FMI, Info, Anul II, 2020-2021
Programare funcțională

Laborator 4
Funcții de nivel înalt

- Reamintiți-vă algoritmul de generare a numerelor prime folosind Ciurul lui Eratostene: https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes. Definiți funcția

  ```
  numerePrimeCiur :: Int -> [Int]
  ```

  care implementează în Haskell acest algoritm (pentru definirea acestei funcții puteți folosi orice metodă doriți).

- **Ordonare folosind comprehensiunea** Pentru început observați comportamentul funcției `and`:

  ```
  Prelude> and [True, False, True]
  False
  Prelude> and [1 < 2, 2 < 3, 3 < 4]
  True
  Prelude> and [1 < 2, 2 < 3, 3 < 1]
  False
  ```

Exerciții

1. Folosind comprehensiunea, funcția `and` și funcția `zip`, completați definiția funcției `ordonataNat` care verifică dacă o listă de valori `Int` este ordonată, relația de ordine fiind cea naturală:

   ```
   ordonataNat [] = True
   ordonataNat [x] = True
   ordonataNat (x:xs) =
   ```

2. Fără comprehensiune, folosind recursie, definiți funcția `ordonataNat1`, care are același comportament cu funcția de mai sus.

3. Scrieți o funcție `ordonata` generică cu tipul

   ```
   ordonata :: [a] -> (a -> a -> Bool) -> Bool
   ```

care primește ca argumente o listă de elemente și o relație binară pe elementele respective. Funcția întoarce `True` dacă oricare două elemente consecutive sunt în relație.

a. Definiți funcția `ordonata` prin orice metodă.

b. Verificați definiția în interpretor pentru diferite valori:
- numere întregi cu relația de ordine;
- numere întregi cu relația de divizibilitate;
- liste (șiruri de caractere) cu relația de ordine lexicografică; obervați că în Haskell este deja definită relația de ordine lexicografică pe liste:

```
Prelude> [1,2] >= [1,3,4]
False
Prelude> "abcd"<"b"
True
```

c. Amintiți-vă teoria de la curs legată de *operatori* sau citiți o scurtă descriere: https://wiki.haskell.org/Section_of_an_infix_operator. Definiți un operator `*<*` cu signatura

```
(*<*) :: (Integer, Integer) -> (Integer, Integer) ->
    Bool
```

care definește o relație pe perechi de numere întregi (alegeți voi relația). Folosind funcția `ordonata` verificați dacă o listă de perechi este ordonată față de relația `*<*`

- Înainte de a trece mai departe, vom face o observație despre **evaluarea funcțiilor în GHCi**. Observați că funcția `sqrt` este o funcție predefinintă; dacă îi dăm o intrare concretă în interpretor, acesta îi calculează corect valoarea.

```
Prelude> sqrt 5.6
2.3664319132398464
```

Însă, dacă dorim să evaluăm funcția

```
Prelude> sqrt
<interactive>:73:1: error:
```

vom obține un mesaj de eroare (ne spune că `sqrt` nu este instanță a clasei `Show`). Practic acest lucru înseamnă ca el nu știe să afișeze valoarea lui `sqrt`, care este o λ-expresie. Același lucru se întâmplă și cu funcții definite de noi, chiar dacă sunt definite ca λ-expresii:

```
Prelude> h = (\x -> x+1)
Prelude> h
<interactive>:73:1: error:
```

Vom discuta despre acest lucru mai târziu, dar rețineți că atunci când o funcție întoarce funcții (liste de funcții, tupluri de funcții, etc) ca valori, ele nu pot fi vizualizate direct în interpretor. Putem însă să cerem informații asupra tipului și putem să le evaluăm pentru valori particulare ale argumentelor:

```
Prelude> :t h
h :: Num a => a -> a
Prelude> h 4
5
```

4. Scrieți o funcție `compuneList` de tip

```
compuneList :: (b -> c) -> [(a -> b)] -> [(a -> c)]
```

care primește ca argumente o funcție și o listă de funcții și întoarce lista funcțiilor obținute prin compunerea primului argument cu fiecare funcție din al doilea argument.

```
*Main> :t compuneList (+1) [sqrt, (^2), (/2)]
```

Conform observației de mai sus, nu putem vizualiza direct rezultatul aplicării funcției `compuneList`. Pentru a verifica funcționalitatea trebuie să calculăm funcțiile în valori particulare.

Scrieți o funcție `aplicaList` de tip

```
aplicaList :: a -> [(a -> b)] -> [b]
```

care primește un argument de tip `a` și o listă de funcții de tip `a -> b` și întoarce lista rezultatelor obținute prin aplicarea funcțiilor din listă pe primul argument:

```
*Main> aplicaList 9 [sqrt, (^2), (/2)]
[3.0,81.0,4.5]
```

Folosind `aplicaList` putem testa `compuneList`:

```
*Main> aplicaList 9 (compuneList (+1) [sqrt, (^2), (/2)])
[4.0,82.0,5.5]
```

- Scrieți funcția `myzip3` folosind numai `map` și `zip`.
- Citiți capitolul *Higher order functions* din

  M. Lipovaca, Learn You a Haskell for Great Good!

  http://learnyouahaskell.com/higher-order-functions