CompArch Final Project Documentation

Yifan (Frank) Zhang, Allister Liu

Prof. Billoo

CU_ECE251

May 6, 2019

Goal & Requirements

The goal of this project is to build a simple calculator program. This program is able to perform simple integer and floating point number arithmetic operations, including addition, subtraction, multiplication, division. The program also supports the use of parentheses. It will take an input string of a series of operations, which is defined as operation = <operand> <arithmetic operation> <operand><arithmetic operation><operand>... , and it will return the result and print it in the terminal.

Function Overviews

Our program supports basically all the normal expressions we can think about. It is execeuted by

```
./final <expression>
```

Some examples of <expression> are:

```
“1      +     2”
“-1-3-4*5”
“(2+3.01)*(-5+6)”
“((4.59 + (-8)) * 100) / 25.7”
```

The expressions must be wrapped in “” or bash will interpret parentheses and space differently. The code will automatically ignore all the spaces. The only issus we have is nested redundant parentheses like

```
“((2+3))”
```

This will create an overflow. However, since this is redundant, we didn’t pay effort to fix it.

Overall-Architecture

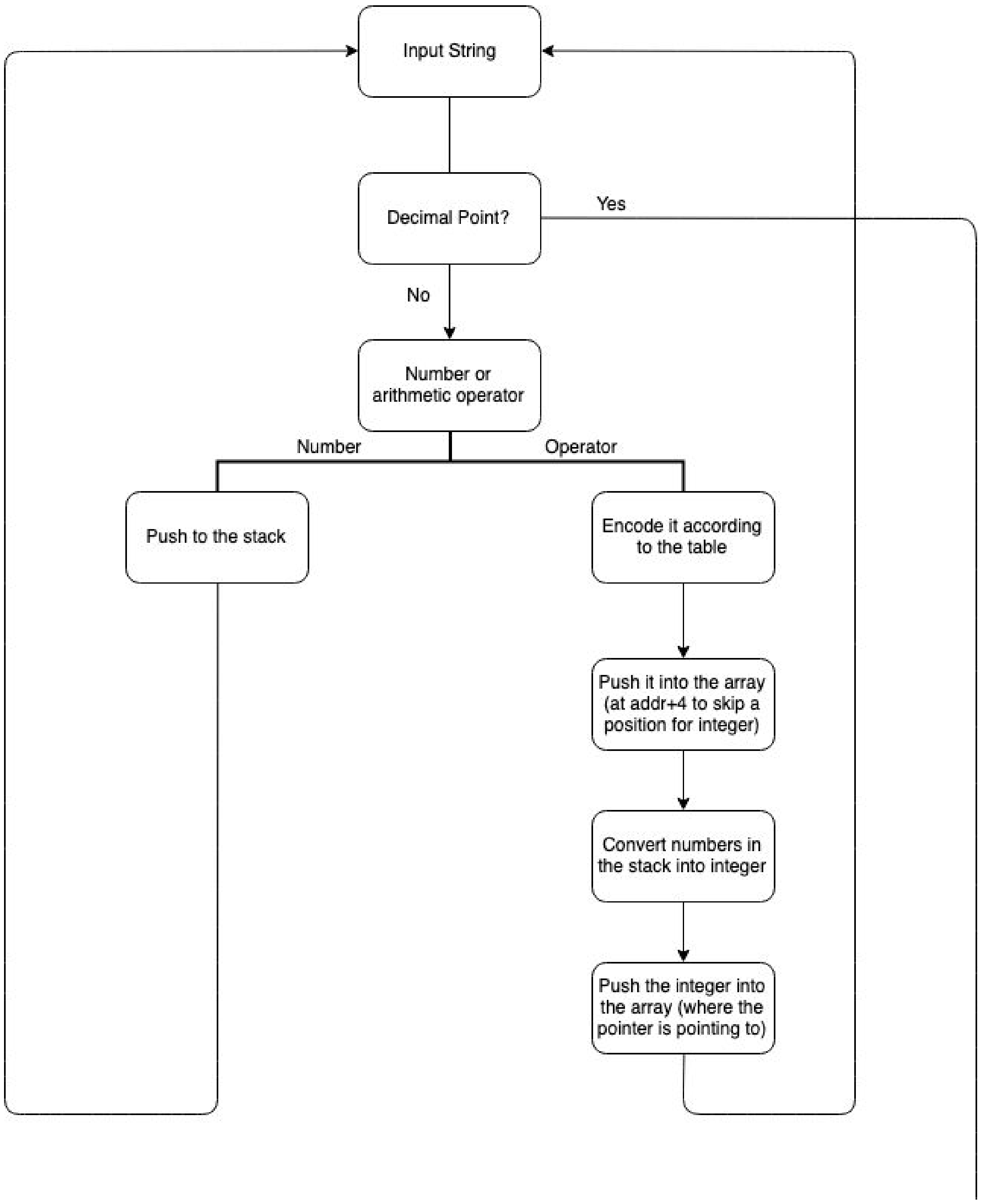
We use Reverse Polish Notation (RPN) to handle the expression. This program is composed of three main parts: preprocessing input, putting input into RPN and creating a doubly linked list, and performing the operations and return the result.

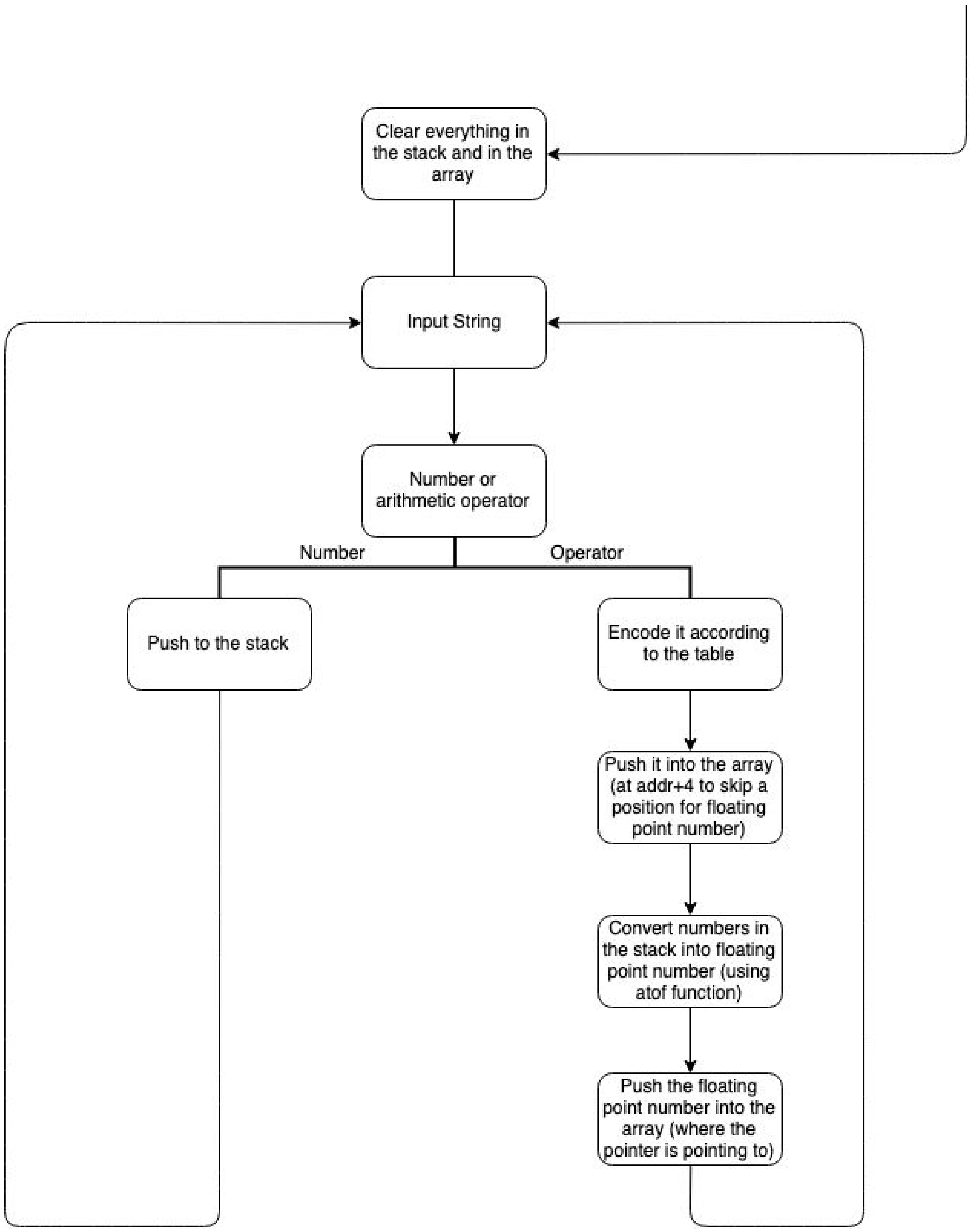
Since this program will take a series of operations in the form of a string as its input, the input it gets will be in ASCII notation; therefore, the program will have to convert the input from ASCII into integer or floating point number -- depending on the situation -- and put them into an integer or floating point number array. Also, to make our life easier, the arithmetic operators will be encoded by the following integers. They are the lowest 32-bits integers and they are rarely used, so this measure won't bring too much restraint to the user. The encoded value for each arithmetic operator used in this program is shown in the table below:

Arithmetic Operator	Encoded Value
(-- left parenthesis	0x800000
- -- subtraction sign	0x800001
+ -- addition sign	0x800002
/ -- division sign	0x800003
* -- multiplication sign	0x800004
) -- right parenthesis	0x800005

Table 1: Look-up Table for the Encoded Value of Arithmetic Operators

We encoded the arithmetic operators used in this program in the way that their encoded numerical values follow the PEMADS precedence order -- except the left parenthesis. This is for the RPN conversion to work correctly, and the result RPN expression still follows the PEMDAS rule.


Input String
Decimal Point?
Yes
No
Number or arithmetic operator
Number
Operator
Push to the stack
Encode it according to the table
Push it into the array (at addr+4 to skip a position for integer)
Convert numbers in the stack into integer
Push the integer into the array (where the pointer is pointing to)


Clear everything in the stack and in the array
Input String
Number or arithmetic operator
Number
Operator
Push to the stack
Encode it according to the table
Push it into the array (at addr+4 to skip a position for floating point number)
Convert numbers in the stack into floating point number (using atof function)
Push the floating point number into the array (where the pointer is pointing to)

Then the conversion process begins. First, a pointer pointing to the address of the resulting array will be saved into a register. Then, the program will start scanning through the input array, one character each time:

- If it is a number, then it will be pushed into a stack, and go to the next element in the input string.
- If it is an operator:
    - The program will first encode it according to the encoding table above.
    - Then it will save the encoded value into the array at *(ptr+4), which leaves 4 bytes of space before the operator -- this is where the operands will be stored (since the operands are integers, and integers take up 4 bytes of memory).
    - After, the numbers in the stack will be converted into integer form.
        - Since the input is a character string, the numbers are ASCII characters; therefore, we can subtract ‘0’, the ASCII character for number 0, to get the numerical value of each number.
        - r4 will be used as a digit multiplier, and it will be initiated to 1; r9 will be used to store the result, and it will be initiated to 0.
        - Then we will pop out a number from the top of the stack, and multiply it by the digit multiplier r4, add the value in r9 to the result, and store the new result into r9.
        - Multiply r4 by 10, and repeat the last step and this step, until it reaches the bottom of the stack.
        - Alternatively, we could also use the atoi() function for this conversion.

    - Finally, the number, in the form of an integer, will be saved at where the pointer is pointing to. The pointer will be increased by 8, and we will go to the next element in the input string.
- If it is a decimal point:
    - Everything in the stack and the array will be erased, and the program will take a similar process.
    - Starting from the beginning of the input array, the program will start scanning through it, one character each time:
        - If it is a number or a decimal point, then it will be pushed into a stack, and go to the next element in the input string.
        - If it is an operator:
            - The program will first encode it according to the encoding table above.
            - Then it will save the encoded value into the array at *(ptr+4), which leaves 4 bytes of space before the operator -- this is where the operands will be stored (since the operands are single precision floating point numbers, and they take up 4 bytes of memory).
            - After, the numbers in the stack will be converted into the form of floating point number by using the atof() function.
            - Finally, the number, in the form of a floating point number, will be saved at where the pointer is pointing to. The pointer will be increased by 8, and we will go to the next element in the input string.

This process will keep looping until we reach the end of the input string, and we will end up with an array of preprocessed instructions ready for further operations.

After all the inputs are put into an array, they will still need to be put into the reverse polish notation(RPN) so that our computer would be able to understand the operations. The reverse polish notation, different from how our input definition, is in the form of <operand> <operand> … <arithmetic operation> …. As the computer executes a series of operations in RPN, it will look for the first operator and take the two numbers before it as the two operands, then perform the operation accordingly. To convert our input into the reverse polish notation, a queue and a stack will be used, the queue will contain the RPN of our input. The program will start to push elements from the array into the queue:

- If it is an integer (or a floating point number), it will be directly pushed into the queue.
- If it is an operator, it will be compared to the operator at the top of the stack:
    - If the stack is empty or the current operator has higher precedence, the current operator will be pushed into the stack.
    - If the current operator has the same or lower precedence, the operator at the top of the stack will be popped up of the stack and pushed into the queue. Then, the current operator will be compared to the operator at the top of the stack again -- until the current operator has higher precedence than the operator at the top of the stack or until the stack is empty, the current operator will be pushed into the stack.
- If it is a parenthesis:
    - If it is a left parenthesis, it will be directly pushed into the stack.

- If it is a right parenthesis, the program will start to pop the operators from the top of the stack into the queue, until a left parenthesis appears; then both the left and the right parenthesis will be discarded.

- As it reaches the end of the input array, all the operators in the stack will be popped out and pushed into the queue.

Once this process is complete, we will end up with an empty stack, and the queue will contain our input in RPN. Then we will need to create a doubly linked list. A doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. In this case, each node in the doubly linked list will be 12 bytes long: the first 4 bytes will contain the data from the queue, the middle 4 bytes saved a pointer pointing to the previous node, and the last 4 bytes is a pointer pointing to the next node.

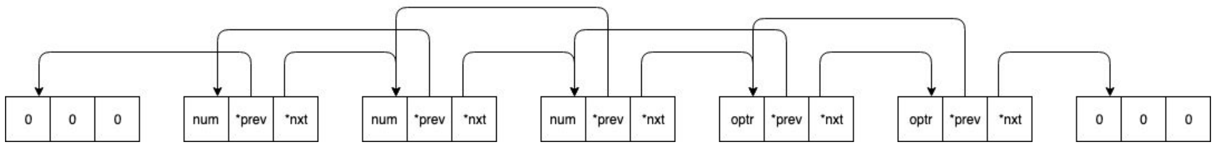


Figure 1: Concept Diagram of Doubly Linked List used in this program

Finally, the program will perform calculations according to our inputs. To do this, a pointer pointing to the start of the doubly linked list will be used.

- First, the data from the where the node that the pointer is pointing to, along with the data from the next two nodes, will be stored into three different registers:

  r1 = node1

  r2 = node1(nextNode) /* node2 */

  r3 = node2(nextNode) /* node3 */

- Then the program will check if the third element is an operator -- this will keep looping (and incrementing every time) until the third element is an operator:

    - If it is not an operator, the program will set the pointer equal to the next-pointer (*nxt) of the current node, so that the pointer will be pointing to the next node. Then it will go back to the first step and check this condition again.
    - If it is an operator, the program will perform calculation corresponding to the third element with the first two elements as operands.
- The result of this calculation will be stored into the node where the pointer is pointing.
- The next-pointer (*nxt) of the node that the pointer is pointing to will be set equal to the next-pointer (*nxt) of the second node after the current node. And The previous-pointer (*prev) of the third node after the node that the pointer is pointing to will be set equal to the previous-pointer (*prev) of the first node after the current node. By doing this, the operands and the operator of the operations those are already done will be discarded from the doubly linked list.

```
node1(nextNode) = node3(nextNode)
node3(nextNode).lastNode = node1
```

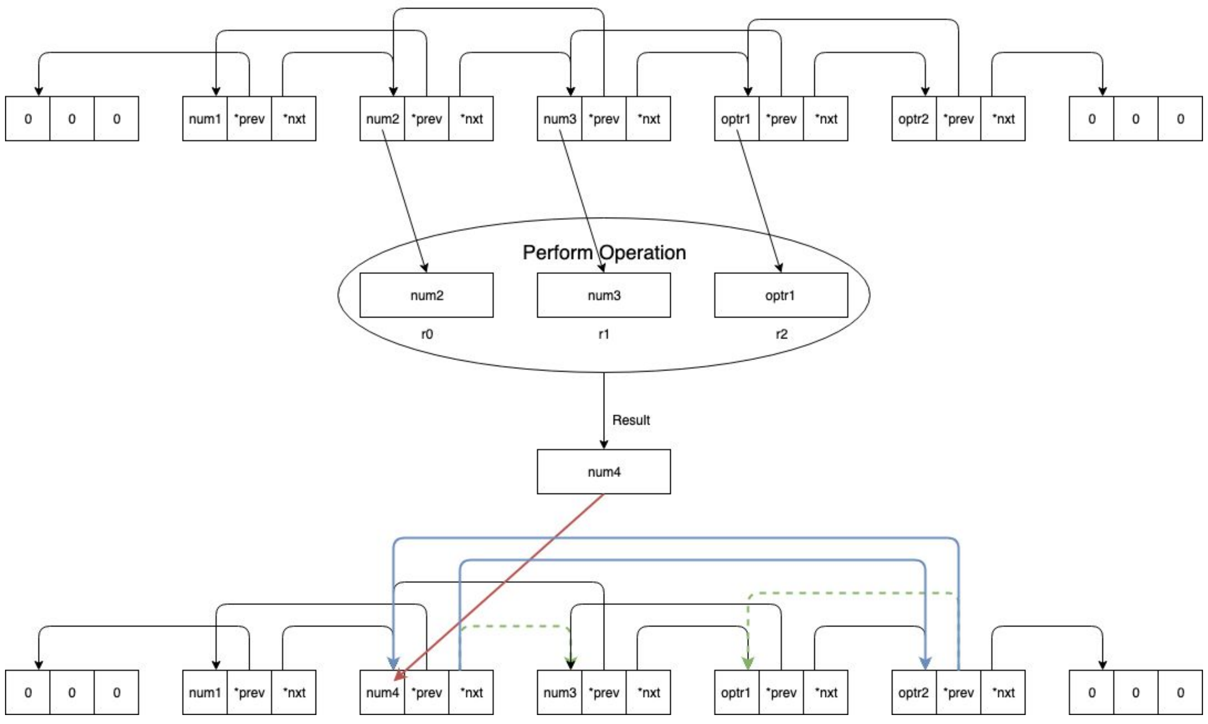


Figure 2: Concept diagram of one iteration of calculation

After every iteration, the program will check if the next node is null:

- If the next node is null, in other words, the next-pointer (*nxt) of the current is pointing to null, then all the calculations are completed, and the program will return the value stored in the current node as the final answer.
- If the next node is not null, then the program will check if the previous node is null:
    - If the previous node is null, in other words, the previous-pointer (*prev) of the current is pointing to null, then the program will go straight to the next iteration by going back to the first step again.
    - If the previous node is not null, then the pointer will be set equal to the previous-pointer (*prev) of the current node so that it will point to one node before the current node, and the program will go back to the first step to start the next iteration.

Challenge

The greatest challenge is to make each part of the code to work. The logic inside each part is complicated, plus that the variable and memory needs to be handled carefully. Often, In order to backtrace a bug, we have to use gdb to set a breakpoint at different points and examine both the registers and the stacks. Also, the three parts are programmed separately and were then combined together. Lots of conflicts happened during the combination and we had to examine each definition and the actual implementation carefully.

Multiple problems can happen together in different parts, and it is hard to differentiate them. When we believe the problem was from the RPN conversion part, we later found that the problem was actually from an improperly placed pointer that restrains the boundary of an array. This problem propagated to all the later code, so we couldn't locate the problem without checking memory step by step using gdb.

Float points don't present too much trouble since the logic to handle them is the same as integers. As the difference only comes from converting strings to numbers and the type of instruction used, we can directly substitute them in the program to ensure float point compatibility.

It is less intuitive to program in a shell, which creates trouble for us. When programming with vim, we accidentally pressed dd (delete a line) somewhere without realizing it. Hence, the program didn't work properly. Because the delete happens long ago, we couldn't undo our actions to restore the line. It takes half an hour for us to locate and fix the missing line.