Lucrări de laborator la Programarea funcțională.

Notă: Fiecare lucrare se va realiza într-un script separat, care va conține cel puțin jumătate din problemele propuse la fiecare lucrare de laborator.

Numele scriptului fa fi de forma: **nume_prenume_n.hs**, unde *n* este numărul lucrării.

Laboratorul nr.1. Evaluarea expresiilor. Funcții elementare

Haskell e un limbaj de progamare functionala, spre deosebire de C/C++, Visual Basic, Java care sunt imperative. Programarea functionala poate fi vazuta ca un *stil* de programare in care principala metoda de calcul o reprezinta aplicarea de functii asupra unor argumente.

Caracteristicile limbajului Haskell sunt:

- **Programe concise**
  Deoarece e un limbaj de programare de nivel inalt; a fost scris cu aceasta trasatura in minte, avand putine cuvinte cheie si obligand la folosirea indentatiei pentru a indica structura programelor; dimensiune de 2 pana la 10 ori mai mica a programelor, in comparatie cu scrierea lor in limbajele curente(imperative).
- **Sistem de tipuri performant**
  Majoritatea limbajelor de programare moderne include un sistem de tipuri de date pentru a detecta erorile de incompatibilitate tipul adunarii unui caracter la un numar. Haskell are un sistem de tipuri ce necesita putine informatii din partea programatorului dar permite detectarea unei clase largi de erori de incompatibilitate, sistem numit “type inference” (deducerea tipurilor). Acest sistem de tipuri permite functii polimorfice si supraincarcarea functiilor.
- **Manipularea listelor**
  Una din cele mai uzuale metode de a structura si manipula date in informatica este folosirea listelor. In Haskell *lista* este conceptul de baza, impreuna cu un mecanism simplu, dar puternic, de manipulare si generare de noi liste, eliminand de cele mai multe ori necesitatea recursivitatii.
- **Functii recursive**
  In Haskell, mecanismul de baza folosit pentru obtinerea ciclurilor este dat de functiile recursive. Multe calcule au o definitie simpla, naturala, in cazul folosirii functiilor recursive.
- **Functii de ordin inalt**
  In Haskell functiile pot primi ca argumente alte functii si pot returna drept rezultat alte functii. Prin folosirea functiilor de ordin inalt se poate defini compunerea a doua functii ca o functie.
- **Monadic effects**
- **Lazy evaluation**
  Programele Haskell sunt executate folosind o tehnica numita *lazy evaluation* (evaluare intarziata) care se bazeaza pe ideea ca nici un calcul nu trebuie realizat pana nu este nevoie de el. Pe langa evitarea calculelor inutile, aceasta tehnica asigura terminarea programelor atunci cand e posibil si permite folosirea structurilor cu un numar infinit de elemente.

In cadrul laboratorului vom folosi Haskell Platform și interpretorul GHCi, disponibil la adresa https://www.haskell.org/platform/ sau https://www.haskell.org/platform/prior.html.

La pornire sistemul Hugs incarca fisierul Prelude.hs apoi asteapta expresii pentru a le evalua. Acest fisier, incarcat la initializare, defineste functii care opereaza asupra numerelor intregi precum: adunare, scadere, inmultire, impartire, ridicare la putere:

```
> 2 + 3
5

> 2 - 3
-1

> 2 * 3
6

> 7 `div` 2
3

> 2 ^ 3
8
```

Se observa ca functia div rotunjeste prin scadere pana la cel mai apropiat intreg.

Ridicarea la putere are prioritate mai mare decat inmultirea si impartirea, care la randul lor au prioritate mai mare decat adunarea si scaderea. Mai mult, ridicarea la putere e asociativa la dreapta (**2^3^4** inseamna **2^(3^4)**) pe cand celelalte 4 operatii sunt asociative la stanga (**2-4+3** inseamna **(2-4)+3**).

In plus fata de functiile pentru lucrul cu numere intregi fisierul de librarie defineste functii de lucru cu liste. In Haskell elementele unei liste sunt inchise intre paranteze patrate si separate prin virgula. Cele mai uzuale sunt:

- Selectarea primului element dintr-o lista nevida
  ```
  > head [1,2,3,4,5]
  1
  ```
- Eliminarea primului element dintr-o lista nevida
  ```
  > tail [1,2,3,4,5]
  [2,3,4,5]
  ```
- Selectarea elementului *n* dintr-o lista (pornind de la **0**)
  ```
  > [1,2,3,4,5]!!2
  3
  ```
- Selectarea primelor *n* elemente dintr-o lista
  ```
  > take 3 [1,2,3,4,5]
  [1,2,3]
  ```
- Stergerea primelor *n* elemente dintr-o lista
  ```
  > drop 3 [1,2,3,4,5]
  [4,5]
  ```
- Lungimea unei liste
  ```
  > length [1,2,3,4,5]
  5
  ```
- Suma elementelor listei
  ```
  > sum [1,2,3,4,5]
  15
  ```
- Produsul elementelor listei

```
> product [1,2,3,4,5]
120
```

- Concatenarea listelor
```
> [1,2] ++ [3,4,5]
[1,2,3,4,5]
```
- Inversarea ordinii elementelor unei liste
```
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

Aplicarea functiilor

In Haskell, aplicarea functiilor are prioritate mai mare decat a oricaror altse functii(operatori). Pentru a intelege notatia folosita in limbaj priviti tabelul de mai jos care face o paralela intre notatia matematica si cea Haskell.

In matematica	In Haskell
`f(x)`	`f x`
`f(x,y)`	`f x y`
`f(g(x))`	`f (g x)`
`f(x,g(y))`	`f x (g y)`
`f(x) g(y)`	`f x * g y`

Se observa ca parantezele sunt necesare in expresia Haskell **`f (g x)`** deoarece **`f g x`** ar fi interpretat ca apelul functiei **`f`** cu 2 parametri **`g`** si **`x`**, cand ceea ce dorim a obtine este aplicarea functiei **`f`** asupra rezultatului functiei **`g`** cu argumentul **`x`**. Observatia e valabila si pentru **`f x (g y)`**.

Haskell Scripts

Din linia de comanda a intrepretorului Hugs nu este posibila definirea unor functiilor. Functiile trebuie define in scripturi Haskell, fisiere text ce contin o secventa de definitii. Prin conventie, aceste fisiere vor avea extensia *.hs* .

Sa presupunem saq vrem sa definim doua functii **`double`** si **`quadruple`**. Pentru aceasta creem un fisier text in care scriem definitiile acestor 2 functii:

```
double x = x + x
quadruple x = double (double x)
```

si poi salvam fisierul cu numele *test.hs*.
Pentru a incarca fisierul in Hugs dam comanda:

```
> :load test.hs
```

In acest moment atat Prelude.hs cat si test.hs sunt incarcate. Putem scrie de exemplu:
> quadruple 10
40

> take (double 2) [1,2,3,4,5]
[1,2,3,4]
Adaugam inca 2 functii scriptului nostru:

```
factorial n = product [1..n]
average ns = sum ns `div` length ns
```

Observatie: functie average se mai poate scrie si

```
average ns = div (sum ns) (length ns)
```

O functie cu 2 argumente se poate scrie si infixat prin incadrarea numelui intre apostroafe intoarse(stanga tastei 1).
Pentru a folosi functiile nou adaugate trebuie sa reincarcam scriptul:

```
> :reload

> factorial 10
3628800

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

Tabelul urmator insumeaza cele mai uzuale comenzi. Precizez ca fiecare comanda poate fi folosite abreviat prin prima sa litera: **`:edit`** prin **`:e`**.

Comanda	Semnificatie
:load name	incarca scriptul name
:reload	reincarca scriptul curent
:edit name	editeaza scriptul name
:edit	editeaza scriptul curent
:type expr	afiseaza tipul lui expr
:?	afiseaza toate comenzile
:quit	inchide Hugs

Conventii de nume

La definirea unei noi functii, numele functiei si a argumentelor functiei trebuie sa inceapa cu o litera mica si poate fi urmat de zero sau mai multe litere(mici sau mari), cifre, underscor sau apostrof . Nume valide sunt:

```
myFunc    fun1      arg_2     x'
```

Nu pot fi folosite ca literali cuvintele cheie:

```
case  class     data      default   deriving  do    else
if    import    in    infix     infixl    infixr    instance
let   module    newtype   of    then      type      where
```

Prin conventie argumentele de tip lista au adaugat sufixul *s* la nume pentru a sublinia ca pot contine mai multe valori. De exemplu o lista de numere poate fi notata **`ns`**, una de valori arbitrare **`xs`**, iar o lista de liste de caractere **`css`**.

Comentariile

Un script Haskell poate contine 2 tipuri de comentarii:

- Pe o singura linie, din momentul aparitiei pana la sfarsitul acesteaia – sunt marcate prin '--'
- Pe mai multe linii – incep prin "{-" si se termina prin "-}"

Exercitii

1. Cum adaugati 4 la sfarsitul listei [1,2,3] ?

2. Gasiti cel putin 2 definitii echivalente pentru functia de librarie *last* (returneaza ultimul element al unei liste nevide), folosind functiile explicate in acest laborator.
3. Gasiti cel putin 2 definitii echivalente pentru functia de librarie *init*(sterge ultimul element dintr-o lista nevida), folosind functiile explicate in acest laborator.
4. Scrieti o functie *fib n* care returneaza al n-lea element din sirul lui fibonacci.
   Scrieti o functie *fibonacci n* care returneaza lista cu primele *n* elemente din sirul lui fibonacci (folosind, eventual, functia *fib*)

Resurse:

www.haskell.org – pagina limbajului
www.haskell.org/tutorial - A Gentle Introduction to Haskell
Alte resurse disponibile în Materialele de curs din Microsoft Teams

Laboratorul nr.2. Variabile, tipuri de date

Deoarece Haskell e un limbaj de programare pur functional, toate calculele sunt efectuate prin evaluarea unor *expresii*(termeni sintactici) pentru a obtine *valori*(entitati abstracte pe care le privim ca raspuns). Fiecare valoare are un tip de date asociat. Exemple de expresii:

- valori atomice precum: valoarea intreaga **`5`**, caracterul '**`a`**'
- functia: **`x -> x + 1`**
- structuri precum lista **`[1,2,3]`** si perechea **`('b', 4)`**

Cum expresiile desemneaza valori, expresiile cu tipuri desemneaza tipuri. Exemplele de expresii de tipuri cuprind:

- tipuri atomice: **`Integer`**(numere intregi cu precizie infinita), **`Char`**(caractere),
  `Integer->Integer`(functie ce mapeaza Integer la Integer);
- structuri de tipuri precum: **`[Integer]`** (lista omogena de Integer),
  `(Char, Integer)` (perechi caracter, nr. intreg);

In Haskell toate valorile pot fi trimise ca argumente functiilor, intoarse ca rezultat al unei functii, folosite in structuri de date. Tipurile de date, insa, nu se pot folosi in acest mod, ele descriu valori, iar asocierea dintre o valoare si un tip de date se numeste *typing*. Exemple:

```
5 :: Integer
'a' :: Char
inc :: Integer -> Integer
[1,2,3] :: [Integer]
('b',4) :: (Char, Integer)
```

Semnul "**`::`**" poate fi citit precum "este de tipul".

Functiile Haskell sunt, de obicei, definite prin ecuatii. Functia **`inc`** se defineste prin ecuatia:

```
inc n = n + 1
```

O ecuatie este un exemplu de *desclaratie*. Un alt tip de declaratie este *declaratia signaturii tipului*, prin care putem declara explicit tipul lui **`inc`**:

```
inc   :: Integer -> Integer
```

In Haskell toate expresiile trebuie sa aibe un tip, care este calculat inainte de evaluarea expresiei prin "type inference"(deductia tipului). In particular, exista un set de reguli de aplicare a tipului care sunt folosite pentru a calcula tipul unei expresii pornind de la componentele sale. Regula de baza se refera la aplicarea functiilor si precizeaza ca daca o functie *f* primeste un argument de tipul *A* si returneaza unul de tipul *B,* iar *e* este o expresie de tipul *A,* atunci aplicare fuctiei *f* lui *e* va returneaza un rezultat de tipul *B.* Pe scurt:

if f :: A -> B and e :: A, then f e :: B

Deoarece deducerea tipurilor precede evaluarea, programele Haskell sunt *type safe*, in sensul ca erorile de tip nu se pot produce in timpul executiei. In practica, deductia tipurilor detecteaza o clasa foarte larga de erori de programare si este una din cele mai folositoare caracteristici ale limbajului. De notat ca folosirea deductiei tipurilor nu elimina posibilitatea aparitiei altor tipuri de erori. Un exemplu il constituie expresia **`1`div`0`** care este corecta din punctul de vedere al tipului, dar va produce o eroare la evaluarea ei deoarece impartirea prin **0** este nedefinita.

Tipurile de baza:

- **Bool** – valori logice
  Contine valorile logice **False** si **True**.
- **Char** – caractere
  Contine toate caracterele disponibile de la tastatura precum **'a'**, **'_'**, **'V'**, precum si un numar de caractere de control precum **'\n'**, **'\t'**.
- **String** – siruri de caractere
  Acest tip contine toate secventele de caractectere, precum **"abc"**, **"1+2=3"** si sirul vid **""**.
- **Int** – numere intregi intre -2^{31} si $2^{31}-1$
- **Integer** – numere intregi cu precizie infinita
- **Float** - single-precision floating-point numbers

Tipul lista:

- Secventa(chiar si infinita) de elemente de acelasi tip:
  - [True, False, False] :: [Bool]
  - ['a','b','c','d'] :: [Char]
  - ["unu", "doi", "trei"] :: String
- [[]] este diferita de []
- nu exista restrictii asupra tipurilor ce pot forma liste!
  - [['a', 'b'], ['a', 'c', 'd']] :: [[Char]]
- 3:(2:(1:[])) , lista construita cu 'cons'.
- datorita evaluarii tarzii listele infinite sunt posibile in Haskell.

Tipul uplu:

- secventa finita de componente, posibil de tipuri diferite:
  - (False, True) :: (Bool, Bool)
  - (False, 'a', True) :: (Bool, Char, Bool)
  - ("Yes", True, 'a') :: (String, Bool, Char)
- numarul de componente dintr-un uplu se numeste aritate

- uple de aritate 1 precum (False) nu sunt permise deoarece intra in conflict cu folosirea parantezelor pentru a forta ordinea unei evaluari. Ex. (1+2)*3
- tipul unui uplu ii denota aritatea
    - (Bool, Bool) ulpu de aritate 2
    - (String, Bool, Char) ulpu de aritate 3
- nu exista restrictii asupra tipurilor ce pot forma uple:
    - ('a', (False, 'b')) :: (Char, (Bool , Char))
    - (['a', 'b'], [False, True]) :: ([Char], [Bool])

Tipul functie:

- functia este o mapare de la argumente de un tip la rezultate de alt tip.
- **T_1 -> T_2** este tipul tuturor functiilor ce mapeaza argumente de tipul T_1 la rezultate de tipul T_2:
    - **isDigit::Char->Bool**
    - **and::[Bool]->Bool**
- nu exista restrictia ca functiile sa fie totale (sa fie definite ptr orice argument valid). Functia de biblioteca **tail** nu e definita ptr lista vida.
- In Haskell e standard ca functiile sa fie precedate de declaratia de tip, care e verificata la compilare cu cea obtinuta prin *type inference*.
- functiile cu mai multe argumente se pot defini in 2 feluri:
    - "impachetand" argumentele in uple sau liste:
        - ```
          add :: (Int, Int) -> Int
          add (x,y) = x + y
          ```
        - ```
          sum :: [Int] -> Int
          sum x:xs = x + sum xs
          ```
    - prin functii Curry (metoda nativa Haskell):
        - sunt functii ce intorc alte functii
        - ```
          add' :: Int->(Int->Int)
          add' x y = x + y
          ```
        - declaratia functiei ar fi identica chiar si fara paranteze, fiind asociativa la dreapta.
        - definitia functiei e asociativa la stanga:
            - ((add' x) y) = add' x y
        - in afara de cazurile in care trimiterea parametrilor sub forma de uple e strict necesara, toate functiile in Haskell cu parametri multipli sunt definite ca functii curry.

Tipuri polimorfice:

- functiile pot fi definite pentru mai multe tipuri: functia **length** trebuie sa poata fi aplicata pe orice tip de lista!
- este posibil prin definirea functiei pe *tipuri variabile:*
    - **length :: [a] -> Int**
    - **head :: [a] -> a**
    - **id :: a -> a**
    - **fst :: (a,b) -> a**
- variabilele de tip incep prin litere mici si se noteaza de obicei: a, b, c
- un tip ce contine unul sau mai multe tipuri/variabile de tip se numeste *polimorfic.*
          ```

Tipuri supraincarcate:

- operatorii **+, -, *, ^** se aplica mai multor tipuri
- tipul acestora contine variabile de tip supuse unor constrangeri de clasa ce se scriu sub forma:
    - **C a** unde **C** este numele clasei si **a** variabila de tip
    - **(+) :: *Num* a => a-> a->a** adica pentru orice tip **a**, instanta a clasei ***Num***, functia **(+)** are tipul **a->a->a**
- acestea se numesc tipuri *supraincarcate*

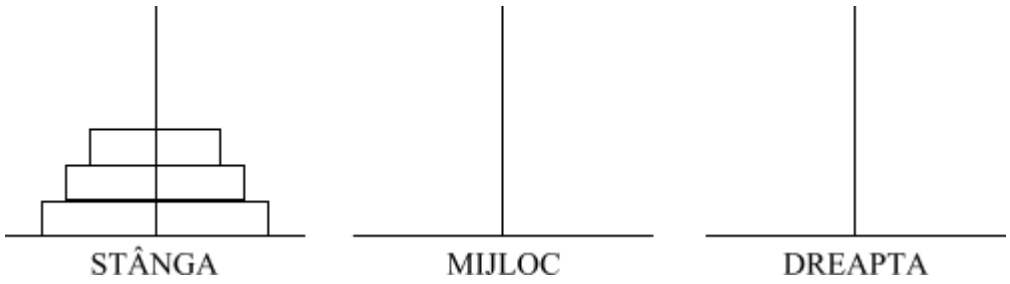
Obs: tipul unei expresii se poate afla prin comanda **:type <expresie>**

Elemente de sintaxa Haskell prin exemple:

- repeat :: a -> [a]
  repeat x = xs where xs = x:xs
- doubleLen :: [a] -> Int
  doubleLen xs = let n=length xs in n+n
- isPositive :: Num a => a -> Bool
  isPositive n = case signum n of
  1 -> True
  _ -> False

Exercitii:

1. Care este tipul urmatoarelor date:
   ['a', 'b', 'c']
   ('a', 'b', 'c')
   [(False, 'O'), (True, '1')]
   ([False, True], ['0', '1'])
   [tail , init , reverse]
2. Functia
   map :: (a -> b) -> [a] -> [b]
   aplica o alta functie pe toate elementele unei liste, rezultand o noua lista. Scrieti o definitie echivalenta e functiei de librarie **length** folosind functia **map**.
   Scrieti o functie mapAll asemanatoare cu map, dar care primeste o lista de functii ca prim argument.
3. Scrieti o functie Haskell care testeaza daca un numar intreg este palindrom.
4. Definiti o functie
   block :: Int -> [a] -> [[a]]
   astfel incat block n xs imparte xs in blocuri de n elemente. De exemplu:
   block 3 "hello clouds" = ["hel", "lo ", "clo", "uds"]
5. Turnurile din Hanoi. Se dau trei tije si n discuri de diferite dimensiuni, stivuite pe tija STANGA în ordine descrescatoare a dimensiunilor lor, formând un <turn> ca în figura. Sa se scrie o functie Haskell care care calculeaza cate mutari sunt necesare pentru a muta cele n discuri de pe tija STANGA pe tija DREAPTA, astfel încât ele sa fie ordonate ca la început.



Mutarile se fac cu urmatoarele restrictii:

- la fiecare miscare se muta doar un disc;
- un disc nu poate fi plasat peste unul mai mic;
- tija MIJLOC poate fi utilizata ca pozitie intermediara.

Problema mutarii celor n discuri poate fi redusa la problema mutarii a n-1 discuri, astfel:

- se muta primele n-1 discuri în pozitia MIJLOC, folosind tija DREAPTA;
- se muta discul ramas în pozitia DREAPTA;
- se muta cele n-1 discuri din pozitia MIJLOC în pozitia DREAPTA, folosind tija STANGA drept pozitie intermediara.

Puteti gasi o strategie bazata pe recursivitate pentru a rezolva puzzel-ul?

Daca mai adaugam o tija cum se modifica strategia? Cate mutari sunt necesare de aceasta data?

Laboratorul nr.3. Clasele de baza din Haskell

O clasa este o colectie de tipuri ce suporta anumiti operatori supraincarcati numiti *metode*.

- **Eq –** clasa care contine tipuri ale caror valori pot fi comparate pentru egalitate sau diferenta folosind urmatoarele doua metode:
  - `(==) :: a -> a -> Bool`
  - `(/=) :: a -> a -> Bool`

  Toate tipurile de baza: *Bool*, *Char*, *String*, *Int*, *Integer* si *Float* sunt instante ale clasei **Eq**, la fel ca si listele si uplele ale caror elemente sunt instante ale clasei.

  De exemplu:

```
> False == False
True
> 'a' == 'b'
False
> "abc" == "abc"
True
> [1, 2] == [1, 2, 3]
False
> ('a', False) == ('a', False)
True
```

- **Ord** – aceasta clasa contine tipuri ce sunt instante ale clasei **Eq** si in plus ale caror valori sunt total ordonate si astfel pot fi comparate folosind urmatoarele 6 metode:
  - `(<) :: a -> a -> Bool`
  - `(<=) :: a -> a -> Bool`
  - `(>) :: a -> a -> Bool`
  - `(>=) :: a -> a -> Bool`
  - `min :: a -> a -> a`
  - `max :: a -> a -> a`

Toate tipurile de baza: *Bool*, *Char*, *String*, *Int*, *Integer* si *Float* sunt instante ale clasei **Ord**, la fel ca si listele si uplele ale caror elemente sunt instante ale clasei.
De exemplu:

```
> False < True
True
> min 'a' 'b'
'a'
> "elegant" < "elephant"
True
> [1, 2, 3] < [1, 2]
False
> ('a', 2) < ('b', 1)
True
> ('a', 2) < ('a', 1)
False
```

A se observa ca sirurile de caractere, listele si uplele sunt ordonate lexicografic!

- **Show** – clasa ce contine tipuri ale caror valori pot fi convertite in siruri de caractere folosind urmatoarea metoda:
  - `show :: a -> String`

  Toate tipurile de baza: *Bool*, *Char*, *String*, *Int*, *Integer* si *Float* sunt instante ale clasei **Show**, la fel ca si listele si uplele ale caror elemente sunt instante ale clasei. De exemplu:

```
> show False
"False"
> show 'a'
"'a'"
> show 123
"123"
> show [1, 2, 3]
"[1,2,3]"
> show ('a', False)
"('a',False)"
```

- **Read** – clasa duala cu Show, contine tipuri ale caror valori pot fi convertite din siruri din caractere folosind urmatoarea metoda:
  - `read :: String -> a`

  Toate tipurile de baza: *Bool*, *Char*, *String*, *Int*, *Integer* si *Float* sunt instante ale clasei **Read**, la fel ca si listele si uplele ale caror elemente sunt instante ale clasei. De exemplu:

```
> read "False" :: Bool
False
> read "'a'" :: Char
'a'
> read "123" :: Int
123
> read "[1,2,3]" :: [Int ]
[1, 2, 3]
> read "('a',False)" :: (Char, Bool )
('a', False)
```

S-a folosit "::" in exemple pentru a specifica tipul rezultatului. Adesea, in practica, nu e necesar deoarece tipul e dedus automat din context. Exemplu: `not (read "True")` .

- **Num** – tipuri numerice. Contine tipuri ce sunt instante ale claselor **Eq** si **Show** si in plus contin valori numerice ce pot fi procesate cu urmatoarele 6 metode:
  - `(+) :: a -> a -> a`
  - `(-) :: a -> a -> a`
  - `(*) :: a -> a -> a`
  - `negate :: a -> a`
  - `abs :: a -> a`
  - `signum :: a -> a`

  Tipurile *Int, Integer* si *Float* sunt instante ale acestei clase. De exemplu:

```
> 1 + 2
3
> 1.1 + 2.2
3.3
> negate 3
-3
> abs (-3)
3
> signum (-3.3)
-1
```

  Clasa **Num** nu are o metoda pentru impartire, aceasta este implementata in urmatoarele doua clase.
- **Integral** – numere intregi. Contine tipuri instante ale clasei **Num** ce in plus suporta urmatoarele 2 metode:
  - `div :: a -> a -> a`
  - `mod :: a -> a -> a`

  Tipurile Int si Integer sunt instante ale acestei clase.
- **Fractional** – aceasta clasa contine tipuri instante ale clasei **Num**, dar in plus ale caror valori nu sunt intregi si astfel suporta metodele de impartire si reciproca:
  - `(/) :: a -> a -> a`
  - `recip :: a -> a`

  Tipul de baza *Float* este instanta a acestui tip. De exemplu:

```
> 7.0 / 2.0
3.5
> recip 2.0
0.5
```

Exercitii:

1. Scrieti o functie *powers* cu proprietatea ca **`powers n = [n¹, n², n³,…]`**
2. Definiti o functie **longest** care returneaza cele mai lungi cuvinte dintr-o propozitie(*String*). Exemplu:
   `> longest "examples of functional algorithms"`
   `["functional", "algorithms"]`
   Ajutor: puteti folosi functia de biblioteca **`words`**.
3. Scrieti o functie care calculeaza intersectia a doua multimi(liste).
   Ajutor: puteti folosi functia de librarie **`elem`**.

4. Scrieti o functie care transforma o lista de valori *Bool* intr-un numar intreg, considerand lista ca fiind reprezentarea lui in baza 2. De exemplu: [True, True, False] se va evalua la 6. Scrieti apoi functia inversa.
5. Scrieti o functie care calculeaza cel mai mare divizor comun a doua numere.
6. Implementati unul din algoritmii de sortare cunoscuti. Datele se vor reprezenta sub forma de lista.
7. Scrieti o functie ce evalueaza expresii sub forma poloneza postfixata. Expresia de calculat va fi transmisa functiei sub forma unui sir de caractere. Exemplu:
   "1 2 3 + *" se va evalua la 7, adica 1+2*3
   " 1 3 + 2 *" se va evalua la 8, adik (1+3)*2

Laboratorul nr.4. Moduri de definire a functiilor

- pornind de la functii existente:
```
and, or   :: [Bool] -> Bool
and        = foldr (&&) True
or         = foldr (||) False
```

- prin expresii conditionale: ***if*** *condiţie* ***then*** *res1* ***else*** *res2*
```
until          :: (a -> Bool) -> (a -> a) -> a -> a
until p f x     = if p x then x else until p f (f x)
```

- "guarded ecuations"
```
dropWhile            :: (a -> Bool) -> [a] -> [a]
dropWhile p []        = []
dropWhile p xs@(x:xs')
          | p x       = dropWhile p xs'
          | otherwise = xs
```

- "pattern matching"
```
null              :: [a] -> Bool
null []            = True
null (_:_)         = False
```

- lambda expresii
```
> (\x-> x+x) 2
> 4
```

- sectiuni
```
(1+) = \y -> 1+y
(1/) = \y -> 1/y
(*2) = \x -> x*2
(/2) = \x -> x/2
```

Exercitii

1. Se considera o functie **safeTail::[a]->[a]** ce se comporta similar cu functia de librarie **tail**, cu exceptia ca in cazul listei vide **safeTail** va returna aceeasi lista vida, pe cand **tail** genereaza eroare. Definiti **safeTail** folosind:

- o expresie conditionala
- "guarded ecuations"
- "pattern matching"

2. Scrieti o functie **subList** care verifica daca o lista e inclusa in alta.
3. Definiti o functie pentru calculul produsului a 2 polinoame. Un polinom se va reprezenta prin lista coeficientilor.
4. Scrieti o functie **pascal :: Int -> [Int]** , unde **pascal n** returneaza randul n din triunghiul lui Pascal.

Laboratorul nr.5. List comprehension

1. Scrieti o functie *pythagoreans* care calculeaza toate tripletele (x,y,z) cu proprietatea $x^2+y^2=z^2$, unde x, y, z sunt pana la o limta data. De exemplu:
```
> pythagoreans 10
[(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
```

2. Un numar natural se numeste perfect daca este egal cu suma divizorilor sai, excluzandu-se pe sine. Folosind list comprehension, definiti o functie**perfects::Int->[Int]** care o lista cu toate numere perfecte panal la o anumita limita. De exemplu:
```
> perfects 500
 [6,28,496]
```

3. Implementati cifrul lui Cezar. Se vor define 2 functii **encode** pentru criptare si **decode** pentru decriptare.
   Ajutor: folositi functiile *chr* si *ord*.

Laboratorul nr.6.

Exercitii:

1. Definiti o functie care calculeaza valoarea lui e^x utilizand seria Maclaurin:
   Solutia va trebui sa neglijeze termenii mai mici de 10^{-6} ai seriei.
   Ajutor: puteti folosi functiile *zipWith* si *takeWhile.*

2. Definiti o functie **digits :: Integer -> Integer** astfel incat **digits n** sa returneze numarul de cifre al celui mai mic multiplu, **m**, al lui **n**, cu proprietatea ca **m** se scrie in baza zece folosind doar cifra 1. De exemplu:
```
> digits 9901
12
> digits 3
3
> digits 7
6
```

3. Considerati urmatorul algoritm pentru generarea unei secvente de numere. Se porneste de la un numar natural **n** oarecare. Daca n este par se imparte la 2, alfel se inmulteste cu 3 si se adauga 1. Repetati pana cand se ajunge la voarea 1. De exemplu pentru n=22 se obtine urmatoarea secventa: 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1.

Definiti o functie **solve :: Int -> Int -> (Int, Int, Int)** care dandu-se doua numere **i** si **j** va returna o tripleta de forma **(i, j, k)** unde **k** este lungimea secventei maxime, generate conform algoritmului de mai sus, considerandu-se ca punct de start, pe rand, toate numere din intervalul **[i..j]**. De exemplu:

```
> solve 1 10
(1,10,20)
> solve 100 200
(100,200,125)
> solve 201 210
(201,210,89)
> solve 900 1000
(900,1000,174)
```

4. Definiti o functie **isJumper :: [Int] -> Bool** care testeaza daca o lista de n > 0 numere indeplineste conditia ca valoarea absoluta a diferentei dintre elemente succesive ia toate valorile din intervalul **[0..n-1]**. O astfel de lista este [1,4,2,3] deoarece diferentele absolute sunt 3, 2 si respectiv 1. Se va sonsidera ca orice lista cu un singur element indeplibeste conditia.

```
> isJumper [1,4,2,3]
True
> isJumper [1,4,2,-1,6]
False
```

5. Definiti o functie care, dandu-se o lista de numere intregi si un numar intreg, returneaza un sir de caractere reprezentand o expresie matematica formata din numerele din lista si operatorii +, -, *, / ce se evalueaza la valoarea celui de-al doilea argument, in cazul in care e posibil, altfel returneaza sirul vid. Sunt permise doar operatii cu numere intregi (se utilizeaza impartirea doar daca rezultatul e intreg). Exemplu:

```
> solve [2,4,5,9,75] 658
"(2*(9+(4*(5+75))))"
> solve [9,1,4,8] 26
""
> solve [9,1,4,8] (-17)
"(1-((9*8)/4))"
```

Laboratorul nr.7. Intrările și ieșirile

1. Scrieti o actiune IO() care concateneaza 2 fisiere a caror nume se citeste de la tastatura.
2. Definiti o actiune IO() readSort, care citeste numere intregi, de la tastatura, de pe linii succesive pana la intalnirea unei linii ce nu contine un intreg. Dupa citirea ultimei linii valide trebuie sa afiseze numerele sub forma unei secvente ordonate.
3. Construiti un joc de **x** si **zero** folosind biblioteca grafica Graphics.SOE (opțional)