Introduction to Haskell
Lab 1

1 Preparing the work environment.

1.1 Installation

Installation of the Haskell environment can be done by following the instructions here: `https://www.haskell.org/ghcup/install/`.

1.2 Editor

For Haskell you can use any regular text editor. It is advisable to choose a familiar text editor that provides syntax coloring, alignment, formatting code, etc. VSCode is okay.

2 The `ghci` Interpreter

Along with the installation of the Haskell platform, two executables are also installed that we will use intensively: `ghc` and `ghci`. The first executable (`ghc`) is a compiler for the Haskell language, and its name is short for *Glasgow Haskell Compiler*. The second executable (`ghci`) is an interpreter for Haskell.

To begin with, we use Haskell in interactive mode, that is, the `ghci` interpreter that we run directly from the command line like this:

```
cmd> ghci
GHCi, version 8.6.3:  http://www.haskell.org/ghc/ :?  for help
Prelude>
```

Immediately after executing the `ghci` command, the current version is displayed and a prompt starts where we can enter new commands. For example, a very useful first command is the one that displays all the commands we can execute:

```
cmd> :?
Commands available from the prompt:
...
```

To exit `ghci` we will use the command:

```
cmd> :quit
```

We can sometimes, for certain commands, use the short version:

```
cmd > :q
```

Another very useful command is `:!CMD` where `CMD` is a command that can be run directly in the terminal. For example, `:!ls` on a unix system will display the contents of the current directory. On Windows, the `:!dir` command will have the same behavior.

As we use `ghci`, we will learn more such commands. Note that all these commands are preceded by ":".

2.1 Expression Evaluation

In `ghci` we can evaluate expressions in a simple way.

Exercițiul 2.1. Evaluate in `ghci` the following expressions:

```
cmd > 2
cmd > 2 + 3
cmd > 2 + 3 * 5
cmd > (2 + 3) * 5
cmd > 3 / 5
cmd > 45345345346536 * 54425523454534333
cmd > 3 / 0
cmd > True
cmd > False
cmd > True && False
cmd > True || False
cmd > not True
cmd > 2 <= 3
cmd > not (2 <= 3)
cmd > (2 <= 3) || True
cmd > "aaa" == "aba"
cmd > "aba" == "aba"
cmd > "aaa" ++ "aba"
```

As can be seen in Exercise 2.1, the syntax of the expressions is the usual one. However, in Haskell, the above expressions can also be written in prefixed form. For example, the expression `2 + 3` can be written as `((+) 2 3)`.

Exercițiul 2.2. Evaluate all the above expressions in prefixed form. Pay attention to the priorities of the operators!

2.2 The `:t` command

A very useful `ghci` command is `:t` or `:type`. This command allows us to find out the type of an expression:

```
cmd > :t True
True ::  Bool

cmd > :t not
not ::  Bool -> Bool
```

Notice that `True` has type `Bool` and `not` has type `Bool -> Bool`, i.e. it takes an argument of type `Bool` and returns a result of type `Bool`.

Exercițiul 2.3. Use `:t` to find the types of expressions: `True`, `False`, `True && False`, `True && (2 <= 4)` .

Exercițiul 2.4. Use `:t` to find out the type of the expression: `"aaa"`. Ask the lab teacher to explain the type shown.

Exercițiul 2.5. Use `:t` to find out the types of expressions: `2`, `2 + 3`, `(+)`. Ask the lab teacher to explain the types shown.

Exercițiul 2.6. Evaluate in `ghci` the expression `not 2`. What do you get?

Evaluating the expression `not 2` in Exercise 2.6 produces an error:

```
<interactive>:42:5:  error:
  • No instance for (Num Bool) arising from the literal '2'
  • In the first argument of 'not', namely '2'
    In the expression:  not 2
    In an equation for 'it':  it = not 2
```

The error tells us that the argument type for `not` is not the expected one, i.e. a boolean argument.

Exercițiul 2.7. Use the command `:t` to find the type of `not` and then the type of the argument `2`. What do you notice?

3 Functions and function calls.

3.1 Function calls.

If you solved Exercise 2.2 you already learned how to call functions in Haskell. The addition operation `(+)` is a function. Calling this function is done like this: on the first position we put the name of the function, and on the following positions we find the arguments separated by spaces. So the call is `((+) 2 3)`.

Exercițiul 3.1. In Haskell there are already predefined functions: `succ` – which calculates the successor of a number, `pred` – which calculates the predecessor of a number, `max` – which calculates the maximum between two numbers, `min` – which calculates the minimum of two numbers. Use the `:t` command to find out the types of these functions. Call all these functions in `ghci` and check if you get the correct output.

3.2 Defining functions.

The syntax for defining functions in Haskell is very simple and we will explain it with an example:

```
id x = x
```

The above function is the identity function. The name of the function is `id`, the name of the argument is `x`, and after the symbol `=` is the body of the function.

Exercițiul 3.2. Write the above function in `ghci` and call the function.

The function that calculates the sum of three numbers can be defined as follows:

```
sumThree x y z = x + y + z
```

Exercițiul 3.3. Write the above function in `ghci` and call the function.

Exercițiul 3.4. Write a function that calculates the product of three numbers and test the function in `ghci`.

Because it is more difficult to edit functions on the command line, we prefer to write the code in files. A Haskell file usually has the extension `.hs`.

Exercițiul 3.5. Create a file that we'll call `functii.hs` that will contain the definitions of the `id` and `sumThree` functions (defined above).

To load this file into `ghci`, we will use the following command line:

```
cmd > ghci functions.hs
GHCi, version 8.6.3:  http://www.haskell.org/ghc/ :?  for help
[1 of 1] Compiling Main ( functii.hs, interpreted )
Ok, one module loaded.
Main>
```

Alternatively, we can load the file into `ghci` using the `:l` or `:load` command:

```
cmd > ghci
GHCi, version 8.6.3:  http://www.haskell.org/ghc/ :?  for help
Prelude> :l functii.hs
[1 of 1] Compiling Main ( functii.hs, interpreted )
Ok, one module loaded.
```

After any change we make to the file, it must be reloaded using the command `:r` or `:reload`:

```
*Main> :r
[1 of 1] Compiling Main ( functii.hs, interpreted )
Ok, one module loaded.
```

Exercițiul 3.6. Again call the `id` and `sumThree` functions which are now defined in the `functions.hs` file.

When writing functions in Haskell, it is recommended that we also write the type of the functions to be sure that they will only be called on the arguments that we intend to process in that function. The Haskell language comes with a type inference mechanism. For example, if we don't explicitly specify the type of a function, that mechanism uses the information it has in the body of the function to calculate the type of the function.

Exercițiul 3.7. What type is the `sumThree` function? Discuss with the lab teacher how the type of the function was inferred. Call the function over the arguments `3.2`, `2` and `4`.

We explicitly specify the type of the `sumThree` function like this:

```
sumThree ::  Int -> Int -> Int -> Int
sumThree x y z = x + y + z
```

Exercițiul 3.8. What type does `ghci` display for the `sumThree` function now? Call the function over the arguments `3.2`, `2` and `4`. What happened?

Next, we define a function that calculates the maximum between two numbers:

```
myMax ::  Int -> Int -> Int
myMax x y = if x <= y then y else x
```

Exercițiul 3.9. What type is the function `myMax`? Test the function in `ghci`.

Exercițiul 3.10. Define a function that calculates the maximum of 3 integers and test the function in `ghci`.

3.3 Funcții recursive.

Așa cum era de așteptat, în Haskell putem defini funcții recursive. Funcția de mai jos calculează pentru un număr dat suma numerelor naturale până la acel număr. În cazul în care argumentul este un număr negativ, funcția va returna valoarea 0.

```
mySum ::  Int -> Int
mySum x = if x <= 0 then 0 else x + mySum (x - 1)
```

Exercițiul 3.11. Testați funcția `mySum` în `ghci`.

Exercițiul 3.12. Definiți o funcție recursivă care returnează elementul de pe poziția dată ca argument din șirul lui Fibonacci.

Exercițiul 3.13. Definiți o funcție recursivă care returnează cel mai mare divizor comun a două numere.

3.4 Recursive functions.

As expected, in Haskell we can define recursive functions. The function below calculates for a given number the sum of the natural numbers up to that number. If the argument is a negative number, the function will return the value 0.

```
mySum ::  Int -> Int
mySum x = if x <= 0 then 0 else x + mySum (x - 1)
```

Exercițiul 3.14. Test the `mySum` function in `ghci`.

Exercițiul 3.15. Define a recursive function that returns the element at the given position as argument in the Fibonacci sequence.

Exercițiul 3.16. Define a recursive function that returns the greatest common divisor of two numbers.