Assignment 4 – Calc Report Template

Nickolas Tran

CSE 13S – Winter 24

Purpose

In this assignment, you will be implementing stack data structures to replicate the Reverse Polish Notation of a basic scientific calculator in C. Upon execution, the program prompts the user to input a mathematical expression. It then parses the input expression, tokenizes it, and evaluates it using the defined operators and functions. The calculator handles both binary operations, such as addition and multiplication, and unary operations, such as sine and cosine. It also ensures proper error handling, including division by zero and invalid input expressions.

Questions

- Are there any cases where our sin or cosine formula can't be used? How can we avoid this?
  Some cases in which sin or cosine formulas cannot be used are through inverse sine, non-right triangles, and complex numbers. We can avoid this through using appropriate formulas and through library functions, and input validation.

- What ways (other than changing epsilon) can we use to make our results more accurate? [1]
  1. Symbolic representation
  2. Conditioning
  3. Numerical integration
  4. Error analysis

- What does it mean to normalize input? What would happen if you didn't?
  Normalizing input means that you create a consistent set of results and a better generalization of the output. If you didn't normalize input, you would create invalid operations, loss of precision, and unsupported formats.

- How would you handle the expression 321+? What should the output be? How can we make this a more understandable RPN expression?
  For example: Infix notation (standard mathematical notation): 3 + 2. Equivalent postfix (RPN) notation: 3 2 +
  To make the expression "321+" more understandable in RPN, we need to add an operand after the operator "+".
  Infix notation: 3 + 2 + 1. RPN notation: 3 2 + 1 +. This expression would calculate the sum of 3 and 2 first (yielding 5), and then add 1 to the result (yielding 6).

- Does RPN need parenthesis? Why or why not?
  No because the order of operations is determined by the position of the operators relative to their operand and it makes it more straightforward and eliminates the need for complex precedence rules.

[1] hint: Use trig identities

Testing

- Unit testing: Test each function with various input values to ensure correct behavior and handling of edge cases.
- Integration testing: Test the interaction between different components of the calculator program.
- Valid input testing: Test the program with different types of input, including valid expressions, invalid expressions, and edge cases.
- Error handling testing: Ensure that appropriate error messages are displayed when errors occur and that the program gracefully handles exceptional conditions.
- Boundary testing: Test the program with boundary values for stack capacity, operand range, and expression length.
- UI testing:Test the user interface by providing inputs through standard input and verifying the output displayed to the user.

How to Use the Program

The user uses the program to compile and run the code.

```
make clean
make
make format
```

The user then uses "./calc" to run the program.

```
./calc
5 10 +
```

Program Design

- Stack Data Structure: The stack supports operations such as push, pop, peek, clear, and printing.
- Main Algorithms: The core algorithm of the program involves reading expressions in RPN format, tokenizing them, and evaluating them using stack-based operations.
- Input/Output Handling: The program displays prompts for user input and outputs the result of expression evaluation.
- Error Reporting: Detailed error messages help users understand the nature of the problem and take appropriate actions.

Pseudocode

For stack.c

```
stack_push(item):
    if stack_size < STACK_CAPACITY:
        stack[stack_size] = item
        stack_size += 1
        return true // Push successful
    else:
        return false // Stack is full

stack_peek(item):
    if stack_size > 0:
        item = stack[stack_size - 1]
        return true // Peek successful
    else:
        return false // Stack is empty
```

```
stack_pop(item):
    if stack_size > 0:
        item = stack[stack_size - 1]
        stack_size -= 1
        return true // Pop successful
    else:
        return false // Stack is empty

stack_clear():
    stack_size = 0

stack_print():
    print "Stack contents:"
    for i from 0 to stack_size - 1:
        print stack[i]
END FUNCTION
```

For operator.c

```
operator_add(lhs, rhs):
    return lhs + rhs

operator_sub(lhs, rhs):
    return lhs - rhs

operator_mul(lhs, rhs):
    return lhs * rhs

operator_div(lhs, rhs):
    if rhs == 0.0:
        return nan
    return lhs / rhs

apply_binary_operator(op):
    if not stack_pop(rhs) or not stack_pop(lhs):
        return false
    result = op(lhs, rhs)
    stack_push(result)
    return true

apply_unary_operator(op):
    if not stack_pop(x):
        return false
    result = op(x)
    stack_push(result)
    return true

parse_double(s, d):
    *d = strtod(s, &endptr)
    return endptr != s
END FUNCTION
```

For mathlib.c

```
Abs(x):
    if x < 0:
        return -x
    else:
        return x
```

```
Sqrt(x):
    if x < 0:
        return NaN
    else:
        return sqrt(x)

Sin(x):
    result = x
    term = x
    factorial = 1.0
    power = x
    sign = -1
    for i from 1 to 10:
        power *= x * x
        factorial *= (2 * i) * (2 * i + 1)
        term = power / factorial
        result += sign * term
        sign *= -1
    return result

Cos(x):
    result = 1.0
    term = 1.0
    factorial = 1.0
    power = 1.0
    sign = -1
    for i from 1 to 10:
        power *= x * x
        factorial *= (2 * i) * (2 * i - 1)
        term = power / factorial
        result += sign * term
        sign *= -1
    return result

Tan(x):
    return Sin(x) / Cos(x)
```

For calc.c

```
main()
    Display prompt for user input
    Loop until user enters 'q':
        Read input from user
        Tokenize input string
        For each token in the input:
            If token is a number:
                Convert token to double and push onto stack
            Else if token is an operator:
                If operator is binary:
                    Pop two operands from stack
                    Apply binary operator to operands
                    Push result back onto stack
                Else if operator is unary:
                    Pop one operand from stack
                    Apply unary operator to operand
                    Push result back onto stack
                Else:
                    Print error message for unknown operator
            Else:
```

```
            Print error message for unknown token

    Peek the result from the stack
    Print the result
    Clear the stack for next calculation

Exit the program
```

Function Descriptions

DO NOT JUST PUT THE FUNCTION SIGNATURES HERE. MORE EXPLANATION IS REQUIRED.

For function stack push():

- Input: 'item'
- Output: Boolean value if true, false if stack is full.
- Purpose: Pushes an item onto the stack

For function stack peek():

- Input: None
- Output: Boolean value if true, false if stack is empty. Updates value of 'item' to the top of the stack
- Purpose: Allows peeking at the top item of the stack without removing it.

For function stack pop():

- Input: None
- Output: Boolean value if true, false if stack is empty. Updates value of 'item' to popped item from the stack
- Purpose: Pops an item from the top of the stack.

For function stack clear():

- Input: None
- Output: None
- Purpose: Clears the stack but setting the variable to zero.

For function stack print():

- Input: None
- Output: None
- Purpose: Prints the contents of the stack

For function operator add():

- Input: 'lhs' and 'rhs'
- Output: Result of adding 'lhs' and 'rhs'
- Purpose: Compute the addition of two numbers

For function operator sub():

- Input: 'lhs' and 'rhs'
- Output: Result of subtracting 'lhs' and 'rhs'
- Purpose: Compute the subtraction of two numbers

For function operator mul():

- Input: 'lhs' and 'rhs'
- Output: Result of multiplying 'lhs' and 'rhs'
- Purpose: Compute the multiplication of two numbers

For function operator div():

- Input: 'lhs' and 'rhs'
- Output: Result of dividing 'lhs' and 'rhs'
- Purpose: Compute the division of two numbers

For function apply binary operator():

- Input: Function pointer 'op' of type 'binary operator fn'
- Output: Boolean value indicating success or failure
- Purpose: Apply a binary operator function to the top two elements of the stack

For function apply unary operator():

- Input: Function pointer 'op' of type 'unary operator fn'
- Output: Boolean value indicating success or failure
- Purpose: Apply a unary operator function to the top element of the stack

For function parse double():

- Input: String 's' representing a double value, pointer to 'd' where the parsed value will be stored
- Output: Boolean value indicating whether parsing was successful or not
- Purpose: Parse a string representation of a double and convert it to a double value

For function Abs():

- Input: 'x'
- Output: Absolute value of 'x'
- Purpose: Computes the absolute value of a number

For function Sqrt():

- Input: 'x'
- Output: Square root of 'x' or nan if negative
- Purpose: Computes the square root of a non-negative number

For function Sin():

- Input: 'x'

- Output: Sine of 'x'
- Purpose: Computes the sine of an angle using Taylor series

For function Cos():

- Input: 'x'
- Output: Cosine of 'x'
- Purpose: Computes the cosine of an angle using Taylor series

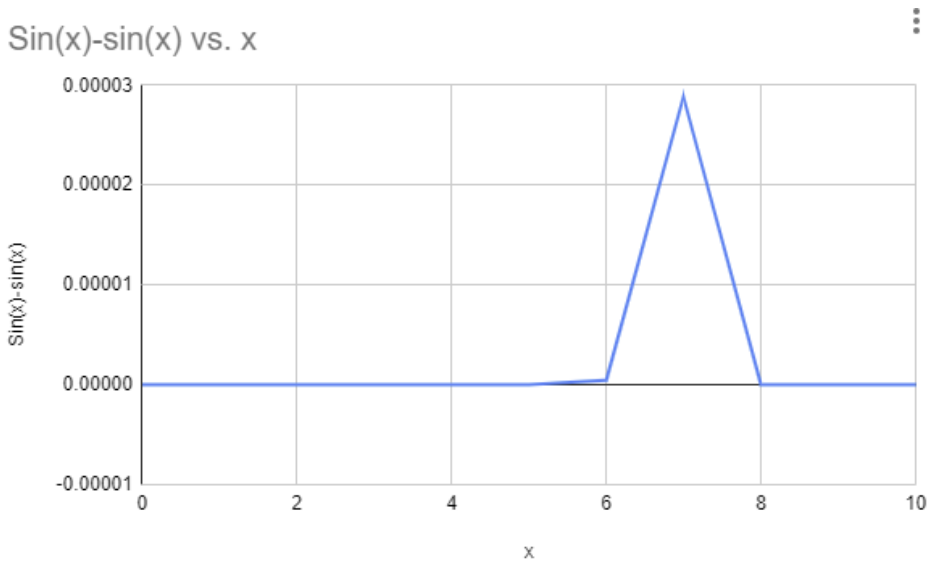
For function Tan():

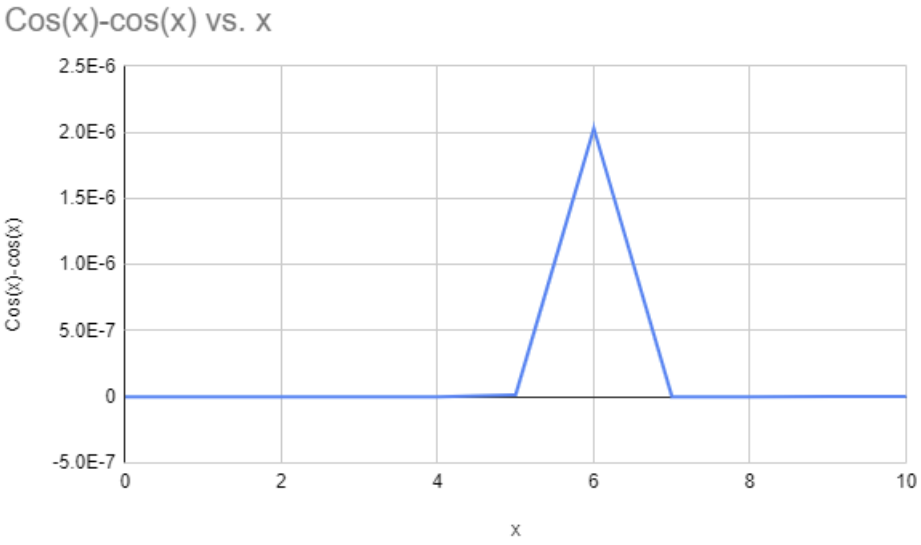
- Input: 'x'
- Output: Tangent of 'x'
- Purpose: Computes the tangent of an angle

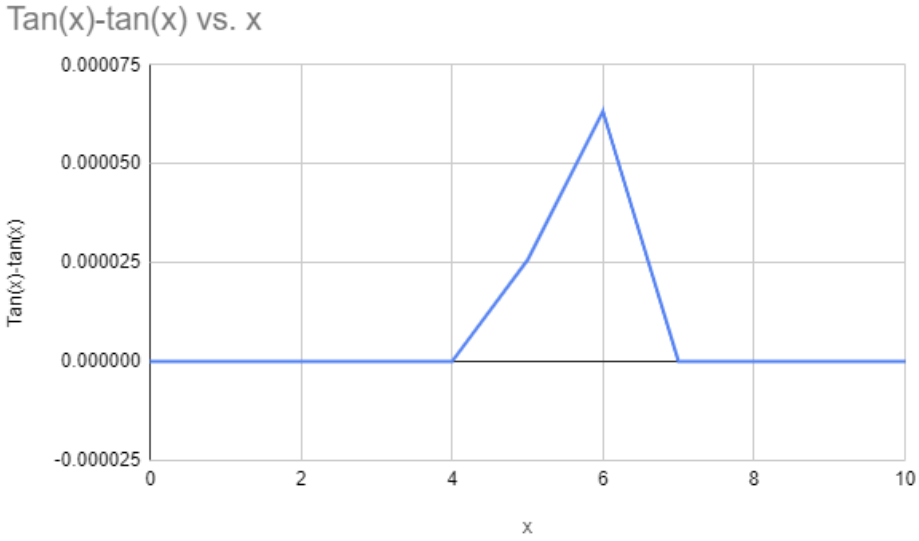
For function main():

- Input: User input of calculator equation in RPN format
- Output: The output of the calculator of user input
- Purpose: The main function of the calculator program. It reads expressions in Reverse Polish Notation (RPN) from the user, evaluates them, and prints the result

Results




Cos(x)-cos(x) vs. x
2.5E-6
2.0E-6
1.5E-6
1.0E-6
5.0E-7
0
-5.0E-7
Cos(x)-cos(x)
0
2
4
6
8
10
x


Tan(x)-tan(x) vs. x
0.000075
0.000050
0.000025
0.000000
-0.000025
Tan(x)-tan(x)
0
2
4
6
8
10
x