Funcții de ordin superior
Laborator 4

1 Currying

Vom explica termenul *currying* pe un exemplu. Să considerăm funcțiile `f` și `g` definite astfel:

```
f ::  (Int, Int) -> Int
f (x,y) = x + y

g ::  Int -> Int -> Int
g x y = x + y
```

Observați că funcțiile `f` și `g` au un comportament asemănător, dar totuși ele sunt diferite.

În Haskell, toate funcțiile au doar un singur argument. Funcția `f` primește un singur argument (un tuplu) și returnează un element de tipul `Int`:

```
*Main> :t (f (2,3))
(f (2,3)) ::  Int
```

În Haskell, atunci când specificăm tipul unei funcții trebuie să ținem cont că simbolul `->` este asociativ la dreapta. Astfel, prin `g ::  Int -> Int -> Int` înțelegem de fapt `g ::  Int -> (Int -> Int)`. Așadar, funcția `g` primește și ea un singur argument (un număr întreg) și returnează o *funcție* de tipul `Int -> Int`

```
*Main> :t (g 2)
(g 2) ::  Int -> Int
```

Mai departe, această nouă funcție primește ca argument un număr întreg și returnează tot un număr întreg:

```
*Main> :t ((g 2) 3)
((g 2) 3) ::  Int
```

Funcția `g` este forma *curried* a funcției `f`. Această formă este preferată în Haskell deoarece ea permite aplicarea parțială a funcțiilor. În Haskell toate funcțiile sunt considerate ca fiind în forma *curried*.

Exercițiul 1.1. Scrieți varianta *curried* pentru funcția:

```
addThree ::  (Int, Int, Int) -> Int
addThree (x,y,z) = x + y + z
```

2 Funcții de ordin superior

Am văzut în secțiunea precedentă că funcțiile în Haskell pot returna alte funcții. Mai mult, funcțiile pot primi ca argumente alte funcții. Funcția `process` de mai jos primește ca argumente o funcție de tipul `Int -> Int` și un număr întreg. Ea aplică funcția dată ca argument peste numărul întreg și returnează un alt număr întreg:

```
process ::  (Int -> Int) -> Int -> Int
process f x = f x
```

Un posibil apel al funcției este:

```
*Main> process (+ 2) 4
6
```

O altă posibilitate de a defini funcția `process` este următoarea:

```
process ::  (a -> a) -> a -> a
process f x = f x
```

Aici, `a` este o *variabilă de tip* care poate fi instanțiată cu orice tip:

```
*Main> process (+ 2) 4
6
Main> process (&& True) False
False
```

Funcția este apelată prima dată pe argumente cu tipurile `Int -> Int` și `Int`. La al doilea apel, argumentele au tipurile `Bool -> Bool` și `Bool`.

Exercițiul 2.1. Scrieți o funcție care are tipul `(Int -> Int) -> Int -> Int -> Int` și aplică funcția de tipul `Int -> Int` tuturor valorilor cuprinse între două numere întregi date ca argument. Funcția va returna suma valorilor obținute.

Exercițiul 2.2. Scrieți o funcție care returnează compunerea a două funcții.

Exercițiul 2.3. Scrieți o funcție care primește ca parametru o listă de funcții și returnează compunerea lor.

Exercițiul 2.4. Scrieți o funcție care calculează suma elementelor dintr-o listă. Utilizați tipul de date pentru liste predefinit în librăria standard.

Exercițiul 2.5. Scrieți o funcție care aplică o funcție fiecărui element al unei liste și returnează lista obținută.

Exercițiul 2.6. Scrieți o funcție care va returna lista elementelor pentru care o funcție de tipul `a -> Bool` returnează `True`.

Exercițiul 2.7. Scrieți o funcție care implementează comportamentul fold `(foldr, foldl)` pe lista definită în laboratorul anterior.

Exercițiul 2.8. Scrieți trei funcții, care primesc ca parametri de intrare, rădăcina unui arbore binar de căutare și o funcție (`f`), care va fi aplicată fiecărui nod în manieră `preordine, postordine, inordine`. Funcțiile vor returna o listă cu rezultatele aplicării funcției `f`. Utilizați structura de arbore binar de căutare definită în laboratorul anterior.

Exercițiul 2.9. Pornind de la exercițiul anterior, scrieți o **singură** funcție de parcurgere pentru un arbore binar de căutare, care să primească strategia de parcurgere (inordine, postordine, preordine, orice-ordine) sub forma unei funcții.

3 Sortare prin comparare

Exercițiul 3.1. Implementați un algoritm de sortare prin comparare care primește ca argumente:

1. o listă `::  [a]` de elemente de ordonat;
2. o funcție `::  a -> a -> Bool` de comparare a două elemente.

Puteți alege ce algoritm de sortare doriți. Nu vă concentrați pe eficiență. Alegeți semnificația funcției de comparare într-un mod rezonabil.

Exercițiul 3.2. Implementați `http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Either.html`.

Exercițiul 3.3. Implementați arborele de căutare binar, discutat la laboratorul anterior, astfel încât să poată conține orice tip de date (care face parte din clasa `Ord`).

Exercițiul 3.4. Scrieți câte o funcție care rezolvă problema căutării (secvențială, binară), în mod clasic și utilizând `foldr/foldl`.

4 Bonus: TABA

TABA (there and back again) este o paradigmă de programare care permite scrierea unor funcții într-un mod mai eficient decât în mod obișnuit, prin evitarea construirii unor structuri de date suplimentare.

Exercițiul 4.1. Scrieți o funcție `fromend` care primește o listă `L` și un număr natural `n` și calculează al `n`-lea element al listei `L`, numărând de la sfârșit spre început.

```
> fromend [1, 7, 5] 0
   Just 5
   > fromend [1, 7, 5] 1
   Just 7
   > fromend [1, 7, 5] 100
   Nothing
```

Exercițiul 4.2. Scrieți o funcție `convolute` care primește două liste `L1` și `L2` și construiește convoluția lor, cu lista `L2` inversată.

```
> convolute [1, 7, 5] [1, 2, 3]
   [(1, 3), (7, 2), (5, 1)]
```

Iată o metodă de a scrie o funcție similară cu `fromend`, care are avantajul că nu efectuează decât o singură parcurgere a listei.

```
fromendaux :: [a] -> Int -> (a, Int)

fromendaux [x] index = (x, 0)
fromendaux (x:xs) index = let (x', index') = fromendaux xs index in
```

```
                              if index' == index then
                                (x', index')
                              else
                                (x, index' + 1)

fromend :: [a] -> Int -> Maybe a
fromend []  = Nothing
fromend (x:xs) index = let (x', index') = fromendaux (x:xs) index in
                           if index == index' then
                             Just x'
                           else
                             Nothing
```

Exercițiul 4.3. Scrieți o implementare a funcției `convolute` care funcționează în mod similar, printr-o singură parcurgere.

Exercițiul 4.4. Exprimați cele două funcții (`convolute` și `fromend`) cu ajutorul unui `fold`. Rezultatul întors de `foldl` poate fi postprocesat (în $O(1)$).