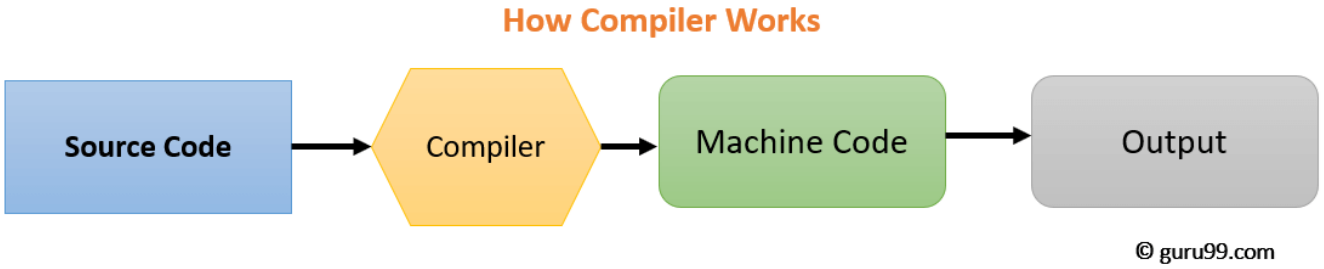
Introduction

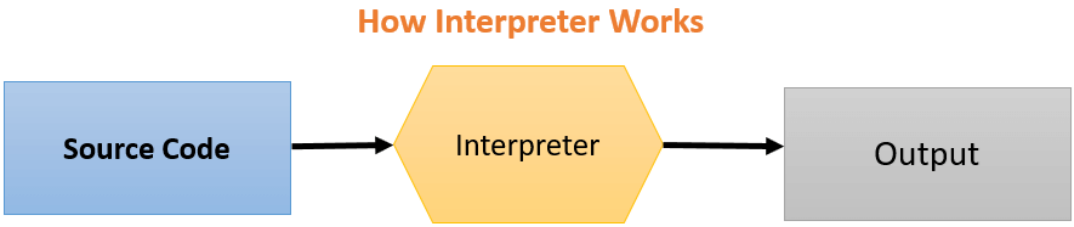
A **Compiler** is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. It is a standalone program and it generates a separate file of compiled machine code every time the code is compiled.

An **Interpreter** is a computer program that directly executes, i.e. performs instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program. Therefore, the interpreter gives an instant output but is a slower alternative to a compiler. The goal of an **Interpreter** or a **Compiler** is to translate a source program in some high-level language into some other form.

Examples of Interpreted language include Python, Javascript, LISP, Ruby, etc.

There are advantages and disadvantages to both compiler and interpreter and the problem of which one to use when depends on the problem being solved. Should one need speed but does not care about evaluating the program repeatedly to achieve that speed, one should use a compiler. If one is willing to compromise on speed for accuracy, line by line execution and debugging, then an Interpreter can be used.





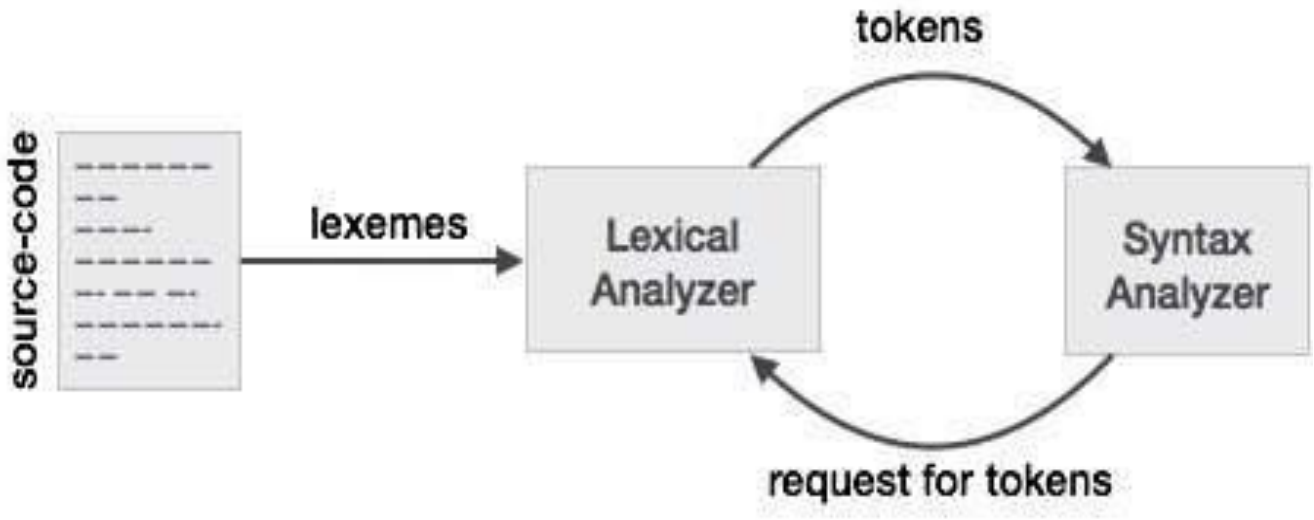
Terminologies

Token - A token is an object that has a type and a value

Lexical Analysis - The process of breaking the input string into tokens is called lexical analysis. The part of the interpreter that does it is called a lexical analyzer, or lexer for short.

Lexeme - A lexeme is a sequence of characters that form a token.

Parsing - The process of recognizing a phrase in the stream of tokens is called parsing. The part of an interpreter or compiler that performs that job is called a parser. Parsing is also called syntax analysis, and the parser is also aptly called a syntax analyzer.



Problem Statement

To implement an interpreter in C that performs basic mathematical operations such as addition, subtraction, multiplication, division and modulus.

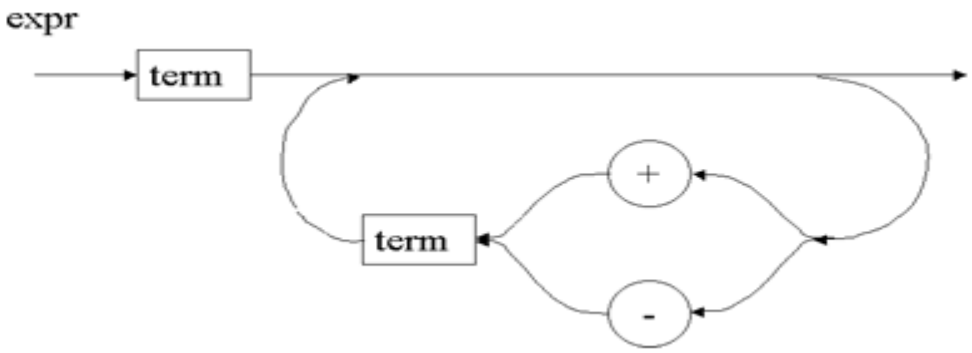
Literature Review

One of the main areas that vastly explored is the inter-mediate representation of the source code. This representation varied from simple AST's, CST's, byte-code, machine instructions etc. Any representation form that is chosen is based on the requirements and constraints of the language and platforms. Most of these representations are developed keeping in mind the desktop computer which has sufficient computing resources like memory, CPU and disk space with other OS capabilities like multi-threading and multi-tasking. Except for KVM there is no other known implementation that works satisfactorily for the smaller electronic devices. In object oriented programming language like C++, methods of the class are being called by the instance of the class (object) through the function pointer.

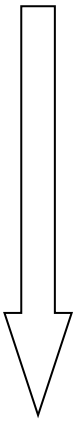
Python Interpreter was developed in 1990 by Guldo Von Rossum. Initially a very slow fork of the LISP Interpreter, It was improved with many bug fixes and improvisations until it became one of the most widely used languages in the industry today.

Javascript Interpreter was developed in the 1980s as a compiled language. However it was changed to an interpreted language with the expansion of the internet. Today it is one of the most used languages and makes up 90% of all web applications.

Graphically, the arithmetic expressions in this can be represented with the following syntax diagram:



A **syntax diagram** is a graphical representation of a programming language's syntax rules. One can read the above syntax diagram as following: a term optionally followed by a plus or minus sign, followed by another term, which in turn is optionally followed by a plus or minus sign followed by another term and so on.



expr

factor ((MUL | DIV) factor)*

factor MUL factor ((MUL | DIV) factor)*

factor MUL factor DIV factor

INTEGER MUL INTEGER DIV INTEGER

3 * 7 / 2

The above is a derivation of a grammar for expression 3 * 7 / 2.

If 7 - 3 - 1 is treated as 7 - (3 - 1) then the result would be unexpected 5 instead of the expected 3.

In ordinary arithmetic and most programming languages addition, subtraction, multiplication, and division are left-associative. This is precedence of operators. When an operand like 3 in the expression 7 + 3 + 1 has plus signs on both sides, a convention is needed to decide which operator applies to 3. Is it the one to the left or the one to the right of the operand 3? The operator + associates to the left because an operand that has plus signs on both sides belongs to the operator to its left and so + is said to be left-associative.

Below is the precedence table,

	Operator Precedence
1	! Logical not (Highest)
2	( ) Parenthesis
3	*, /, %
4	+, -
5	>, >=, <, <=
6	==, !=
7	&& (AND)
8	|| (OR)
9	= (Lowest)

Grammar:

expr : term ((PLUS | MINUS) term)*
term : factor ((MUL | DIV) factor)*
factor : (MINUS) factor | INTEGER

By the above grammar, one can identify that the operations possible would be Addition, Multiplication, Division and Subtraction. However, one could add another operator Modulus that returns remainder of a division. Also, one can add negative sign to the first factor allowing negative integers for mathematical operations.

Hardware and Software Requirements

Hardware

1 GB RAM

8 GB Hard Disk

Software

GCC Compiler Collection

Operating System: Microsoft Windows / Linux

Design

Algorithm

Step 1: Start

Step 2: Input a Mathematical Expression

Step 3: From left to right, consider each number or operator as a token

Step 4: If token is a number, then it is a factor (INTEGER), Store it and proceed to next token

Step 5: If token is an operator, compare it with defined operators.

Step 6: If operator is one of defined operators then proceed to the next token.

Step 7: Perform the required operation on the first and second factor and store it in a variable [End If]

Step 8: Repeat Steps 4 – 7 until EOF is encountered

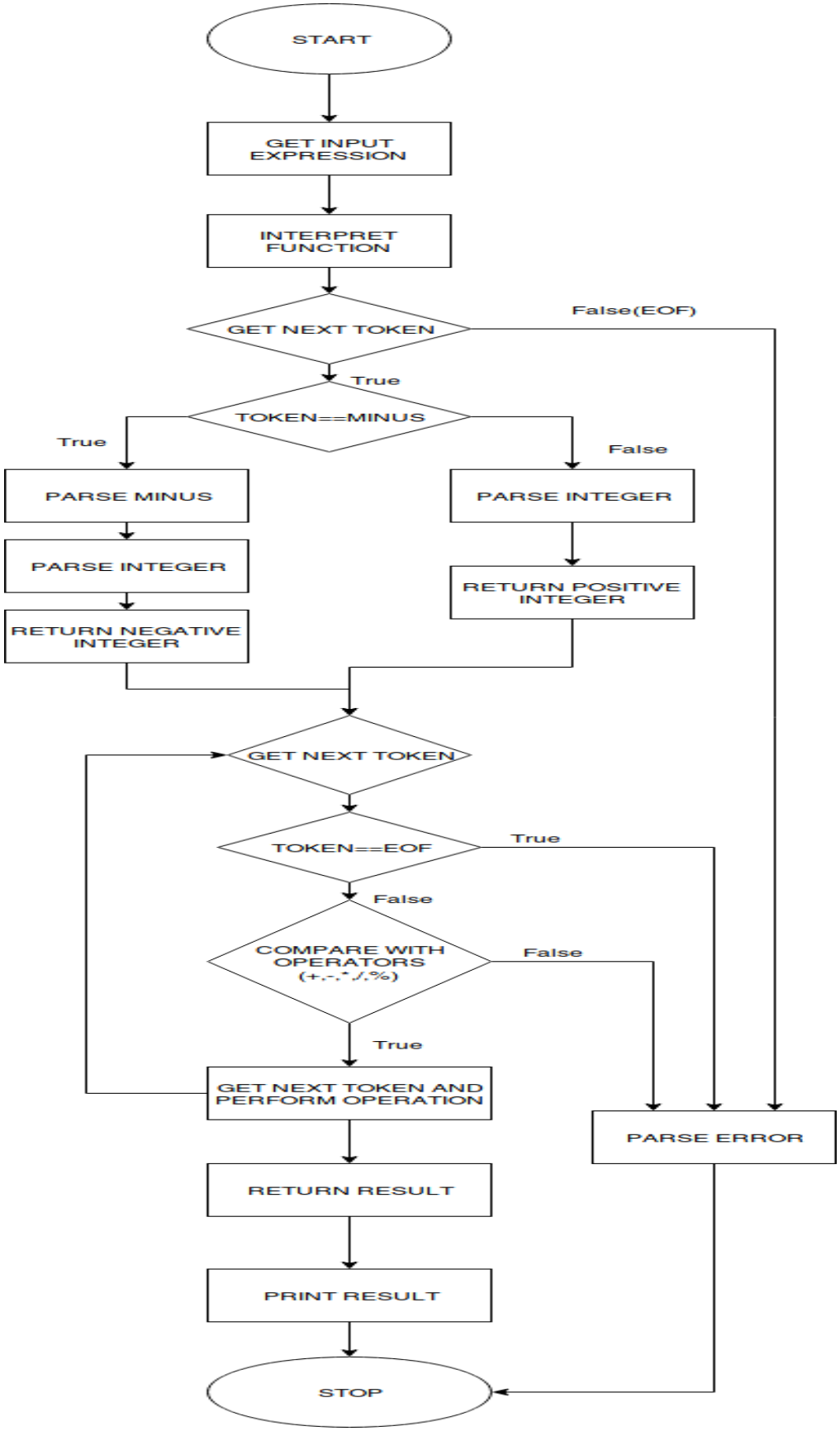
Step 9: When EOF is encountered, Output the result of the mathematical Expression.

Step 10: Stop.

Grammar

expr : term ((PLUS | MINUS | MODULUS) term)
term : factor ((MULTIPLY | DIVIDE) factor)
factor : (MINUS) factor | INTEGER

Flowchart


START
GET INPUT EXPRESSION
INTERPRET FUNCTION
GET NEXT TOKEN
False(EOF)
True
TOKEN==MINUS
True
False
PARSE MINUS
PARSE INTEGER
PARSE INTEGER
RETURN POSITIVE INTEGER
RETURN NEGATIVE INTEGER
GET NEXT TOKEN
TOKEN==EOF
True
False
COMPARE WITH OPERATORS (+,-,*,/,%)
False
True
GET NEXT TOKEN AND PERFORM OPERATION
PARSE ERROR
RETURN RESULT
PRINT RESULT
STOP

Implementation

1. **Header Function str.h**

```
#ifndef STR_H_INCLUDED
#define STR_H_INCLUDED

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct String
{
   char *ptr;
   size_t len;
} string;

void init_string(string *,int);
void display_string(string);

#endif // STR_H_INCLUDED
```

2. **String Functions str_fnt.c**

```
#include "str.h"


void init_string(string *s,int len)
{
   s->len = len;
   s->ptr = malloc(s->len+1);


   if (s->ptr == NULL)
   {
      fprintf(stderr, "malloc() failed\n");
      exit(EXIT_FAILURE);
   }
```

```
    s->ptr[0] = '\0';
}

void display_string(string s)
{
    printf("%s\n",s.ptr);
}
```

3. **Interpreter Implementation calc1.c**

```
#include <ctype.h>
#include <setjmp.h>
#include "str.h"
#define MAXLEN 256

jmp_buf resume_here;

typedef struct Token
{
    char *type;
    string value;
} Token;

typedef struct Interpreter
{
    char *text;
    int pos;
    Token current_token;
} Interpreter;

int factor(Interpreter *);
```

```
int term(Interpreter *);
void expr(Interpreter *);
void interpret(Interpreter *);
Token get_next_token(Interpreter *);
void parse (Interpreter *,char *);
void parse_error();

int factor(Interpreter *i)
{
        int curfactor;
        Token fact = i->current_token;

        if (strcmp(i->current_token.type,"MINUS") == 0)
        {
                parse(i,"MINUS");

                fact = i->current_token;
                parse(i,"INTEGER");

                return (-1) * atoi(fact.value.ptr);
        }
        else
                parse(i,"INTEGER");

        curfactor = atoi(fact.value.ptr);
        return curfactor;
}

int term(Interpreter *i)
{
```

```
	int t = factor(i);
	if (strcmp(i->current_token.type,"INTEGER") == 0)
		parse_error();
	while ((strcmp(i->current_token.type,"MULTIPLY") == 0) ||
(strcmp(i->current_token.type,"DIVIDE") == 0))
	{
		Token tok = i->current_token;
		if (strcmp(tok.type,"MULTIPLY") == 0)
		{
			parse(i,"MULTIPLY");
			t = t * factor(i);
		}
		else
		{
			parse(i,"DIVIDE");
			t = t / factor(i);
		}
	}
	return t;
}

void expr(Interpreter *i)
{
	int res = term(i);
	while ((strcmp(i->current_token.type,"PLUS") == 0) || (strcmp(i->current_token.type,"MINUS")
== 0) || (strcmp(i->current_token.type,"MODULUS") == 0))
	{
		Token tok = i->current_token;
		if (strcmp(tok.type,"PLUS") == 0)
		{
```

```
                parse(i,"PLUS");
                res = res + term(i);
            }
            else if (strcmp(tok.type,"MINUS") == 0)
            {
                parse(i,"MINUS");
                res = res - term(i);
            }
            else
            {
                parse(i,"MODULUS");
                res = res % term(i);
            }
        }
        printf("%d\n",res);
}

void interpret(Interpreter *i)
{
        i->pos = 0;
        i->current_token.type = "";
        init_string(&i->current_token.value,MAXLEN);
        i->current_token = get_next_token(i);
        expr(i);
}
void parse(Interpreter *i,char *type)
{
   if(strcmp(i->current_token.type,type) == 0)
      i->current_token = get_next_token(i);
   else
```

```
        parse_error();
}

void parse_error()
{
    printf("Error parsing input\n");
    longjmp(resume_here,1);
}

Token get_next_token(Interpreter *i)
{
    string current_char;
    init_string(&current_char,MAXLEN);

    while(isspace(i->text[i->pos]))
        current_char.ptr[0] = i->text[++i->pos];

    if (i->pos > strlen(i->text) - 1)
        return (Token)
    {"EOF",current_char
    };

    current_char.ptr[0] = i->text[i->pos];

    if(isdigit(current_char.ptr[0]))
    {
        int j = 0;
        while(isdigit(i->text[++i->pos]))
            current_char.ptr[++j] = i->text[i->pos];
        current_char.ptr[++j] = '\0';
```

```
    return (Token)
    {"INTEGER",current_char
    };
}
else if(current_char.ptr[0] == '+')
{
    i->pos++;
    return (Token)
    {"PLUS",current_char
    };
}
else if(current_char.ptr[0] == '-')
{
    i->pos++;
    return (Token)
    {"MINUS",current_char
    };
}
else if(current_char.ptr[0] == '*')
{
    i->pos++;
    return (Token)
    {"MULTIPLY",current_char
    };
}
else if(current_char.ptr[0] == '/')
{
    i->pos++;
    return (Token)
    {"DIVIDE",current_char
```

```
        };
    }
    else if(current_char.ptr[0] == '%')
    {
        i->pos++;
        return (Token){"MODULUS",current_char};
    }
    else
    {
        free(current_char.ptr);
        parse_error();
        return (Token)
        {"", {"",0}
        };
    }
}

int main()
{
    //Turn off buffering in stdout
    setvbuf(stdout, NULL, _IONBF, 0);

    Interpreter i;
    i.text = malloc(sizeof(char*) * MAXLEN);

    while(1)
    {
        setjmp(resume_here);
        printf("calc> ");
        //scanf("%s",i.text);  //for whitespaces use gets
```

```
        gets(i.text);
        if(strcmp(i.text,"exit") == 0)
            break;

        interpret(&i);
    }

    free(i.text);

    return 0;
}
```

Explanation

When you enter an expression 3+5 on the command line your interpreter gets a string “3+5”. In order for the interpreter to actually understand what to do with that string it first needs to break the input “3+5” into components called tokens. A token is an object that has a type and a value. For example, for the string “3” the type of the token will be INTEGER and the corresponding value will be integer 3. The method get_next_token of the Interpreter class is the lexical analyzer. Every time it is called, the next token from the input of characters is returned to the interpreter.

The input is stored in the variable text that holds the input string and pos is an index into that string (think of the string as an array of characters). pos is initially set to 0 and points to the character ‘3’. The method first checks whether the character is a digit and if so, it increments pos and returns a token instance with the type INTEGER and the value is set to the integer value of the string ‘3’, which is an integer 3.

The pos now points to the ‘+’ character in the text. The next time when the method is called, it tests if a character at the position pos is a digit and then it tests if the character is a plus sign, which it is. As a result the method increments pos and returns a newly created token with the type PLUS and value ‘+’

The pos now points to character ‘5’. When the method get_next_token is called again, the method checks if it’s a digit, which it is, so it increments pos and returns a new INTEGER token with the value of the token set to integer 5. The pos index is now past the end of the string “3+5”, so the get_next_token method returns the EOF token every time you call it

The function *get_next_token* of the *Interpreter* structure is the lexical analyzer. Every time one calls it, the next token created from the input of characters is passed to the interpreter.

There are two rules in the grammar used for the interpreter: expr rule and factor rule. Let’s start with the factor rule (production). According to the method, one needs to create a method called factor that has a single call to the parse method to consume the INTEGER token

The rule expr becomes the expr method. The body of the rule starts with a reference to factor that becomes a factor() method call. The optional grouping (…)* becomes a while loop and (MUL | DIV) alternatives become an if-elif-else statement. By combining those pieces together the expr method is obtained. The *expr* method first calls the *term* method. Then the *expr* function has a *while* loop which can execute zero or more times. And inside the loop the parser makes a choice based on the token (whether it’s a plus or minus sign).

Current Functions the Code is capable of :

1. Storing and Processing Integers more than 1 digit
2. Ignoring whitespace between operators and operands
3. Outputs Error when input is wrong
4. Uses precedence of operators to give priority to different operators

Testing

Simple TESTS

Input	Interpreter Output	Calculator Output
10+10	20	20
10-10	0	0
1+2	3	3
1+2+3-5	1	1
1*1	1	1
-1*-1	1	1
10/10	1	1
100/-1	-100	-100

Complex TESTS

Input	Interpreter Output	Calculator Output
1*100/100-100*100	-9999	-9999
100*100/100*100-100	9900	9900
10*100-/100/100	Error Parsing Input	syntax error
100/100/100	0	0
()()()()	Error Parsing Input	syntax error
1234565+12341231	13575796	13575796

Output Screenshots

```
$ mingw32-make
gcc calc1.c str_fnt.c -o calc1 -g -F dwarf
$ ls
calc1.c     makefile   str.h      UNIT_TESTS
calc1.exe*  README.md  str_fnt.c
$
```

Build project from source using makefile

```
$ ./calc1.exe
calc> 1+1
2
calc> 10+10
20
calc> 1 - 2
-1
calc> 1 * 100
100
calc> 1--
Error parsing input
calc> 100/-1
-100
calc> 100/50
2
calc> 1 + 2 + 4 - 2
5
calc>
Error parsing input
calc> -1
-1
calc> 1*100/100-100*100
-9999
calc> 100*100/100*100-100
9900
calc> () () () ()
Error parsing input
calc> (100 -100)
Error parsing input
calc> 13432423 + 324232
13756655
calc> 12312 - 212213
-199901
calc> 100/100/100
0
calc> 34@@@ #@$#@$ #$+@$+@#$#@4 23423429 + 323432
Error parsing input
calc> asffss3
Error parsing input
calc> exit
$
```

C_Interpreter

Conclusion

In this project, a simple interpreter was created using C. The same code could be written in python albeit it would be quite slower due to python's interpreted nature. This interpreter has 2 structures namely Token and Interpreter. Another structure String allows manipulation of the input effectively without the character pointer stunts.

The Token keeps track of the input tokens and the Interpreter keeps track of the current token. The input given to the program is string and this is taken as input and processed in the String structure. String structure then passes control to the interpret structure instance to evaluate it.

All the interpreters and compilers are written in either C or x86 Assembly language due to the speed these languages provide compared to other high level languages. These languages emphasize speed and power saving over readability and beauty thus making them perfect for this purpose because no end user is ever going to look at the code of the compiler or the interpreter.

References

- https://ruslanspivak.com/ - Blog followed (information about interpreters and grammar)
- https://wikipedia.org/Interpreter_(computing) – Definition and explanation of an Interpreter