Laboratorul 2: Elemente de limbaj. Funcții

Elemente de limbaj

Există numeroase biblioteci utile de Haskell. Puteți găsi informații despre ele în **Hoogle**: https://hoogle.haskell.org/

Căutați în **Hoogle** funcția `head`. Observați că se găsește în mai multe biblioteci, printre care `Prelude` și `Data.List`.

1. Să presupunem că vrem să generăm toate permutările unei liste. Căutați în **Hoogle** folosind cuvântul-cheie `permutation` (sau ceva asemănător).

Printre rezultatele întoarse, se află și funcția `permutations` din biblioteca `Data.List`. Dați click pe numele funcției (sau al bibliotecii) pentru a citi mai multe detalii. Pentru a folosi funcția în interpretor, va trebui să încărcați biblioteca `Data.List` folosind comanda `import`:

```
Prelude> :t permutations
<interactive>:1:1: error: Variable not in scope: permutations
Prelude> import Data.List
Prelude Data.List> :t permutations
permutations :: [a] -> [[a]]
Prelude Data.List> permutations [1,2,3]
[[1,2,3],[2,1,3],[3,2,1],[2,3,1],[3,1,2],[1,3,2]]
Prelude Data.List> permutations "abc"
["abc","bac","cba","bca","cab","acb"]
```

Atenție! Funcția `permutations` întoarce o listă de liste.

Eliminați biblioteca folosind

```
Prelude> :m - Data.List
```

Bibliotecile se includ în fișiere sursă folosind comanda `import`. Deschideți fișierul `PF2024-Lab2.hs` și adăugați linia de mai jos la începutul său:

```
import Data.List
```

Încărcați fișierul în interpretor și evaluați:

```
*Main> permutations [1..myInt]
```

Ce se întâmplă? Hint: `[1..myInt]` este lista `[1,2,3,..., myInt]` care are multe elemente. (Întrebare bonus: de câte caractere este nevoie pentru a afișa toate elementele listei?)

Putem opri evaluarea unei expresii folosind `Ctrl+C`.

2. Căutați funcția `subsequences` în biblioteca `Data.List`, înțelegeți ce face și testați-o folosind câteva exemple.

Indentare

În Haskell se recomandă ***indentarea*** riguroasă a codului sursă. În anumite situații, nerespectarea regulilor de indentare poate provoca erori la încărcarea programului.

3. Modificați indentarea funcției `double` din fișierul `PF2024-Lab2.hs`. De exemplu:

```
double :: Integer -> Integer
  double x = x+x
```

Reîncărcați programul. Ce observați?

Atenție! În unele editoare se recomandă înlocuirea tab-urilor cu spații.

4. Definiți funcția `maxim`:

```
maxim :: Integer -> Integer -> Integer
maxim x y = if (x > y) then x  else y
```

O variantă cu indentare este:

```
maxim :: Integer -> Integer -> Integer
maxim x y =
    if (x > y)
        then x
        else y
```

Dorim acum să scriem o funcție care calculează maximul a trei numere. Evident, o variantă este:

```
maxim3 x y z = maxim x (maxim y z)
```

Scrieți funcția `maxim3` fără a folosi `maxim`, utilizând direct `if` și indentări.

O altă posibilitate ar fi să scriem funcția `maxim3` folosind expresii `let...in` astfel:

```
maxim3 x y z = let u = (maxim x y) in (maxim u z)
```

Atenție! Expresia `let...in` creează un domeniu de vizibilitate local.

O variantă cu indentări este:

```
maxim3 x y z =
    let
        u = maxim x y
    in
        maxim u z
```

Scrieți o funcție `maxim4` folosind `let..in` și indentări.

Scrieți o funcție care testează funcția `maxim4` prin care să verificați că rezultatul este mai mare (`>=`) decât fiecare din cele patru argumente. (hint: operatorii logici în Haskell sunt ||, &&, `not`).

Citiți mai multe despre indentare: https://en.wikibooks.org/wiki/Haskell/Indentation

Tipuri de date

5. Din exemplele de până acum ați putut observa că în Haskell:

a) există tipuri predefinite: `Integer`, `Bool`, `Char`

b) se pot construi tipuri noi folosind `[...]`

```
*Main> :t [1..myInt]
[1..myInt] :: [Integer]
```

```
Prelude> :t "abc"
"abc" :: [Char]
```

`[a]` este tipul *listă de date de tip* `a`. Tipul `String` este un sinonim pentru `[Char]`.

c) Ați întâlnit tipul `Bool` și valorile `True` și `False`. În Haskell tipul `Bool` este definit astfel:

```
data Bool  = False | True
```

În această definiție, `Bool` este un *constructor de tip*, iar `True` și `False` sunt *constructori de date*.

d) Sistemul tipurilor în Haskell este mult mai complex. Fără a încărca fișierul `PF2024-Lab2.hs`, definiți direct in `ghci` funcția `maxim`:

```
Prelude> maxim x y = if (x > y) then x else y
```

Cu ajutorul comenzii :t aflați tipul acestei funcții. Ce observați?

```
Prelude> :t maxim
maxim :: Ord p => p -> p -> p
```

Răspunsul primit trebuie interpretat astfel: `p` reprezintă un tip arbitrar înzestrat cu o relație de ordine, iar funcția `maxim` are două argumente de tip `p` și întoarce un rezultat de tip `p`.

Așadar, tipul unei operații poate fi definit de noi sau poate fi dedus automat. Vom discuta mai multe despre tipuri în cursurile și laboratoarele următoare.

Funcții

6. Scrieți următoarele funcții:
   a) o funcție cu doi parametri care calculează suma pătratelor lor;
   b) o funcție cu un parametru ce întoarce stringul "par" dacă parametrul este par și "impar" altfel;
   c) o funcție care calculează factorialul unui număr;
   d) o funcție care verifică dacă primul parametru este mai mare decât dublul celui de-al doilea parametru;
   e) o funcție care calculează elementul maxim al unei liste.
7. Scrieți o funcție `poly` cu patru argumente de tip `Double` (`a,b,c,x`) care calculează `a*x^2+b*x+c`. Scrieți și signatura funcției (`poly :: ??`).
8. Scrieți o funcție `eeny` care întoarce stringul "eeny" atunci când primește ca input un număr par și "meeny" când primeste ca input un număr impar. Hint: puteți folosi funcția `even`, despre care puteți citi pe https://hoogle.haskell.org/).

```
eeny :: Integer -> String
eeny = undefined
```

9. Scrieți o funcție `fizzbuzz` care întoarce "Fizz" pentru numerele divizibile cu 3, "Buzz" pentru numerele divizibile cu 5 și "FizzBuzz" pentru numerele divizibile cu ambele. Pentru orice alt număr întoarce șirul vid. Scrieți două definiții pentru funcția `fizzbuzz`: una folosind `if` și una folosind gărzi (condiții). Hint: pentru a calcula restul împărțirii unui număr la un alt număr puteți folosi funcția `mod`.

```
fizzbuzz :: Integer -> String
fizzbuzz = undefined
```

Recursivitate

Una din diferențele dintre programarea declarativă și cea imperativă este modalitatea de abordare a problemei iterării: în timp ce în programarea imperativă folosim bucle (`while`, `for`, ...), în programarea declarativă folosim conceptul de recursivitate.

Un avantaj al folosirii recursivității este acela că ușurează sarcina de scriere și de verificare a corectitudinii programelor prin raționamente de tip inductiv: construim rezultatul pe baza rezultatelor unor subprobleme mai simple (aceeași problemă, dar pentru date de dimensiune mai mică).

Un exemplu simplu este calcularea unui element de la o poziție dată din secvența numerelor Fibonacci, definită recursiv astfel:

$$F_n = \begin{cases} n & \text{dacă } n \in \{0, 1\} \\ F_{n-1} + F_{n-2} & \text{dacă } n > 1 \end{cases}$$

Putem transcrie această definiție în Haskell:

```
fibonacciCazuri :: Integer -> Integer
fibonacciCazuri n
    | n < 2     = n
    | otherwise = fibonacciCazuri (n - 1) + fibonacciCazuri (n - 2)
```

Alternativ, putem da o definiție în stil ecuațional (cu șabloane):

```
fibonacciEcuational :: Integer -> Integer
fibonacciEcuational 0 = 0
fibonacciEcuational 1 = 1
fibonacciEcuational n =
    fibonacciEcuational (n - 1) + fibonacciEcuational (n - 2)
```

10. Numerele `tribonacci` sunt definite astfel:

$$T_n = \begin{cases} 1 & \text{dacă } n = 1 \\ 1 & \text{dacă } n = 2 \\ 2 & \text{dacă } n = 3 \\ T_{n-1} + T_{n-2} + T_{n-3} & \text{dacă } n > 3 \end{cases}$$

Implementați funcția `tribonacci` dând o definiție bazată pe cazuri și una ecuațională, cu șabloane.

```
tribonacci :: Integer -> Integer
tribonacci = undefined
```

11. Scrieți o funcție recursivă care calculează coeficienții binomiali. Coeficienții sunt determinați folosind următoarele ecuații (pentru orice întregi n, k astfel încât $1 \leq k < n$):

```
B(n,k) = B(n-1,k) + B(n-1,k-1)

B(n,0) = 1

B(0,k) = 0
```

```
binomial :: Integer -> Integer -> Integer
binomial = undefined
```

Citiți, citiți, citiți!

- Citiți capitolul *Starting Out* din M. Lipovaca, Learn You a Haskell for Great Good! http://learnyouahaskell.com/starting-out