Basic PMDAS Calculator

Minh-Thai Nguyen, Sophie Jaro

May 2019

1 Program Specifications

This program functions as a basic calculator that applies parentheses, multiplication/division, and addition/subtraction rules respectively to an inputted expression. This program uses ARM assembler for Raspberry Pi. The expression is read as a string from the command line. The program will evaluate an expression of integers, floating point numbers, and up to four operations. Expressions must be inputted as $sum = < operand >_1 < arithmetic\ operator >_1 \ldots < arithmetic\ operator >_n < operand >_{n+1}$ with n $\leq$ 4.

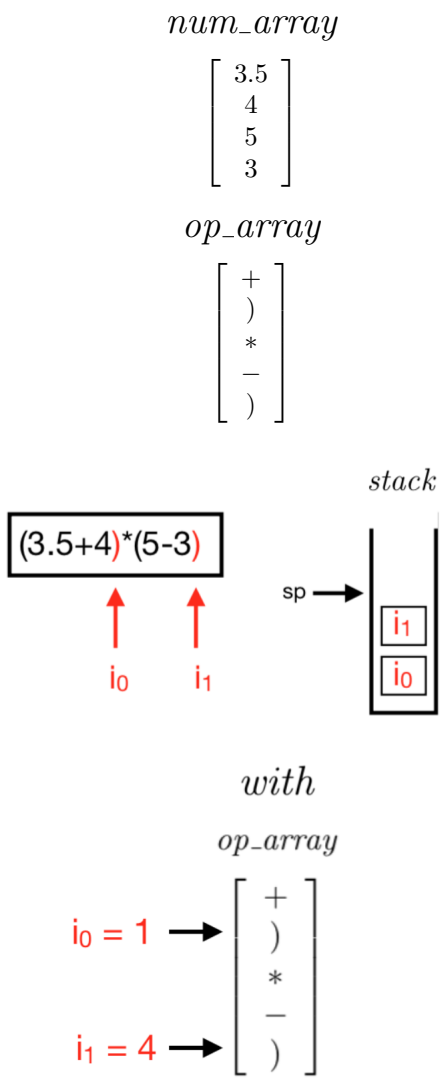
2 Program Architecture

2.1 Reading and Storing Input

The program is a fed a string of operands, operations, and parentheses. The string of inputs is sorted into three arrays. The arrays are operands in the order they appear, operations and right parentheses in the order they appear, and parentheses pushed onto the stack as they appear in the expression.

2.2 Storing Example

The expression $(3.5 + 4) * (5 - 3)$ is stored in the following way:



The above arrays are created by looping through the elements and comparing them to acceptable input characters. First, the element is compared to the null character, which denotes the end of the input string, then ASCII values greater than 57, which are unknown input elements. The program then compares the input to a number. If the element is a number or decimal point, the program moves to the next element. If the element is a left parenthesis, the program

pushes the current index of *op_array* onto the stack. If the element is neither a number or left parenthesis, it is either an operator or an invalid character. The *op_array* index is incremented and the element is compared to addition, subtraction, multiplication, division, and right parentheses. If it is none of those, it is an unknown element.

If the element is a right parentheses, the "Was the last operator a right parenthesis?" value, stored in $r10$, is loaded to $r0$. If $r10$ is 0 if the last operator was not a right parenthesis and 1 if the last operator was a right parenthesis. $r10$ is then set to 1 for the next operator. If last operator was not a right parenthesis, the number is scanned by branching to the *scan_num* function. If the last operator was a right parenthesis, the address of the next character in the input string is stored in the *prev_op* variable.

If the element is an operator, the "Was the last operator a right parenthesis?" value is loaded to $r0$. The "Was the last operator a right parenthesis?" value is set to 0. If last op was not a right parenthesis, the number is scanned by branching to the *scan_num* function. If the last operator was a right parenthesis, the address of the next character in the input string is stored in the *prev_op* variable.

The *scan_num* function works by first setting the current character in the input string to the null character, Ø. The program then passes the value in *prev_op* to the $sscanf$ with the parameters to read a float. By passing the value in *prev_op* and setting the current character to a null character, this function essentially isolates just the number as a string to be read in by $sscanf$. The sscanf function then reads the floating point number and stores it in the next element of the *num_array*. If $sscanf$ throws an error, an error message is displayed as this is indicative of a bad equation format. If not, the program reads the next element in the input string.

End of input occurs when the input string reaches an null character. The last number in the input string is scanned. The value at the bottom of the stack is saved to a register.

2.3 Evaluating Input Arrays

To evaluate the expression, the parentheses must be dealt with in a certain order such that a parent set of parentheses is only evaluated after all the children have been evaluated. By evaluating the parentheses sets from right to left, it can be guaranteed that this condition is held.

$$(1 + 2 * (3 + 4) + 5 * (6 - 7)) + 8$$

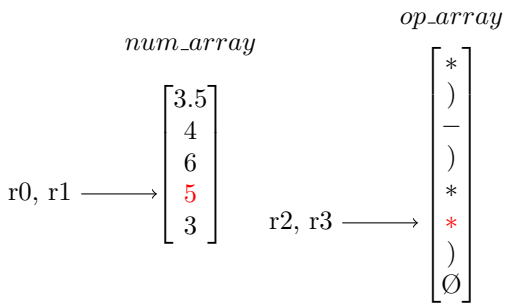
In the equation above, the parentheses set of $(6 - 7)$ should be evaluated first. The order of the evaluation for the parentheses set is given by position of the left parentheses, where the right-most left parentheses is evaluated first. No matter what, the right-most parentheses set in a valid expression should only contain a simple expression (in this case only multiplies, divides, additions, and subtractions). The method employed by this program capitalizes on these

ideas by gradually simplifying the total expression by evaluating the right-most parentheses set until there are no more operations to perform. It chooses which set to evaluate by looking at the top of the stack that stores the indices of the left parentheses. By solving the expression in this manner, only a single *solve* function (a function that solves a simple expression) needs to be written and can be called as many times as needed.

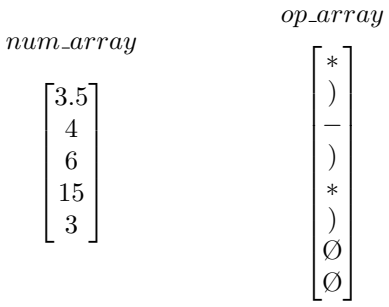
The *solve* function evaluates a simple expression by first initializing the different registers to the starting point of evaluation. These parameters are passed down through r0 and r2 by the calling function. The program then evaluates the multiplies and divides first, rewriting the non-evaluated operators (+ and -) back into the original *op_array*. Once it reaches either a right parentheses or a null character in the *op_array*, the program then moves onto the next pass through, rewriting the parentheses or null character to the *op_array*. If it hit a parentheses, the program will also rewrite all the operators in the *op_array* after it up to and including the null character. This shifting down of the non-evaluated operators is done through a function called $shifts$. After the shifts are done, the program then passes through the same section in the *op_array* and evaluates the addition and subtraction operations. It then does the same shifting process. If the end of the section is signified by a right parentheses, it skips it during the rewriting process.

2.4 Solving Example

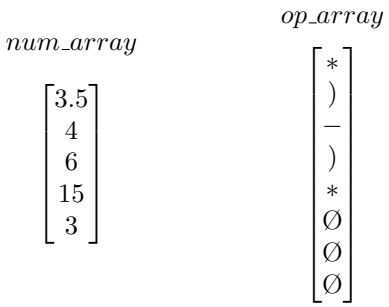
Once sorted, the expression $((3.5 * 4) - 6) * (5 * 3)$ is evaluated as follows:



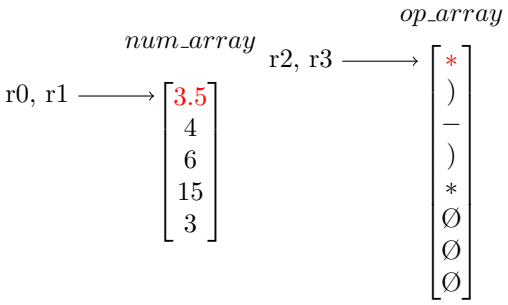
Step 1: Initial state of the two arrays and initialization for the first set of parentheses.



Step 2: Evaluating the $*$ and / operations for the first set of parentheses and shifting the operators and numbers down.



Step 3: Evaluating the $+$ and $-$ operations for the first set of parentheses and shifting the operators and numbers down. In this case, there are no $+$ or $-$ operations, but the parentheses is still skipped in rewriting.



Step 4: Initialization for the second set of parentheses.

$$\begin{array}{cc} num\_array & op\_array \\ \begin{bmatrix} 14 \\ 6 \\ 15 \\ 15 \\ 3 \end{bmatrix} & \begin{bmatrix}) \\ - \\ ) \\ * \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix} \end{array}$$

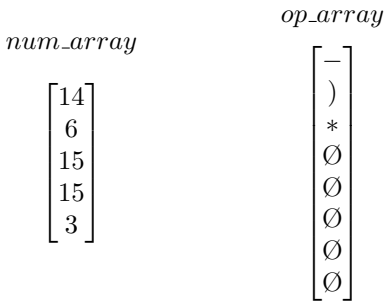
Step 5: Evaluating the $*$ and / operations for the second set of parentheses and shifting the operators and numbers down.

$$\begin{array}{cc} num\_array & op\_array \\ \begin{bmatrix} 14 \\ 6 \\ 15 \\ 15 \\ 3 \end{bmatrix} & \begin{bmatrix} - \\ ) \\ * \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix} \end{array}$$

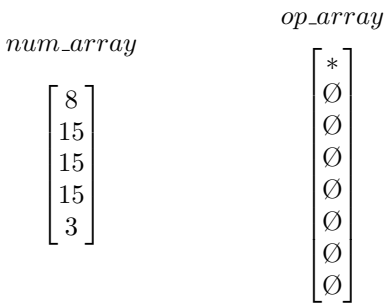
Step 6: Evaluating the $+$ and $-$ operations for the second set of parentheses and shifting the operators and numbers down. In this case, there are no $+$ or $-$ operations, but the parentheses is still skipped in rewriting.

$$\begin{array}{ccccc} & & num\_array & & op\_array \\ \text{r0, r1} & \longrightarrow & \begin{bmatrix} 14 \\ 6 \\ 15 \\ 15 \\ 3 \end{bmatrix} & \text{r2, r3} \longrightarrow & \begin{bmatrix} - \\ ) \\ * \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix} \end{array}$$

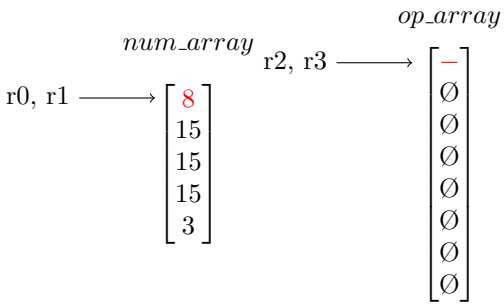
Step 7: Initialization for the third set of parentheses.



Step 8: Evaluating the $*$ and / operations for the third set of parentheses and shifting the operators and numbers down. In this case there are none, so nothing happens.



Step 9: Evaluating the $+$ and $-$ operations for the third set of parentheses and shifting the operators and numbers down. The parentheses is skipped in rewriting.



Step 10: Initialization for the final expression.

$$num\_array \qquad op\_array$$

$$\begin{bmatrix} 120 \\ 15 \\ 15 \\ 15 \\ 3 \end{bmatrix} \qquad \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix}$$

Step 11: Evaluating the $*$ and / operations for the final expression and shifting the operators and numbers down.

$$num\_array \qquad op\_array$$

$$\begin{bmatrix} 120 \\ 15 \\ 15 \\ 15 \\ 3 \end{bmatrix} \qquad \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix}$$

Step 12: Evaluating the $+$ and $-$ operations for the final expression and shifting the operators and numbers down. In this case there are none so nothing happens. The final result is stored in the 0^{th} element of the num_array. This value is then printed to *stdout* using $printf$:

```
The result is:  120.000000
```

3 Errors and Unacceptable Inputs

This program prints errors for invalid user input.

When an invalid character is inputted, an error message is printed and the program ends. The program knows it is an invalid character by eliminating all ASCII values above 57, because no valid input has a higher ASCII value. The program then compares the element to all valid inputs. If the element is not one of those known valid inputs, it must be an invalid input.

When the sscanf function is reading numbers and encounters an error (such as a previous element that was another operation instead of a number), the program will print the bad number format error message. If, when reading a number, more than one decimal point is found between operations, the program will print the bad number format error message.

If an expression started with a left parenthesis, it should end with one. If an expression did not start with a left parenthesis, it should end in a null

character. If neither of these are the case (Example: $input = 4 * 5(-6) + 2$), then the parentheses are misaligned. If it didn't it should end in null. When parentheses are misaligned, the program will print the $error\_bad\_format$ message. The program keeps track of this error by setting a register value to a 1 at the beginning of a parentheses expression and subtracting 1 at the end of the parentheses expression. This value must be zero at the end of the expression. In the case of a mismatch, the value will be 1 or -1, so an error will be thrown.

4 Design Choices and Difficulties

The first notable design choice is the choice to store the position of the left parentheses on the stack. This was done because it allowed for a very easy way of solving nested parentheses because it kept all the operators and operands index aligned. The only draw back was that this method required an additional accompanying function to account for the misalignment of indices due to the other sets of parentheses that lie on the same level as it:

$$(expr1) + (expr2)$$

The added function was the count function which counted the number of right parentheses stored in the array before the start of the of the starting index of the solve. This count serves as the offset for where the index in num_array should lie. The second design choice was to solve the by tackling simple expressions at a time (ie no parentheses). Thanks to the use of the stack and a good method of indexing it was very easy to break down the parentheses into a set of simple expressions. This also saved a lot of code as the same *solve* function could be called at various points in the program. This design also allowed for very trivial ways of testing the validity of expressions by checking sets of parentheses as they're being evaluated. Single precision floating points were using because it didn't seem necessary to have double precision and it allowed the num_array to hold twice as many elements without increasing the size. The drawback to this is that the quantization of numbers is much worse as the numbers get larger. If doubles were to be used, the program would have to account for it by changing the increments, changing the bit alignment, and changing the floating point instructions to account for double precision instead of single precision.

Most of the present difficulties involved dealing with printing floating point numbers. There wasn't too much documentation online that described how to print floating point numbers in ARM using $printf$, but eventually it was figured out. In addition to this, there was some problems with testing with gdb as parentheses wouldn't work as command line arguments, but eventually this was resolved too by surrounding the expression in quotes.