Graded Session Haskell: My Own Programming Language

NAAM: RICHTING:

Some practicalities

- You get two hours two solve this exercise **individually**.
- You can only use the Haskell slides (possibly in printed format with handwritten notes) and the exercices which are available for this course through Toledo. You can also use the manuals which are mentioned on Toledo, and Hoogle.
- In the folder **1415_Graded/Haskell_thursday** on Toledo, you can find the files `MOPL.hs` and `MOPLtest.hs`. The submission module is also there.
  - **Download and open** the file `MOPL.hs`, inhere there is already a template for your solution.
  - As first things you enter your name, student number, and your student program.

    ```
    -- Jan Jansen
    -- r0123456
    -- master cw
    ```
  - You can import extra functions and types.
  - For each exercise there are a few predefined function with accompanying type signature. This signature cannot be changed. Replace in each exercise "undefined" with your implementation. You can put arguments before the equality sign. It is allowed to write extra (helper)functions.
  - It is possible to test your own solution with the help of `MOPLtest.hs`. you can do this by running the next commando in the folder where the `.hs` files are present:

    ```
    runhaskell MOPLtest.hs
    ```
  - When the tests succeed, it doesn't necessarily mean that your program is completely correct or that you deserve the maximum of the points.
  - After two hours, or whenever you are ready, you can submit the file `MOPL.hs` via Toledo.

Exercise

In this exercise we will develop a new programming language: My Own Programming Language (MOPL). This programming language is very simple and contains only 2 types of statements: the assignment of variables and print statements. We will firstly develop the assignments, and thereafter extend the language with print statements.

A program will be represented as a list of statements. This is an example of a valid program:

```haskell
program :: [Statement]
program = [
    assign  "a" (intTerm 8), --a = 8
    printTerm (plus (varTerm "a") (intTerm (-5))), --print(a-5)
    assign "b" (plus (varTerm "a") (intTerm 2)), --b = a+2
    assign "a" (plus (varTerm "a") (varTerm "b")), -- a = a+b
    printTerm (varTerm "a") -- print(a)
    ]
```

When this program is executed, the variable `a` firstly gets the value 8, then the value of `a-5` is printed, so the program prints `3`. After this, `a+2` is assigned to `b`, so `b` gets the value 10, after this `a+b` is assigned to `a`, so `a` becomes 18, and finally `a` gets printed, so the program prints `18`.

```
Main> execute program
3
18
```

We build the programming language step by step. Don't let the number of exercises overwhelm you, each exercise can be solved in a few lines of code.

Exercise 1: Data types for MOPL
Define the data types `Term` and `Statement`. A statement is either an assignment of a term to a variable, or a print statement of a term. A term is either a variable (represented by a `String`), an integer, or an application of a binary operation on 2 terms. In this language we will support addition, subtraction and multiplication, but choose your datatypes so that any binary function over the integers is representable without changing your data type.

Exercise 2: Helper functions for MOPL
To be able to work with the datatypes from exercise 1 more easily, it is a good idea to make functions that help us to write programs more easily. Implement the following functions [1]:

```
assign :: String -> Term -> Statement
printTerm :: Term -> Statement
intTerm :: Int -> Term
varTerm :: String -> Term
plus :: Term -> Term -> Term
times :: Term -> Term -> Term
minus :: Term -> Term -> Term
```

Exercise 3: The state of a MOPL program
A program that is being executed, has a state at every statement. This state consists of the variables and their value. We will represent a state using a list of tuples `[(String,Int)]`.

- Write a function `valueOf :: [(String,Int)] -> String -> Int`, that, for a given String, extracts the first integer out of the list which has that string as a first element in the tuple. You can assume that there is such a tuple at all times.

  Some examples:

  ```
  Main> valueOf [("a",5),("b",9),("c",10)] "b"
  9
  Main> valueOf [("a",1),("a",2),("a",3)] "a"
  1
  ```

- Write a function `insertS :: String -> Int -> [(String,Int)] -> [(String,Int)]`, which adds/changes a tuple in the state. The tuple of that string and integer is added to the state, and if any tuples with the same string exist, they are removed from the list. The order of the resulting list is not important.

  Some examples:

[1] Hint: If you chose your datatypes well, the implementation of each of these functions should contain less than 20 characters

```
Main> insertS "a" 1 []
[("a",1)]
Main> insertS "b" 3 [("a",5),("b",9),("c",10)]
[("a",5),("b",3),("c",10)]
```

Exercise 4: The evaluation of a term
Before being able to execute a whole program, we firstly have to be able to evaluate a term, starting from a certain state. Write the function `evalTerm :: [(String,Int)] -> Term -> Int`, which does exactly that.

```
Main> evalTerm [] (intTerm 5)
5
Main> evalTerm [("a",6)] (varTerm "a")
6
Main> evalTerm [("a",6)] (times (varTerm "a") (intTerm 2))
12
```

Exercise 5: The execution of an assignment
Write the function `execAssign :: String -> Term -> [(String,Int)] -> [(String,Int)]`, which converts the state before an assignment, to the state after an assignment.

```
Main> execAssign "b" (intTerm 6) []
[("b",6)]
Main> execAssign "b" (minus (varTerm "b") (varTerm "a") ) [("a",3),("b",8)]
[("a",3),("b",5)]
```

Exercise 6: The execution of a sequence of statements
Write the function `execPure :: [(String,Int)] -> [Statement] -> [(String,Int)]`, which converts the state of the program, before a list of statements, to the state of the program after these statements, by executing all all the statements. The print statements have no effect on the state of the program, so you can ignore those.

Exercise 7: The full language
Write the function `execute :: [Statement] -> IO ()` which executes a program completely, as in the introduction of this exercise, starting from the empty state.