FMI, Info, Anul III, 2020-2021
Programare declarativă

Laborator 3
Din nou liste. Funcții de nivel înalt

Atenție! Înainte de a continua acest laborator terminați exercițiile din Laboratorul 2!

În rezolvarea exercițiilor folosiți fișierul lab3.hs, ce conține prototipurile funcțiilor pe care trebuie să le implementați și exemple de rulat.

(L3.1) [Definirea listelor prin comprehensiune] Reamintiți-vă definirea listelor prin comprehensiune din Laboratorul 2. Încercați să găsiți valoarea expresiilor de mai jos și verificați răspunsul găsit de voi în interpretor:

```
[ x^2 |x <- [1..10], x `rem` 3 == 2]
[(x,y)| x<- [1..5], y <- [x..(x+2)]]
[(x,y)| x<-[1..3], let k = x^2, y <- [1..k]]
[ x | x<- "Facultatea de Matematica si Informatica", elem x ['A'..'Z']]
[[x..y]| x <- [1..5], y <- [1..5], x < y]
```

Pentru a putea fi ușor copiate în interpretor expresiile sunt și în fișierul sursă.

Exerciții Deși în aceste exerciții vom lucra cu date de tip `Int`, rezolvați exercițiile de mai jos astfel încât rezultatul să fie corect pentru valori pozitive. Definițiile pot fi adapatate ușor pentru valori oarecare folosind funcția `abs`.

1. Folosind numai comprehensiunea definiți o funcție

   ```
   factori :: Int -> [Int]
   ```

   atfel încât `factori n` întoarce lista divizorilor pozitivi ai lui `n`.

2. Folosind funcția `factori`, definiți predicatul `prim n` care întoarce `True` dacă și numai dacă `n` este număr prim.

3. Folosind numai comprehensiunea și funcțiile definite anterior, definiți funcția

   ```
   numerePrime :: Int -> [Int]
   ```

astfel încât `numerePrime n` întoarce lista numerelor prime din intervalul `[2..n]`.

(L3.2) [**Funcția** `zip`] Testați și sesizați diferența:

```
Prelude> [(x,y)| x <- [1..5], y <- [1..3]]
Prelude> zip [1..5] [1..3]
```

Definiți funcția `myzip3` care se comportă asemenea lui `zip` dar are trei argumente:

```
*Main> myzip3 [1,2,3] [1,2] [1,2,3,4]
[(1,1,1),(2,2,2)]
```

Funcții de nivel înalt În Haskell, funcțiile sunt *valori*. Putem să trimitem funcții ca argumente și sa le întoarcem ca rezultat.
Să presupunem că vrem să definim o funcție `aplica2` care primește ca argument o funcție `f` de tip `a -> a` și o valoare `x` de tip `a`, rezultatul fiind `f (f x)`. Tipul funcției `aplica2` este

```
aplica2 :: (a -> a) -> a -> a
```

Se pot da mai multe definiții:

```
aplica2 f x = f (f x)
aplica2 f = f . f
aplica2 =\f x -> f (f x)
aplica2 f = \x -> f (f x)
```

(L3.3) [**map**] Funcția `map` are ca argumente o funcție de tip `a -> b` și o listă de elemente de tip `a`, rezultatul fiind lista elementelor de tip `b` obținute prin aplicarea funcției date pe fiecare element de tip `a`:

```
map :: (a -> b) -> [a] -> [b]
map f xs =[f x | x <- xs]
```

Reamintiți-vă noțiunea de *secțiune* definită la curs: o *secțiune* este aplicarea parțială a unui operator, adică se obține dintr-un operator prin fixarea unui argument. De exemplu
`(*3)` este o funcție cu un singur argument, rezultatul fiind argumentul înmulțit cu `3`,
`(10-)` este o funcție cu un singur argument, rezultatul fiind diferența dintre `10` și argument.
Următoarele exemple - discutate la curs - folosesc secțiuni și funcția `map`:

```
Prelude> map (* 3) [1,3,4]
[3,9,12]
Prelude> map ($ 3) [(4 +), (10 * ), (^ 2), sqrt ]
[7.0,30.0,9.0,1.7320508075688772]
```

Încercați să găsiți valoarea expresiilor de mai jos și verificați răspunsul găsit de voi în interpretor:

```
map (\x -> 2 * x) [1..10]
map (1 `elem`) [[2,3], [1,2]]
map (`elem` [2,3]) [1,3,4,5]
```

Exerciții Rezolvați exercițiile folosind `map`. În fiecare caz scrieți tipul funcției respective.

1. Scrieți o funcție generică `firstEl` care are ca argument o listă de perechi de tip `(a,b)` și întoarce lista primelor elementelor din fiecare pereche:

   ```
   firstEl [('a',3),('b',2), ('c',1)]
   "abc"
   ```

2. Scrieți funcția `sumList` care are ca argument o listă de liste de valori `Integer` și întoarce lista sumelor elementelor din fiecare listă (suma elementelor unei liste de întregi se calculează cu funcția `sum`):

   ```
   sumList [[1,3], [2,4,5], [], [1,3,5,6]]
   [4,11,0,15]
   ```

3. Scrieți o funcție `prel2` care are ca argument o listă de `Integer` și întoarce o listă în care elementele pare sunt înjumătățite, iar cele impare sunt dublate:

   ```
   *Main> prel2 [2,4,5,6]
   [1,2,10,3]
   ```

(L3.4) [map, filter] Funcția `filter` are ca argument o proprietate si o listă de elemente, rezultatul fiind lista elementelor care verifică acea proprietate:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]

Prelude> filter (>2) [3,1,4,2,5]
[3,4,5]
Prelude> filter odd [3,1,4,2,5]
[3,1,5]
```

Exerciții Rezolvați aceste exerciții fără recursie, folosind `filter` și `map`.

1. Scrieți o funcție care primește ca argument un caracter și o listă de șiruri, rezultatul fiind lista șirurilor care conțin caracterul respectiv (folosiți funcția `elem`).

2. Scrieți o funcție care primește ca argument o listă de întregi și întoarce lista pătratelor numerelor impare.

3. Scrieți o funcție care primește ca argument o listă de întregi și întoarce lista pătratelor numerelor din poziții impare. Pentru a avea acces la poziția elementelor folosiți `zip`.

4*. Scrieți o funcție care primește ca argument o listă de șiruri de caractere și întoarce lista obținută prin eliminarea consoanelor din fiecare șir. Rezolvați exercițiul folosind numai `filter`, `map` și `elem`.

```
numaiVocale ["laboratorul", "PrgrAmare", "DEclarativa"]
["aoaou","Aae","Eaaia"]
```

(L3.5) [**mymap, myfilter**] Definiți recursiv funcțiile `mymap` și `myfilter` cu aceeași funcționalitate ca și funcțiile predefinite.

Material suplimentar

- 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