Advanced Calculator Documentation

CMPE 230 - Project 1

Mustafa Oguz Hekim

March 29, 2023

Contents

1 Introduction

This document is a manual and documentation for the first project of CMPE230-Systems Programming course. In this project I implemented an advanced calculator, that works like an python interpreter, using C programming language.

2 Installation and Usage

First, download the project, navigate to the project folder from the terminal. Use `make` command to compile the project and create an executable. After the project is compiled you can run it with `./advcalc` command in the project directory. Once you run the project, you can enter operations you want to do line by line.

- There are 12 operations and functions that can be done with the calculator.
- `%` sign is used for inline comments. Any input after this sign will be ignored.
- Parentheses can be used for giving precedence to operations.
- Unary operators are not supported.
- Values can be assigned to variables (i.e. a=5).
- If any undefined variable is used, it will have value of 0.
- For each line of input, there is a limit of 256 characters.
- Operations can produce 64-bit results at most.
- Bitwise operations are also based on 64-bit.
- If there is a mistake with the syntax or parentheses, `Error!` will be the output.
- You can press `CTRL+D` to exit the program.

Allowed operations and functions are in the table below.

a = b	Assign value of b to the variable a.
a + b	Addition of a and b
a - b	Subtraction of b from a
a * b	Multiplication of a and b
a & b	Bitwise a and b
a | b	Bitwise a or b
xor(a, b)	Bitwise a xor b
ls(a, b)	a shifted b bits to the left
rs(a, i)	a shifted b bits to the right
lr(a, i)	a rotated b bits to the left
rr(a, i)	a rotated b bits to the right
not(a)	Bitwise not of a

2.1 Examples

Here are some example input and outputs for the calculator.

```
% ./advcalc
> x = 2+7 *8
> x
58
> x = x*x
> x
3364
> y = x/58
Error!
> 53+ -2
Error!
```

```
% ./advcalc
> xor(2,3*1+1)
6
> rr(256,10)
4611686018427387904
> rr(256,9)
-9223372036854775808
> not(xor(2, rs(555,4))
Error!
> not(xor(2, rs(555,4)))
-33
> myVariable = 3
> my_variable = 4
Error!
> a
0
```

```
% ./advcalc
> a=(8)
> a
8
> b =               2
> a+b
10
> (((a)) + b )
10
> c
0
> c = -10
Error!
> ls ( xor(a|b , c+25))
Error!
> ls ( xor(a|b , c+25), a)
4864
```

```
% ./advcalc
> not(2,3)
Error!
> not((2,3))
Error!
> not(xor(2,3))
-2
> xor((2,3))
Error!
> not(xor(2,3),2)
Error!
```

3 Implementation

Project consists of several modules with different functionalities. There are four main steps in the process of calculation:

1. Lexical analysis.
2. Validation of input.
3. Converting infix expression to postfix.
4. Evaluating the postfix expression.

3.1 Lexical Analysis

Lexical analysis is entry point of the program. Input should be identified and resolved into predefined stream of tokens. This module is named `lexer` in my program. Here is the source code and explanation to some of the lines from the lexer function:

Lexer traverses input string. Two pointers to input string is defined as `left` and `right`. If current character is not a delimiter, right pointer is incremented.

```
1  Token* lexer(char* input, int* tokenCount, bool *error){
2      Token *tokens = malloc(sizeof(Token)*256);
3      int len = strlen(input);
4      int left = 0;
5      int right = 0;
6      while (right<=len) {
7          if (!isDelimiter(input[right])){
8              right++;
9          }
```

If current character is a delimiter and pointers are not equal, that means the string between two pointers must be captured and used to decide whether the substring is a function, integer or variable. After the decision, pointers can be set equal.

```
1          else if (isDelimiter(input[right]) && left != right){
2              // Get the substring between two indices
3              char *substr = malloc((right-left+1)*sizeof(char));
4              for (int i = left; i < right; i++)
5                  *(substr+i-left) = input[i];
6              *(substr+right-left) = '\0';
7              left = right;
```

For the last case, if current character is a delimiter and pointers are equal, that means both of the pointers are pointing to a single character which is a delimiter. This character can be white space, comment sign, comma, parentheses, null character or an operation. Once that is decided, pointers are incremented.

This is the overall structure of the lexical analysis. After the whole string is processed, a stream of tokens is returned from the function. You can check the source code for a full review of the module.

3.2 Validation

Validation is the error checking functionality of the program. Most -but not all- of the error handling is done here. `validate` function takes stream of tokens that is returned from lexer, and changes state of the program if anything wrong is encountered. Most of this decision is made by considering the position of the token in the stream. Here are the considered cases for validation:

- First token can be integer, variable, function or left parenthesis.
- Last token can be integer, variable or right parenthesis.
- If an equal operator is encountered it must be the second element of the stream. Also the element before the operator must be a variable.
- Token after the comma or operator can be integer, variable, function or left parenthesis.
- Token before the comma or operator can be integer, variable or right parenthesis.
- Token after the function must be left parenthesis.
- Token before the function can be operator, comma or left parenthesis.
- If token is the `not` function, it cannot contain comma between parenthesis (except nested functions).
- Token after the integer or variable cannot be integer, variable, function or left parenthesis.
- Token before the integer or variable cannot be integer, variable, function or right parenthesis.
- Token after the left parenthesis cannot be right parenthesis (empty parentheses error).

3.3 Shunting Yard Algorithm

After lexical analysis and validation of the input, several methods can be used. Building an abstract syntax tree with recursive descent parser is one of them. However I chose to use the shunting yard algorithm invented by Edsger Dijkstra, because of simplicity.

Shunting yard algorithm uses a stack and a precedence rule to convert infix expression to postfix. Here is the precedence rule for the calculator:

$$Multiplication > Addition/Subtraction > And/Or > Assignment$$

Functions do not need any precedence rule as they will be pushed to the stack directly and will be followed by left parenthesis.

After this is set, a stack is created. Algorithm traverses through stream of tokens returned by lexer. There are several steps to check in the algorithm:

1. If the incoming token is an operand (integer or variable), add it to output.
2. If the incoming token is a function or left parenthesis, push it to the stack.
3. If the incoming token is an operator and the stack is empty or contains a left parenthesis on top, push it to the stack.
4. If the incoming token is an operator and has higher precedence than the operator on top of the stack, push it to the stack.
5. If the incoming token is an operator and has lower or same precedence as the operator on top of the stack, pop the stack and add popped tokens to the output until this is not true. Then push the incoming operator.
6. If the incoming token is an operator and has lower or same precedence as the operator on top of the stack, pop the stack and add popped tokens to the output until this is not true. Then push the incoming operator.
7. If the incoming token is comma, pop until left parenthesis (excluding left parenthesis).
8. If the incoming token is a right parenthesis:
   (a) If the stack is empty give an error (unmatched right parenthesis).
   (b) Pop and output the stack until you see a left parenthesis. Pop the left parenthesis. If a function comes next in stack, pop and add it to the output.

9. If all the tokens are processed and stack is not empty, pop and output the stack until it is empty.

After all of the infix tokens is processed, function will output a new stream of tokens. Source code for this algorithm can be found in the `shunting` module.

Here is an example to show how algorithm works:

$$x = (2 + 3) * xor(a, 5)$$

Current Token	Stack (Right is top)	Postfix Stream
x		x
=	=	x
(	= (	x
2	= (	x 2
+	= (+	x 2
3	= (+	x 2 3
)	=	x 2 3 +
*	= *	x 2 3 +
xor	= * xor	x 2 3 +
(	= * xor (	x 2 3 +
a	= * xor (	x 2 3 + a
,	= * xor (	x 2 3 + a
5	= * xor (	x 2 3 + a 5
)	= *	x 2 3 + a 5 xor
		x 2 3 + a 5 xor * =

3.4 Evaluation

Evaluating the postfix stream is the last step of the program. Until now the input string has been tokenized with lexer, checked with validator and converted to postfix tokens with shunting yard algorithm. For the evaluation, variables and their corresponding values should be handled.

Variables are stored in an array at main function. There is also another array for their corresponding values. Whenever a variable is encountered in evaluation, it is searched in variable array. If it is in the array, its value is retrieved from the value array. If it is not, that variable is added to the variable array and its value is set to 0 in the value array.

Evaluation is done by using a stack. Whenever an integer or variable is encountered in the stream, it is pushed to the stack. If the token is operator or function,

two tokens are popped from the stack and the new value is evaluated with necessary operation. After that, new value is pushed to the stack. There are two special cases with this approach. If the token is the `not` function, only one token is popped. And if the token is assignment, the variable is searched in the array and if it exists, its value is updated. If it does not exist that variable is added to the array and its value is assigned in the value array.

Source code for this part can be found in the `calculator` module. To demonstrate how this process is done, following up from the previous example, the postfix expression is

$$x\ 2\ 3\ +\ a\ 5\ xor\ *\ =$$

Current Token	Stack (Right is top)	Explanation
x	x	Push x to the stack.
2	x 2	Push 2 to the stack.
3	x 2 3	Push 3 to the stack.
+	x 5	Pop 2 and 3. Evaluate 2+3 and push it to the stack.
a	x 5 a	Push a to the stack.
5	x 5 a 5	Push 5 to the stack.
xor	x 5 5	Pop a and 5. Get the value of a if it is defined before. If it is not take it as 0. Evaluate xor(a,5) and push it to the stack.
*	x 25	Pop both 5's. Evaluate 5*5 and push it to the stack.
=		Pop x and 25. Set x's value as 25.

3.5 Error Handling

Error handling is done by creating a boolean variable inside the main function. Other functions that are called in main, can change the value of this variable.

As mentioned earlier, error handling is mainly done with `validator` module. However there are some cases checked in other modules as well.

- Any invalid characters in the input are detected in `lexer` module.
- Unmatched right parentheses are detected in `shunting` module.

- Unmatched left parentheses are detected in `calculator` module because input of this function is postfix stream. Postfix stream cannot contain any parenthesis, so it can issue an error.

4 Improvements

Since this was project with certain restrictions and also my first experience in C language, there are parts that can be improved.

Handling variables can be improved by implementing a hash table instead of two separate arrays. Although searching in the value array takes constant time since we know the index, searching in variables array is linear time. It is implemented as an array because maximum number of variables is given as 128 and C language has not built-in hash table.

The other point is with shunting yard algorithm. Currently algorithm outputs a stream of postfix tokens. Instead, it can evaluate the expression on the go. This can be done with two stacks. One for the operations and one for the operands. If this is done `calculator` module can be removed. However this makes the code less modular. Also hurts readability and writability a lot. Since this is my first project in C, I did not want to confuse myself with this approach.