Design of the calculator

Authors:

Andres Otero Garcia, Juan Vercher and Maximilian Feussner

Overview

The objective of this project is to create a working calculator divided into 4 different programs which will work in a sequence to obtain the result (meaning the output from the first program will be the input of the second one and so on). The first one will consist of a tokenizer which will split the input from the user into tokens (integers, floating point numbers, operators...). Then, the infix-to-postfix translator will change the notation to a simpler one, to facilitate the creation of the code in the next step (code generator) which will translate the notation to the instructions, which will be executed by the final program, the Virtual Machine. All these programs will be written in C, without the use of any library apart from the ones included in the language.

Requirement

General requirements:

1. The calculator should calculate with + and -
2. The calculator should calculate with * and /
3. The calculator should calculate with these operators % and ( )
4. The calculator should work with floating-point and integer numbers
5. The calculations should be finished in less than 1 seconds
6. The calculator should take the input as file (textfile)
7. The calculator should be able to calculate 10 different results without crashing
8. The calculator shall support only one inputfile at once and must report the result before taking another input
9. The inputfile shall have only numbers and operators in it
10. When the inputfile contains other characters than stated above, the calculator shall skip these characters
11. When the inputfile contains other characters than stated above, the calculator shall print an errormessage
12. The calculator should consist of four parts: Tokenizer, the infix-to-postfix translator, code generator and the virtual machine
13. Every function of the calculator should have a test suite
14. Every part should have a test suite
15. The calculator should have a test suite

Requirements for the Tokenizer

16. The tokenizer shall be able to process the inputfile
17. The tokenizer shall return a file where all operators and numbers are separated by a new line sign (\n)
18. The tokenizer shall only process the operators specified above and integer or floating-point numbers
19. For every number or operator there should be the “value” (e.g. 1 or +)
20. For every number or operator there shoul the type identifier (e.g. In or Op (for operator))

Requirements for the Infix-to-postfix translator

21. The translator shall process the file from the
22. The translator returns a file where the operators and numbers are in the postfix notation
23. The translator shall be able to process the “value” of the numbers and operators
24. The translator shall be able to process the type identifier of the numbers and operators

Requirements for the code generator

25. The generator shall process the file from the translator
26. The generator shall return a file with instructions
27. The generator shall be able to process the operators and numbers
28. The generator shall translate the numbers and operators to a given instruction set

Requirements for the virtual machine (vm)

29. The vm shall process the file from the generator
30. The vm shall return an integer or floating-point number
31. The vm shall be able to process the instructions
32. The vm shall be able to execute the instructions

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	`Rp:(`
Right parenthesis	`Lp:)`
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:-
int:2
op:+
un:-
Lp:(
int:4
op:+
un:-
un:+
int:3
Rp:)
op:-
un:-
un:-
fp:2.5
```

```
op:/
int: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
$\boldsymbol{x + y}$	$x\ y +$
$\boldsymbol{x - y}$	$x\ y -$
$\boldsymbol{x * y}$	$x\ y *$
$\boldsymbol{x/y}$	$x\ y/$
$\boldsymbol{x\%y}$	$x\ y\ \%$
$\boldsymbol{-x}$	$0\ x -$
$\boldsymbol{+x}$	$0\ x +$
$\boldsymbol{x\ op\ y\ op\ z}$	$x\ y\ op\ z\ op$
$\boldsymbol{x\ op\ (y\ op\ z)}$	$x\ y\ z\ op\ op$

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
```

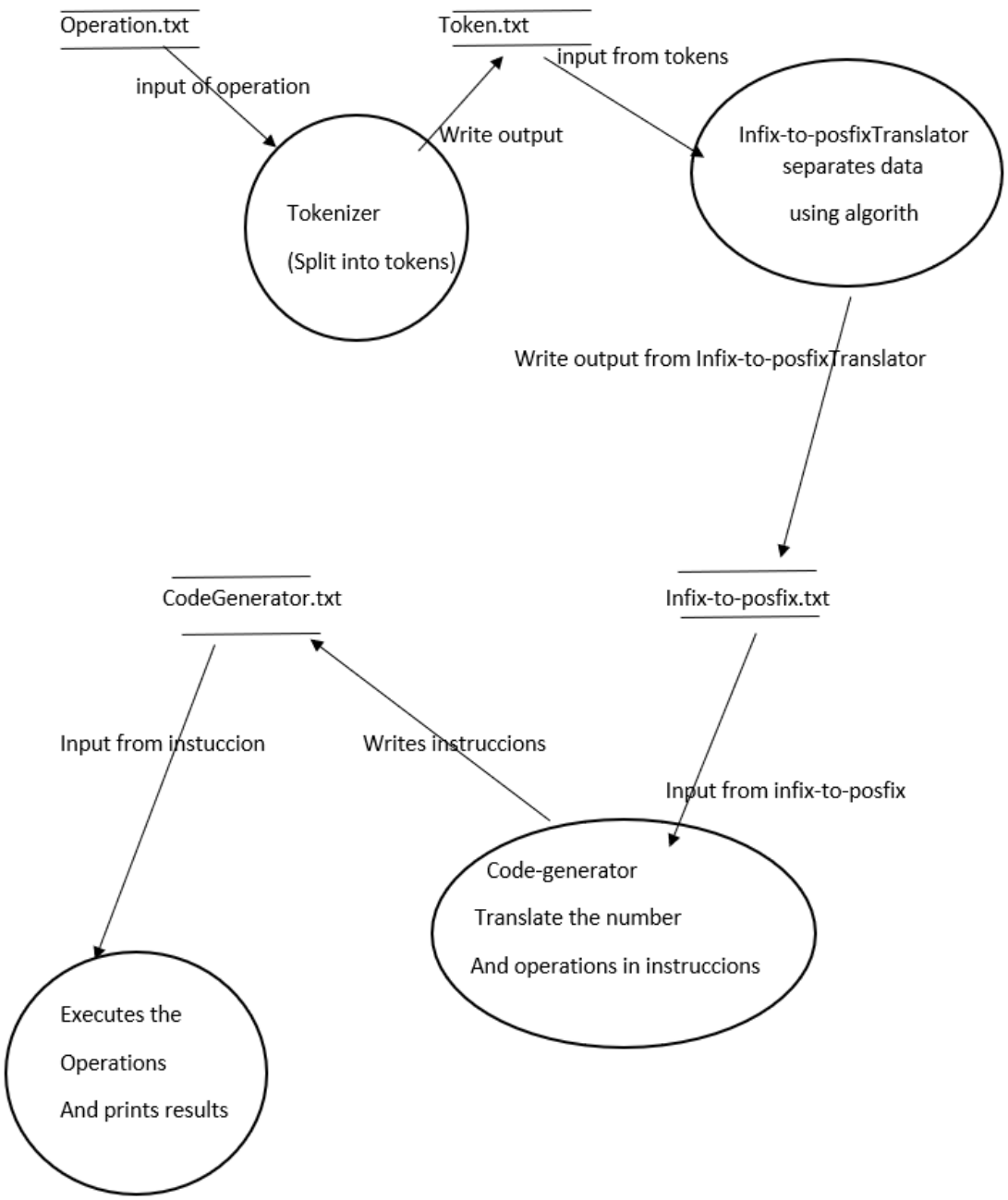
```
LDI 0
LDI 2
SUB
SUB
LDI 0
LDI 4
LDI 0
LDI 0
LDI 3
ADD
SUB
ADD
SUB
LDI 0
LDI 0
LDF 2.5
LDI 3
DIV
SUB
SUB
SUB
ADD
```

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
```

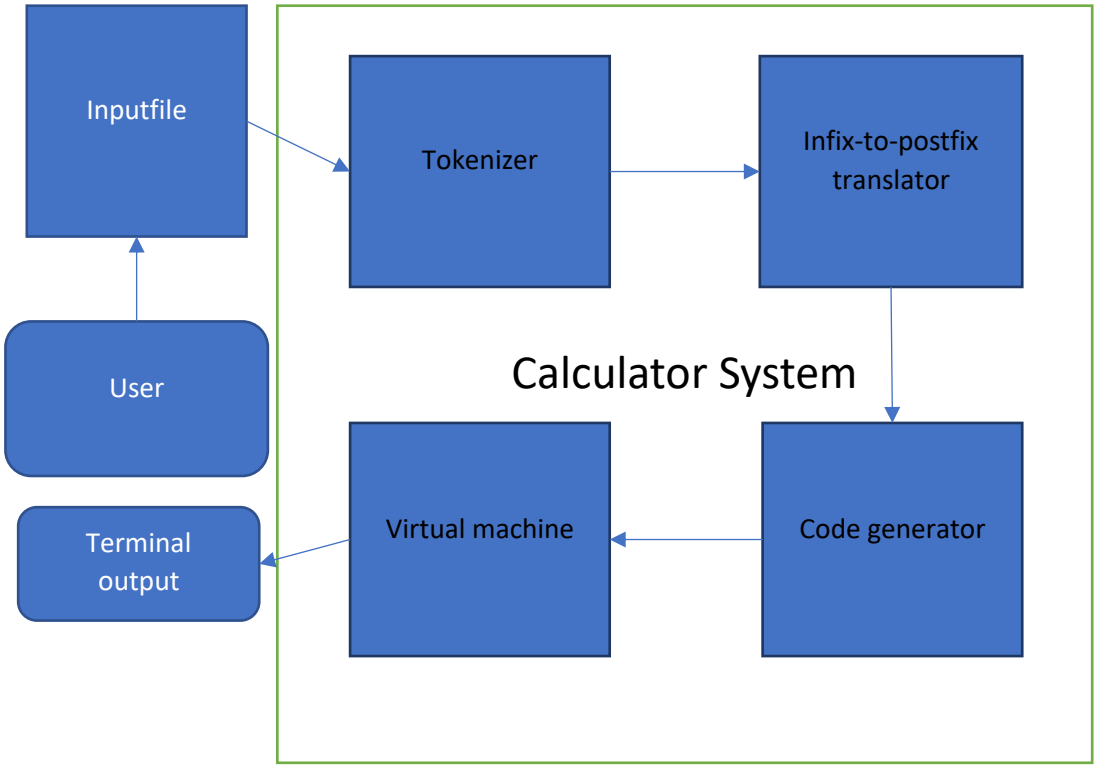
DataFlow Diagram



Graphic 1: DataFlow diagram with customer and the 4 parts of the calculator

The first graphic shows the Dataflow from the calculator. First a customer or a user writes his problem in the inputfile. The inputfile is a textfile which path gets hardcoded in the calculator. After the start of the calculator the 4 parts work together to get the result as described in the Interface chapter or the overview.

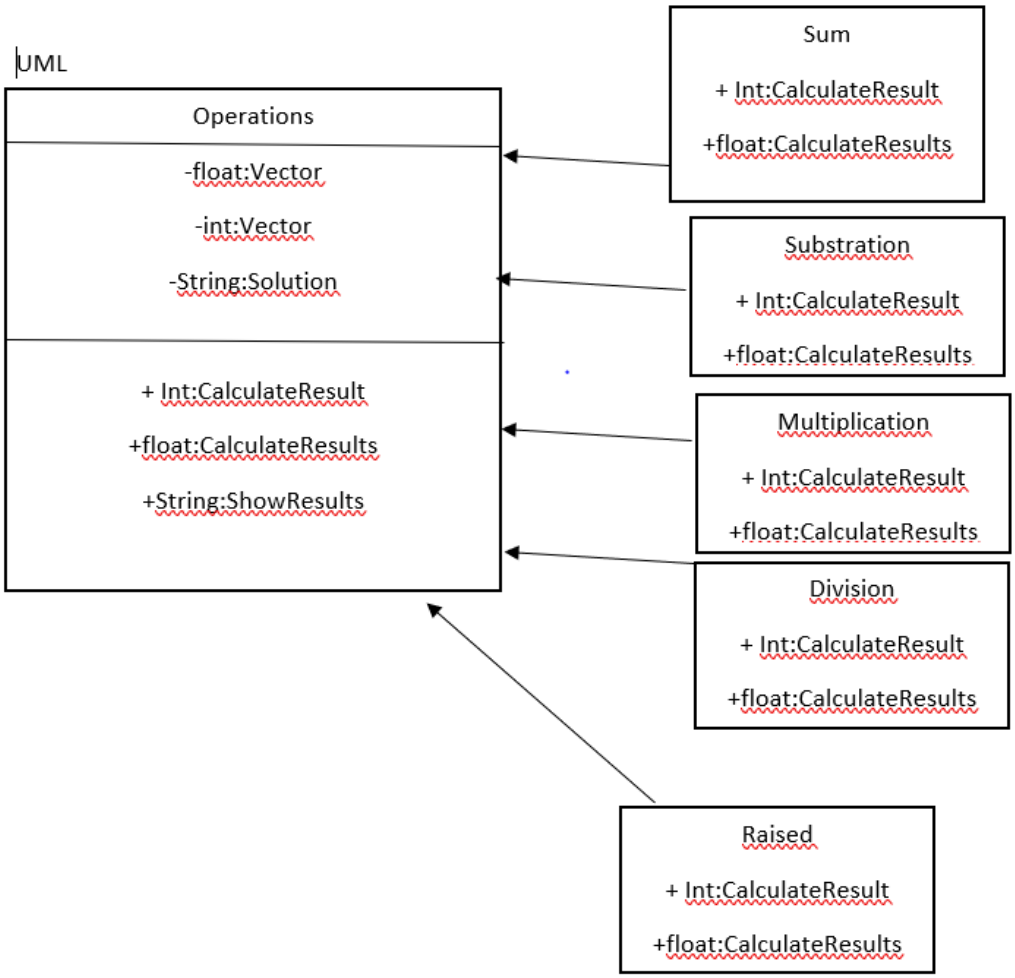
HighLevel Architecture



Graphic 2: HighLevel architecture of the calculator containing the four parts.

The second graphic shows that the calculator exist out of four parts. The parts work individually and consecutive. They are only connected throw a one-way interface. The User can write the number and operators in the inputfile and will get a terminal output containing the result. The execution order is shown in graphic 3.The box inside the green square is the whole system. And the boxes inside are the part of the system .Each of this parts do a specific job. The boxes outsite the box are the ones that the users uses.

UML-Diagram



Graphic 4: UML-diagram from the calculator

The fourth graphic shows the UML-diagram of the calculator with the 5 operators and their attributes.The boxes are the operations that can be done.

Group 13: Andres Otero Garcia, Juan Vercher and Maximilian Feussner