INF-122 Compulsory assignment 1, fall 2020

- Read the entire assignment before you start working out your solutions.
- It is allowed to discuss the approach and possible solutions, but everybody has to write an independent solution on his/her own. Suspicious repetitions of code pieces in distinct answers will qualify as a failure, for all persons sharing them. (What counts as a suspicious repetition is decided by the person evaluating the solutions.)
- Questions about the assignment can be asked at the at the workshops 28-29.9.2020.
- Submit your solutions (using "Lever" button under Oblig1) not later than **Friday, 2th October 2020, at 12:00**. Solutions submitted after that will not be considered.
- Solution must be submitted as a plain-text, interpretable file with the name **Oblig1.hs** (both the name Oblig1 and the suffix .hs are essential). At the end of this assignment, you can find a template for your solution. Copy it or download the file `Oblig1.hs` (from the same catalogue where this text is found) and start writing your solutions under `import Data.Char`.
- Your program must not import any other modules than `Data.Char`.

Introduction: a simple word construction language

Consider the following grammar for a simple, infix word construction language:

```
Expr -> Term + Expr | Term - Expr | Term
Term -> Number * Term | Word
Word -> Letter Word | Letter
Number -> Digit Int | Digit
Letter -> a | b | c | ...| x | y | z | A | B | ...| Y | Z
Digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The start symbol is `Expr`; the terminal ones are digits, letters, +, -, * and space. Any number of whitespaces can be used between any two tokens, e.g. `"2*a+b"`, and `"23 + dd* bq"` are both valid expressions. The expression `5 *` is not correct, missing another `Expr`. Zeros are allowed in front of numbers, e.g., `0012` is a valid representation of 12.

You will parse and evaluate expressions in this language. Each arithmetic operator, `+`, `-` and `*`, has a specific interpretation:

- `"ab + bc"` should be evaluated to `"abbc"`, i.e. concatenation of strings.
- `"cabbage - aabcd"` should be evaluated to `"bge"`, i.e. for each letter in the second expression, remove the first occurrence of it in the first expression. Duplicate letters are considered unique, see `a` in the example. This is akin to `diff` from the weekly exercises.

- `"3 * hei"` should be evaluated to `"heiheihei"`, i.e. replicating the second term by the amount given by the first term.

Note that, based on the grammar, `*` has the **highest precedence**, while `-` and `+` associate to the **right**. This is, of course, not the familiar way of doing this from mathematics. That is, the expression `"abc - b + c"` is parsed as `abc - (b + c)` and evaluates to `"a"`, while the expression `"a-2*a+3*c"` is parsed as `a - ((2 * a) + (3 * c))` and evaluates to `""`.

Important:

- Don't change the grammar, even if you don't like it.
- We will be testing your solution by running some automated tests. Therefore, names and types of data constructors and functions which are specified in the text below **must be exactly the same** in your solution. Problems from having the wrong names or types, will result in **points being deducted**! Use the provided template.

1 parse :: String -> Ast

You are not supposed to use the `Parsing` module (Hutton, chapter 13). If you do, all shortcomings (if any) of your solution will be judged much more severely, than if you used recursive descent parsing from the lectures.

Write a function `parse::String->Ast` which takes an expression (a string in the language of `Expr`) as an argument and produces an abstract syntax tree (AST) for it if it is correct, and raises an `error` when it is not. The datatype for ASTs must have the following format and names of constructors:

```
data Ast
  = Word String
  | Num Int
  | Mult Ast Ast
  | Plus Ast Ast
  | Minus Ast Ast
  deriving (Eq, Show)
```

For instance:

```
> parse "a + b"
Plus (Word "a") (Word "b")

> parse "a - 2 * b + c"
Minus (Word "a") (Plus (Mult (Num 2) (Word "b")) (Word "c"))
```

Please note that the data constructor `Mult`, which represents `*` in the grammar, allows two Asts of any kind here. This is not, however, allowed in the grammar given in the introduction which restricts the first term to be a number. Extra points will be given if `parse` returns an error for such expressions, e.g. `hello * world`.

Hints

- You can test if a character is a letter or a digit by using the following functions from the module `Data.Char`: `isAlpha:: Char -> Bool`, or `isDigit:: Char -> Bool` .
- You can read in an integer (with leading zeros) using the function `read::String->Int` from the Haskell prelude. To find the integer in a prefix of a string `str`, you can do something like `read (takeWhile isDigit str) :: Int`.
- To do the actual parsing, write function `parseExpr:: [String] -> (Ast,[String])` taking a list of tokens as input and returning an AST fragment together with the rest of the list remaining to be parsed. To parse subexpressions, just call `parseExpr` recursively.
- Your `parse` function will probably be just a wrapper around `parseExpr`. Function `tokenize` from one of the previous weekly exercises can be used to generate input to `parseExpr` from the input string.

2 viss :: Ast -> String, *and* vis :: Ast -> IO ()

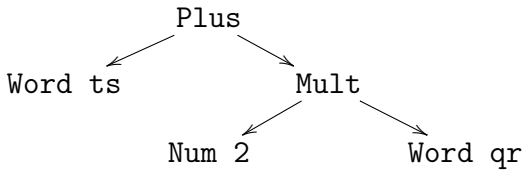
The main objective here is to implement function `vis::Ast->IO ()`, (the type `IO ()` is explained below) which writes the argument Ast on the screen, with each subtree marked by an increasing indentation of ***exactly 3*** spaces. For instance, the result of parsing:

```
> parse "ts + 2 * qr "
   Plus (Word "ts") (Mult (Num 2) (Word "qr")),
```

sent as argument to `vis` is indicated below to the left, with spaces represented by '_' (the signs '_' ***only represent*** spaces and ***shall not appear in the output!***):

```
> vis (Plus (Word "ts") (Mult (Num 2) (Word "qr"))
   Plus
   ___Word ts
   ___Mult
   _____Num 2
   _____Word qr
   >
```

representing the tree:

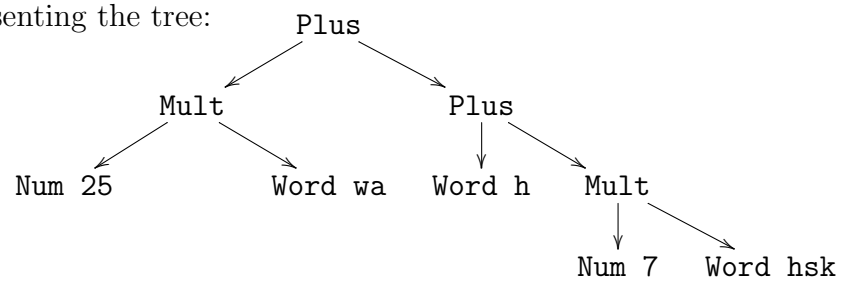



```
> parse "25 * wa + h + 7 * hsk"
   Plus (Mult (Num 25) (Word "wa")) (Plus (Word "h") (Mult (Num 7) (Word "hsk")))
```

```
> vis (Plus (Mult (Num 25) (Word "wa")) (Plus (Word "h") (Mult (Num 7) (Word "hsk"))))
```

```
Plus
__Mult
____Num 25
____Word wa
__Plus
____Word h
____Mult
______Num 7
______Word hsk
>
```

representing the tree:



NB!! The type of the result, `IO ()`, will be described later in the course. Here you shall achieve it by using the built-in function `putStr::String->IO ()`. It writes the string on the screen displaying "\n" as a new line. E.g., given a constant `str = "a\nb\n"`, the following calls from GHCi give the specified results:

```
> str
```
"a\nb\n"

```
> putStr str
  a
  b
```

Do not send the result returned by `putStr` as an argument to any other function, but use `putStr` only as the outermost call in the implementation of `vis`. Write first function `viss::Ast -> String`, generating an appropriate string, such that `vis` can be defined by:

```
vis ast = putStr (viss ast).
```

3 eval :: Ast -> String

You shall program an evaluation function `eval::Ast -> String` which evaluates any Ast into a string based on the rules given in the introduction. For example:

```
> eval (parse "hel + lo")
  "hello"

> eval (parse "hel - lo")
  "he"

> eval (parse "2*a-a")
  "a"

> eval (parse "mi + 2 * la")
  "milala"

> eval (parse "haskell - 1 * laek")
  "hsl"

> eval (parse "5 * ae")
  "aeaeaeaeae"

> eval (parse "hei - h + i")
  "e"

> eval (parse "ost - t - t")
  "ost"
```

Notes

- If you are running the examples above, be wary of consequential errors from using your own implementation of `parse`.
- While the examples above are using `parse`, the first exercise is not a prerequisite for this exercise; we will be testing your implementation of `eval` directly with various Asts.

Remember to include all functions!

Complete the following schema, contained in the file `Oblig1.hs`, which you can copy from MittUiB under the Files tab.

```
-- a comment with your forename + surname and group
module Oblig1 where
import Data.Char

data Ast
   = Word String
   | Num Int
   | Mult Ast Ast
   | Plus Ast Ast
   | Minus Ast Ast
   deriving (Eq, Show)

parse ::  String -> Ast
parse str = undefined

viss ::  Ast -> String
viss ast = undefined

vis ::  Ast -> IO ()
vis ast = putStr (viss ast)

eval ::  Ast -> Int
eval str = undefined
```

If you do not implement some function, retain the appropriate lines in the file you deliver. E.g., if you do not implement function `eval`, retain the lines

```
eval ::  Ast -> Int
eval ast = undefined.
```

In the functions which you do implement, replace `undefined` by your own code.