwise` este echivalentă expresiei `True`. O gardă poate conține orice fel de expresie care se va evalua la o valoarea de tip `Bool`.

Gărzile sunt testate în ordine, fiind evaluată expresia aferentă primei gărzi adevărate. Astfel, un set de gărzi este similar unei expresii `cond` din Racket, cu toate că există anumite restricții în privința locului în care pot fi definite gărzi.

Pentru a ne apropia de definiția matematică, putem scrie aceeași funcție folosind `case` (echivalentul `switch`-ului din C dar mult mai avansat):

```
factorial_case x = case x < 1 of
    True -> 1
    _ -> x * factorial_case (x - 1)
```

Observați regula indentării aplicată și aici. Expresia `_` semnifică orice valoare, indiferent de valoarea ei. Expresia din `case` poate fi oricare.

Fiecare caz este tratat în ordine, prima potrivire este executată.

În final, putem scrie aceeași funcție folosind **pattern matching**:

```
factorial_pm 0 = 1
factorial_pm x = x * factorial_pm (x - 1)
```

Expresiile din interiorul fiecărei ramuri din `case` sau din interiorul pattern-match nu pot fi decât constructorii unui tip (similar cu ce ați făcut la AA). În continuare vom folosi cele 4 stiluri pentru a ilustra funcția care calculează lungimea unei liste

```
length_if l = if l /= [] then 1 + length_if (tail l) else 0
```

```
length_guard l
    | l /= [] = 1 + length_guard (tail l)
    | otherwise = 0
```

```
length_case l = case l of
    (_ : xs) -> 1 + length_case xs
    _ -> 0
```

```
length_pm [] = 0
length_pm (_:xs) = 1 + length_pm xs
```

Observați în exemplele de mai sus expresivitatea limbajului. Folosirea construcției potrivite duce la definiții scurte și ușor de înțeles.

Curry vs Uncurry

Fiind un limbaj funcțional, în Haskell funcțiile sunt valori de prim rang. Astfel, putem afla tipul unei funcții:

```
f x y = x + y
```

```
> :t f
f :: Num a => a -> a -> a
```

Fiecare argument este separat prin `->` de următorul sau de rezultat.

Amintindu-ne de discuția despre funcții curry și uncurry [https://ocw.cs.pub.ro/courses/pp/22/laboratoare/racket/functionale#functii_curryuncurry], rezultatul următor nu trebuie să ne surprindă

```
> :t f 3
f 3 :: Num a => a -> a
```

Toate funcțiile din Haskell sunt în formă **curry**.

Pentru a face o funcție uncurry putem folosi funcția `uncurry :: (a -> b -> c) -> (a, b) -> c` dacă vrem să transformăm o funcție echivalentă sau putem s-o definim în mod direct folosind perechi.

Funcționale uzuale

Haskell oferă un set de funcționale (similar celui din Racket) pentru cazuri de folosire des întâlnite (dacă nu mai știți ce fac, încercați să ghiciți citind semnătura și numele ca o documentație):

```
> :t map
map :: (a -> b) -> [a] -> [b]
> :t filter
filter :: (a -> Bool) -> [a] -> [a]
> :t foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
> :t zip
zip :: [a] -> [b] -> [(a, b)]
> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Folosirea lor duce la un cod mai ușor de citit și de întreținut.

Domenii de vizibilitate

Spre deosebire de Racket [https://ocw.cs.pub.ro/courses/pp/22/laboratoare/racket/legare], unde legarea variabilelor la nivelul cel mai de sus (top-level) este dinamică, Haskell leagă definițiile **static**, acestea fiind vizibile implicit la nivel **global**. De exemplu, o definiție de forma:

```
theAnswer = 42
```

va putea fi utilizată implicit în toate fișierele încărcate de către compilator/interpretor la un moment dat. Domeniul de vizibilitate al definițiilor top-level poate fi însă redus cu ajutorul definirii de module: consultați capitolul 11 din A Gentle Introduction to Haskell

[http://www.haskell.org/tutorial/modules.html] pentru mai multe detalii.

La fel ca Racket, Haskell permite definirea în cadrul domeniilor de vizibilitate locală (mai exact în cadrul funcțiilor), cu ajutorul clauzelor `let` și `where`.

let

Forma generală a clauzei `let` este următoarea:

```
let id1 = val1
    id2 = val2
    ...
    idn = valn
in expr
```

unde `expr` este o expresie Haskell care poate depinde de `id1, id2, ..., idn`. De asemenea, domeniul de vizibilitate ale definițiilor locale este întreaga clauză `let` (similar cu `letrec` în Racket). Astfel, definiția următoare:

```
p = let x = y + 1
        y = 2
        b n = if n == 0 then [] else n : b (n - 1)
    in (x + y, b 2)
```

este corectă. `x` poate să depindă de `y` datorită **evaluării lenețe**: în fapt `x` va fi evaluat în corpul clauzei, în cadrul expresiei `(x + y, b 2)`, unde `y` e deja definit.

where

Clauza `where` este similară cu `let`, diferența principală constând în folosirea acesteia **după** corpul funcției. Forma generală a acesteia este:

```
def = expr
    where
    id1 = val1
    id2 = val2
    ...
    idn = valn
```

cu aceleași observații ca în cazul `let`.

Un exemplu de folosire este implementarea metodei de sortare QuickSort:

```
qsort [] = []
qsort (p : xs) = qsort left ++ [p] ++ qsort right
    where
    left = filter (< p) xs
    right = filter (>= p) xs
```

Clauzele de tip `let` și `where` facilitează **reutilizarea** codului. De exemplu, funcția:

```
inRange :: Double -> Double -> String
inRange x max
    | f < low               = "Too low!"
    | f >= low && f <= high = "In range"
    | otherwise             = "Too high!"
    where
    f = x / max
    (low, high) = (0.5, 1.0)
```

verifică dacă o valoare normată se află într-un interval fixat. Expresia dată de `f` este folosită de mai multe ori în corpul funcției, motiv pentru care este urmărită încapsularea ei într-o definiție. De asemenea, se observă că definițiile locale, ca și cele top-level, permit pattern matching-ul pe constructorii de tip, în cazul acesta constructorul tipului pereche.

Observăm că `where` și `let` sunt de asemenea utile pentru definirea de funcții auxiliare:

```
naturals = iter 0
    where iter x = x : iter (x + 1)
```

`iter` având în exemplul de mai sus rolul de generator auxiliar al listei numerelor naturale.

Traduceri cod din Racket în Haskell

Aici, vom face comparații între Racket și Haskell, în ceea ce privește sintaxa, pe categorii.

Funcții lambda:

- Racket

```
(lambda (x) (+ x 1))
```

- Haskell

```
\x -> x + 1
```

Operatori aritmetici:

- Racket

```
(+ 1 2)
(- 7 2)
(* 2 11)
(/ 5 2)
(quotient 5 2)
(modulo 5 2)
```

- Haskell

```
1 + 2
7 - 2
2 * 11
5 / 2
mod 7 2
quot 7 2
```

Operatori logici:

- Racket

```
(not #t) ; not
(not #f) ; not
(or #t #f) ; or
(and #t #f) ; and
```

- Haskell

```
not True -- not
not False -- not
True || False -- or
True && False -- and
```

Apelare de funcții:

- Racket

```
(max 2 3)
```

- Haskell

```
max 2 3
```

Perechi:

- Racket

```
(cons 1 2) ; construirea unei perechi
(car (cons 1 2)) ; primul element
(cdr (cons 1 2)) ; al doilea element
```

- Haskell - sunt tupluri, mai precis colecții care pot avea elemente de tipuri diferite

```
(1, 2) -- construirea unei perechi
(fst (1, 2)) -- primul element
(snd (1, 2)) -- al doilea element

(1, 2, 3, 4, [], "aaaa") -- un tuplu care are mai multe elemente, de tipuri diferite
```

Liste:

- Racket - listele sunt eterogene, pot conține elemente de tipuri diferite

```
null ; lista goala
'() ; lista goala

(list 1 2 3 4) ; o lista de numere

(car (list 1 2 3 4)) ; primul element din lista
(cdr (list 1 2 3 4)) ; restul listei, fara primul element

(cons 0 (list 1 2 3 4)) ; adaugarea unui element la inceputul unei liste
```

```
(null? (list 1 2 3 4)) ; se verifica daca lista este goala

(length (list 1 2 3 4)) ; lungimea unei liste

(append (list 1 2 3 4) (list 5 6 7)) ; concatenarea dintre doua liste

(member 4 (list 1 2 3 4)) ; se verifica daca un element exista intr-o lista
```

- Haskell - listele sunt omogene, au elemente de același tip

```
[] -- lista goala

[1, 2, 3, 4] -- o lista de numere

head [1, 2, 3, 4] -- primul element din lista
tail [1, 2, 3, 4] -- restul listei, fara primul element

0 : [1, 2, 3, 4] -- adaugarea unui element la inceputul unei liste

null [1, 2, 3, 4] -- se verifica daca lista este goala
[] ==  [1, 2, 3, 4] -- se verifica daca lista este goala (trebuie sa elementele sa poata fi comparate - vom discuta despre Eq la laboratorul de clase in Haskell)

length [1, 2, 3, 4] -- lungimea unei liste

[1, 2, 3, 4] ++ [5, 6, 7] -- concatenarea dintre doua liste

elem 4 [1, 2, 3, 4] -- se verifica daca un element exista intr-o lista
4 `elem` [1, 2, 3, 4] -- se verifica daca un element exista intr-o lista
```

Sintaxa if:

- Racket

```
(if (< a 0)
    (if (> a 10)
        (*a a)
            0)
    -1))
```

- Haskell

```
if a < 0 then
    if (a > 10)
        then a * a
        else 0
    else -1
```

Definirea unei funcții:

- Racket

```
(define (sum-list l)
  (if (null? l)
      0
      (+ (car l) (sum-list (cdr l)))))
```

- Haskell

```
-- cu if-else-then
sumList :: [Int] -> Int
sumList l = if null l then 0 else head l + sumList (tail l)

-- pattern matching
sumList2 :: [Int] -> Int
sumList2 [] = 0
sumList2 (x:xl) = x + sumList2 xl

-- cu garzi
sumList3 :: [Int] -> Int
sumList3 l
    | null l = 0
    | otherwise = head l + sumList3 (tail l)

-- cu case of
sumList4 :: [Int] -> Int
sumList4 l = case l of
    [] -> 0
    (x:xl) -> x + sumList4 xl
```

Funcționale:

- Racket

```
; map
(map (λ (x) (+ x 1)) (list 1 2 3 4)) ; '(2 3 4 5)
(map add1 (list 1 2 3 4)) ; '(2 3 4 5)
```

```
; filter
(filter (λ (x) (equal? (modulo x 2) 0)) (list 1 2 3 4 5 6)) ; '(2 4 6)

; foldl
(reverse (foldl (lambda (x acc) (cons x acc)) '() (list 1 2 3 4 5))) ; '(1 2 3 4 5)

; foldr
(foldr (lambda (x acc) (cons x acc)) '() (list 1 2 3 4 5)) ; '(1 2 3 4 5)
```

- Haskell

```
-- map
map (\x -> x + 1) [1, 2, 3, 4] -- [2, 3, 4, 5]
map (+ 1) [1, 2, 3, 4] -- [2, 3, 4, 5]

-- filter
filter (\x -> mod x 2 == 0) [1, 2, 3, 3, 4, 5, 6] -- [2, 4, 6]

-- foldl
reverse $ foldl (\acc x -> x : acc) [] [1, 2, 3, 4, 5] -- [1, 2, 3, 4, 5]

-- foldr
foldr (\x acc -> x : acc) [] [1, 2, 3, 4, 5] -- [1, 2, 3, 4, 5]
```

Un fapt notabil în ceea ce diferența dintre Haskell și Racket este în faptul că diferă ordinea parametrilor la funcția anonimă dată ca parametru între tipurile de fold (în Racket nu diferă ordinea, care este element → acumulator, acest fapt se poate observa în exemplele de mai sus), mai precis:

- `foldl`: acumulator → element
- `foldr`: element → acumulator

Legări:

- Racket

```
(define (f a)
    (let ((c a) (b (+ a 1)))
        (+ c b)))

(define (g a)
    (let* ((c a) (b (+ c 1)))
        (+ c b)))

;(define (h a)
;    (letrec ((c b) (b (+ a 1)))
;        (+ c b)))               ; aici vom avea eroare, pentru ca la legarea lui b la c, b-ul nu este definit
```

- Haskell - `let` în Haskell se comporta precum `letrec` din Racket

```
-- cu let
f a =
    let c = a
        b = a + 1
    in (c + b) -- let din Racket

g a =
    let c = a
        b = c + 1
    in (c + b) -- let* din Racket

h a =
    let c = b
        b = a + 1
    in (c + b) -- letrec din Racket, aici nu avem eroare datorita evaluarii lenese

-- cu where
f' a = (c + b)
    where
        c = a
        b = a + 1  -- let din Racket

g' a = (c + b)
    where
        c = a
        b = c + 1  -- let* din Racket

h' a = (c + b)
    where
        c = b
        b = a + 1  -- letrec din Racket, aici nu avem eroare datorita evaluarii lenese
```

Resurse

- Cheatsheet (partea cu list comprehensions, funcționale utile și operatori se aplică pentru laboratorul 7) [https://github.com/cs-pub-ro/PP-laboratoare/raw/master/haskell/intro/haskell-cheatsheet-1.pdf]
- Schelet [https://ocw.cs.pub.ro/courses/_media/pp/22/laboratoare/haskell/intro-schelet.zip]

- Soluții [https://ocw.cs.pub.ro/courses/_media/pp/22/laboratoare/haskell/intro-solutii.zip]

Referințe

- *Learn You a Haskell [http://learnyouahaskell.com/chapters]*, este utilă pentru toate laboratoarele de Haskell plus ceva extra
- *Hoogle [http://www.haskell.org/hoogle/]*, search for Haskell functions starting from types
- *Hackage [http://hackage.haskell.org/packages/hackage.html]*, repository pentru pachete Haskell
- *Local definitions [http://en.wikibooks.org/wiki/Haskell/Variables_and_functions#Local_definitions]*
- *Comparație [http://www.lambdadays.org/static/upload/media/14254629275479beameraghfuneval2.pdf]* între diferiți algoritmi de sortare implementați in limbaje funcționale
- *Let vs Where [https://wiki.haskell.org/Let_vs._Where]*