CmpE230 Project 1 Documentation
- *AdvCalc* -

Muhammet Ali Topcu - *2020400147*
Abdullah Enes Güleş - *2021400135*

01.04.2023

Purpose of the Project

Implementing an interpreter for an advanced calculator using the C programming language. The advanced calculator (AdvCalc) will accept expressions and assignment statements.

Design

In this project, we first need to take the input from the user, create tokens from the given input with lexical analyzer. Then parse the tokens in order to create a parse tree (or abstract syntax tree) from the tokens. And finally, evaluate the tree and find the result. If it is an assignment, assign it to the identifier and store it in a hash table. If it is an expression, just return the result.

How to Run the Program

1. Open a terminal at the root folder of the program.
2. Type 'make' to create an executable of the program. (make command runs the code in Makefile. It compiles the program and creates an executable named 'advcalc.exe'
3. Then, with './advcalc.exe' command, you can run the program.

Implementation Details

The program runs mainly in main.c file.

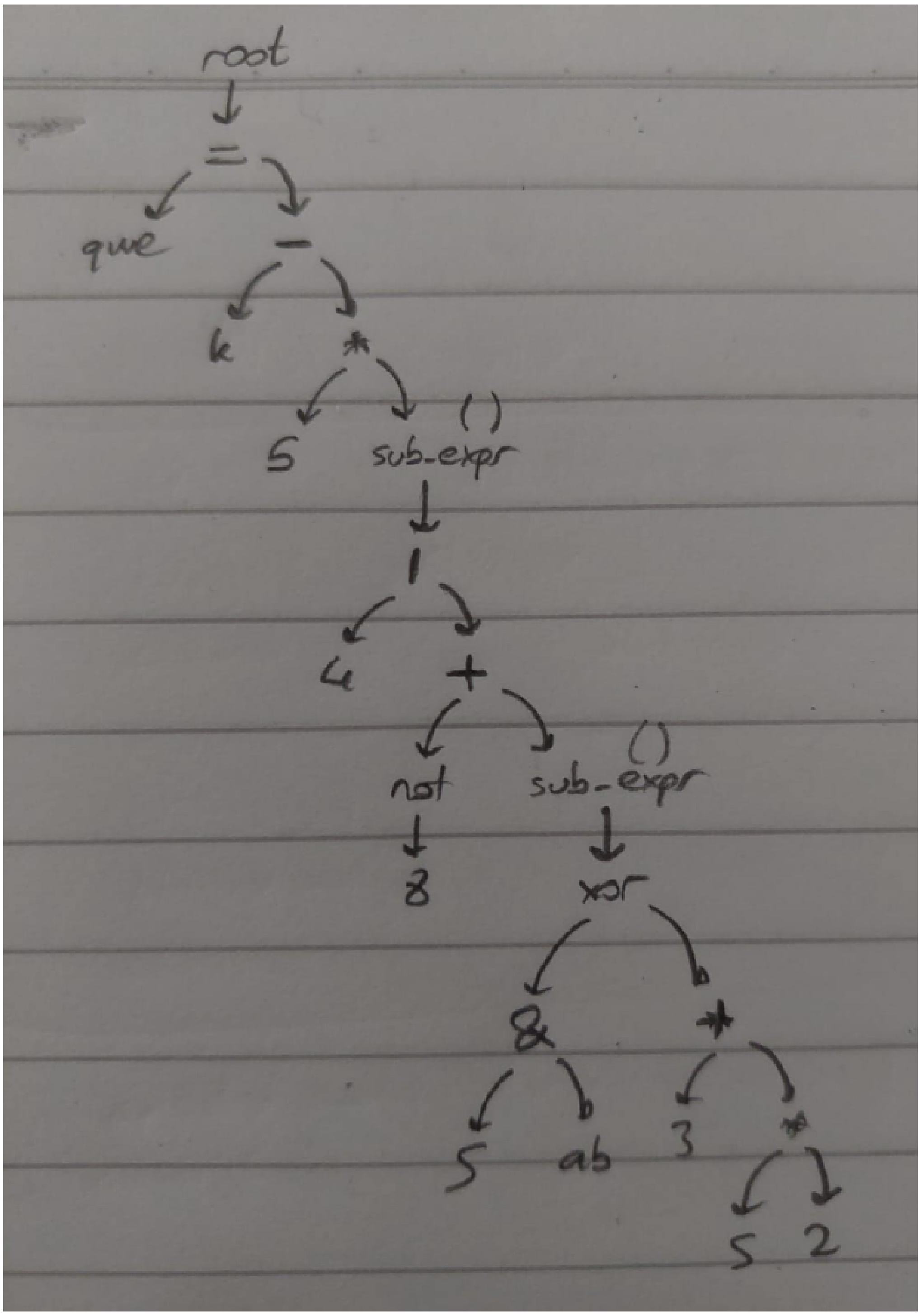
Firstly, the program takes the input from the user with fgets function. Then, it creates a temporary array to store the input. After that, program creates an array of Tokens, which is a type and defined at token.h header file to store the tokens properly.

Then, program proceeds and sends the temporary input array and tokens array to lexer function, which can be found at lexer.c. The lexer function takes the input array and tokenizes the input. If an error occurs, the function returns 0 to notify that the input has an error and so that the program will not proceed anymore and continues to take the next input. If no error occurs, the function returns 1 and proceeds as expected.

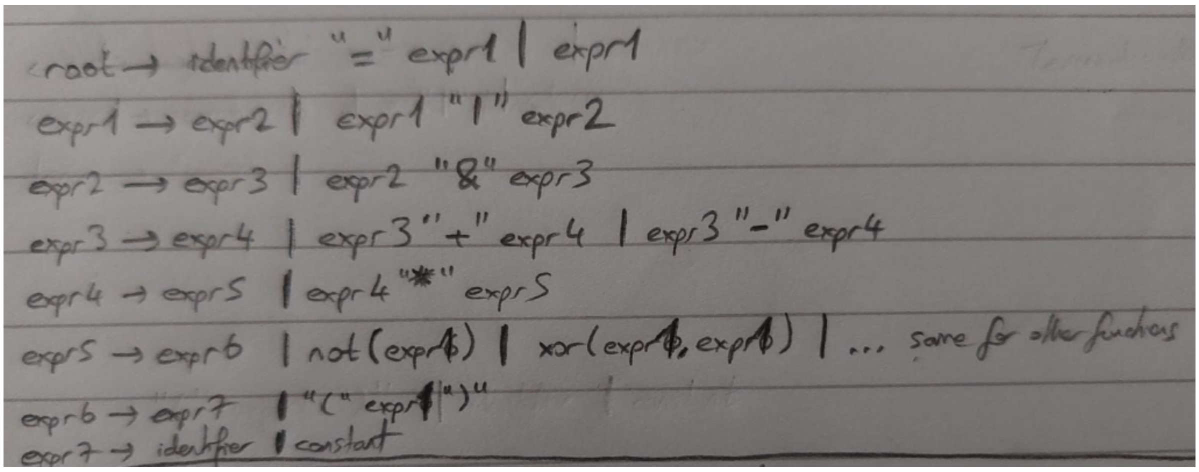
After tokenizing the input, we need to create a parse tree to store the input in postfix format so that we can evaluate the postfix expression later with DFS.

At this stage, it was necessary to draw an example parse tree to see the operator precedences. So, we drew a parse tree to see all possible tokens in action.

In this example, the input was: **qwe = k - 5 * (4 | not(8) + xor(5 & ab, 3 + 5*2)))**



This input have covered all possible operator, function and parantheses precedences. And then, with the help of this example parse tree, we created a BNF to later use in parser functions:



```
root → identifier "=" expr1 | expr1
expr1 → expr2 | expr1 "|" expr2
expr2 → expr3 | expr2 "&" expr3
expr3 → expr4 | expr3 "+" expr4 | expr3 "-" expr4
expr4 → expr5 | expr4 "*" expr5
expr5 → expr6 | not(expr1) | xor(expr1, expr1) | ... same for other functions
expr6 → expr7 | "(" expr1 ")"
expr7 → identifier | constant
```

Using this BNF, we coded the parser functions. At the main function, after tokenizing the input with lexer function, program creates a root node for AST. We implemented AST in ast.c to store the inputs in postfix format. At this stage, program has two possibilities: Either input is an expression or an assignment.

In case of an assignment, parser function takes the tokens and creates the tree and returns it. If there exists a problem, it returns null. Otherwise, it returns the result tree. Since this is an assignment, we need to store the identifier. So, we also implemented a hash table in lookup.c. After evaluating the right hand side of the tree, program inserts the identifier with its value as (key, value) pair in the hashmap.

In case of an expression, parser function takes the tokens and creates the tree and returns it, same as the above example.

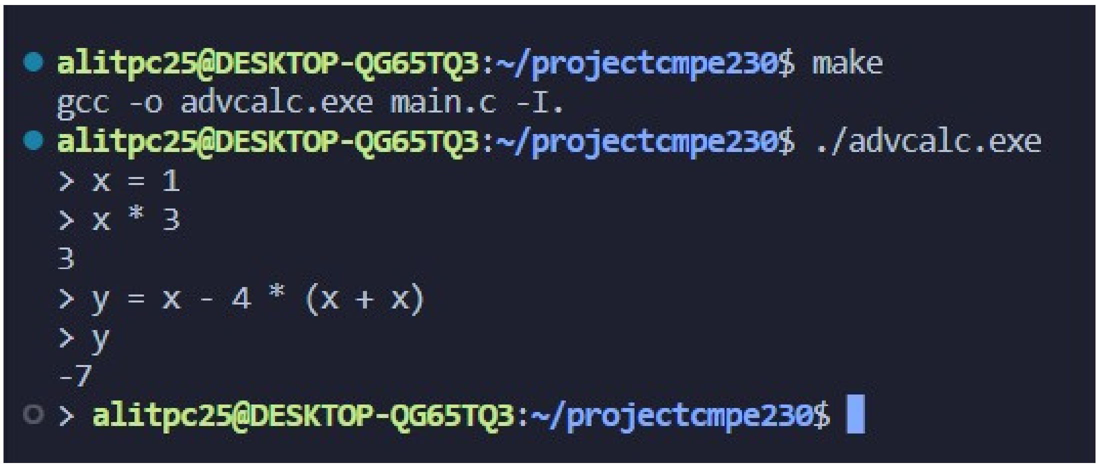
Evaluation of the tree is implemented in evaluator.c with eval function. This function takes the root of the tree, traverses the tree recursively and evaluates the result and returns it.

Difficulties Encountered

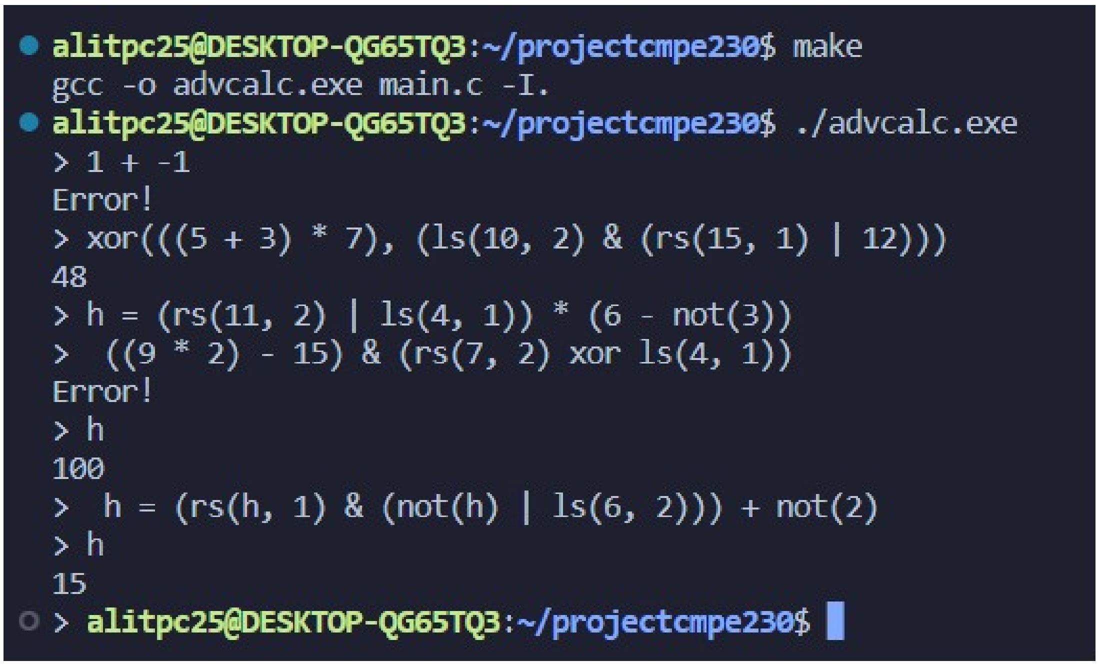
We faced difficulties during memory management and dealing with strings. First problem was storing the tokens and strings in array. After spending about 2 hours, we figured out that it is necessary to use malloc for dynamic memory allocation.

The other problem was dealing with strings. We should not have forgotten that strings have length of their characters + '\0' at the end. Our program was terminating without any error. Later, we figured out that it is necessary to end strings with '\0'.

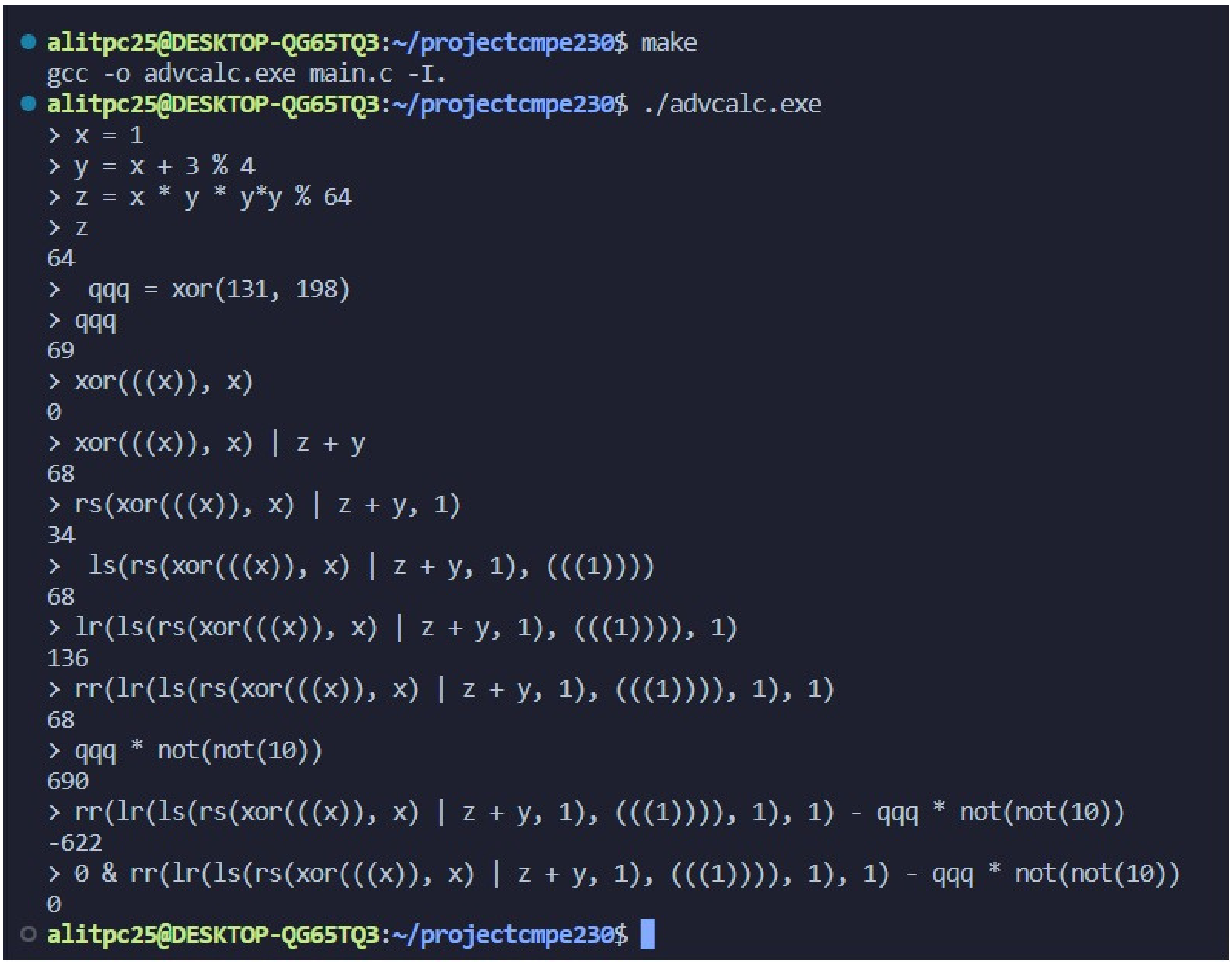
Example Inputs & Outputs



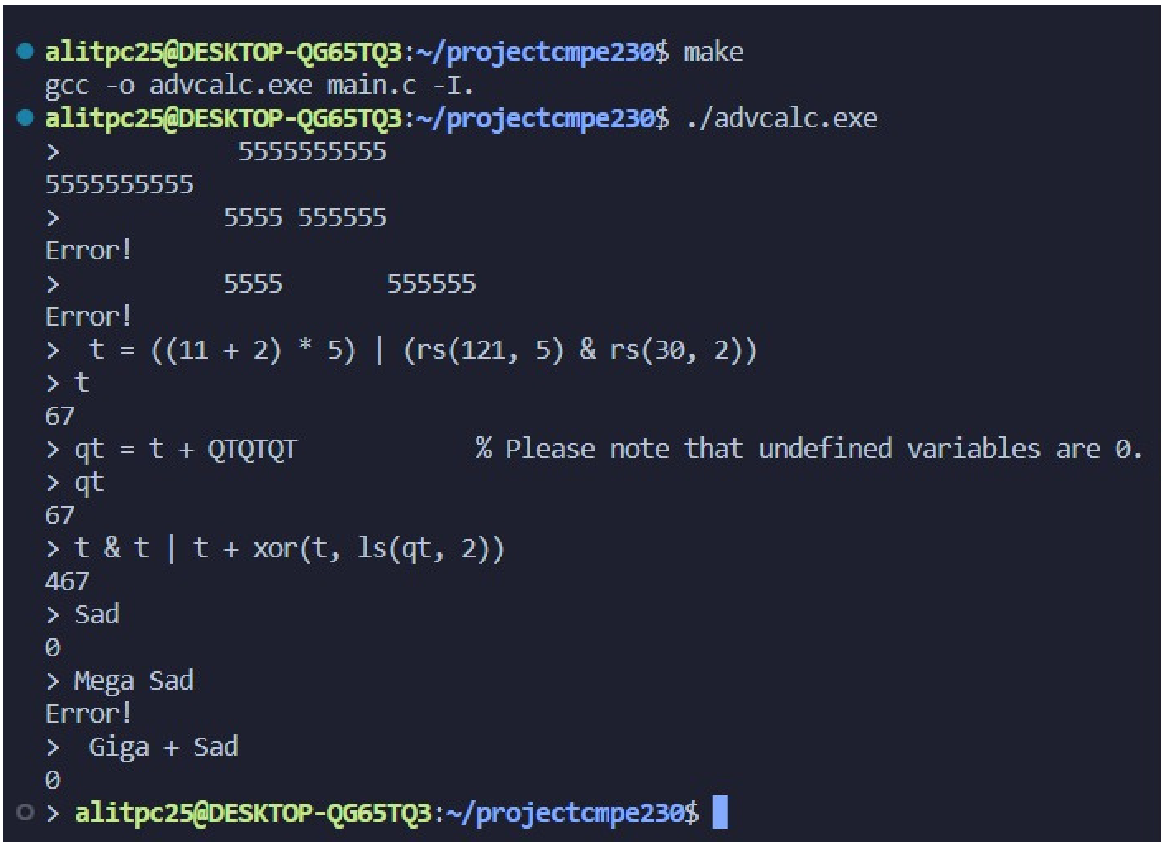
```
● alitpc25@DESKTOP-QG65TQ3:~/projectcmpe230$ make
  gcc -o advcalc.exe main.c -I.
● alitpc25@DESKTOP-QG65TQ3:~/projectcmpe230$ ./advcalc.exe
  > x = 1
  > x * 3
  3
  > y = x - 4 * (x + x)
  > y
  -7
○ > alitpc25@DESKTOP-QG65TQ3:~/projectcmpe230$ 
```



```
● alitpc25@DESKTOP-QG65TQ3:~/projectcmpe230$ make
  gcc -o advcalc.exe main.c -I.
● alitpc25@DESKTOP-QG65TQ3:~/projectcmpe230$ ./advcalc.exe
  > 1 + -1
  Error!
  > xor(((5 + 3) * 7), (ls(10, 2) & (rs(15, 1) | 12)))
  48
  > h = (rs(11, 2) | ls(4, 1)) * (6 - not(3))
  >  ((9 * 2) - 15) & (rs(7, 2) xor ls(4, 1))
  Error!
  > h
  100
  >  h = (rs(h, 1) & (not(h) | ls(6, 2))) + not(2)
  > h
  15
○ > alitpc25@DESKTOP-QG65TQ3:~/projectcmpe230$ 
```



```
alitpc25@DESKTOP-QG65TQ3:~/projectcmpe230$ make
gcc -o advcalc.exe main.c -I.
alitpc25@DESKTOP-QG65TQ3:~/projectcmpe230$ ./advcalc.exe
> x = 1
> y = x + 3 % 4
> z = x * y * y*y % 64
> z
64
>  qqq = xor(131, 198)
> qqq
69
> xor(((x)), x)
0
> xor(((x)), x) | z + y
68
> rs(xor(((x)), x) | z + y, 1)
34
>  ls(rs(xor(((x)), x) | z + y, 1), (((1))))
68
> lr(ls(rs(xor(((x)), x) | z + y, 1), (((1)))), 1)
136
> rr(lr(ls(rs(xor(((x)), x) | z + y, 1), (((1)))), 1), 1)
68
> qqq * not(not(10))
690
> rr(lr(ls(rs(xor(((x)), x) | z + y, 1), (((1)))), 1), 1) - qqq * not(not(10))
-622
> 0 & rr(lr(ls(rs(xor(((x)), x) | z + y, 1), (((1)))), 1), 1) - qqq * not(not(10))
0
alitpc25@DESKTOP-QG65TQ3:~/projectcmpe230$
```



```
alitpc25@DESKTOP-QG65TQ3:~/projectcmpe230$ make
gcc -o advcalc.exe main.c -I.
alitpc25@DESKTOP-QG65TQ3:~/projectcmpe230$ ./advcalc.exe
>             5555555555
5555555555
>           5555 555555
Error!
>           5555       555555
Error!
>  t = ((11 + 2) * 5) | (rs(121, 5) & rs(30, 2))
> t
67
> qt = t + QTQTQT            % Please note that undefined variables are 0.
> qt
67
> t & t | t + xor(t, ls(qt, 2))
467
> Sad
0
> Mega Sad
Error!
>  Giga + Sad
0
> alitpc25@DESKTOP-QG65TQ3:~/projectcmpe230$
```