FLEE 1.0
(Formal Language Expression Evaluator)

by
Filip Andrei-Dan

Contents:

1. Motivation and Objectives

The motivation behind this paper and the affiliated program (FLAT 1.0) is to provide a way to visually demonstrate the working mechanism behind Dijkstra's Two Stack algorithm[1]. This objective was realized by creating a graphical user interface (GUI) that presents step-by-step the inner states of the aforementioned algorithm. Both the source and the executable can be found here: https://github.com/filip256/FLEE.

2. Algorithm Implementation

The executable was written in C++ and compiled for 32-bit versions of Microsoft Windows. For the algorithm itself two stack type structures were used, one for operator objects and one for operand objects. An additional class was created for abstract syntax trees (AST), the main algorithm needing to directly write terms into such an object.

The evaluation algorithm has the following structure:

```
for i in string
    if i is operator
        while i.precedence <= operator_stack.last_elem.precedence
            if operator_stack.last_elem.arity == 2
                aux = operand_stack.last_elem
                operand_stack.pop()
                tree.add(operator_stack.last_elem, aux, operand_stack.last_elem)
                operand_stack.pop()
            else
                tree.add(operator_stack.last_elem, aux, operand_stack.last_elem)
                operand_stack.pop()
        operator_stack.push(i)
    else
        operand_stack.push(i)

while operator_stack.size > 0
    if operator_stack.last_elem.arity == 2
        aux = operand_stack.last_elem
        operand_stack.pop()
        tree.add(operator_stack.last_elem, aux, operand_stack.last_elem)
        operand_stack.pop()
    else
        tree.add(operator_stack.last_elem, aux, operand_stack.last_elem)
        operand_stack.pop()
    operator_stack.pop()
```

In order to recognize operators from operands in the input string an operator set must be defined. By default, FLEE 1.0 provides the following set in the form of (symbol, precedence, arity):

```
('+', 1, 2) //ADD
('-', 1, 2) //SUB
('*', 2, 2) //MLT
('/', 2, 2) //DIV
('-', 4, 1) //OPS
('^', 3, 2) //PWR

('!', 5, 1) //NEG
('&', 4, 2) //AND
('|', 3, 2) //OR
('$', 3, 2) //XOR
('>', 2, 2) //IMP
('=', 1, 2) //EQU
```

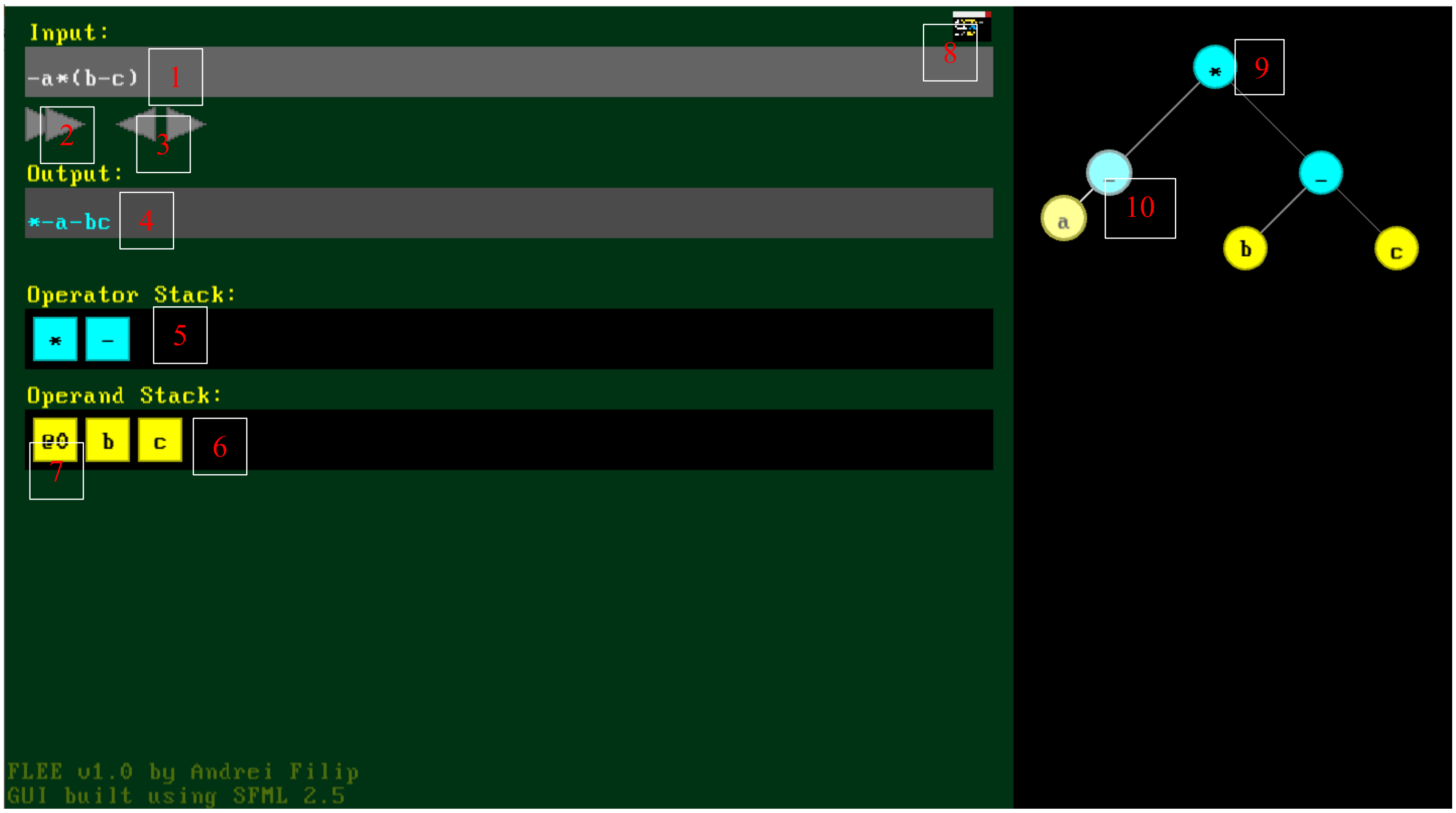
In the case of parentheses a simple rule is implemented: when '(' is found increment the precedence level and when ')' is found decrement it. The precedence of each operator is then calculated by increasing it's original value by precedence level times the number of operators.

To obtain the prefix notation of the given string, the AST is parsed recursively in the order: root, first leaf, second leaf, each symbol encountered being added to a string that is returned.

The output of the algorithm can be the expression in prefix notation together with the AST representation if the string is a well-formed mathematical expression or an error message otherwise.

3. The Graphical User Interface

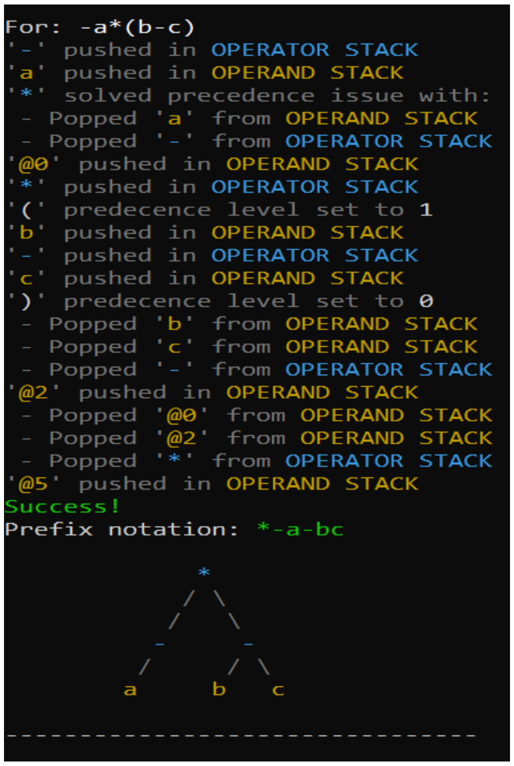
The GUI of the application was build using SFML 2.5[2], a library that offers support for 2D graphics in C++, amongst many other features. The interface shows each step of evaluation algorithm, while also offering a visual representation of the generated AST. It consists of the following elements:



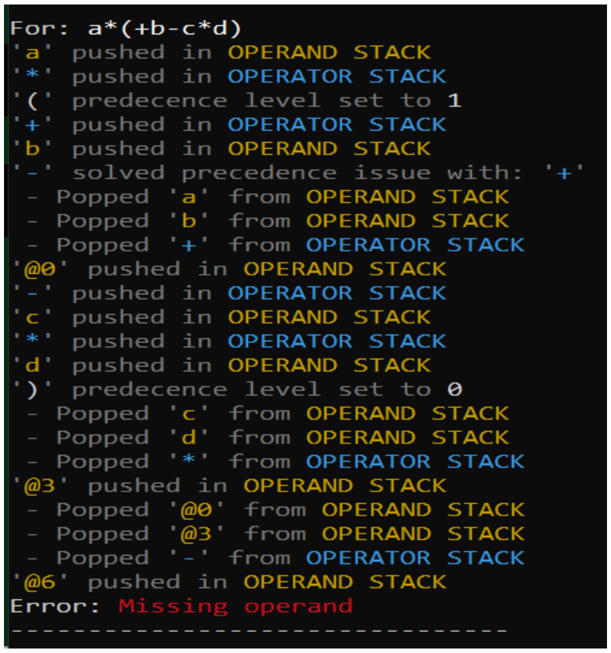
1. Input box, use ctrl+C and ctrl+V to copy and paste respectively
2. Run the algorithm on the given input
3. Move back and forth the steps
4. Output box, use ctrl+C to copy
5. Operator Stack, updated for each step
6. Operand Stack, updated for each step
7. Object box containing a reference, click to highlight the corresponding branch of the AST
8. Open and close the log console
9. Visual representation of the AST, use the keyboard arrows to move the view
10. Highlighted branch of the AST

4. The Log Console

The console was built using the Win32 API in order to provide an easy-to-read log of all the steps, allowing the output to be easily copied by the standard means.

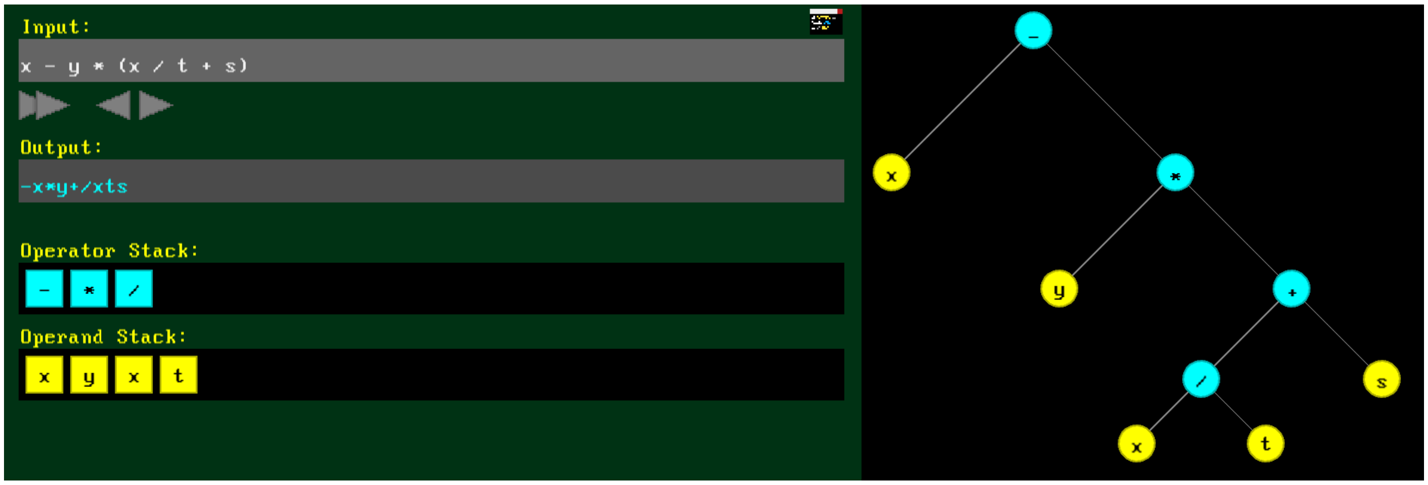


```
For: -a*(b-c)
'-' pushed in OPERATOR STACK
'a' pushed in OPERAND STACK
'*' solved precedence issue with:
 - Popped 'a' from OPERAND STACK
 - Popped '-' from OPERATOR STACK
'@0' pushed in OPERAND STACK
'*' pushed in OPERATOR STACK
'(' predecence level set to 1
'b' pushed in OPERAND STACK
'-' pushed in OPERATOR STACK
'c' pushed in OPERAND STACK
')' predecence level set to 0
 - Popped 'b' from OPERAND STACK
 - Popped 'c' from OPERAND STACK
 - Popped '-' from OPERATOR STACK
'@2' pushed in OPERAND STACK
 - Popped '@0' from OPERAND STACK
 - Popped '@2' from OPERAND STACK
 - Popped '*' from OPERATOR STACK
'@5' pushed in OPERAND STACK
Success!
Prefix notation: *-a-bc

             *
            / \
           /   \
          -     -
         /     / \
        a     b   c

--------------------------------
```



```
For: a*(+b-c*d)
'a' pushed in OPERAND STACK
'*' pushed in OPERATOR STACK
'(' predecence level set to 1
'+' pushed in OPERATOR STACK
'b' pushed in OPERAND STACK
'-' solved precedence issue with: '+'
 - Popped 'a' from OPERAND STACK
 - Popped 'b' from OPERAND STACK
 - Popped '+' from OPERATOR STACK
'@0' pushed in OPERAND STACK
'-' pushed in OPERATOR STACK
'c' pushed in OPERAND STACK
'*' pushed in OPERATOR STACK
'd' pushed in OPERAND STACK
')' predecence level set to 0
 - Popped 'c' from OPERAND STACK
 - Popped 'd' from OPERAND STACK
 - Popped '*' from OPERATOR STACK
'@3' pushed in OPERAND STACK
 - Popped '@0' from OPERAND STACK
 - Popped '@3' from OPERAND STACK
 - Popped '-' from OPERATOR STACK
'@6' pushed in OPERAND STACK
Error: Missing operand
--------------------------------
```

5. Examples



Input:

x - y * (x / t + s)

Output:

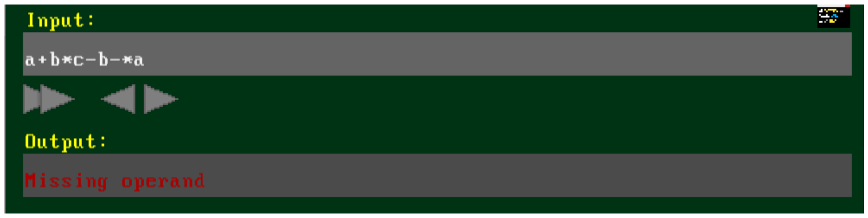
-x*y+/xts

Operator Stack:

| - | * | / |

Operand Stack:

| x | y | x | t |

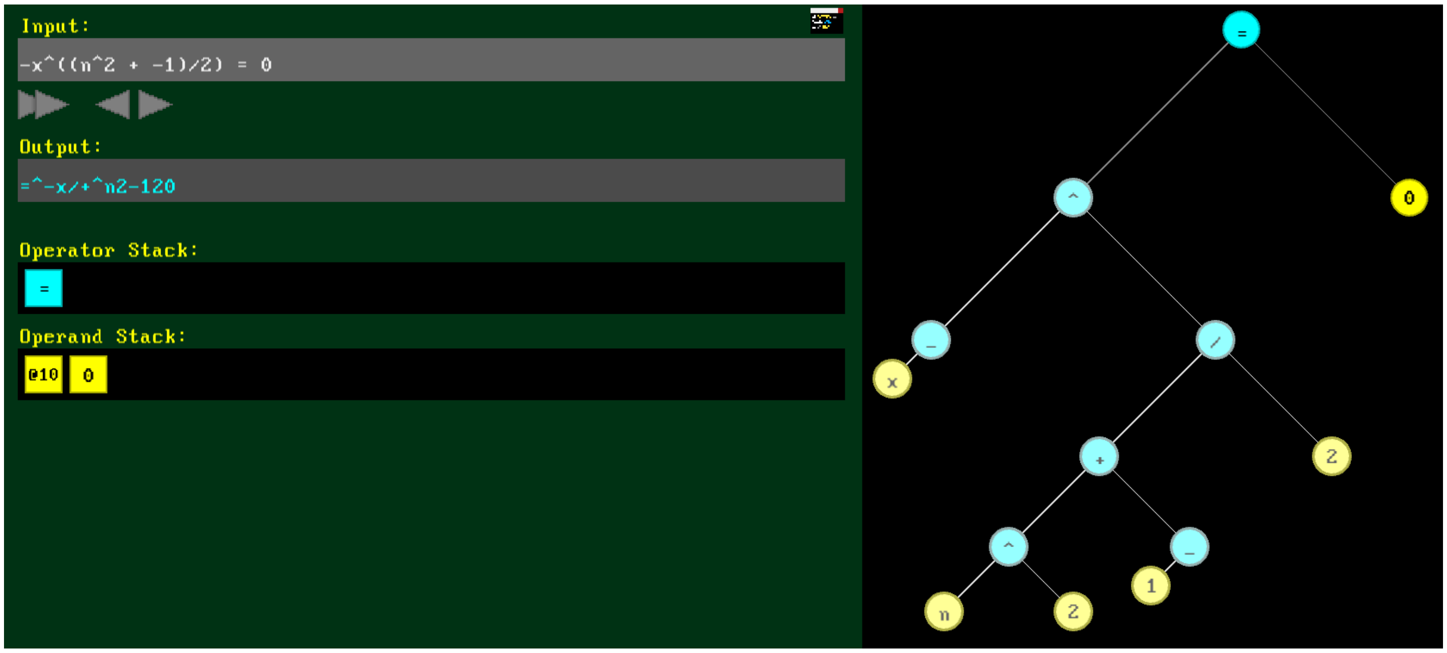


Input:

a+b*c-b-*a

Output:

Missing operand



Input:

```
-x^((n^2 + -1)/2) = 0
```

Output:

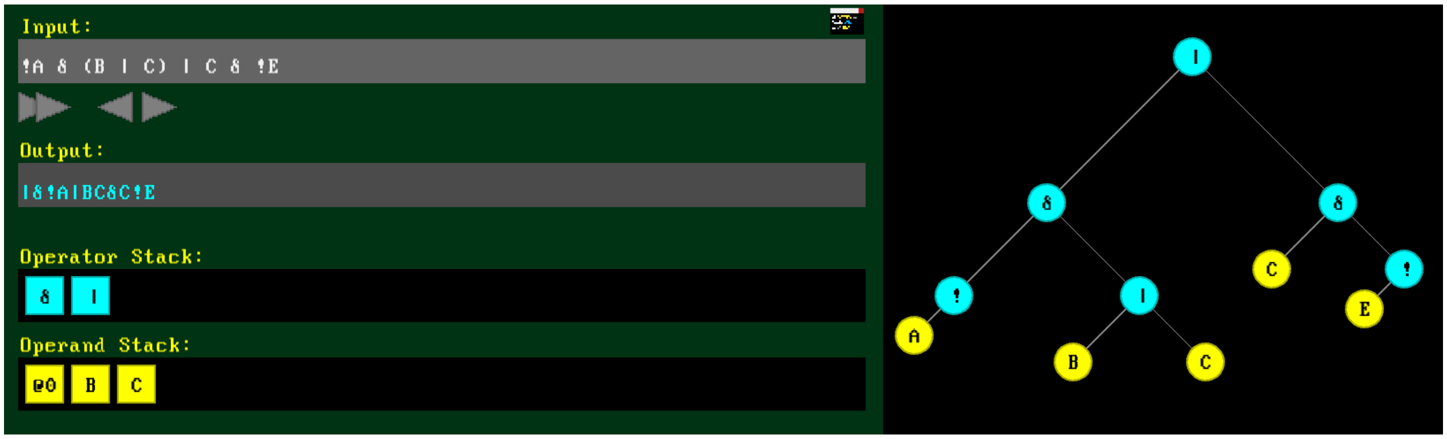
```
=^-x/+^n2-120
```

Operator Stack:

```
=
```

Operand Stack:

```
@10 0
```



Input:

```
!A & (B | C) | C & !E
```

Output:

```
|&!A|BC&C!E
```

Operator Stack:

```
& |
```

Operand Stack:

```
@0 B C
```

6. References

[1] Dijkstra's original description of the closely related Shunting-yard algorithm: https://www.cs.utexas.edu/~EWD/MCReps/MR35.PDF

[2] The SFML frontpage: https://www.sfml-dev.org