

Assignment 3 – RPN Calculator

Aiden Trager

CSE 13S – Fall 2023

Purpose

The RPN (Reverse Polish Notation) Calculator is a command-line program designed to evaluate mathematical expressions in postfix notation. Its primary purpose is to provide a tool for users to perform arithmetic operations, including addition, subtraction, multiplication, and division, as well as advanced mathematical functions such as trigonometric calculations (sine, cosine, tangent), and square root operations.

How to Use the Program

In order to run the program first make the program and then run it. These are what the commands should look like in your terminal:

```
make calc
```

or

```
make all
```

Followed by:

```
./calc
```

Example: To calculate $4 + 4$, enter it as follows:

```
4 4 +
```

For square root, use "r," for sine, use "s," for absolute value, use "a," for cosine, use "c," and for tangent, use "t." Example: To find the square root of 16, enter it as:

```
16 r
```

Get the result: After entering an expression, the calculator will provide the result. Example output for "4 4 +":

```
Result: 8.0000000000
```

Please note that the calculator checks for well-formed expressions and will notify you of any errors.

Program Design

The RPN (Reverse Polish Notation) Calculator program is organized with a clear separation of responsibilities between different components, including the main evaluation engine, operators, stack management, and mathematical functions.

Data Structures

The program primarily utilizes the following data structures:

Stack: The central data structure for managing operands and results. The stack is implemented as an array with a fixed capacity to store double-precision floating-point numbers. It holds intermediate values during expression evaluation.

Enums and Strings: Various operators and functions are represented as string values and associated with enum constants for easy recognition and processing.

Algorithms

This is the psuedocode for the main evaluation that my program does.

```
function evaluate_expression(input_expression):
    Initialize an empty stack
    Split the input expression into tokens
    For each token in the expression:
        If the token is an operator:
            Apply the corresponding binary or unary operator
        If the token is a numeric value:
            Push it onto the stack
    Check the stack for the final result
    If the stack contains one value:
        Display the result
    Else:
        Notify of an invalid expression
```

This algorithm evaluates RPN expressions by maintaining a stack to manage operands and operators.

Function Descriptions

evaluate_expression(input_expression): This function processes and evaluates RPN expressions. It splits the input expression into tokens and processes each token, applying operators or pushing numeric values onto the stack. It ensures that the expression is well-formed and provides the result.

apply_binary_operator(op): This function applies binary operators such as addition, subtraction, multiplication, and division. It ensures there are enough operands on the stack for the operation.

apply_unary_operator(op): This function applies unary operators such as sine, cosine, tangent, and square root. It checks for a valid number of operands on the stack. Do not simply use your code to describe this. This section should be readable to a person with little to no code knowledge.

parse_double(s, d): This function attempts to parse a double from the input string s. If successful, it stores the parsed value in d and returns true. If parsing fails, it returns false, indicating an invalid token.

stack_push(item): This function pushes a double item onto the stack. It returns true if the operation succeeds, indicating a successful push. If the stack is full, it returns false.

stack_peek(item): This function retrieves the top item from the stack without removing it. If the stack is not empty, it stores the top item in item and returns true. If the stack is empty, it returns false.

stack_pop(item): This function pops the top item from the stack. If the stack is not empty, it stores the popped item in item and returns true. If the stack is empty, it returns false.

stack_clear(): This function resets the stack, clearing all items and setting the stack size to zero.

The program's functions are designed to handle various aspects of RPN expression evaluation, including the processing of numeric values, binary operators, unary operators, and stack management.

Error Handling and Testing

Error Handling:

I have implemented error handling in the code to handle various types of errors and provide error messages to the user. For example, I check for stack underflows, invalid binary operators, and invalid unary operators, and I report appropriate error messages when encountered. I also check for insufficient memory space when pushing values onto the stack and provide an error message if needed. I handle the cases of invalid characters and invalid strings and provide error messages for these cases.

Testing: I have a dedicated testing function, `test_evaluate_expression`, to test my code's functionality. This function tests both valid and invalid expressions, checking that the results match the expected values or that appropriate error messages are generated.

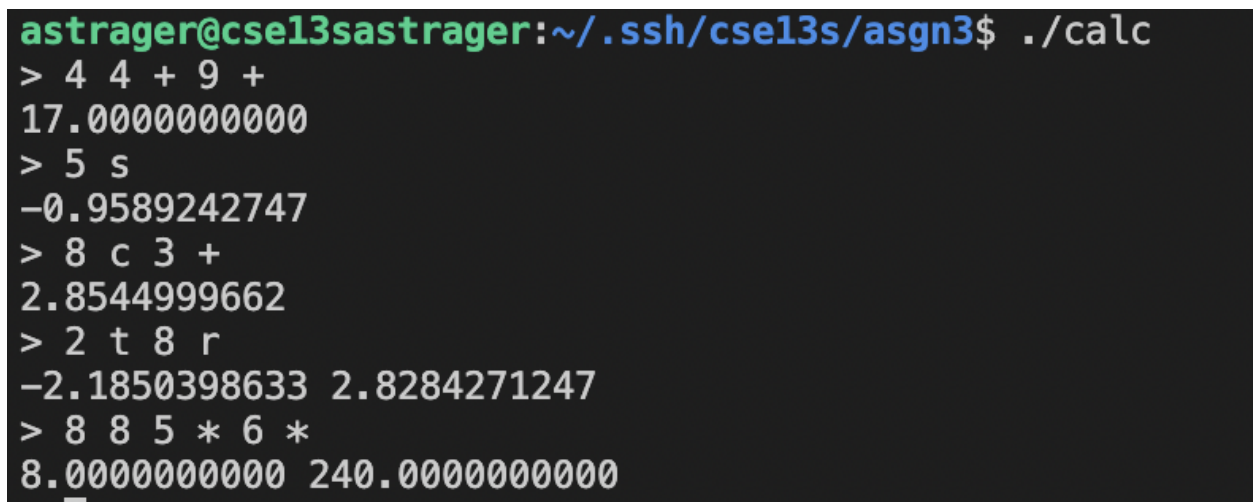
Test Cases: I have provided test cases for valid expressions, including arithmetic operations and trigonometric functions. This helps ensure that my code calculates the results correctly.

Invalid Expression Testing: I have test cases for invalid expressions, ensuring that the error handling code is functioning as expected.

Assertions: I use assertions to compare the actual results with expected results in my test cases. This is a good practice to automatically verify the correctness of my code.

Results

My code is designed to evaluate mathematical expressions in Reverse Polish Notation (RPN). It is expected to handle a variety of mathematical operations and trigonometric functions while providing error messages for invalid expressions. The code is also meant to be configurable through command-line options, allowing the user to choose between using the standard math library or the custom math functions. It works as intended and has allowed me to greatly improve my ability to code in c. I was unable to create a graph of the difference between my trig functions and the math.h ones because I was unable to figure out how to do so effectively with enough data. This was further inhibited by the cancellation of the office hours on Sunday. I attempted to go to the earlier ones as well, but joined slightly too late and could not secure time with Ben.



```
astrager@cse13s:~/.ssh/cse13s/asgn3$ ./calc
> 4 4 + 9 +
17.0000000000
> 5 s
-0.9589242747
> 8 c 3 +
2.8544999662
> 2 t 8 r
-2.1850398633 2.8284271247
> 8 8 5 * 6 *
8.0000000000 240.0000000000
```

Figure 1: Screenshot of the program running.