# Interfaces

## Tokenizer

The tokenizer will be able to read any input written in the traditional mathematical notation, meaning that the multiplicative binary operations (\* / and %) will have a higher precedence than the additive ones (+ and -), excepting those cases in which parenthesis are used. It will be able to read integer number, as well as floating point numbers. The only unary operations allowed will be +, which will not change the value of the number it precedes, and -, which will assign a negative value to the following number.

Examples of valid inputs:

+1-4/2

1.0-2\*-2+(+2)-2

--2+-(4+-+3)---2.5/3

After parsing the input, the tokenizer will write an output to a txt file which will contain the different tokens separated by a newline character (\n). Each token will be described by a type identifier (can be int, fp, op or un), followed by a colon (:) and the “value” of such token afterwards.

### Key table

|  |  |
| --- | --- |
| **Item** | **Token** |
| **Integer number x** | In:x |
| **Floating point number x** | Fp:x |
| **Unary +** | Un:+ |
| **Unary -** | Un:- |
| **Left parenthesis** | Op:( |
| **Right parenthesis** | Op:) |
| **Plus sign** | Op:+ |
| **Minus sign** | Op:- |
| **Multiplication sign** | Op:\* |
| **Division sign** | Op:/ |
| **Modulo sign** | Op:% |

### Example

Output corresponding to the third example of valid input:

Un:-

Un:-

In:2

Op:+

Un:-

Lp:(

In:4

Op:+

Un:-

Un:+

In:3

Rp:)

Op:-

Un:-

Un:-

Fp:2.5

Op:/

In:3

(the first operator will always be considered the binary operator, whereas the following + or – operators will be counted as unary operators which will only modify the value of the integer of floating point number they precede).

## Infix to Postfix converter

Given the output of the tokenizer as the input of this program, the converter will translate it into the postfix notation. This means that the numbers for the operation are listed first, and then the operators in the order of the priority they must be executed (separated by spaces)

Note: Unary operators are interpreted as a binary operation where the first operand is 0 and the second one is the number to which we want to apply such operator. (for example, the integer “-5” will become (0-5).

### Key Table

|  |  |
| --- | --- |
| **Expression** | **Notation** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

### Example

Therefore, taking the example output from the previous section we obtain:

## 0 0 2 - - 0 4 0 0 3 + - + - + 0 0 2.5 3 / - - -

## Code generator

This program will read the output of the previous one.

It will read the postfix notation and will generate an output of an instruction for each element of the equation.

### Key table

|  |  |
| --- | --- |
| **Item** | **Code** |
| **Integer number x** | LDI X |
| **Floating point number x** | LDF X |
| **Plus sign** | ADD |
| **Minus sign** | SUB |
| **Multiplication sign** | MUL |
| **Division sign** | DIV |
| **Modulo sign** | MOD |

### Example

The following text shows the code which would be generated using the previous example as input:

LDI 0

LDI 0

LDI 2

SUB

SUB

LDI 0

LDI 4

LDI 0

LDI 0

LDI 3

ADD

SUB

ADD

SUB

LDI 0

LDI 0

LDF 2.5

LDI 3

DIV

SUB

SUB

SUB

ADD

## Virtual Machine

The Virtual Machine will read the instructions generated in the previous step and it will output the result of computing such operations. Following the previous example, the result would be:

0.1666666