Laboratorul 1

În acest laborator ne vom aminti unele concepte de programare funcțională și vom experimenta folosirea unui interpretor al unui mini-limbaj funcțional, pe care îl vom numi `miniHaskell` datorita sintaxei ușor asemănatoare cu aceea a limbajului `Haskell`.

Limbajul `miniHaskell` ne va însoți pe parcursul întregului semestru, deoarece vom avea teme de laborator care ne vor ajuta să ne implementăm propriul nostru interpretor `miniHaskell`.

Limbajul `miniHaskell`

`miniHaskell` este un limbaj funcțional minimalist, având la bază doar λ-calcul (variabile, abstracții de funcții, aplicație), și folosind extensii *sintactice* pentru Booleeni, nr naturale, optiuni, perechi, liste și definiții (inclusiv definiții recursive).

Prin extensii *sintactice* înțelegem că toate extensiile sunt de fapt codări în limbajul de bază folosind ideile de Church encodings. De aceea, de exemplu, funcțiile care manipulează numere naturale au o complexitate relativ crescută, deoarece un număr `n` este similar unei liste cu `n` elemente, iar orice operație elementară asupra sa este similară unei operații de agregare (`foldr`) a unei liste.

Sintaxa `miniHaskell`

```
Exp ::=
   x
 | '\' x '->' Exp
 | Exp Exp
 | 'let' x '=' Exp 'in' Exp
 | 'letrec' f '=' Exp 'in' Exp
```

Expresiile pot fi formate prin una din următoarele operații:

- variablile (`x`), care pot fi orice identificator sau operator definibil în Haskell

  Atenție: operatorii vor fi folosiți tot ca funcții, dar fără a îi mai înconjura între paranteze. De exemplu, compunerea a două funcții va fi `. f g` în loc de `f . g` sau `(.) f g`

- λ-abstracții, care folosesc sintaxa din Haskell
- aplicarea unei expresii altei expresii
- legarea unui nume la o expresie în scopul folosirii sale în evaluarea altei expresii (`let`)
- legarea unui nume la o expresie (care poate folosi recursiv acel nume) în scopul folosirii sale în evaluarea altei expresii (`letrec`)

Extensii sintactice

Pe lângă expresiile de bază, limbajul mai acceptă și alte tipuri de expresii (definite în limbajul de bază). În prezentarea lor vom folosi signaturi de funcții de tipul celor din `Haskell`.

Atenție: deși signaturile funcțiilor sunt prezentate folosind tipuri, limbajul `miniHaskell` este fără tipuri, deci este responsabilitatea voastră să le folosiți conform tipului indicat. În partea a doua a semestrului vom implementa și un verificator de tipuri pentru a elimina expresiile cu tipuri greșite.

Funcții de bază

- `id :: a -> a`
- `const :: a -> b -> a`
- `. :: (b -> c) -> (a -> b) -> (a -> c)`
- `flip :: (a -> b -> c) -> (b -> c -> a)`.

Tipul `Bool`

- `True :: Bool`
- `False :: Bool`
- `if :: Bool -> a -> a -> a` este instrucțiunea condițională `if_then_else_`
- `&& : Bool -> Bool -> Bool`
- `|| : Bool -> Bool -> Bool`
- `not :: Bool -> Bool`

Tipul `Maybe a`

- `Nothing :: Maybe a`
- `Just :: a -> Maybe a`
- `maybe :: b -> (a -> b) -> Maybe a -> b`
- `fromMaybe :: a -> Maybe a -> a`
- `isNothing :: Maybe a -> Bool`
- `isJust :: Maybe a -> Bool`
- `mapMaybe :: (a -> b) -> Maybe a -> Maybe b` este `fmap` pentru instanța de `Functor` a lui `Maybe`

Tipul pereche `(a,b)`

- `pair :: a -> b -> (a,b)` este constructorul de perechi
- `fst :: (a, b) -> a`
- `snd :: (a, b) -> a`

Tipul `Natural`

- orice număr natural poate fi folosit ca constantă
- `Z :: Natural` o altă formă a constantei `0`

- `S :: Natural -> Natural` funcția succesor
- `+ :: Natural -> Natural -> Natural`
- `* :: Natural -> Natural -> Natural`
- `pred :: Natural -> Natural` funcția predecesor (întoarce `0` pentru `0`)
- `- :: Natural -> Natural -> Natural` întoarce `0` dacă scăzătorul e mai mare
- `isZero :: Natural -> Bool` testează dacă argumentul e `0`
- `<= :: Natural -> Natural -> Bool`
- `>= :: Natural -> Natural -> Bool`
- `== :: Natural -> Natural -> Bool`
- `< :: Natural -> Natural -> Bool`
- `> :: Natural -> Natural -> Bool`
- `max :: Natural -> Natural -> Natural`
- `fact :: Natural -> Natural` calculează factorialul numărului dat ca argument

Tipul listelor `[a]`

- orice listă de tipul `[e1, e2, e3]` este o expresie
- `null :: [a]` este un alias pentru lista vidă (`[]`)
- `: :: a -> [a] -> [a]` este constructorul de liste
- `(++) :: [a] -> [a] -> [a]`
- `length :: [a] -> Natural`
- `isNull :: [a] -> Bool` testează dacă argumentul e lista vidă
- `uncons :: [a] -> Maybe (a, [a])`
- `head :: [a] -> Maybe a` variantă sigură a funcției head
- `tail :: [a] -> Maybe [a]` variantă sigură a funcției tail
- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- `map :: (a -> b) -> [a] -> [b]`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `foldl :: (b -> a -> b) -> b -> [a] -> b`
- `reverse :: [a] -> [a]`

Funcții speciale pentru liste de numere naturale `[Natural]`

- `nat2list :: Natural -> [Natural]` numerele naturale de la `1` la argumentul dat, în ordine inversă

- `product :: [Natural] -> Natural`
- `sum :: [Natural] -> Natural`
- `maximum :: [Natural] -> Natural`

Exerciții

0. Interpretorul

Executați interpretorul `miniHaskell`. Dacă nu aveți probleme în executarea lui (de genul *missing libraries*, *bad architecture*) ar trebui să vedeți un prompt:

```
miniHaskell>
```

Acum puteți scrie orice expresie folosind limbajul descris mai sus și interpretorul va încerca să o evalueze. De exemplu:

```
miniHaskell> letrec fct = \n -> if (isZero n) 1 (* n (fct (pred n))) in fct 4
24
miniHaskell>
```

Important: fiecare expresie pe care o vrem evaluată trebuie să fie scrisă pe o singură linie

Pentru a ieși din interpretor trebuie sa folosiți comanda `:q` sau `:quit` ca în `ghci`

1. Functii lambda

Definiti o funcție `squareSum :: Natural -> Natural -> Natural` care primește două numere naturale ca argument și întoarce suma pătratelor celor două numere.

Exemplu:

```
miniHaskell> let squareSum = ... in squareSum 2 3
13
```

2. Recursie

Definiți prin recursie o funcție `revRange :: Natural -> [Natural]` care primește un număr natural ca argument și întoarce lista numerelor naturale mai mici decât `n`, în ordine descrescătoare.

Exemplu:

```
miniHaskell> letrec revRange = ... in revRange 4
[3, 2, 1, 0]
```

Folosind funcția anterioară, definiția funcția `range` care întoarce numerele mai mici ca `n` în ordine crescătoare.

```
miniHaskell> let range = ... in range 4
[0, 1, 2, 3]
```

3. Filter, map

Definiți, fără a folosi explicit recursie, o funcție `justList :: [Maybe Natural] -> [Natural]` care primește o listă cu elemente `Maybe Natural` și întoarce

lista formată din acei `n` pentru care `Just n` apare în listă, păstrând ordinea și multiplicitatea.

Exemplu:

```
miniHaskell> let justList = ... in justList [Just 4, Nothing, Just 5, Just 7, Nothing]
[4, 5, 7]
```

4. Fold

Fără a folosie explicit recursie, scrieți funcția `all :: (Natural -> Bool) -> [Natural] -> Bool` care verifică dacă toate elementele unei liste satisfac un predicat.

Exemplu:

```
miniHaskell> let all = ... in all isZero [0, 0, 0]
True
miniHaskell> let all = ... in all isZero [0, 1]
False
```