ISTANBUL TECHNICAL UNIVERSITY
COMPUTER ENGINEERING

BLG 233E DATA STRUCTURES
HOMEWORK – 3

Due: December 18th, 2016.

Summary

Your assignment is to code up an **interpreter for prefix arithmetic expressions**. The interpreter **will use a stack** in its infrastructure and **implement all of the operations** defined in the Implementation section. For each successive evaluation, the result will be **stored in a queue**. Upon termination, the program will print the contents of this queue in order.

Work Flow

1. **Initialize** your stack and the interpreter prompt. The prompt must be displayed on the console and wait for user input.

```
user@ssh ~ $ ./interpreter.exe

Initializing the interpreter...

>> _
```

2. **Scan** the input expression, then **convert** it to postfix notation and **print** the converted expression.

3. **Evaluate** the postfix expression using the stack. If you **detect invalid syntax**, print them on the interpreter console. Otherwise, **store the answer** and **print** it on the console.

4. Display the prompt again to wait for user input, and **return to step 2**. Inputting the string `**exit**' in the prompt should terminate the program.

```
>> sum ( exp 2 3 ) ( sqrt 4 )

Postfix string: 2 3 exp 4 sqrt sum
Answer: 10

>> ans mod 3

Syntax error.

>> mod 3 ans

Postfix string: 3 ans mod
Answer: 1

>> exit

Terminating the interpreter...
Answer queue: 10 -> 1

user@ssh ~ $ _
```

Requirements

1. You must **implement your own stack and queue** structures using linked lists. Use the same stack for both postfix conversion and expression evaluation, and store **`ans`** values in the queue after each evaluation.

2. Assume that all of your tokens (operands, operators, parentheses) are delimited with a **single** space character. You may throw syntax errors for the cases that do not comply with this rule.

3. Throw syntax errors for expressions that do not evaluate to numbers (even if they are not really syntax errors), *e.g.* **`div 3 0`**.

4. Only accept integer operands as input, and store them in variables with the primitive type **`long`**. Make sure that you accept negative values as well.

   For any operation that would yield a fractional output, round the result down to the nearest integer.

5. Use the predefined maximal/minimal values **`LONG_MAX`** and **`LONG_MIN`** from the **`limits`** library to internally store plus and minus infinity, respectively.

   Make sure that expressions that evaluate to infinity always return the designated **`long`** values and display **`inf`** at the output rather than other large **`long`** values.

   For instance, **`div inf 5`** should evaluate to **`inf`**, not **`LONG_MAX/5`**.

6. For operations that accept both unary and binary argument syntax, **look ahead** in the expression string and determine by yourself which version was used before conversion/evaluation.

7. The usage of parentheses for disambiguation is **optional**, but you are still required to properly handle expressions with parentheses.

8. Your program must run in console mode by default when invoked without parameters. This is the mode described in the Work Flow section.

   If the program is invoked with a file name parameter, then it must run in batch mode. In this mode, read the input file line by line, consider each line to be an expression, and then evaluate them. Print only the answer queue on the console.

input.txt

```
1    abs ( product -2 -4 -8 )
2    div ( sum ( exp 2 3 ) ( sqrt ans ) ) 3
3    mod 4 ans
```

```
user@ssh ~ $ ./interpreter.exe input.txt

Answer queue: 64 -> 5 -> 1

user@ssh ~ $ _
```

Implementation

You must implement all of the following variables and operations for your interpreter application. In your implementations, feel free to make use of the `math` library functions such as `pow()` and `log()`.

Static Variables

`ans`	The result of the last evaluation
`inf`	**`LONG_MAX`**
`-inf`	**`LONG_MIN`**

Unary Operations

`abs n`	$\lvert n \rvert$
`sqrt n`	$\lfloor \sqrt{n} \rfloor$

Binary Operations

`sub` $\mathbf{n_1}$ $\mathbf{n_2}$	$n_1 - n_2$
`div` $\mathbf{n_1}$ $\mathbf{n_2}$	$\lfloor n_1 \div n_2 \rfloor$

Unary/Binary Operations

`exp n`	$\lfloor e^n \rfloor$
`exp x n`	$\lfloor x^n \rfloor$
`log n`	$\lfloor \log_e n \rfloor$
`log x n`	$\lfloor \log_x n \rfloor$
`mod n`	$n \ (mod\ 10)$
`mod x n`	$n \ (mod\ x)$

Polynary Operations

sum n_1 n_2 … n_k	$\sum_k n_k$
product n_1 n_2 … n_k	$\prod_k n_k$
min n_1 n_2 … n_k	$\min_k n_k$
max n_1 n_2 … n_k	$\max_k n_k$

Submission

1. Make sure that you write your name and number in all of the files in your project, in the following format:

```
/* @Author
 * Student Name: <student_name>
 * Student ID: <student_id>
 * Date: <date>
 */
```

2. Use comments whenever necessary in your code to explain what you did.

3. **Compile the code in the Secure Shell Client (SSH) before you send the assignment.**

4. After you make sure that everything is compiling smoothly, archive all files into a zip file. Submit this file through `www.ninova.itu.edu.tr`. Ninova enables you to overwrite your submission until the submission deadline, so you can upload preliminary submissions well in advance of the deadline to be safe.

   Do not miss the submission deadline. **Do not** leave your submission until the last minute, where the submission system tends to become less responsive due to high network traffic. **HOMEWORKS SENT VIA E-MAIL WILL NOT BE GRADED**.

Academic dishonesty including but not limited to cheating, plagiarism and collaboration is unacceptable. Your assignments will be checked with plagiarism detection software. Any student found guilty will receive 0 as their grade for the assignment, and be subject to disciplinary actions.

If you have any questions about the assignment, feel free to contact the teaching assistant Umut Sulubacak via e-mail (sulubacak@itu.edu.tr) or on IRC (channel #itu-cs on Freenode).