Assignment 3 – Calculator

Johann Lee

CSE 13S – Fall 2023

Purpose

The purpose of this program is to emulate a scientific calculator that makes use of the ¡math.h¿. This calculator will use Reverse Polish Notation and will be able to calculate trigonometric functions such as cos, sin, and tangent.

How to Use the Program

This program will consist of a Makefile that was created by the writer and will require the command "make calc" to compile and "./calc" to open up the calculator program. Some command line prompts while typing "./calc" are flags -m and -h. "./calc -m" will swap the trigonometric functions to follow the ones in the math library instead of the custom-written trig functions. "./calc -h" will show all pre-execute commands which for this project are only -m and -h as mentioned before. The calculator requires all inputs to be in the form of Reverse Polish Notation meaning that numbers must be given and followed by an operation and the program reads from left to right until it reaches an operation. An example of this would be 2 + 2 which in Reverse Polish Notation would be 2 2 +. The operations signs for this calculator are as follows, "+ - * / s c t". The operations are for the most point self- explanatory with the trig functions being represented by their first letter. Any incompatible inputs will return their respective error message. To exit the program Ctrl+D or Ctrl+C can be used.

Program Design

This program consists of 11 files, 5 of which should not be edited. The files that can be edited are as listed:

- Makefile: Contains all the rules to compile and create an executable
- calc.c: Contains the main program which holds all the other files together
- tests.c: Contains a main function to test other code
- mathlib.c: Contains all required math functions using approximations
- operators.c: Contain basic operations in their wrapper functions
- stack.c: Contains implementation of stack ADT

Functions

The functions within these files and their descriptions are as follows:

- double Abs(double x)
  - This function accepts a double and returns the absolute value of the double x.

```
double abs (double x)
    if x is greater than 0
        do nothing
    if x is less than 0
        multiply it by -1
    return x
```

- double Sqrt(double x)
  - This function returns the square root of the double x.
  - The code for this is given to us by Jess Srinivas and Ben Grant on asgn3.pdf page 4 and is as follows

```
double Sqrt(double x) {
    // Check domain.
    if (x < 0) {
        return nan("nan");
    }

    double old = 0.0;
    double new = 1.0;

    while (Abs(old - new) > EPSILON) {
        // Specifically, this is the Babylonian method--a simplification of
        // Newton's method possible only for Sqrt(x).
        old = new;
        new = 0.5 * (old + (x / old));
        }

    return new;

    }
```

- double Sin(double x)
  - This function returns the sine of the input double x which should be given in radians. This will not use the sin from the math library and outputs a normalized angle between zero and 2pi.
  - The pseudo code for this function is located in the Algorithms section
- double Cos(double x)
  - This function returns the cosine of the input double x which should be given in radians. This will not use the cos from the math library and outputs a normalized angle between zero and 2pi.
  - The pseudo code for this function is located in the Algorithms section
- double Tan(double x)
  - This function returns the tangent of the input double x which should be given in radians. This will not use the sin, cos, or tan from the math library and outputs a normalized angle between zero and 2pi.
  - The pseudo code for this function is located in the Algorithms section
- bool apply_binary_operator(binary_operator_fn op)

- This functions takes in an operator and accesses the global stack. It then pops the first 2 elements on the stack, and calls the op function with those two (first element is right side, second is left). It then pushes the result of the op on these elements. If there are no two elements then it returns false.

```
bool apply_binary_op (binary_op_fn op)
    if stack has less than 2 elements
        return false
    else
        right hand = pop element
        left hand = pop element
        op with right hand and left hand
        push op result to stack
        return true
```

- bool apply_unary_operator(unary_operator_fn op)
  - This function takes in an operator and accesses the global stack. It pops element then applies the the operation before pushing back to the stack. If there are no elements to pop then returns false.
  - The code for this function is given by Jess Srinivas and Ben Grant asgn3.pdf page 3
- double operator_add(double lhs, double rhs)
  - Takes the sum of doubles lhs and rhs and returns it.

```
add function
    variable = lhs + rhs
    return variable
```

- double operator_sub(double lhs, double rhs)
  - Takes the difference of doubles lhs and rhs and returns it.

```
sub function
    variable = lhs - rhs
    return variable
```

- double operator_mul(double lhs, double rhs)
  - Takes the product of doubles lhs and rhs and returns it.

```
mul function
    variable = lhs * rhs
    return variable
```

- double operator_div(double lhs, double rhs)
  - Takes the quotient of doubles lhs and rhs and returns it.

```
div function
    variable = lhs / rhs
    return variable
```

- my_fmod(double lhs, double rhs)
  - Returns the remainder of division between lhs and rhs.

```
[frame=single]
                mod function
                    variable = fmod(lhs, rhs)
                    return variable
```

- bool parse_double(const char *s, double *d)
  - This function attempts to parse a double-precision floating point number from string s. If it is successful, it stores the number in the location pointed by d and returns true. If the string is not a valid number, it returns false.
  - This function is given to us by Jess Srinivas and Ben Grant asgn3.pdf page 6
- bool stack_push(double item)
  - Pushes an item to the top of the stack and updates the stack_size. Returns true if it was possible else if the stack is at capacity it returns false.

```
stack push
    if stack size >= stack capacity
        print error message
        return false
    else
        push item to top of stack
        add 1 to stack size
        add 1 to top variable
        return true
```

- stack_peek(double *item)
  - Copies the first item of stack to address pointed by item. Returns false if stack is empty.

```
stack peek
    if stack size is 0
        print error message
        return false
    else
        copy top of stack to item
        return true
```

- stack_pop(double *item)
  - Stores the first item of the stack in memory pointed at item. If the stack is empty it does not modify item and returns false. Updates stack_size if pop goes through.

```
stack pop
    if stack size =0
        print error message
        return false
    else
        remove top of stack and copy to item
        subtract 1 to stack size
        subtract 1 to top variable
        return true
```

- stack_clear(void)

- Sets the size of the stack to zero

```
stack clear
    stack size = 0
    top variable = -1
```

- stack_print(void)
  - This function prints every element in stack to stdout and is separated by only spaces. There is no newline after. If the stack is 0 it returns a blank. If not, it prints each each element of the stack with a for loop. This function is given to us by Jess Srinivas and Ben Grant asgn3.pdf page7
- main function
  - This is the main function of my code.

```
main argc argv
    int opt for getopt
    bool if my math is being used = true

    while opt for getopt hm != -1
        switch case
            m -> swap to math library
            h -> print help
            unknown -> continue to program

    infinite while
        char expression
        char pointer save
        boolean error default false
        double holding parse variable

        print "> "
        accept line of input
            remove '\n' that comes with fget

        pointer token
        while token is not NULL or error is false
            iterate through line with token
        check if token is in binary or unary operators
            if so set value to true that for switch case
        else parse_double token and check if valid double
                if yes then push to stack
                if no then pop error

            if binary is flipped true
                switch case
                    + -> if !apply_binary_op(add)
                            print error binary
                            error = true
                         break
        do this for each character in binary operators

            if unary is flipped true and my math is true
                switch case
                    s -> if !apply_unary_op(Sin)
```

```
                            print error unary
                            error = true
                        break
    do this for each character in my unary operators

            if unary is flipped true and my math is false
                switch case
                    s -> if !apply_unary_op(sin)
                            print error unary
                            error = true
                        break
    do this for each character in library unary operators

        within token while -> token = strtok_r

        within forever while ->
            if error is false
                print stack
            clear stack

    return 0
```

Algorithms

The algorithms section are meant to describe the the functions for sin, cos, tan, and sqrt.

- Sine
  - Sine utilizes a method that uses the Maclaurin series in order to calculate itself. This can be be best viewed as portions. There are portions that are not mentioned below which is the normalization of double x within 0 to 2pi which will help with accuracy.
    * The first portion would be the top of the fraction being x * x on each interval. This is easy as the last term can be multiplied by x * x.
    * The second portion is harder as it is the factorial of the bottom of the fraction. For sin this changes by increasing the factorial value by 2 while starting from 1!. This means that it must have some relation with n if we are to use the last term. Taking into concern that the first n is 0 and may not work we set n to 1 and compensate by changing our initial variables to the values of sin maclaurin at n = 0. This gives us the equation of ((n + (n - 1)) * (2n)). This will always increase the previous factorial by 2.
    * The final portion is the alternating 1 and -1 which can just be put into a separate variable and changed on each iteration of n.
    * The summation of these are what represents sin using a Maclaurin series. The pseudocode is below:

```
sin (x)
    set x to 0-2pi

    int n = 1;
    double radians = x as that is x at n=0
    double holder = x, previous term that will be multiplied
    int swap = -1 as we are already on n=1

    while (will always run)
        holder = holder multiplied by x^2/((n + (n - 1)) * (2n))
```

```
                    radians = radians + swap * holder
                    swap = - swap (here 1 is changing to -1 to vice versa)
                    n increase by 1

                    if (absolute value of holder is < epsilon)
                        run one more iteration of adding holder to radians
                        break

                set radians to between 0-2pi
                return radians
```

- Cosine
  - Cosine utilizes a method that uses the Maclaurin series in order to calculate itself. This can be be best viewed as portions. There are portions that are not mentioned below which is the normalization of double x within 0 to 2pi which will help with accuracy. Cosine is very similar to sine being shifted over on the x-axis.
  - The first portion would be the top of the fraction being x * x on each interval. This is easy as the last term can be multiplied by x * x.
  - The second portion is harder as it is the factorial of the bottom of the fraction. For sin this changes by increasing the factorial value by 2 while starting from 1!. This means that it must have some relation with n if we are to use the last term. Taking into concern that the first n is 0 and may not work we set n to 1 and compensate by changing our initial variables to the values of sin maclaurin at n = 0. This gives us the equation of ((n + (n - 1)) * (2n)). This will always increase the previous factorial by 2.
  - The final portion is the alternating 1 and -1 which can just be put into a separate variable and changed on each iteration of n.
  - The summation of these are what represents sin using a Maclaurin series. The pseudocode is below:

```
            cos (x)
                set x to 0-2pi

                int n = 1;
                double radians = 1 as that is radians at n=0
                double holder = 1, previous term that will be multiplied
                int swap = -1 as we are already on n=1

                while (will always run)
                    holder = holder multiplied by x^2/((n + (n - 1)) * (2n))
                    radians = radians + swap * holder
                    swap = - swap (here 1 is changing to -1 to vice versa)
                    n increase by 1

                    if (absolute value of holder is < epsilon)
                        run one more iteration of adding holder to radians
                        break

                set radians to between 0-2pi
                return radians
```

- Tangent
  - Tangent is much simpler as it just requires our sine function and cosine function from before and creating a quotient between them with sine being the numerator and cosine being the denominator.

```
Tan (x)
    variable = sin(x) / cos(x)
    return variable
```

- Square root
  - Square root is a given function. It is shown earlier and explaining it begins with the first if statement. This statement checks if the given input is less than 0 which it returns nan or "undefined" which makes sense if square rooting a negative number. The next portion is running a reverse squared function through a while loop until it is the absolute value of the old value - the new value is less than the given epsilon.

Testing

My testing is relatively simple and is mostly comprised of testing my unary functions to the values of the internal math library functions. This is done by writing the difference between my custom functions and the functions provided by the math library in C into a CSV file. These CSV files were then output and represented as graphs.

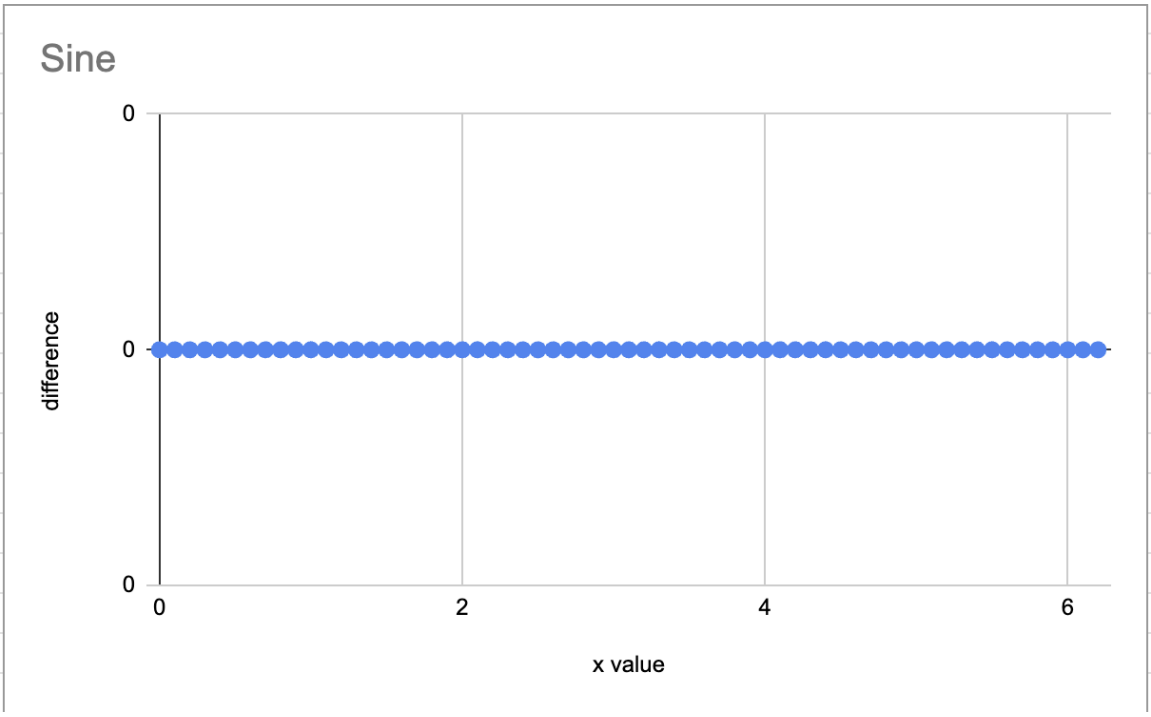

Figure 1: Here is the sine graph with the x-axis from 0 to 2pi. The y-axis shows the difference in output between custom Sin and actual sin with a range of -1e-14 to 1e-14. It can be seen that there is no difference between the two values showing that the custom Sin function accurately portrays real sine.

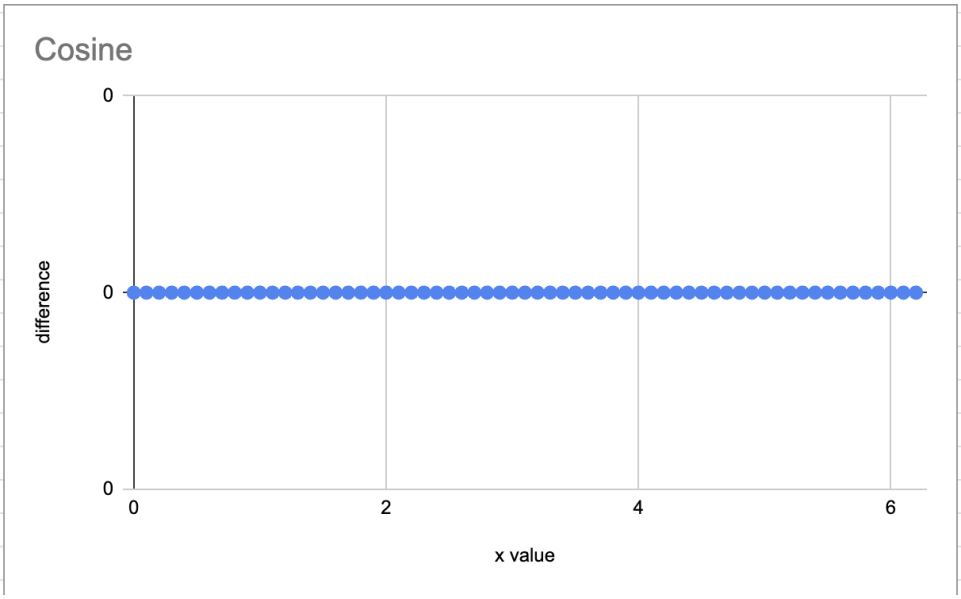

Figure 2: Here is the cosine graph with the x-axis from 0 to 2pi and the y-axis from -1e-14 to 1e-14. Once again just like the sine graph there is no movement of the y-axis between 0 and 2pi showing that the custom cos function is a good representation of actual cosine.

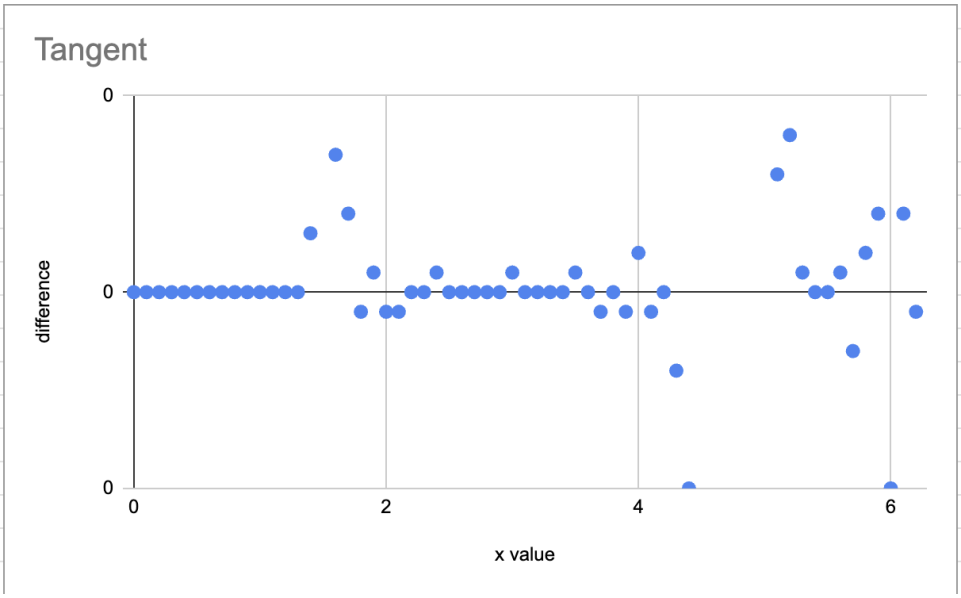

Figure 3: Here is the tangent graph with x-axis from 0 to 2pi and y-axis from -1e-14 to 1e-14. Unlike sine and cosine, we can see a clear differentiation between custom Tan and actual Tan. While this difference is minor as the is nearly 14 places away from the decimal it shows a tangible difference. This difference is due to the inaccuracies of double in C and its reliability only until 15-16 decimal places and is highlighted by the implementation of tan using both custom sin and custom cos.

Results

The results of the calculator were successful as it is able to use its stack of size 64 to its full extent and is able to use all given operators. By inputting any series of numbers (within the limits of 64 numbers) in reverse polish notation my calculator is able to convert that into a product of all the input operations. It is also

able to differentiate between characters that are operations and garbage characters. If a garbage character is given it will raise an error before asking for a new input. A bug that was present in both the given calc program and my calc program is that if there is a garbage letter attached to a number with no space it will read the number until the character.

Figure 4: Here is the given calc program and an example of the bug

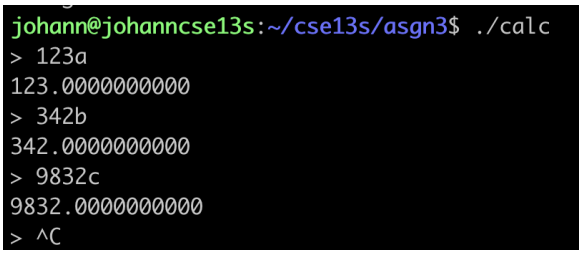

Figure 5: and here is an image of my program which has the same bug

Given that the resources calculator program also contains this bug I had assumed that this functionality was okay given that we are meant to use the given calculator as a reference. Besides this, my calculator runs as programmed seen in the images below.

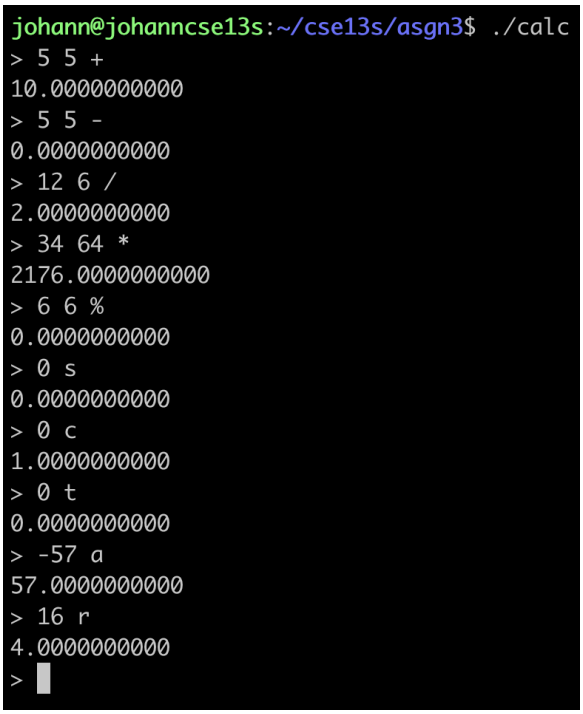

Figure 6: Here is an example of all outputs and how they work as expected.

References

- asgn3.pdf - Jess Srinivas and Ben Grant
  - Pages used: 3,4,6,7,8,9