Programare funcțională

Introducere în programarea funcțională folosind Haskell

C01 - Organizare. Intro în Haskell

Claudia Chiriță
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Welcome

Organizare

Instructori

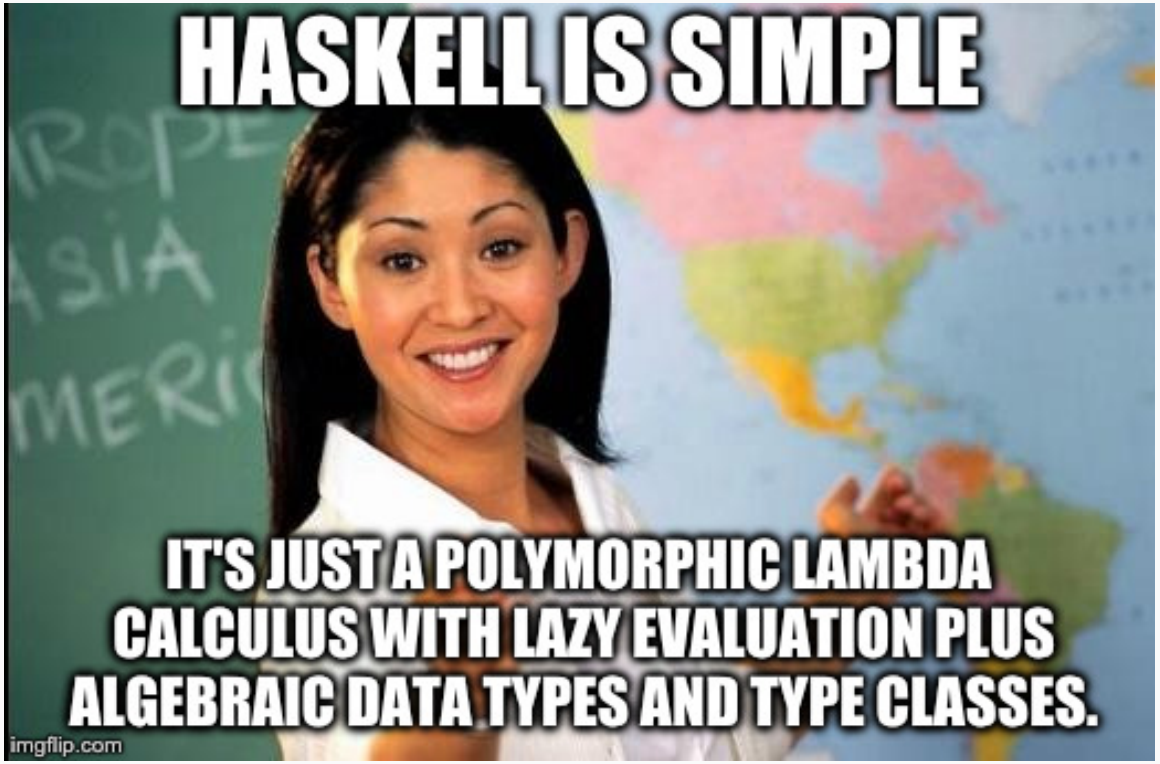
- Curs
  - Claudia Chiriță (webpage) (seria 23)
  - Denisa Diaconescu (webpage) (seriile 24, 25)
- Laborator
  - Ioan Ionescu (241, 242)
  - Ștefan Mocanu (233, 234)
  - Alex Oltean (232, 251, 252)
  - Stelian Stoian (231)
  - Dafina Trufaș (243, 244)

Suport de curs

- Materiale pentru curs și laborator:
  `https://tinyurl.com/PF2025-git`
- Canal Teams:
  `https://tinyurl.com/PF2025-MTeams`

Prezență

Prezența la curs nu este obligatorie, dar încurajată.



Evaluare

Notare

- Sesiune: 1 (oficiu) + notă laborator + examen
- Restanță 1: la alegere una din variantele de mai jos
  - 1 (oficiu) + notă laborator + examen (dacă aveți notă la laborator și vreți să o păstrați)
  - 1 (oficiu) + examen (altfel)
- Restanță 2: 1 (oficiu) + examen

Condiție de promovabilitate

- cel puțin 5 > 4.99

Laborator

Notă laborator

- max. 2 puncte
- se acordă pentru exercițiile rezolvate în cadrul laboratorului

Bonus

- max. 1 punct
- se acordă pentru activitate excepțională (de ex., rezolvarea exercițiilor suplimentare)

Examen

- max. 7 puncte
- durata 2 ore
- în sesiune, fizic
- 28 întrebări grilă
- un cheatsheet comun
    - Concurs "Best Cheatsheet"
    - mai multe detalii în decembrie

Resurse suplimentare

- Pagina Haskell
  `http://haskell.org`
- Hoogle
  `https://www.haskell.org/hoogle`
- Haskell Wiki
  `http://wiki.haskell.org`
- Cartea online „Learn You a Haskell for Great Good”
  `http://learnyouahaskell.com/`

De ce programare funcțională?

Sondaj

Ce știți despre programarea funcțională?

https://tinyurl.com/PF-C01-Quiz1

Programare Imperativă vs. Declarativă

Imperativă (*the How*)

- Îi dai calculatorului o listă clară de pași.
- Gândește-te la un **checklist** sau la o **rețetă**: faci asta, apoi asta, apoi asta.
- Programare procedurală / orientată pe obiecte.

Declarativă (*the What*)

- Îi spui doar ce vrei sa obții, nu fiecare pas de urmat.
- Ca la restaurant: tu spui doar "vreau paste"; bucătarul decide cum gătește pastele.
- Programare logică / baze de date / funcțională

Programare funcțională

Programare funcțională în limbajul vostru preferat
(Java 8, C++11, Python, JavaScript, . . .)

- Funcții anonime
- Funcții de procesare a fluxurilor de date: filter, map, reduce

Agregarea datelor dintr-o colecție (JS)

C. Boesch, Declarative vs Imperative Programming - Talk.JS

https://www.youtube.com/watch?v=M2e5sq1rnvc

Will I ever prefer to read declarative javascript?

```
function multiply(array) {
  return array.reduce( (a,b) => a*b, 1);
}

function multiply(array) {
  var total = 1;
  for (var i = 0; i < array.length; i++){
    total = total * array[i];
  }
  return total;
}
```

Agregarea datelor dintr-o colecție (JS)

C. Boesch, Declarative vs Imperative Programming - Talk.JS

https://www.youtube.com/watch?v=M2e5sq1rnvc

Reasons to be More Declarative

- Better readability
- Better scalability
- Fewer state-related bugs
- Stand on the shoulders of giants

De ce Haskell?

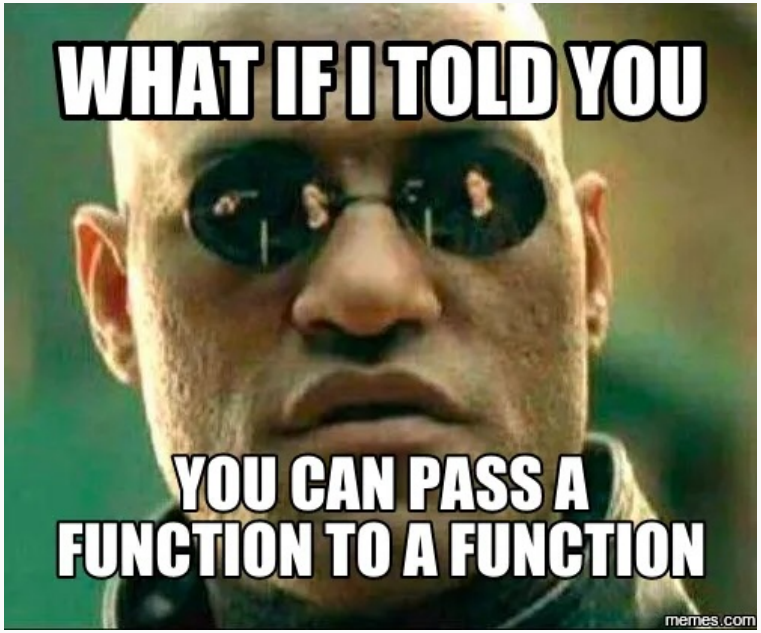
Programare funcțională pură

- Totul este o funcție
- O funcție produce mereu același rezultat pentru aceeași intrare
- O funcție nu poate modifica intrarea și nu poate produce *side effects* (e.g., citire dintr-un fișier, afișarea unui text)

De ce Haskell?

Funcțiile sunt *first-class citizens*

- Funcțiile sunt folosite ca valori
- Funcțiile pot fi transmise ca argumente și pot fi returnate de alte funcții



De ce Haskell?

Lazy

- Amânarea calculelor
  - orice expresie este evaluată doar atunci când chiar ai nevoie de rezultat
- Schimbă modul de gândire
  - programele nu mai sunt doar o succesiune de pași executați imediat, ci devin un set de descrieri pe care mașina le "activează" la momentul potrivit
- Colecții infinite
  - poți lucra elegant cu structuri de date infinite, de exemplu lista [1..], fără să explodeze programul
  - poți decide explicit când și cât dintr-o structură de date să fie evaluat
- Optimizare automată
  - evaluarea lazy evită calcule inutile, ceea ce poate reduce timpul de execuție

De ce Haskell?

Strongly Typed

- Orice are un tip
- Compilatorul verifică tipurile la compile-time
- Compilatorul deduce tipurile

De ce Haskell?

Limbaj elegant

- Concepte puternice din matematică
  - recursivitate, compunerea de funcții, functori, monade
  - λ-calcul
- Rigurozitate
  - gândim mai mult înainte și producem cod mai curat
- Curbă de învățare în trepte
  - putem scrie programe mici destul de repede
  - expertiza în Haskell necesită multă gândire și mult exercițiu
  - descoperirea unei lumi noi poate fi fun
    `http://wiki.haskell.org/Humor`

Haskell

https://serokell.io/blog/10-reasons-to-use-haskell

Proiecte din industrie în Haskell

- Compilatorul GHC pentru Haskell
- pandoc
  - convertor de documente versatil, care suportă multe formate de fișiere.
- Xmonad
  - *tiling window manager* dinamic pentru sistemul X Window.
- PostgREST
  - server web care expune o bază de date PostgreSQL printr-un API RESTful.
- Cardano
  - blockchain

Ce vom învăța: imagine de ansamblu

Programare funcțională în Haskell

- Tipuri, funcții
- Recursivitate, liste
- Funcții de nivel înalt
- Polimorfism
- Tipuri de date algebrice
- Clase de tipuri
- Functori
- Monade

Exemplu 1

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (p:xs) =
   (qsort lesser) ++ [p] ++ (qsort greater)
    where
        lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

Exemplu 2

```
repMax :: Ord a => [a] -> [a]
repMax [] = []
repMax [x] = [x]
repMax (x:y:xs) = max m r : (r:rs)
  where r:rs = repMax (m : xs)
        m = max x y
```

Magia din spate: λ-calcul

În 1929-1932, Alonzo Church a propus λ-calculul ca sistem formal pentru logica matematică. În 1935 a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată în λ-calcul.

$$
\begin{array}{lll}
t = & x & \text{(variabilă)} \\
& \mid \lambda x.\, t & \text{(abstractizare)} \\
& \mid t\ t & \text{(aplicare)}
\end{array}
$$

Magia din spate: λ-calcul

- Independent, în 1935, Alan Turing a introdus mașina Turing.
- În 1936, Turing a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată de o mașină Turing.
- Turing a arătat echivalența celor două modele de calcul.
- Această echivalență a constituit o indicație puternică asupra "universalității" celor două modele (Teza Church-Turing).

Elemente de bază. Primii pași

Sintaxă

Comentarii

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

Identificatori

- șiruri formate din litere, cifre, caracterele _ și ’ (apostrof)
- identificatorii pentru variabile încep cu literă mică sau _
- identificatorii pentru tipuri și constructori încep cu majuscule
- Haskell este case sensitive

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

Blocuri și 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
trei  =  let a = 1; b = 2 in a + b
```

Variabile

Presupunem că fișierul `test.hs` conține

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

Ce valoare are x?

Variabile

Presupunem că fișierul `test.hs` conține

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

Ce valoare are x?

```
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
  | ^
```

Variabile

În Haskell, variabilele sunt imutabile *(immutable)*, adică:

- = **nu** este operator de atribuire
- x = 1 reprezintă o legatură *(binding)*
- din momentul în care o variabilă este legată la o valoare, acea valoare nu mai poate fi schimbată

Legarea variabilelor

let .. in ... este o expresie care creează un domeniu de vizibilitate local.

Presupunem că fișierul `testlet.hs` conține

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

Ce valoare au z și x?

Legarea variabilelor

let .. in ... este o expresie care creează un domeniu de vizibilitate local.

Presupunem că fișierul `testlet.hs` conține

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

Ce valoare au z și x?

```
-- z = 3
-- x = 1
```

Legarea variabilelor

let .. in ... este o expresie care creează un domeniu de vizibilitate local.

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

Ce valoare are x?

Legarea variabilelor

let .. in ... este o expresie care creează un domeniu de vizibilitate local.

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

Ce valoare are x?

```
-- x = 12
```

Legarea variabilelor

let .. in ... este o expresie care creează un domeniu de vizibilitate local.

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

Ce valoare are x?

```
-- x = 12
```

Dar în situația de mai jos, ce valoare are x?

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

Legarea variabilelor

let .. in ... este o expresie care creează un domeniu de vizibilitate local.

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

Ce valoare are x?

```
-- x = 12
```

Dar în situația de mai jos, ce valoare are x?

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

```
-- x = 5
```

Legarea variabilelor

clauza **... where ...** creează un domeniu de vizibilitate local

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

Ce valoare are f 1?

Legarea variabilelor

clauza **... where ...** creează un domeniu de vizibilitate local

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

Ce valoare are f 1?

```
-- x = 4
```

Legarea variabilelor

let .. in ... este o expresie

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

where este o clauză, disponibilă doar la nivel de definiție

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

Legarea variabilelor

let .. in ... este o expresie

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

where este o clauză, disponibilă doar la nivel de definiție

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

Variabile pot fi legate și prin *pattern matching* la definirea unei funcții sau expresii **case**.

```
h x | x == 0    = 0
    | x == 1    = y + 1
    | x == 2    = y * y
    | otherwise = y
  where y = x*x
```

```
f x = case x of
            0 -> 0
            1 -> y + 1
            2 -> y * y
            _ -> y
  where y = x*x
```

Quiz time!

https://tinyurl.com/PF-C01-Quiz2

Pe săptămâna viitoare!