SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District - 603203

18CSC304J/ COMPLIER DESIGN

MINI PROJECT REPORT

MINI COMPILER USING C++

Gudied by:

Dr. Anand M

Submitted By:

ADITYA YALAVARTHY - RA2011031010120

IBRAHIM MUFEEZ - RA2011031010100

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this mini project report "**MINI COMPILER USING C++**" is the bonafide work of **ADITYA YALAVARTHY (RA2011031010120), IBRAHIM MUFEEZ (RA2011031010100)** who carried out the project work under my supervision.

SIGNATURE

Dr.M.Anand
Assistant Professor
CSE
SRM Institute of Science and Technology

TABLE OF CONTENTS

ABSTRACT

A compiler translates the code written in one language to another without changing the program's meaning. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.

The compilation is a process of converting the source code into object code. It is done with the help of the compiler. The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.

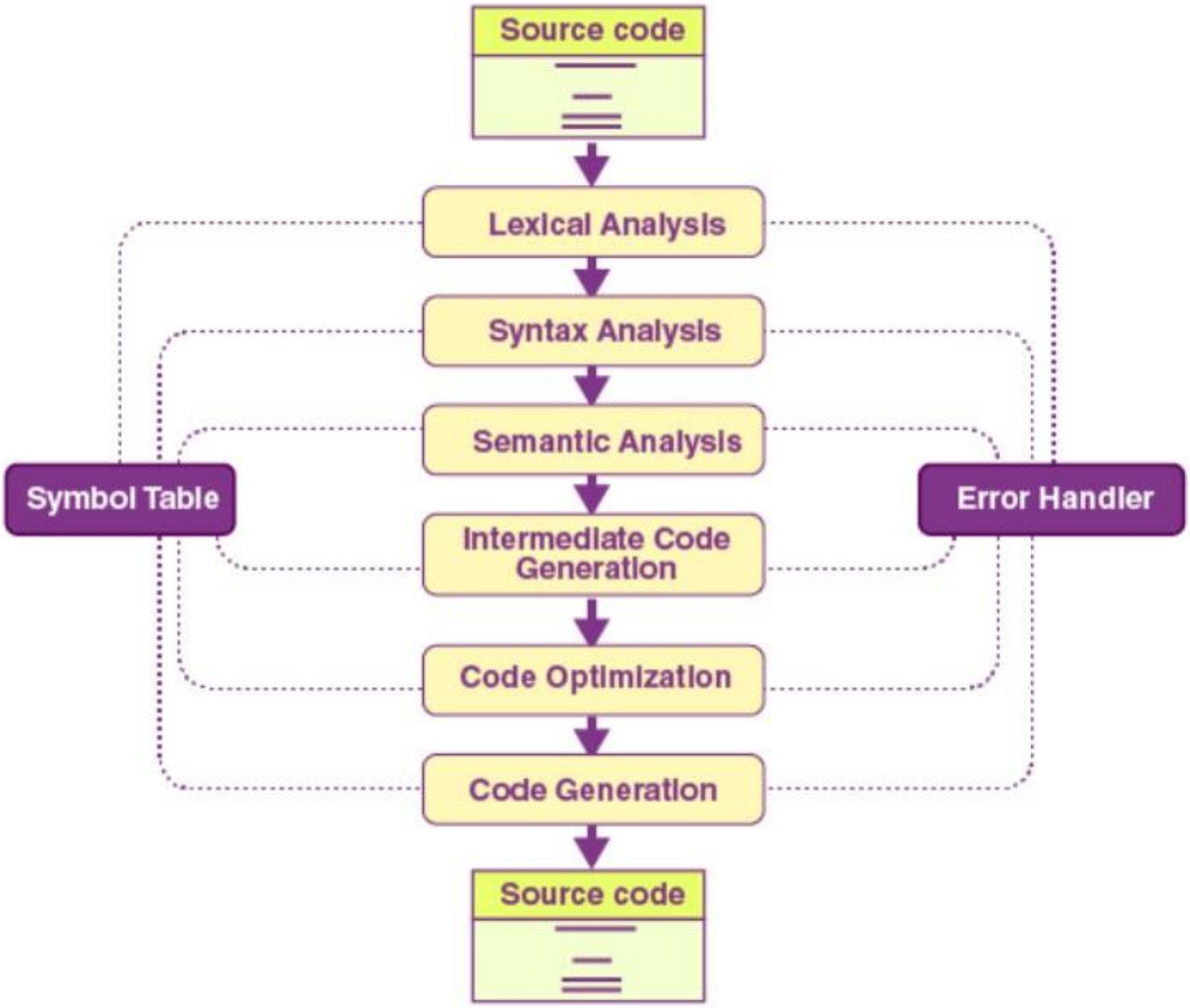
We basically have two phases of compilers, namely the Analysis phase and Synthesis phase. The analysis phase creates an intermediate representation from the given source code. The synthesis phase creates an equivalent target program from the intermediate representation. The analysis of a source program is divided into mainly three phases. They are:

- Linear Analysis involves a scanning phase where the stream of characters is read from left to right. It is then grouped into various tokens having a collective meaning.
- Hierarchical Analysis phase, based on a collective meaning, the tokens are categorized hierarchically into nested groups.
- Semantic Analysis phase is used to check whether the components of the source program are meaningful or not.

Our project focuses on building a mini-compiler for an arbitrary language. Where the raw input to the compiler, is taken from a file during execution, and is pre-processed and checked if it has some specific functionality (like some basic operations – add, subtract, divide, etc.), or any if-else condition, or any loop. If yes, then the respective operation is executed and displayed to the console as the output. For the output and building of the compiler, the C++ language is chosen as it can run on any platform with no unique setup.

A compiler has 6 phases they are:

1. Lexical Analysis
2. Syntactic Analysis or Parsing
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation



1.**Lexical Analysis:** Lexical analysis or Lexical analyser is the initial stage or phase of the compiler. This phase scans the source code and transforms the input program into a series of a token. A token is basically the arrangement of characters that defines a unit of information in the source code.

2. **Syntax Analysis:** In the compilation procedure, the Syntax analysis is the second stage. Here the provided input string is scanned for the validation of the structure of the standard grammar. Basically, in the second phase, it analyses the syntactical structure and inspects if the given input is correct or not in terms of programming syntax.

3. **Semantic Analysis:** In the process of compilation, semantic analysis is the third phase. It scans whether the parse tree follows the guidelines of language. It also helps in keeping track of identifiers and expressions. In simple words, we can say that a semantic analyser defines the validity of the parse tree, and the annotated syntax tree comes as an output.

4. **Intermediate Code Generation:** The parse tree is semantically confirmed; now, an intermediate code generator develops three address codes. A middle-level language code generated by a compiler at the time of the translation of a source program into the object code is known as intermediate code or text.

5. **Code optimizer:** Now coming to a phase that is totally optional, and it is code optimization. It is used to enhance the intermediate code. This way, the output of the program is able to run fast and consume less space. To improve the speed of the program, it eliminates the unnecessary strings of the code and organises the sequence of statements.

6. **Code Generator:** The final stage of the compilation process is the code generation process. In this final phase, it tries to acquire the intermediate code as input which

is fully optimised and map it to the machine code or language. Later, the code generator helps in translating the intermediate code into the machine code.

What is a Symbol Table?

The symbol table is mainly known as the data structure of the compiler. It helps in storing the identifiers with their name and types. It makes it very easy to operate the searching and fetching process.

The symbol table connects or interacts with all phases of the compiler and error handler for updates. It is also accountable for scope management.

It stores:

- It stores the literal constants and strings.
- It helps in storing the function names.
- It also prefers to store variable names and constants.
- It stores labels in source languages.

OBJECTIVE

We have been studying compiler design, how it works and how a language is being compiled. This created a curiosity about how these compilers are being made and how they are designed, and the practical implementation of a concept is the best way to understand and learn. Following are some features that we wanted to implement –

- How compilers do the lexical and semantic analysis
- How do compilers take a raw input as string from file and processes it to make some meaningful code to execute
- How compilers do identify an operator or an if-else condition or a loop and do the meaningful executions accordingly
- How can we can teach a new language and new functionality to a compiler and see how will the compiler respond to it
- Error detection is also a difficult part to design for a compiler, what will happen if we encounter an error in the code which is not handled.

LIMITATIONS OF EXISTING METHODS

Existing compilers take a pl0 file as input and convert it into assembly language code. However, it is difficult for humans to understand it, making it difficult to modify and maintain the program. Additionally, assembly language codes are more prone to bugs which can be very difficult to locate and eliminate.

Some compilers use intermediate steps between the lexer, parser, and the code generator.

They generate an intermediate file for the lexer output, which the parser then uses.

The parser also outputs a file that the generator then uses to generate code.

Hence, for our project, we have built a compiler that takes a pl0 file as input and converts it into a corresponding C language file.

Since, C is a high-level programming language, it is programmer-friendly. It is portable as well, making it easy to share the results.

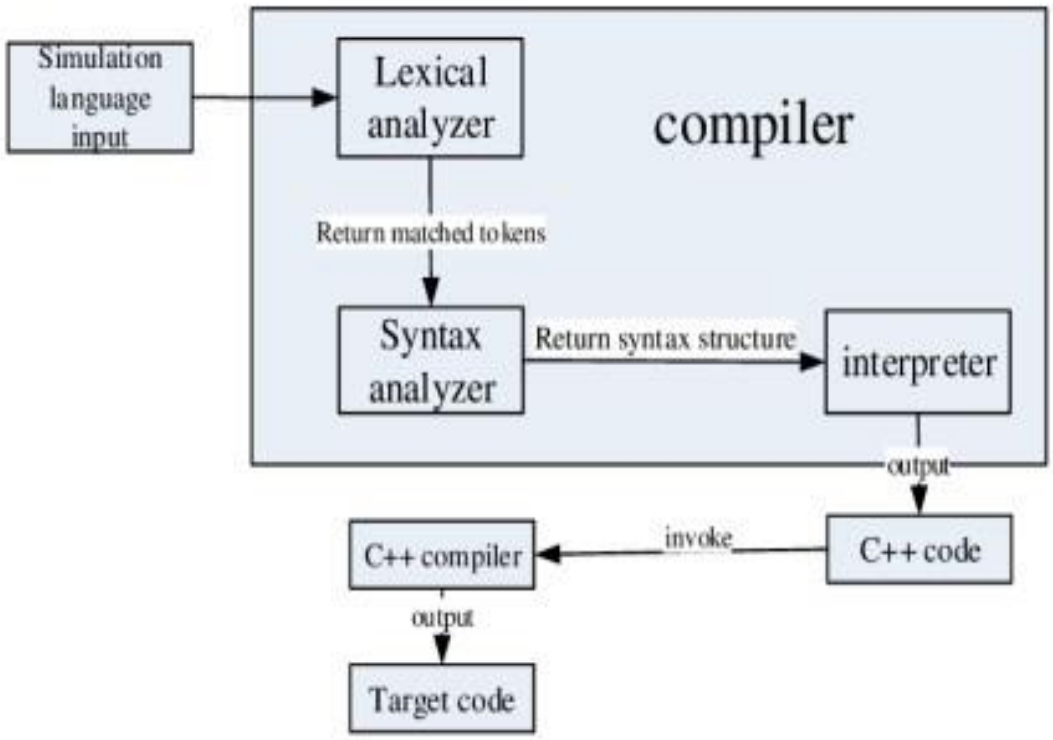
Our project has made it so that the output is directly fed into the next step, thus eliminating the need for intermediate files.

This speeds up the compiler and does not occupy space for generating files. Even though these metadata files are deleted at the end, if something goes wrong, it can start stacking up and lead to useless memory occupation.

This doesn't happen in our program as GCC will clean up the metadata that we created during execution for us, and since it's not files, it will get cleaned up every time by the compiler after the program finishes running In the end, we execute the output C file to check the correctness of the C code generated.

PROPOSED METHOD WITH ARCHITECTURE

The rough workflow of our compiler as shown below in the diagram:



We will break up these different parts of the compiler organizationally. There are three main parts for this compiler: the pre-processing, the parser, and the code generator. In pre-processing part, out task is to take the raw free-form source code and turn it into a series of tokens, the individual units of the language. Then we move to the parser The parser's part is to take that series of tokens and ensure that their ordering follows the rules of the language syntax. And then the code generator checks what type of operators and conditions are there and executes them, and then we print them as output.

MODULES WITH DESCRIPTION

The compiler consists of three main modules - the Pre-processor, the parser, and the code generator, as explained in the workflow.

Pre-processor:

In this phase we aim to accept an arbitrary source code file and output all its constituent tokens in proper format. It should accept all valid tokens and should reject invalid tokens.

According to the grammar, the reserved words are out, end, if, then, while, do, and loop. The symbols are '.' , '=' , ',' , ';' , ':=', '#', '<', '>', '+', '-' , '*' , '/'.

The identifiers and numbers cannot be known in advance.

The first step of the lexer is to move past the whitespaces for that we are using trimString().

Next, the checkVariable() function reads in all letters, numbers, and underscores until it encounters a character that isn't one of those or checkOutput() returns true. Next, iterate down the list of reserved words, and if there is a match, return the correct token type for that reserved word. Else, report the token type as an identifier.

Parser

For its implementation, checkCalculation() function is used, the parser will can tell the checkVariable() to fetch the next token from the source code and record all of the useful information for that token: the token type and, if needed, the token string.

Code Generation

The code generator is added between the pre-processor and the parser. The pre-processor does not talk to the code generator. And the code generator never talks to the pre-processor or the parser.The only communication is from the parser to the code generator. It gets called when we know parsing is complete and successful, meaning that we have a valid code.

Some essential functions used by the code generator:

- checkIfElse(): It checks the presence of If-Else condition in the passed string and executes the condition and returns the output.

- isIfElse(): This function simply returns the bool expression, telling if the passed string has any conditional operator in it or not.

- checkLoop(): This statement is used to check and execute the loop present in the statement, if present. To prevent very long lines and at least give us some semblance of C code output, it prints a semicolon followed by a newline.

- includesOperator(): This function simply returns the bool expression, telling if the passed string has any binary operator in it or not.

SOURCE CODE

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <map>
#include <string>

using namespace std;

     // recursive tree logic of our checkCalculation function
     // for ex : B + A + C * D
     // tree logic of this
     //          +
     //        /   \
     //      B       +
     //            /  \
     //          A     *
     //              /   \
     //             C     D
     //

string checkCalculation(string str, map<string,string>&vars);
string checkIfElse(string str, map<string,string>&vars);

// remove all white spaces
string trimString(string s) {
    int lefti = 0,righti = s.size()-1;
    while (s[lefti] == ' ') lefti++;
    while (s[righti] == ' ') righti--;
    return s.substr(lefti, righti-lefti+1);
}

bool includesOperator(string s) {
    for (int i = 0; i < s.size(); ++i) {
        if (s[i] == '+' || s[i] == '/' || s[i] == '-' || s[i] == '*') return
true;
    }
    return false;
}

bool isIfElse(string str) {
```

```
    for (int i = 0; i < str.size(); ++i) {
        if (str[i] == '>' || str[i] == '<' || str[i] == '=' || str[i] == '!')
return true;
    } return false;
}

string checkLoop(string str, map<string,string> &vars, string key) {
    //check if the key exists in our variables map
    if (vars.find(key) == vars.end()) vars[key] = "0";
    int index = str.find("TIMES");
    string left = trimString(str.substr(0,index));
    int count = stoi(trimString(str.substr(index+5)));
    for (int i = 0; i < count; ++i) {
        vars[key] = str.find("IF") != string::npos ? checkIfElse(left, vars) :
checkCalculation(left, vars);
    } return "";
}


string checkIfElse(string str, map<string,string> &vars) {
    int index = str.find("IF");
    int index2 = str.find("THEN",index+1);
    int index3 = str.find("ELSE",index2+1);

    string condition = trimString(str.substr(index+2,index2 - index - 2));
    string ifRes = trimString(str.substr(index2+4,index3 - index2 - index-4));
    string elseRes = trimString(str.substr(index3+4));
    string left,right,op = ">=";

    ifRes = includesOperator(ifRes) ? checkCalculation(ifRes, vars) :
vars[ifRes];
    elseRes = includesOperator(elseRes) ? checkCalculation(elseRes, vars) :
vars[elseRes];

    // find the operator
    index = condition.find(">=");
    if (index == string::npos) {index = condition.find(">");op = ">";}
    if (index == string::npos) {index = condition.find("<=");op = "<=";}
    if (index == string::npos) {index = condition.find("<");op = "<";}
    if (index == string::npos) {index = condition.find("==");op = "==";}
    if (index == string::npos) {index = condition.find("!=");op = "!=";}

    if (index != string::npos) {
```

```
        string left_side = trimString(condition.substr(0,index));
        string right_side = trimString(condition.substr(index+op.size()));

        if (vars.find(left_side) != vars.end()) left = vars[left_side];
        else if (includesOperator(left_side)) left = checkCalculation(left_side,
vars);
        else left = left_side;
        if (vars.find(right_side) != vars.end()) right = vars[right_side];
        else if (includesOperator(right_side)) right =
checkCalculation(right_side, vars);
        else right = right_side;

        if (op == ">") return (stod(left) > stod(right) ? ifRes : elseRes);
        else if (op == "<") return stod(left) < stod(right) ? ifRes : elseRes;
        else if (op == "<=") return stod(left) <= stod(right) ? ifRes : elseRes;
        else if (op == ">=") return stod(left) >= stod(right) ? ifRes : elseRes;
        else if (op == "==") return stod(left) == stod(right) ? ifRes : elseRes;
        else if (op == "!=") return stod(left) != stod(right) ? ifRes : elseRes;
    } return "";
}


string checkCalculation(string str, map<string,string>&vars)
{
    bool intCalc = false;
    string left,right;
    // decide the operator and calculation class with priority
    // paranthese priority
    int index = str.find("(");
    while (index != string::npos) {
        int index2 = str.rfind(")");
        str.replace(index, index2-index+1,
checkCalculation(str.substr(index+1,index2-index-1), vars));
        index = str.find(str, index+1);
    }
    // continue calculation without parantheses
    char op = '+';
    index = str.find("+");
    if (index == string::npos) {index = str.find("-");op='-';}
    if (index == string::npos) {index = str.find("/");op='/';}
    if (index == string::npos) {index = str.find("*");op='*';}
    // check if there is a math calc
    if (index != string::npos)
```

```
    {
        string left_side = trimString(str.substr(0,index));
        string right_side = trimString(str.substr(index + 1));

        // check left side
        if (vars.find(left_side) != vars.end()) left = vars[left_side];     //
if it exists in my map
        else if (includesOperator(left_side)) left = checkCalculation(left_side,
vars);  // recursive - tree logic
        else left = left_side;
        // check right side
        if (vars.find(right_side) != vars.end()) right =
vars[right_side];      // if it exists in my map
        else if (includesOperator(right_side)) right =
checkCalculation(right_side, vars);
        else right = right_side;

        if (left.find(".") == string::npos && right.find(".") == string::npos)
intCalc = true ;     // check if the calc is between integers or doubles

        switch (op) {
            case '+':
            return to_string(intCalc ? stoi(left) + stoi(right) : stod(left) +
stod(right)); break;
            case '-':
            return to_string(intCalc ? stoi(left) - stoi(right) : stod(left) -
stod(right)); break;
            case '*':
            return to_string(intCalc ? stoi(left) * stoi(right) : stod(left) *
stod(right)); break;
            case '/':
            return to_string(intCalc ? stoi(left) / stoi(right) : stod(left) /
stod(right)); break;
        }
    }
    return trimString(str);
};


void checkVariable(string str, map<string,string>&vars) {
    int index = str.find("=");
    if (index != string::npos) {
        // it's an assignment
```

```
        int loopind = str.find("LOOP");  // if there is a loop
        string key = loopind != string::npos ?
trimString(str.substr(loopind+4,index-loopind-4)) :
trimString(str.substr(0,index));
        if (loopind != string::npos) checkLoop(str.substr(index + 1), vars,
key);
        else vars[key] = str.find("IF") != string::npos ?
checkIfElse(str.substr(index + 1), vars) :  checkCalculation(str.substr(index +
1),vars);  // check if it's a calculation assignment and store the val
    }
};


// check if we are done
bool checkOutput(string str) {return str.find("OUT", 0) != string::npos ? true :
false;}


int main() {
    map<string,string> vars;
    string str;

    //read and process the input
    ifstream input;
    input.open("example9.inp");
    while (getline(input,str)) {
        if (checkOutput(str)) break;
        checkVariable(str, vars);
        // checkIfElse(str, vars);
    }
    input.close();
    cout<<"Executing file: example9.inp\n";
    // checking if we get the vars correctly
     for (auto a : vars) {
         cout << a.first << " ==== " << a.second << endl;
     }

    // print the output
    auto output = vars[trimString(str.substr(str.find("OUT")+3))];
    cout << "output : " << output << endl; // log the output
    ofstream ofile;
    ofile.open("output.out");
    ofile << output;
```

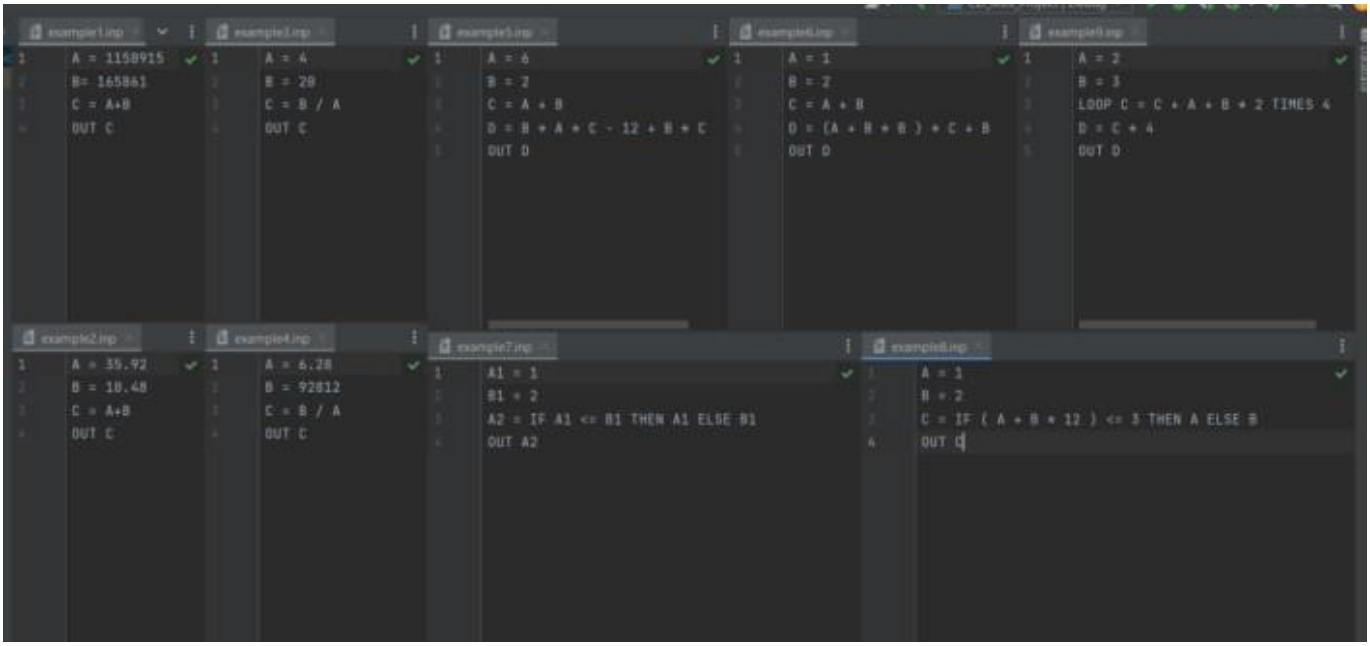
```
    ofile.close();

    // system("Pause");

    return 0;
}

// time: 3-4 days
```

INPUT FILES



OUTPUT

Example 1:

```
Executing file: example1.inp
A ==== 1158915
B ==== 165861
C ==== 1324776.000000
output : 1324776.000000

Process finished with exit code 0
```

Example 2:

```
A ==== 35.92
B ==== 10.48
C ==== 46.400000
output : 46.400000

Process finished with exit code 0
```

Example 3:

```
Executing file: example3.inp
A ==== 4
B ==== 20
C ==== 5.000000
output : 5.000000

Process finished with exit code 0
```

Example 4:

```
Executing file: example4.inp
A ==== 6.28
B ==== 92812
C ==== 14778.980892
output : 14778.980892

Process finished with exit code 0
```

Example 5:

```
Executing file: example5.inp
A ==== 6
B ==== 2
C ==== 8.000000
D ==== 100.000000
output : 100.000000

Process finished with exit code 0
```

Example 6:

```
Executing file: example6.inp
A ==== 1
B ==== 2
C ==== 3.000000
D ==== 17.000000
output : 17.000000

Process finished with exit code 0
```

Example 7:

```
Executing file: example7.inp
A1 ==== 1
A2 ==== 1
B1 ==== 2
output : 1

Process finished with exit code 0
```

Example 8:

```
Executing file: example9.inp
A ==== 2
B ==== 3
C ==== 32.000000
D ==== 128.000000
output : 128.000000

Process finished with exit code 0
```

Example 9:

CONCLUSION

The mini compiler that we set out to make works and compiles arbitrary code code in an acceptable fashion even though it doesn't convert it into orthodox forms of code i.e., infix, Postfix or assembly code, generating C++ code has been a comparatively hard task.

There are many ways for us to improve this including allocating a string data type instead of using character arrays and error checking input, output and adding other data structure functionality We have thus learnt the working of compilers and its hidden parts through this project

REFERENCES

https://www.geeksforgeeks.org/introduction-of-lexical-analysis/

https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm

https://www.cs.cmu.edu/~aplatzer/course/Compilers/waitegoos.pdf

https://www.webopedia.com/definitions/high-level-language/

https://en.wikipedia.org/wiki/Principles_of_Compiler_Design

https://softwareengineering.stackexchange.com/questions/165543/how-to-write-a-verybasiccompiler