Laboratorul 4: Exerciții

Liste

Reamintiți-vă definirea listelor prin selecție din **Laboratorul 3**. 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]
-}
```

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 metoda prin selecție definiți o funcție

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

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.

```
prim :: Int -> Bool
prim = undefined
```

3. Folosind numai metoda prin selecție și funcțiile definite anterior, definiți funcția

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

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

Funcția `zip`

Testați și sesizați diferența:

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

```
Prelude> zip [1..5] [1..3]
```

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

```
myzip3 [1,2,3] [1,2] [1,2,3,4] == [(1,1,1),(2,2,2)]
```

Secțiuni

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.

Ordonare folosind selectie

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
```

5. Folosind metoda prin selecție, 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 :: [Int] -> Bool
ordonataNat [] = True
ordonataNat [x] = True
ordonataNat (x:xs) = undefined
```

6. Folosind doar recursie, definiți funcția `ordonataNat1`, care are același comportament cu funcția de mai sus.

7. Scrieți o funcție `ordonata` generică cu tipul

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

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
```

8. Definiți un operator `*<*`, asociativ la dreapta, cu precedenta 6, 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 `*<*`.

9. 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)]
```

Nu putem vizualiza direct rezultatul aplicării funcției `compuneList`. Atunci când o funcție întoarce funcții (liste de funcții, tupluri de funcții, etc) ca valori, ele nu pot fi vizualizate direct în interpretor. Pentru a verifica funcționalitatea trebuie să calculăm funcțiile în valori particulare.

10. 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]
```