PROGRAMARE FUNCȚIONALĂ

Facultatea de Matematică și Informatică
Universitatea din București
Informatică ID, anul II

Cuprins

1 Administrativ

Instructor: Natalia Ozunu, natalia.ozunu@g.unibuc.ro

Obiectivele cursului: Cursul are scopul de a pune baza programării în Haskell.

Cursurile vor acoperi următoarea materie:

- elemente de bază de limbaj;
- tipuri de date, funcții;
- liste;
- funcții de nivel înalt;
- tipuri de date algebrice și clase de tipuri.

Notare: Examenul va consta într-un set de întrebări grilă din toată materia și exerciții de programare asemănătoare cu cele făcute în timpul semestrului.

Examinarea se va face pe calculatoarele din facultate.

La examen este permis ca material ajutător doar suportul de curs sub format pdf care este oferit de către profesor prin intermediul platformei Teams. Nu este permisă folosirea altor materiale online/fizice. Mai multe detalii vor fi oferite înainte de examinare.

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/`

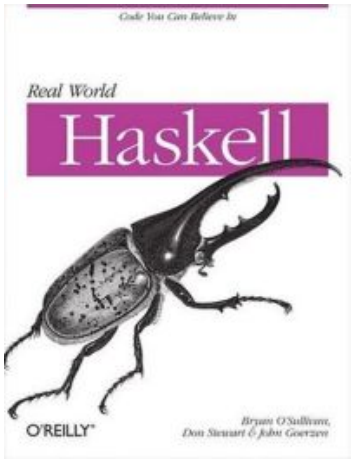
2 De ce programare funcțională?

Putem implementa programare funcțională și în alte limbaje de programare:

- Java 8, C++11, Python, JavaScript, ...
- Funcții anonime (λ-abstracții)

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

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

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

2.1 Proprietăți ale acestui limbaj:

- **Haskell este un limbaj funcțional pur**
  - **Funcțiile sunt valori.**
    - În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții.
  - **Funcțiile sunt pure:** aceleași rezultate pentru aceleași intrări.
    - O bucată de cod nu poate corupe datele altei bucăți de cod.
    - Există o distincție clară între părțile pure și cele care comunică cu mediul extern.
- **Oferă suport pentru paralelism și concurență.**
- **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 e leneș** *(lazy)***:** orice calcul e amânat cât de mult posibil
  - Schimbă modul de concepere al programelor
  - Permite lucrul cu colecții potențial infinite de date precum [1..]
  - Evaluarea leneșă 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 defecte ... rezolvând totuși problema.

```
numbers = [1,2,3,4,5]
total = foldl (*) 1 numbers
doubled = map (* 2) numbers
```

Exemplu

Quicksort în Haskell

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

Quicksort în C++

```
void QuickSort(int v[], int st, int dr)
{
        if(st < dr)
        {
                int m = (st + dr) / 2;
                int aux = v[st];
                v[st] = v[m];
                v[m] = aux;
                int i = st , j = dr, d = 0;
                while(i < j)
                {
                if(v[i] > v[j])
                        {
                                aux = v[i];
```

```
                    v[i] = v[j];
                    v[j] = aux;
                    d = 1 - d;
                }
                i += d;
                j -= 1 - d;
            }
            QuickSort(v, st , i - 1);
            QuickSort(v, i + 1 , dr);
        }
}
```

2.2 Haskell in industrie

Programarea funcțională e din ce în ce mai importantă în industrie

- Haskell e folosit în proiecte de Meta, Google, Microsoft etc.
- Alte proiecte mari scrise in Haskell:
  - `https://typeable.io/`
  - `https://serokell.io/`
  - `https://xmonad.org/`
  - `https://jaspervdj.be/hakyll/`

Figure 1: 10 motive pentru a folosi Haskell

Mai multe detalii la:

- Haskell in industrie - Wiki
- Serokell Blog Post: Best Haskell open source projects
- Typeable Blog Post: 7 Useful Tools Written in Haskell
- Serokell Blog Post: Why Fintech Companies Use Haskell
- Wasp Blog Post: How to get started with Haskell in 2022 (the straightforward way)

3 Elemente de bază

3.1 Instalare Haskell

Pentru a instala Haskell puteți folosi GHCup, urmărind instrucțiunile de aici [Instalare]

Deschideți un terminal si introduceți comanda `ghci` (în Windows este posibil să aveți instalat WinGHCi). După câteva informații despre versiunea instalată va apare
ghci> sau **Prelude**> (în funcție de versiunea instalată)

`ghci` este librăria standard: `https://hackage.haskell.org/package/base-4.17.0.0/docs/ghci.html`

În interpretor puteți:

- să introduceți expresii, care vor fi evaluate atunci când este posibil:

```
ghci> 2+3
 5
ghci> False || True
 True
ghci> x
 <interactive>:10:1: error: Variable not in scope: x
ghci> x=3
ghci> x
 3
ghci> y=x+1
ghci> y
 4
ghci> head [1,2,3]
 1
ghci> head "abcd"
 'a'
ghci> tail "abcd"
 'bcd'
```

Funcțiile `head` și `tail` aparțin modulului standard **ghci**.

- să introduceți comenzi, orice comandă fiind precedată de ":"

  :? - este comanda *help*

  :q - este comanda *quit*

  :cd - este comanda *change directory*

  :t - este comanda *type*

  :l - comanda de încărcare (*load*) pt fișiere

  :r - comanda de reîncărcare (*reload*) a fișierelor încărcate

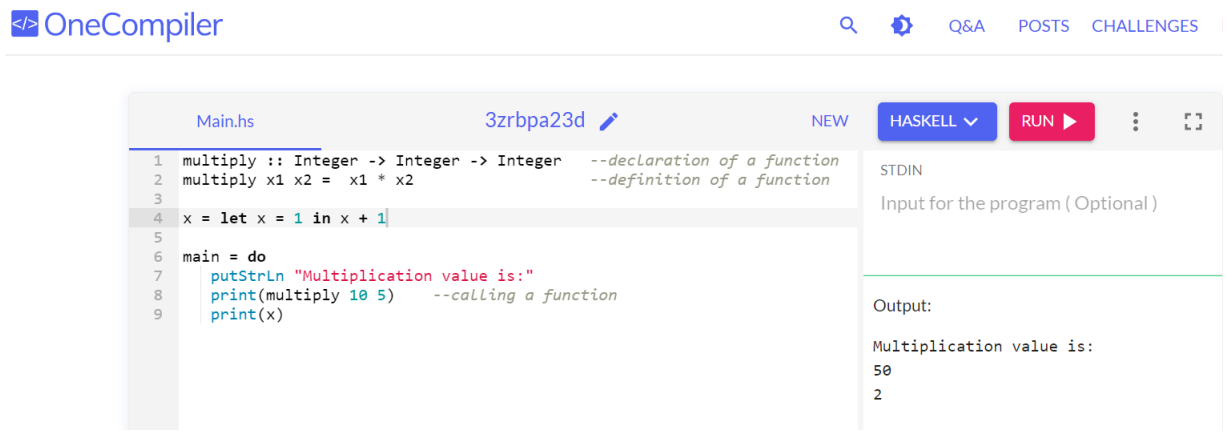
```
ghci> :t True
True :: Bool

ghci> :t head
head :: [a] -> a

ghci> :t tail
tail :: [a] -> [a]
```

Citiți mai mult despre GHCi:
`https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html`

3.2 Interpretor online Haskell

Puteți folosi orice interpretor online de Haskell pentru a lucra probleme simple.

- `https://replit.com/~`
- `https://www.jdoodle.com/execute-haskell-online/`
- `https://onecompiler.com/haskell`



3.3 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 fisierul `"elem_baza.hs"` conține

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

Ce valoare are x?

```
Prelude> :l elem_baza.hs
[1 of 1] Compiling Main             ( elem_baza.hs, interpreted )

elem_baza.hs:2:1: error:
    Multiple declarations of 'x'
    Declared at: elem_baza.hs:1:1
                 elem_baza.hs:2:1
  |
2 | x=2
  | ^
Failed, no modules loaded.
```

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 crează scop local (permite declararea de variabile și funcții locale).

Presupunem că fișierul `testlet.hs` conține

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

```
Prelude> :l testlet.hs
[1 of 1] Compiling Main             ( testlet.hs, interpreted )
Ok, one module loaded.
*Main> z
3
*Main> x
1
*Main>
```

Alte exemple:

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

Ce valoare are x?

-- *x=12*

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

Ce valoare are x?

```
-- x=5
```

Clauza `... where ...` creaza scop local (permite declararea de variabile și funcții locale).

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

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
```

Să definim funcția `maxim`

```
maxim :: Integer -> Integer -> Integer
maxim x y = if (x > y) then x  else y
```

Varianta cu indentare este:

```
maxim :: Integer -> Integer -> Integer
maxim x y =
    if (x > y)
       then x
       else y
```

Dorim acum să scriem o funcție care calculează maximul a trei numere. Evident, o varianta este

```
maxim3 x y z = maxim x (maxim y z)
```

Scrieți funcția `maxim3` fără a folosi `maxim`, utilizând direct `if` și scrierea indentată.

Putem scrie funcția `maxim3` folosind expresia `let...in` astfel

```
maxim3 x y z = let u = (maxim x y) in (maxim u z)
```

Atenție! Expresia `let...in` crează scop local.

Varianta cu indentare este

```
maxim3 x y z =
    let
        u = maxim x y
    in
        maxim u z
```

Scrieți o funcție `maxim4` folosind varianta cu `let..in` și indentare.

Citiți mai multe despre indentare: `https://en.wikibooks.org/wiki/Haskell/Indentation`

3.4 Tipuri de date. Sistemul tipurilor

Sistemul tipurilor de date are urmatoarele proprietăți:

- `tare` – garantează absența anumitor erori
- `static` – tipul fiecărei valori este calculat la compilare
- `dedus automat` – compilatorul deduce automat tipul fiecărei expresii

```
ghci> :t [('a',1,"abc")]
[('a',1,"abc")] :: Num b => [(Char, b, [Char])]
```

Exemple de tipuri de date:

- `Tipurile de bază`: Int, Integer, Float, Double, Bool, Char, String
- `Tipuri compuse`: tupluri (secvențe de tipuri deja existente) si liste

```
ghci> :t ('a', True)
('a', True) :: (Char, Bool)

ghci> :t (1 :: Int, 'a', "ab")
(1 :: Int, 'a', "ab") :: (Int, Char, [Char])

ghci> fst (1,'a') -- numai pentru perechi
ghci> snd (1,'a')

ghci> :t ["ana", "ion"]
["ana", "ion"] :: [[Char]]

ghci>:t [True, False, True]
[True, False, True] :: [Bool]
```

- `Tipuri noi definite de utilizator:`

```
data RGB = Rosu | Verde | Albastru
data Point a = Pt a a     -- tip parametrizat
-- a este variabila de tip
```

- `Integer:` 4, 0, -5
- `Float:` 3.14
- `Char:` 'a','A', '\n'
- `Bool:` True, False
- `String:` "prog\ndec"

```
type String = [Char] -- sinonim pentru tip
```

Tipuri. Clase de tipuri. Variabile de tip - Va fi discutat mai tarziu..

Ce răspuns primim în GHCi dacă introducem comanda?

```
ghci> :t 1
```

Răspunsul primit este:

```
1 :: Num a => a
```

Semnificația este următoarea:

- `a` este un *parametru de tip*
- `Num` este o clasă de tipuri
- `1` este o valoare de tipul `a` din clasa `Num`

```
ghci> :t [1,2,3]
[1,2,3] :: Num t => [t]
```

3.5 Funcții în Haskell. Terminologie

În Haskell funcțiile sunt formate din mai multe componente:

- prototipul funcției: format din numele funcției, tipurile de date ale parametrilor și tipul de date rezultat.
- definiția funcției: formată din numele funcției, parametri formali și corpul funcției (expresia care dă rezultatul)
- aplicarea funcției: numele funcției și parametri actuali

Exemplu: funcție cu un argument

- **Prototipul funcției** double :: **Integer** −> **Integer**
  - numele funcției: `double`
  - signatura funcției: `Integer -> Integer`
- **Definiția funcției** double **elem** = **elem** + **elem**
  - numele funcției: `double`
  - parametrul formal: `elem`
  - corpul funcției: `elem + elem`
- **Aplicarea funcției** double 5
  - numele funcției: `double`
  - parametrul actual (argumentul): `5`

Exemplu: funcție cu două argumente

- **Prototipul funcției** add:: **Integer** −> **Integer** −> **Integer**
  - numele funcției: `add`
  - signatura funcției: `Integer -> Integer -> Integer`
- **Definiția funcției** add elem1 elem2 = elem1 + elem2
  - numele funcției: `add`
  - parametrii formali: `elem1, elem2`
  - corpul funcției: `elem1 + elem2`
- **Aplicarea funcției** add 3 7
  - numele funcției: `add`
  - argumentele: 3, 7

Exemplu: funcție cu un argument de tip tuplu

```
dist :: (Integer, Integer) -> Integer
dist (x,y) = abs(x-y)
-- apel: dist(2,4)
```

Tipuri de funcții

Putem da tipul unei funcții explicit, ca în exemplele de mai sus, sau putem omite linia respectivă caz în care interpretorul găsește automat signatura funcției, în funcție de implementare, funcțiile și operațiile folosite de noi. Aceasta poate să nu fie exactă cum am fi dat-o noi.

În Haskell există o comandă `:t` care afisează tipul unei funcții.

Exemple:

```
ghci> :t abs
abs :: Num a => a -> a
ghci> :t div
div :: Integral a => a -> a -> a

ghci> :t (:)
(:) :: a -> [a] -> [a]
ghci> :t (++)
(++) :: [a] -> [a] -> [a]

ghci> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

Pentru a defini o funcție putem folosi mai multe metode. Putem trata cazurile diferite folosind `if`, ecuații (șabloane), cazuri etc.

De exemplu, pentru a defini funcția `fact` care calculează factorialul unui număr dat, putem folosi următoarele variante:

```
fact :: Integer -> Integer
```

- `Definiție folosind if`

```
fact n = if n == 0 then 1 else n * fact(n-1)
```

- `Definiție folosind ecuații (șabloane)`

```
fact 0 = 1
fact n = n * fact(n-1)
```

- `Definiție folosind cazuri`

```
fact n
  | n == 0     = 1
  | otherwise = n * fact(n-1)
```

Funcția *semn* o putem defini astfel

$$semn\ n = \begin{cases} -1, & \textbf{dacă n} < \textbf{0} \\ 0, & \textbf{dacă n} = \textbf{0} \\ 1, & \textbf{altfel} \end{cases}$$

În Haskell, condițiile devin `gărzi`:

```
semn n
  | n < 0  = -1
  | n == 0 =  0
  | otherwise = 1
```

Sau versiunea cu șabloane:

```
semn :: Integer -> Integer
semn 0 = 0
semn x
  | x > 0     = 1
  | otherwise = -1
```

În scrierea cu șabloane

- variabilele și valorile din partea stângă a semnului = sunt *șabloane*;
- când funcția este apelată se încearcă potrivirea parametrilor actuali cu șabloanele, ecuațiile fiind încercate *în ordinea scrierii*;
- în definiția factorialului, 0 și n sunt șabloane: 0 se va potrivi numai cu el însuși, iar n se va potrivi cu orice valoare de tip `Integer`.

3.6 Fișiere sursă

Fișierele sursă sunt fișiere cu extensia `.hs`, pe care le puteți edita cu un editor la alegerea voastră. Deschideți fişierul `lab1.hs` care conține următorul cod:

```
myInt =
   55555555555555555555555555555555555555555555555555555555555
double :: Integer -> Integer
double x = x+x
```

Fără a încărca fișierul, încercați să calculați double myInt:

ghci> double myInt

Observați mesajele de eroare. Acum încărcați fișierul folosind comanda `load (:l)` și încercați din nou să calculați `double myInt`:

```
ghci> :l lab1.hs
[1 of 1] Compiling Main             ( lab1.hs, interpreted )
Ok, 1 module loaded.
*Main> double myInt

*Main> double 2000
```

Modificați fișierul adăugând o funcție `triple`. Dacă fișierul este deja încărcat, puteți să îl reîncărcați folosind comanda `reload (:r)`.

Puteți reveni în `ghci` folosind `:m -`

```
ghci> :l lab1.hs
[1 of 1] Compiling Main             ( lab1.hs, interpreted )
Ok, 1 module loaded.
*Main> :m - Main
ghci>
```

3.7 Operatori

Operatorii sunt funcții cu două argumente, pot fi apelați folosind notația infix, pot fi definiți folosind numai "simboluri" (ex: *!*) - caz în care în definiția tipului operatorul este scris între paranteze.

- Operatori predefiniți

```
(||) :: Bool -> Bool -> Bool
(:) :: a -> [a] -> [a]
(+) :: Num a => a -> a -> a
```

- Operatori definiți de utilizator

```
(&&&) :: Bool -> Bool -> Bool  -- atentie la paranteze
True &&& b = b
False &&& _ = False
```

Putem defini și funcții ca operatori.

```
ghci> mod 5 2
1
ghci> 5 `mod` 2
1
```

Operatorii care sunt definiți în formă infix, sunt apelați în formă prefix folosind paranteze

```
2 + 3 == (+) 2 3
```

Operatorii care sunt definiți în formă prefix, sunt apelați în formă infix folosind ` ` *(backtick)*

```
mod 5 2 == 5 `mod` 2

elem :: a -> [a] -> Bool
ghci> 1 ` elem` [1,2,3]
True
```

Precedență și asociativitate:

```
ghci> 3+5*4:[6]++8-2+3:[2]==[23,6,9,2]||True==False
True
```

Precedence	Left associative	Non-associative	Right associative
9	!!		.
8			^, ^^, **
7	*, /, `**div**`, `**mod**`, `**rem**`, `**quot**`		
6	+, −		
5			:, ++
4		==, /=, <, <=, >, >=, `**elem**`, `**notElem**`	
3			&&
2			||
1	>>, >>=		
0			$, $!, `**seq**`

Asociativitate:

Operatorul - asociativ la stânga
$5 - 2 - 1 == (5 - 2) - 1$ $-- /= 5 - (2 - 1)$

Operatorul : asociativ la dreapta
$5 : 2 : [] == 5 : (2 : [])$

Operatorul ++ asociativ la dreapta

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

l1 ++ l2 ++ l3 ++ l4 ++ l5 == l1 ++ (l2 ++ (l3 ++ (l4 ++ l5)))

Pentru a seta asociativitatea unui operator vom folosi următoarea notație:

```
infix [l|r] NUMAR <operatori-separati-prin-virgula>
```

- **infix** - neasociativ
- **infixr** - asociativ la dreapta
- **infixl** - asociativ la stânga
- NUMAR - precedența (între 0 și 9)

Pentru a descoperi asociativitatea unui operator se poate folosi comanda :i în GHCI
Exemple:

```
infix  4  ==, /=, <, <=, >=, >

infixr 3  &&
(&&) :: Bool -> Bool -> Bool

infixr 2  ||
(||) :: Bool -> Bool -> Bool
```

```
infixl 9 !!
(!!) :: [a] -> Int -> a

infixl 7 `div`
div :: Integral a => a -> a -> a
```

3.8 Exerciții

Exercițiul 3.1 Alegeți varianta corectă!

1. Cum se comentează o linie în Haskell?
   a) $--$
   b) $/**/$
   c) //
   d) !
2. Ce valoare are x în x = **let** x = 3 **in** x $*$ 5?
   a) 3
   b) 15
   c) 20
   d) Eroare
3. Ce valoare are x în x = **let** x = 3, y = 4 **in** x $*$ y
   a) 3
   b) 4
   c) 12
   d) eroare
4. Ce tip are o funcție f care are două argumente, primul argument de tip Char, iar al doilea argument de tip Bool, și întoarce un rezultat de tip Bool?
   a) f : **Char** −> **Bool** −> **Bool**
   b) f :: **Bool** −> **Char** −> **Bool**
   c) f :: **Char** −> **Bool** −> **Bool**
   d) nu se poate defini
5. Ce tip are expresia [**True**, 'a', "FP"]?
   a) (**Bool**, **Char**, **Char**)
   b) eroare

c) [**Bool**, **Char**, [**Char**]]

d) [**Bool**, **Char**, **Char**]

6. Ce tip are expresia (**True**, 'a', "FP")

   a) eroare

   b) (**Bool**, **Char**, **Char**)

   c) (**Bool**, **Char**, [**Char**])

   d) [**Bool**, **Char** [**Char**]]

7. Ce tip are o funcție f care are două argumente, o funcție de la Char la Bool și, respectiv, un Char, și întoarce un Bool?

   a) nu se poate defini

   b) f : (**Char** −> **Bool**) −> **Char** −> **Bool**

   c) f :: **Char** −> **Bool** −> **Char** −> **Bool**

   d) f :: (**Char** −> **Bool**) −> **Char** −> **Bool**

8. Ce valoare are f 3 în f 5 = **let** x = 3 **in** x + x?

   a) 6

   b) 5

   c) excepție (nu se potrivește niciun caz din definiția lui f)

   d) 10

9. Ce valoare are f 5 în f x = **let** x = 3 ; y = 4 **in** x + y?

   a) 9

   b) 7

   c) 5

   d) eroare

Exercițiul 3.2

Să se scrie următoarele funcții:

a) funcție cu 2 parametri care calculează suma pătratelor celor două numere;

b) funcție cu un parametru ce întoarce mesajul "par" dacă parametrul este par și "impar" altfel;

c) funcție care verifică dacă primul parametru este mai mare decât dublul celui de-al doilea parametru.

3.9 Secțiuni ("operator sections")

Aplicarea parțială a funcțiilor matematice:

Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(a, b) = c$ unde $a \in A$, $b \in B$ și $c \in C$.

Pentru $a \in A$ și $b \in B$ (arbitrare, fixate) definim

$f_a : B \to C$, $f_a(b) = c$ dacă și numai dacă $f(a, b) = c$,

$f^b : A \to C$, $f^b(a) = c$ dacă și numai dacă $f(a, b) = c$.

Funcțiile f_a și f_b se obțin prin `aplicarea parțială` a funcției f.

Matematic, secțiunile corespund aplicării parțiale a unei funcții.

Secțiunile operatorului binar `op` sunt `(op e)` și `(e op)`.

Exemple:

- secțiunile lui `++` sunt `(++ e)` și `(e ++)`

```
ghci> :t (++ " world!")
(++ " world!") :: [Char] -> [Char]
ghci> (++ " world!") "Hello" -- atentie la paranteze
"Hello world!"
ghci> :t (++ " world!") "Hello"
(++ " world!") "Hello" :: [Char]
ghci> ++ " world!" "Hello"
  error

ghci> :t ("Hello" ++)
("Hello" ++) :: [Char] -> [Char]
ghci> ("Hello" ++) " world!"
"Hello world!"
ghci> :t ("Hello" ++) " world!"
("Hello" ++) " world!" :: [Char]
```

- secțiunile lui `<->` sunt `(<-> e)` și `(e <->)`, unde

```
ghci> let x <-> y = x-y+1  -- definit de utilizator
ghci> :t (<-> 3)
(<-> 3) :: Num a => a -> a
ghci> (<-> 3) 4
2
```

- secțiunile operatorului `(:)`

```
ghci> (2:) [1,2]
[2,1,2]
ghci> (:[1,2]) 3
[3,1,2]
```

- secțiunile operatorului `(+)`

```
ghci> f = (+2)
ghci> :t f
f :: Num a => a -> a
ghci> f 7
9
ghci> :t (+2)
(+2) :: Num a => a -> a
ghci> (+2) 7
9
```

- secțiunile operatorului `(-)`

```
ghci> (subtract 1) 11
10
ghci> (11 -) 1
10
ghci> :t (subtract 1)
(subtract 1) :: Num a => a -> a
ghci> :t (11 -)
(11 -) :: Num a => a -> a

ghci> map (subtract 1) [1,2,3,4]
[0,1,2,3]
```

Atenție!

```
ghci> :t -1   -- aici " - " nu este operator de sectiune
-1 :: Num a => a
ghci> -1
-1
ghci> :t (-1)
(-1) :: Num a => a
```

- secțiunile operatorului `( ^ )`

```
ghci> (^2) 3
9
ghci> (2^) 3
8
ghci> :t (2^)
(2^) :: (Integral b, Num a) => b -> a
ghci> :t (^2)
(^2) :: Num a => a -> a
ghci> ((-3)^) 2
9
```

Exercițiul 3.3

a) Scrieți o funcție cu 2 parametri care calculează suma pătratelor celor două numere folosind secțiuni.

```
f = (^2)   -- f x = x^2
g x y = f x + f y

ghci> g 2 3
13
```

b) Scrieți o funcție cu 3 parametri care verifică dacă se poate forma un triunghi dreptunghic având lungimile laturilor date de cei 3 parametri. Rezolvați acest exercițiu folosind secțiuni.

c) Scrieți o funcție poly care are patru argumente de tip Double, a,b,c,x și calculează $a * x^2 + b * x + c$. Scrieți și signatura funcției (poly :: ceva).

d) Scrieți o funcție eeny care întoarce "eeny" pentru input par și "meeny" pentru input impar. Hint: puteti folosi funcția even (puteți căuta pe `https://hoogle.haskell.org/`).

```
eeny :: Integer -> String
eeny = undefined
```

e) Scrieți o funcție fizzbuzz care întoarce "Fizz" pentru numerele divizibile cu 3, "Buzz" pentru numerele divizibile cu 5 și "FizzBuzz" pentru numerele divizibile cu ambele. Pentru orice alt număr se întoarce șirul vid. Pentru a calcula modulo a două numere puteți folosi funcția mod. Să se scrie această funcție în 2 moduri: folosind if și folosind gărzi (condiții).

4 Recursivitate

Una dintre diferențele dintre programarea declarativă și cea imperativă este modalitatea de abordare a problemei iterării: în timp ce in programarea imperativă acesta este rezolvată prin bucle (while, for, ...), în programarea declarativă rezolvarea iterării se face prin conceptul de recursie.
Un avantaj al recursiei față de bucle este acela că usurează sarcina de scriere și verificare a corectitudinii programelor prin raționamente de tip inductiv: construiește rezultatul pe baza rezultatelor unor subprobleme mai simple (aceeași problemă, dar pe o dimensiune mai mică a datelor).
Un foarte simplu exemplu de recursie este acela al calculării unui element de index dat din secvența numerelor Fibonacci, definită recursiv de:

$$F_n = \begin{cases} n & \text{dacă } n \in \{0, 1\} \\ F_{n-1} + F_{n-2} & \text{dacă } n > 1 \end{cases}$$

Putem transcrie această definiție direct in Haskell:

```
fibonacciCazuri :: Integer -> Integer
fibonacciCazuri n
    | n < 2      = n
    | otherwise = fibonacciCazuri (n - 1) + fibonacciCazuri (n -
        2)
```

Alternativ, putem folosi o definiție în stil ecuațional (cu șabloane):

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

Fibonacci liniar O problemă cu definiția de mai sus este aceea că este timpul ei de execuție este exponențial. Motivul este acela că rezultatul este compus din rezultatele a 2 subprobleme de mărime aproximativ egală cu cea inițială.

Dar, deoarece recursia depinde doar de precedentele 2 valori, o putem simplifica cu ajutorul unei funcții care calculează recursiv perechea (F_{n-1}, F_n).

Exercițiul 4.1

Completați definiția funcției fibonacciPereche

Observație 1. Folosiți principiul de inducție: ne bazăm pe faptul ca fibonacciPereche (n−1) va calcula perechea (F_{n-2}, F_{n-1}) și o folosim pe aceasta pentru a calcula perechea (F_{n-1}, F_n).

Observație 2. Recursia este liniară *doar dacă* expresia care reprezintă apelul recursiv apare o singură dată. Folosiți **let**, **case** sau **where** pentru a vă asigura de acest lucru.

```
fibonacciLiniar :: Integer -> Integer
fibonacciLiniar 0 = 0
fibonacciLiniar n = snd (fibonacciPereche n)
  where
    fibonacciPereche :: Integer -> (Integer, Integer)
    fibonacciPereche 1 = (0, 1)
    fibonacciPereche n = undefined
```

Exercițiul 4.2

Numerele tribonacci sunt definite de ecuația

$$T_n = \begin{cases} 1 & \text{dacă } n = 1 \\ 1 & \text{dacă } n = 2 \\ 2 & \text{dacă } n = 3 \\ T_{n-1} + T_{n-2} + T_{n-3} & \text{dacă } n > 3 \end{cases}$$

Să se implementeze functia tribonacci atât cu cazuri cât și ecuațional.

```
tribonacci :: Integer -> Integer
tribonacci = undefined
```

Exercițiul 4.3

Să se scrie o funcție care calculează coeficienții binomiali, folosind recursivitate. Aceștia sunt determinați folosind urmatoarele ecuații.

B(n,k) = B(n-1,k) + B(n-1,k-1)
B(n,0) = 1
B(0,k) = 0

```
binomial :: Integer -> Integer -> Integer
binomial = undefined
```

5 Funcții anonime (Lambda Expresii)

În Haskell funcțiile pot avea ca argument alte funcții. O modalitate de a transmite ca parametru funcții este utilizarea lambda expresiilor (funcții anonime).

Sintaxa seamană cu sintaxa lambda expresiilor din logică.

```
\ x1 x2 ..xn -> expresie
```

```
ghci> (\x -> x + 1) 3
4
ghci> inc = \x -> x + 1
ghci> inc 3
4
ghci> :t inc
inc :: Num a => a -> a

ghci> add = \x y -> x + y
ghci> add 4 6
10
ghci> :t add
add :: Num a => a -> a -> a

ghci> aplic = \f x -> f x
ghci> :t aplic
aplic :: (t1 -> t2) -> t1 -> t2

ghci> :t aplic add
aplic add :: Num t1 => t1 -> t1 -> t1
ghci> aplic add 2 3
5
ghci> :t aplic inc
aplic inc :: Num t2 => t2 -> t2
ghci> aplic inc 3
4

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

6 Compunerea funcțiilor — operatorul .

Matematic

Date fiind $f : A \to B$ și $g : B \to C$, compunerea lor, notată $g \circ f : A \to C$, este dată de formula

$$(g \circ f)(x) = g(f(x))$$

În Haskell.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(g . f) x = g (f x)
```

Exemplu

```
g :: Int -> Bool
g x = x > 0
f :: Int -> Int
f x = x + 1

ghci> :t (g.f)
(g.f) :: Int -> Bool

ghci> (g.f) 0
True
ghci> (g.f.f.f) (-2)
True
```

Exercițiul 6.1

Definiți 2 funcții care să calculeze dublul și triplul unui număr și găsiți toate modalitățile de compunere a acestor funcții.

```
h,t :: Int -> Int
```

7 Liste

Orice listă poate fi scrisă folosind doar constructorul `(:)` și lista vidă `[]`.

- $[1,2,3] == 1 : (2 : (3 : [])) == 1 : 2 : 3 : []$
- "abcd" == ['a', 'b', 'c', 'd'] == 'a' : ('b' : ('c' : ('d' : [])))
  $==' a' :' b' :' c' :' d' : []$

Definiție recursivă. O **listă** este

- `vidă`, notată `[]`; sau
- `compusă`, notată `x:xs`, dintr-un un element `x` numit **capul listei** *(head)* și o listă `xs` numită **coada listei** *(tail)*.

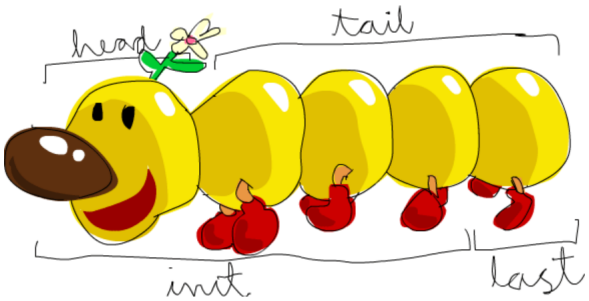
Se pot defini `intervale și progresii artimetice`, dând regula progresiei între primii doi termeni din listă.

```
interval = ['c'..'e']          -- ['c','d','e']
progresie = [20,17..1]         -- [20,17,14,11,8,5,2]
progresie' = [2.0,2.5..4.0]    -- [2.0,2.5,3.0,3.5,4.0]
```

`Operații`

- !! - folosit pentru a găsi elementul de pe o anumită poziție din listă.
- ++ - concatenează două liste.
- : - concatenează un element la o listă.
- `head` - determină primul element dintr-o listă
- `tail` - determină lista fără primul element.
- `init` - determină lista fără ultimul element.
- `last` - determină ultimul element dintr-o listă.
- `length` - determină lungimea unei liste.
- `null` - determină dacă o listă este vidă.
- `take n l` - extrage primele n elemente dintr-o listă l.
- `drop n l` - șterge primele n elemente dintr-o listă l.
- `maximum/minimum` - retunează cel mai mare/mic element dintr-o listă.
- `sum/product` - retunează suma/produsul elementelor dintr-o listă.
- `elem e l` - determină dacă elementul e aparține listei l.

```
ghci> [1,2,3] !! 2
3
ghci> "abcd" !! 0
'a'
ghci> [1,2] ++ [3]
[1,2,3]
ghci> import Data.List

ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
ghci> 'P':"rogramareFunctionala"
"ProgramareFunctionala"
ghci> "P"++"rogramareFunctionala"
"ProgramareFunctionala"

ghci> b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
ghci> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b !! 2
[1,2,2,3,4]
ghci> b !! 2 !! 3
3

ghci> head "ProgramareFunctionala"
'P'
ghci> tail "ProgramareFunctionala"
"rogramareFunctionala"

ghci> init "ProgramareFunctionala"
"ProgramareFunctional"
ghci> last "ProgramareFunctionala"
'a'
```



```
ghci> head []
*** Exception: ghci.head: empty list
ghci> length "ProgramareFunctionala"
21

ghci> null [1,2,3]
False
ghci> null []
True

ghci> reverse "ProgramareFunctionala"
"alanoitcnuFeramargorP"

ghci> take 4 "ProgramareFunctionala"
"Prog"
ghci> drop 10 "ProgramareFunctionala"
"Functionala "

ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
ghci> sum [8,4,2,1,4,5]
24
ghci> product [8,4,2,1,4,5]
1280
ghci> elem 8 [8,4,2,1,4,5]
True
ghci> elem 9 [8,4,2,1,4,5]
False
ghci> 5 `elem` [8,4,2,1,4,5]
True
ghci> 6 `elem` [8,4,2,1,4,5]
False
```

Șabloane (patterns) pentru liste

Listele sunt construite folosind constructorii (:) și []

```
[1,2,3] == 1:[2,3]   -- == 1:2:[3] == 1:2:3:[]
```

Observați:

```
ghci> let x:y = [1,2,3]
ghci> x
1
ghci> y
[2,3]
```

Ce s-a întâmplat?

- `x:y` este un șablon pentru liste

- potrivirea dintre `x:y` și `[1,2,3]` a avut ca efect:
  - "deconstrucția" valorii [1,2,3] în 1:[2,3]
  - legarea lui x la 1 și a lui y la [2,3]

Definiții folosind șabloane

Atenție! `șabloanele sunt definite folosind constructori.` De exemplu, operația de concatenare pe liste este

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

Însă, [x] ++ [1] = [2,1] nu va avea ca efect legarea lui x la 2.

Încercând să evaluăm x vom obține un mesaj de eroare:

```
ghci> [x] ++ [1] = [2,1]
ghci> x
```

error: Variable not in scope: x

În Haskell șabloanele sunt `liniare`, adică o variabilă apare cel mult odată. Șabloane în care o variabilă apare de mai multe ori provoacă mesaje de eroare:

```
ghci> x:y:[1] = [2,2,1]
ghci> x
2
ghci> y
2

ghci> x:x:[1] = [2,2,1]
```

error: Multiple declarations of 'x'

```
ttail (x:x:t) = t

foo x x = x^2
```

error: Conflicting definitions for x

O soluție este folosirea gărzilor:

```
ttail (x:y:t)
 | (x==y) = t
 | otherwise = ...

foo x y
 | (x == y) = x^2
 | otherwise = ...
```

7.1 Recursivitate pe Liste

Recursivitatea pe liste se bazeaza pe definiția inductivă a listelor, prezentată mai sus. Funcțiile care parcurg recursiv listele vor folosi șabloanele pentru listă:

- lista vidă - [ ]
- listă compusă - $x : xs$

Exemplu: Dată fiind o listă de numere întregi, să se scrie o funcție semiPareRec care elimină numerele impare și le înjumătățește pe cele pare.

```
-- semiPareRec [0,2,1,7,8,56,17,18] == [0,1,4,28,9]
```

Prima implementare propusă este realizabilă in orice limbaj, folosind testul **null**, și ,,destructorii" **head** și **tail**.

```
semiPareRecDestr :: [Int] -> [Int]
semiPareRecDestr l
  | null l     = l
  | even h     = h `div` 2 : t'
  | otherwise = t'
  where
    h = head l
    t = tail l
    t' = semiPareRecDestr t
```

A doua implementare (preferată) folosește șabloane peste constructorul de listă : pentru a descompune lista:

```
semiPareRec :: [Int] -> [Int]
semiPareRec [] = []
semiPareRec (h:t)
  | even h     = h `div` 2 : t'
  | otherwise = t'
      where t' = semiPareRec t
```

Exemplu: Dată fiind o listă să se determine lista inversată.

```
reverse2 [ ] = [ ]
reverse2 (x:xs) = (reverse2 xs) ++ [x]
  -- x:xs se potriveste cu liste nevide
```

Exercițiul 7.1 Să se scrie următoarele funcții folosind recursivitate:

a) myreplicate - pentru un întreg n și o valoare v întoarce lista de lungime n ce are doar elemente egale cu v. Să se scrie și prototipul funcției.

b) sumImp - pentru o listă de numere întregi, calculează suma valorilor impare. Să se scrie și prototipul funcției.

c) totalLen - pentru o listă de șiruri de caractere, calculează suma lungimilor șirurilor care încep cu caracterul 'A'.

d) Să se scrie o funcție 'nrVocale' care pentru o listă de șiruri de caractere, calculează numărul total de vocale ce apar în cuvintele palindrom. Pentru a verifica dacă un șir e palindrom, puteți folosi funcția 'reverse', iar pentru a căuta un element într-o listă puteți folosi funcția 'elem'. Puteți defini oricâte funcții auxiliare.

Ex.: nrVocale ["sos", "civic", "palton", "desen", "aerisirea"] = 9

e) Să se scrie o funcție care primește ca parametru un număr și o listă de întregi, și adaugă elementul dat după fiecare element par din listă. Să se scrie și prototipul funcției.

Ex.: f 3 [1,2,3,4,5,6] = [1,2,3,3,4,3,5,6,3]