Programare funcțională

Introducere în programarea funcțională folosind Haskell

C01 - Organizare. Elemente de bază

Claudia Chiriță

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Organizare

Instructori

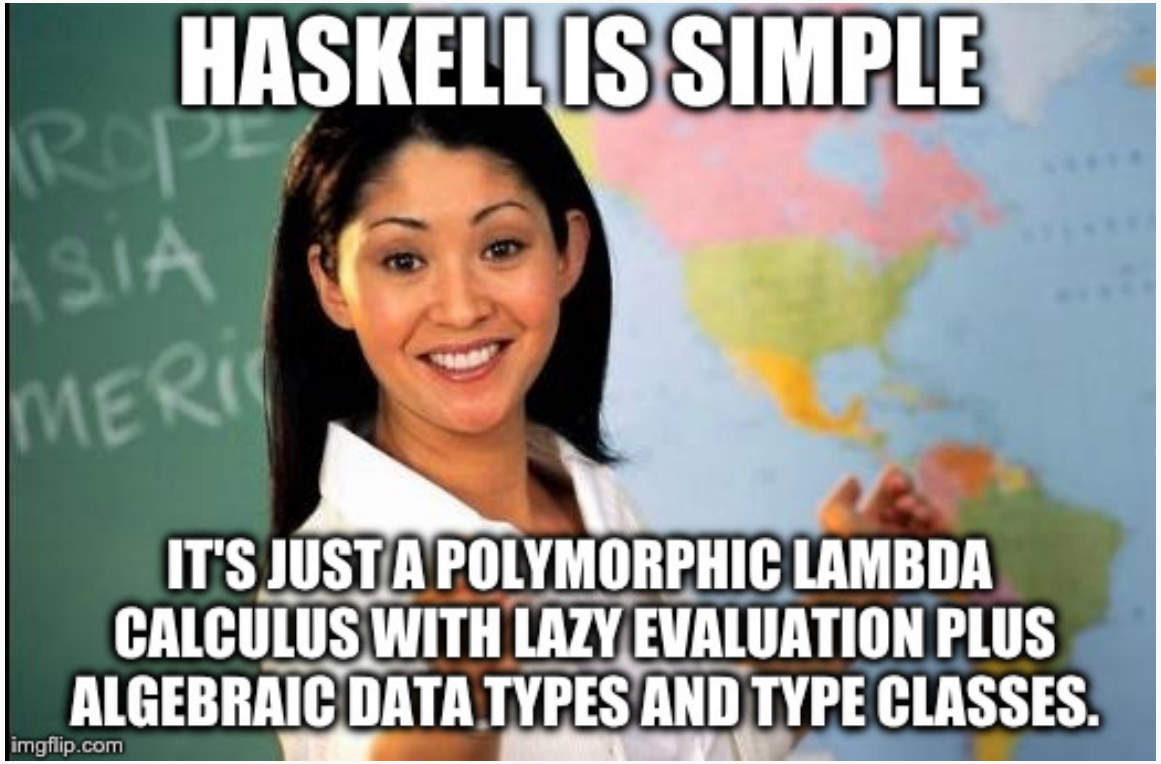
- Curs
  - Claudia Chiriță (seria 23)
  - Denisa Diaconescu (seriile 24,25)
- Laborator
  - Andrei Burdușa
  - Horațiu Cheval
  - Ioan Ionescu
  - Alina Georgescu
  - Alex Oltean
  - George-Mihai Radu
  - Stelian Stoian

Suport curs

- Suporturi de curs și laborator:
  https://tinyurl.com/PF2024-Drive
- Canal Teams:
  https://tinyurl.com/PF2024-MTeams

Prezență

Prezența la curs sau la laboratoare nu este obligatorie,
dar extrem de încurajată!



Evaluare

Notare

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

Condiție de promovabilitate

- cel puțin 5 > 4.99

Parțial

- valorează 3 puncte din nota finală
- durează 40 min
- în săptămâna 9 în cadrul cursului (25-29 noiembrie)
- nu este obligatoriu și nu se poate reface
- va conține 15 întrebări grilă, asemănătoare cu cele din curs
- materiale ajutătoare: suporturile de curs și de laborator, notițe proprii
- fără aparate electronice (laptop, tabletă, telefon etc.)

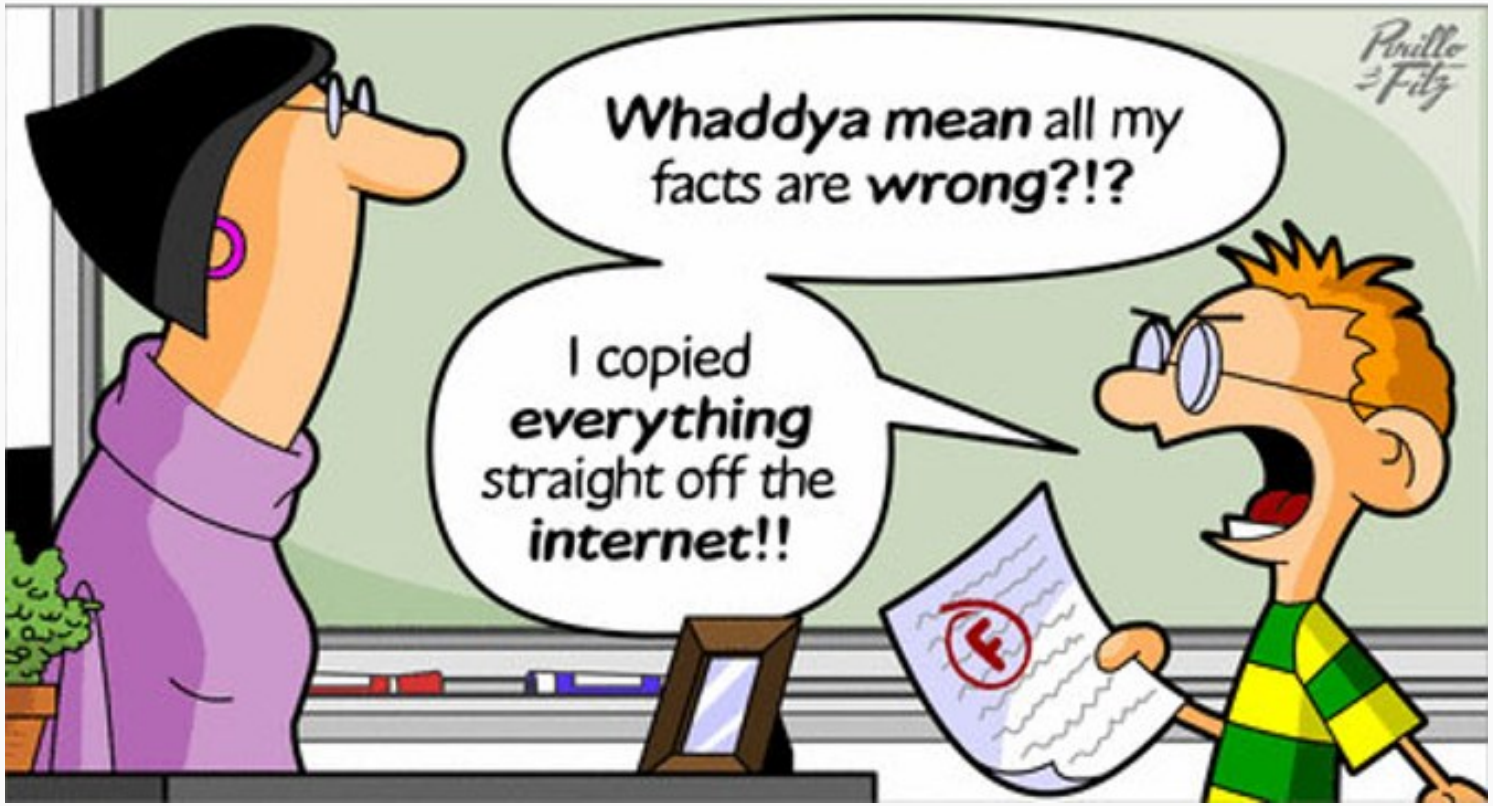
Examen final

- valorează 6 puncte din nota finală
- durează 1 oră
- în sesiune, fizic
- acoperă toată materia
- va conține exerciții asemănătoare cu cele de la laborator
- materiale ajutătoare: suporturile de curs și de laborator, notițe proprii
- se poate susține pe laptop propriu sau pe un calculator din facultate

Bonus

- maxim 1 punct pentru activitatea de la laborator
- nu se poate folosi pentru a satisface condiția de promovabilitate

Nu trișați, cereți-ne ajutorul!



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

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/

The Haskell Foundation

- https://haskell.foundation/
- Fundație independentă și non-profit ce are ca scop îmbunătățirea experienței cu limbajul Haskell
- Oferă suport pentru biblioteci, tool-uri, educație, cercetare

De ce programare funcțională?

Sondaj

Ce știți despre programarea funcțională?

https://tinyurl.com/PF2023-C01-Quiz1

Programare declarativă vs. imperativă – Ce vs. cum

Programare imperativă (Cum)

Explic mașinii, pas cu pas, algoritmic, cum să facă ceva și se întâmplă ce voiam să se întâmple ca rezultat al execuției mașinii.

- limbaje de programare procedurale
- limbaje de programare orientate pe obiecte

Programare declarativă (Ce)

Îi spun mașinii ce vreau să se întâmple și o las pe ea să se descurce cum să realizeze acest lucru :-)

- limbaje de programare logică
- limbaje de interogare a bazelor de date
- limbaje de programare funcțională

Programare funcțională

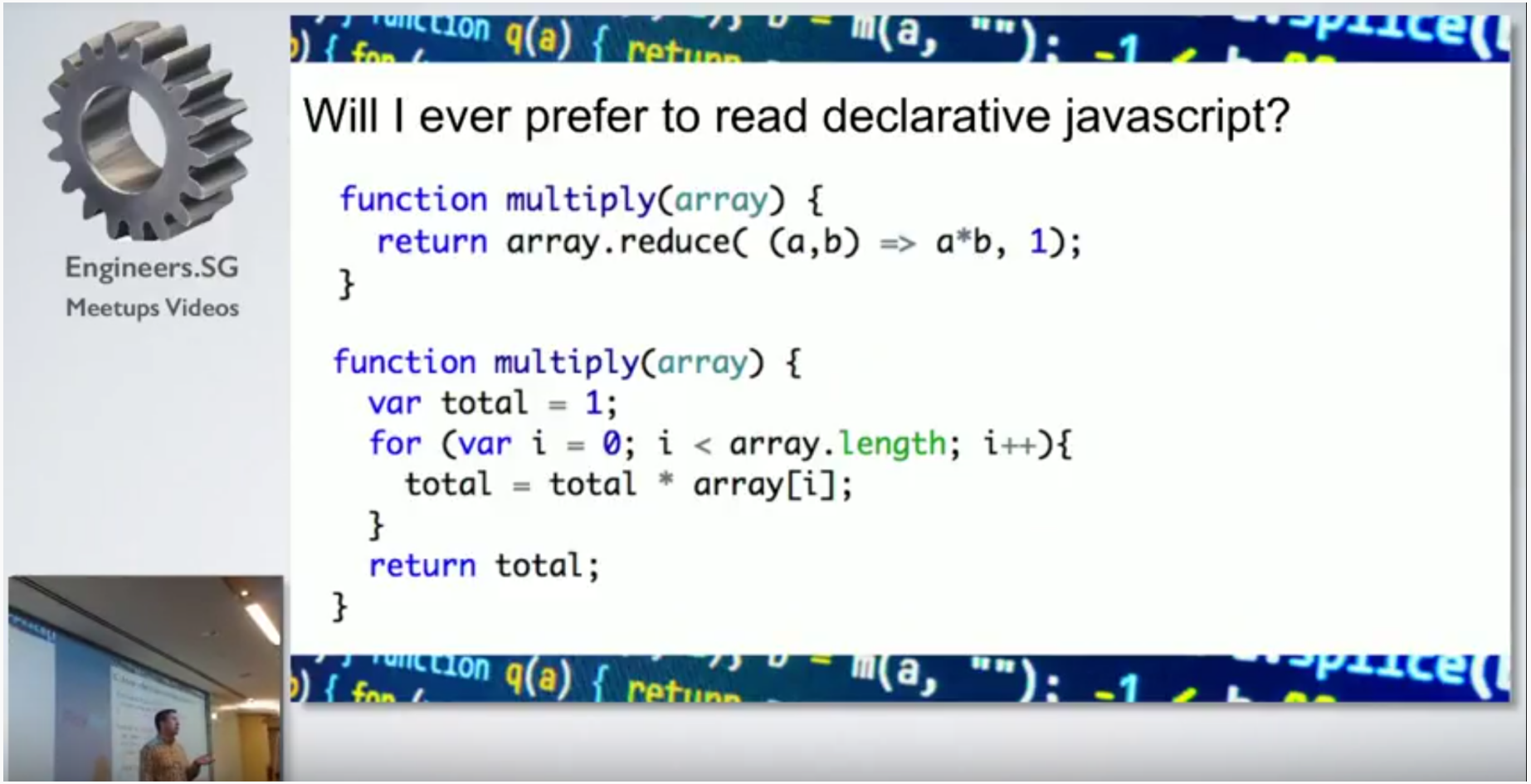
Programare funcțională în limbajul vostru preferat de programare (Java 8, C++11, Python, JavaScript, . . .)

- 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

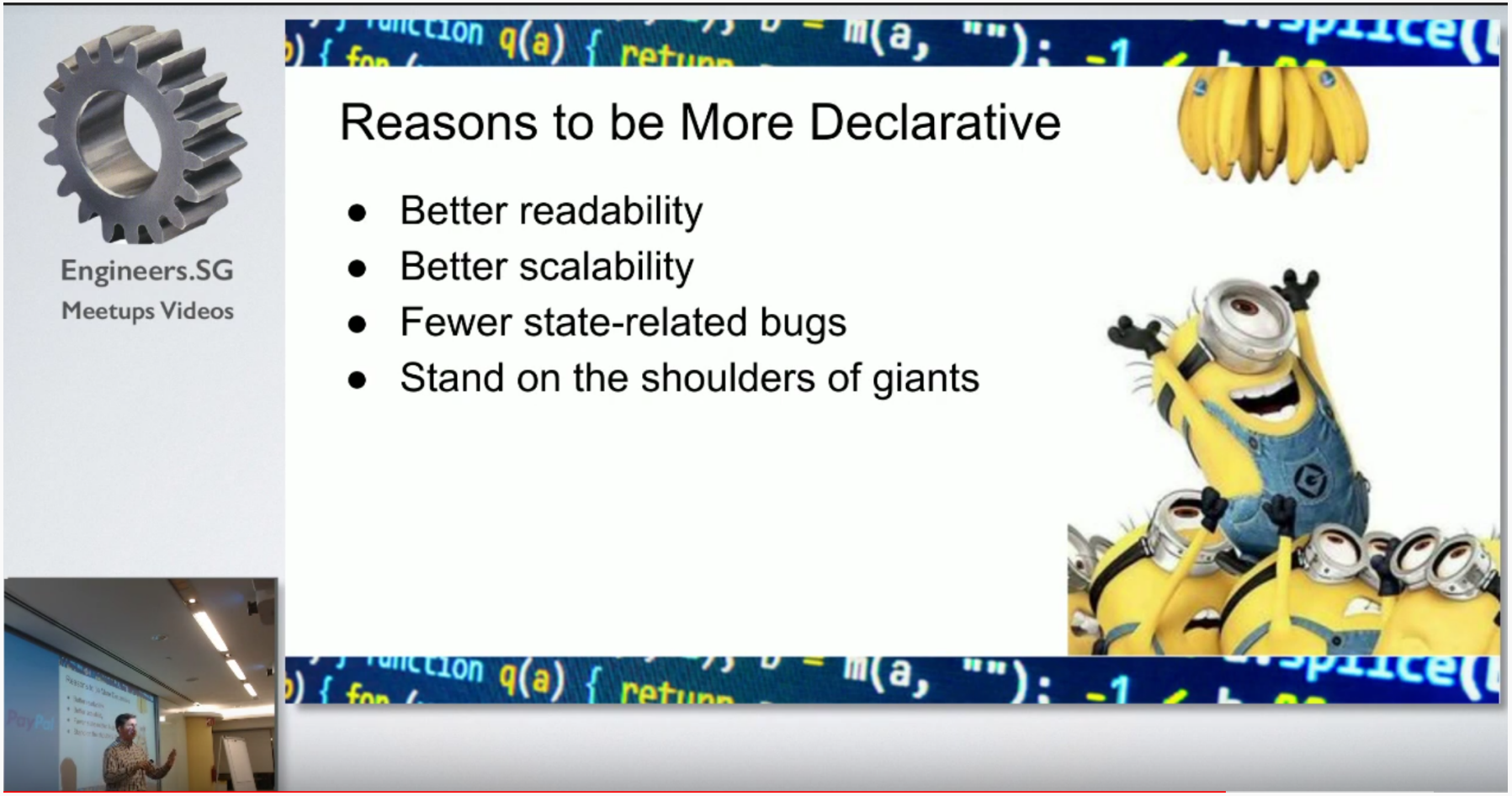
https://www.youtube.com/watch?v=M2e5sq1rnvc



Agregarea datelor dintr-o colecție (JS)

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

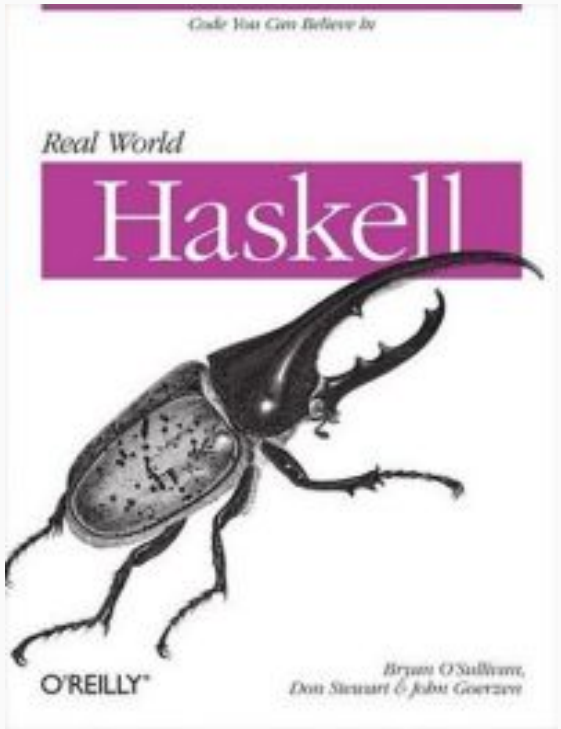
https://www.youtube.com/watch?v=M2e5sq1rnvc



De ce Haskell?

(din cartea "Real World Haskell")

The illustration on our cover is of a Hercules beetle. These beetles are among the largest in the world. They are also, in proportion to their size, the strongest animals on Earth, able to lift up to 850 times their own weight. Needless to say, we like the association with a creature that has such a high power-to-weight ratio.



Exemplu (Ciurul lui Eratostene)

https://ro.wikipedia.org/wiki/Ciurul_lui_Eratostene

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

Haskell este un limbaj funcțional pur

- Funcțiile sunt *first-class citizens*.
  - Funcțiile sunt folosite ca valori.
  - De exemplu, funcțiile pot fi transmise ca argumente și pot fi returnate de alte funcții.
- Funcțiile sunt pure.
  - Produc aceleași rezultate pentru aceleași intrări.
  - O bucată de cod nu poate corupe datele altei bucăți de cod.
  - Distincție clară între părțile pure și părțile care comunică cu mediul extern.

Haskell este un limbaj elegant

- Idei abstracte din matematică devin instrumente puternice în practică.
  - recursivitate, compunerea de funcții, functori, monade
  - folosirea lor permite scrierea de cod compact și modular
- Rigurozitate.
  - ne forțează să gândim mai mult înainte, dar ne ajută să scriem cod mai corect și mai curat
- Curbă de învățare în trepte.
  - Putem scrie programe mici destul de repede
  - Expertiza în Haskell necesită multă gândire și practică
  - Descoperirea unei lumi noi poate fi un drum distractiv și provocator `http://wiki.haskell.org/Humor`

Haskell este leneș și minimalist

- *Lazyness*: orice calcul este amânat cât de mult posibil
  - Schimbă modul de concepere al programelor
  - Permite lucrul cu colecții potențial infinite de date precum [1..]
  - Evaluarea lenișă poate fi exploatată pentru a reduce timpul de calcul fără a denatura codul

```
firstK k  = take k primes
```

- Haskell e minimalist: mai puțin cod, în mai puțin timp, și cu mai puține greșeli
  - ...rezolvând totuși problema :-)

```
multiply = foldl (*) 1
doubled = map (* 2)
```

- Oferă suport pentru paralelism și concurență.

Exemplu

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

Exemplu - generalizare

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

Haskell în industrie

Programarea funcțională este din ce în ce mai utilizată în companii.

Haskell e folosit la Meta, Google, Microsoft, IOHK etc.

Alte proiecte mari scrise în Haskell:

- https://typeable.io/
- https://serokell.io/
- https://xmonad.org/
- https://jaspervdj.be/hakyll/

Linkuri suplimentare:

- Typeable Blog Post: 7 Useful Tools Written in Haskell
- Serokell Blog Post: Best Haskell open source projects
- Serokell Blog Post: Why Fintech Companies Use Haskell
- Wasp Blog Post: How to get started with Haskell in 2022

Haskell în industrie

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

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 literă mare
- Haskell este sensibil la majuscule (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/PF2023-C01-Quiz2

Pe săptămâna viitoare!