SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District - 603203

18CSC304J/ COMPLIER DESIGN

MINI PROJECT REPORT

Mini-C-Compiler

Guided by:

Dr M Karthikeyan

Submitted By:

Utkarsh Saboo (RA2011003011248)

Jatin Singhania (RA2011003011247)

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this mini project report "**Mini-C-Compiler**" is the Bonafide work of **UTKARSH SABOO (RA2011003011248), JATIN SINGHANIA (RA20110030112447)** who carried out the project work under my supervision.

SIGNATURE

Dr M Karthikeyan
Assistant Professor
Department of Computing Technologies
SRM Institute of Science and Technology

ACKNOWLEDGEMENT

We express our heartfelt thanks to our honorable **Vice Chancellor Dr. C. MUTHAMIZHCHELVAN**, for being the beacon in all our endeavors.

We would like to express my warmth of gratitude to our **Registrar Dr. S. Ponnusamy,** for his encouragement

We express our profound gratitude to our **Dean (College of Engineering and Technology) Dr. T. V. Gopal,** for bringing out novelty in all executions.

We would like to express my heartfelt thanks to Chairperson, School of Computing **Dr. Revathi Venkataraman,** for imparting confidence to complete my course project

We wish to express my sincere thanks to **Course Audit Professor Dr. Annapurani Panaiyappan, Professor and Head, Department of Networking and Communications** and **Course Coordinators** for their constant encouragement and support.

We are highly thankful to my Course project Faculty **Dr M Karthikeyan, Assistant Professor, Department of Computing Technologies** for her assistance, timely suggestion and guidance throughout the duration of this course project.

We extend my gratitude to our **HoD Dr. M Pushapaltha, Professor and Head, Department of Computing Technologies** and my Departmental colleagues for their Support.

Finally, we thank our parents and friends near and dear ones who directly and indirectly contributed to the successful completion of our project. Above all, I thank the almighty for showering his blessings on me to complete my Course project.

AGENDA

ABSTRACT

This report presents the development of a Mini C Compiler, which is a software tool that translates C programming language code into machine-readable instructions. The project was undertaken to provide a better understanding of the compilation process and to gain practical experience in implementing a compiler.

The Mini C Compiler was developed using the Lex and Yacc tools, which are commonly used for building compilers. The compiler was designed to support a subset of the C language, including basic data types, control structures, and functions. The compiler was also able to generate assembly code for a hypothetical processor architecture.

The development process involved several stages, including lexical analysis, syntax analysis, and intermediate code generation with optimization as well error finding functionalities. The lexical analyzer was responsible for breaking down the input code into tokens, while the syntax analyzer checked the syntax of the code against a grammar specification. Finally, the optimizer improved the efficiency of the generated code.

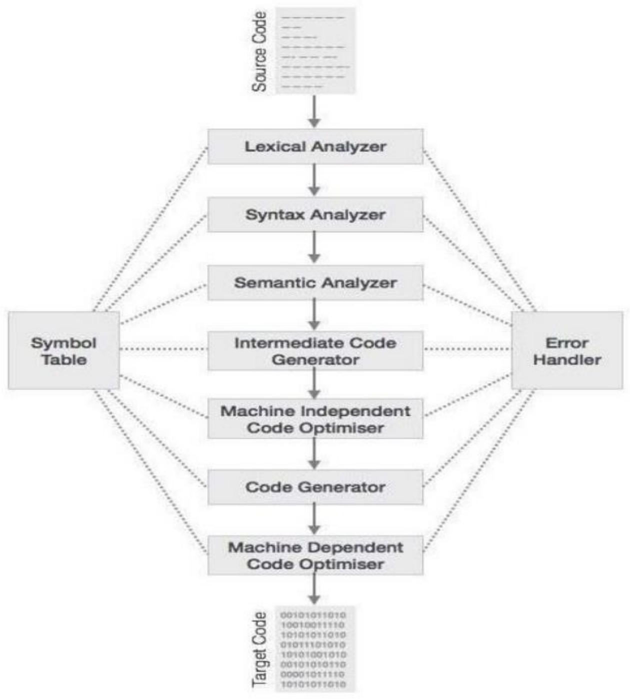
The Mini C Compiler was tested using a set of sample programs, and the results showed that it was able to correctly compile and execute the programs. Not just successfully running but also spotting errors and works for n number of test cases simultaneously which is tried and tested. The project provided valuable insights into the complexities of compiler design and implementation, and demonstrated the importance of careful planning and testing in software development.

INTRODUCTION

Compiler Design Phases:

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 processors use. A compiler can broadly be divided into two phases based on the way they compile.

- **Analysis phase**: Known as the front-end of the compiler, the analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table. This phase consists of:



1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation

- **Synthesis phase**: Known as the back-end of the compiler, the synthesis phase generates the target program with the help of intermediate source code representation and symbol table. This phase consists of:
  1. Code Optimization
  2. Code Generator

Lexical Analysis

Lexical analysis is the first phase of a compiler. It takes the modified source code from language pre-processors that are written in the form of sentences. The lexical analyser breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyser finds a token invalid, it generates an error. The lexical analyser works closely with the syntax analyser. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyser when it demands.

Syntax Analysis

In computer science, syntax analysis is the process of checking that the code is syntactically correct. The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. Tokens are valid sequence of symbols, keywords, identifiers etc. The parser needs to be able to handle the infinite number of possible valid programs that may be presented to it. The usual way to define the language is to specify a grammar. A grammar is a set of rules (or productions) that specifies the syntax of the language (i.e., what is a valid sentence in the language). There can be more than one grammar for a given language. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. It generates a parse tree (or syntax tree) as output.

Semantic Analysis

Semantic analysis is the third phase of a compiler. Semantic analyser checks whether the parse tree constructed by the syntax analyser follows the rules of language.

Intermediate Code generation

Intermediate code generator receives input from its predecessor phase, semantic analyser, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code. A three-address code has at most three address locations to calculate the expression. Hence, the intermediate code generator will divide any expression into sub-expressions and then generate the corresponding code. A three-address code can be represented in two forms: quadruples and triples

Code Optimization

In this phase, code optimization of the intermediate code is done. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language.

Mini Project is divided into three parts which are mentioned below with purpose they are serving:

1. Lexical Analysis
    - Token Generation
    - Line Numbers
    - Lexical Errors
    - Record Lexemes
    - Identify Keywords
    - Symbol Table

2. Parser/Syntax Analysis
    - Abstract Syntax Tree Construction
    - Global & Local Variables Distinction
    - Class Definition
    - Arrays/Pointers definition
    - Define struct, union
    - Shift Reduce, Reduce Error correction
    - Valid Actions for grammar productions
    - Error Handing and Error Recovery

3. Intermediate Code Generator
    - Generate three address code
    - Quadruple format
    - Insert Temporaries to Symbol Table
    - AST
    - Intermediate Code Optimization
    - Eliminate dead code
    - Constant Folding, Propagation

REQUIREMENT ANALYSIS

HARDWARE REQUIREMENT [minimum requirement]:

- Minimum RAM: - 1GB
- Minimum Hard Disk: - 128GB
- Processor: - Intel Pentium 4(1.50 GHz) or above

SOFTWARE REQUIREMENT [minimum requirement]:

- Operating System: - Support for both LINUX and WINDOWS 8.1 or greater users
- Front End Language: - C/C++
- PowerShell X86 or above, Command Prompt or any terminal which supports the OS

CODES

1. Phase I:

➢ Lexical Analyzer:(scanner.l):

```
%{
        #include <stdio.h>
        #include <string.h>


        struct symboltable
        {
                char name[100];
                char type[100];
                int length;
        }ST[1001];

        struct constanttable
        {
                char name[100];
                char type[100];
                int length;
        }CT[1001];

        int hash(char *str)
        {
                int value = 0;
                for(int i = 0 ; i < strlen(str) ; i++)
                {
                        value = 10*value + (str[i] - 'A');
                        value = value % 1001;
                        while(value < 0)
                                value = value + 1001;
                }
                return value;
        }

        int lookupST(char *str)
        {
                int value = hash(str);
                if(ST[value].length == 0)
                {
                        return 0;
```

```
			}
			else if(strcmp(ST[value].name,str)==0)
			{
				return 1;
			}
			else
			{
				for(int i = value + 1 ; i!=value ; i = (i+1)%1001)
			{
				if(strcmp(ST[i].name,str)==0)
				{
					return 1;
				}
			}
			return 0;
	}
}

int lookupCT(char *str)
{
	int value = hash(str);
	if(CT[value].length == 0)
		return 0;
	else if(strcmp(CT[value].name,str)==0)
		return 1;
	else
	{
		for(int i = value + 1 ; i!=value ; i = (i+1)%1001)
		{
			if(strcmp(CT[i].name,str)==0)
			{
				return 1;
			}
		}
		return 0;
	}
}

void insertST(char *str1, char *str2)
{
	if(lookupST(str1))
	{
```

```
            return;
    }
        else
        {
            int value = hash(str1);
            if(ST[value].length == 0)
            {
                strcpy(ST[value].name,str1);
                strcpy(ST[value].type,str2);
                ST[value].length = strlen(str1);
                return;
            }

            int pos = 0;

            for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
            {
                if(ST[i].length == 0)
                {
                    pos = i;
                    break;
                }
            }

            strcpy(ST[pos].name,str1);
            strcpy(ST[pos].type,str2);
            ST[pos].length = strlen(str1);
        }
}

void insertCT(char *str1, char *str2)
{
        if(lookupCT(str1))
            return;
        else
        {
            int value = hash(str1);
            if(CT[value].length == 0)
            {
                strcpy(CT[value].name,str1);
                strcpy(CT[value].type,str2);
                CT[value].length = strlen(str1);
                return;
            }

            int pos = 0;

            for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
            {
                if(CT[i].length == 0)
                {
                    pos = i;
                    break;
                }
            }

            strcpy(CT[pos].name,str1);
            strcpy(CT[pos].type,str2);
            CT[pos].length = strlen(str1);
        }
}

void printST()
{
        for(int i = 0 ; i < 1001 ; i++)
        {
            if(ST[i].length == 0)
            {
                continue;
            }

            printf("%s\t%s\n",ST[i].name, ST[i].type);
        }
}

void printCT()
{
        for(int i = 0 ; i < 1001 ; i++)
        {
            if(CT[i].length == 0)
                continue;

            printf("%s\t%s\n",CT[i].name, CT[i].type);
        }
```

```
            }

%}

DE "define"
IN "include"

operator [[<][=]|[>][=]|[=][=]|[!][=]|[>]|[<]|[\|][\|]|[&][&]|[\!]|[=]|[\^]|[\+][=]|[\-
][=]|[\*][=]|[\/][=]|[\%][=]|[\+][\+]|[\-][\-]|[\+]|[\-]|[\*]|[\/]|[\%]|[&]|[\|]|[~]|[<][<]|[>][>]]



%%
\n   {yylineno++;}
([#][" "]*({IN})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|" "|"\t"] {printf("%s \t-Pre
Processor directive\n",yytext);}          //Matches #include<stdio.h>
([#][" "]*({DE})[" "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\/|" "|"\t"] {printf("%s \t-Macro\n",yytext);}
//Matches macro
\/\/(.*) {printf("%s \t- SINGLE LINE COMMENT\n", yytext);}
\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/  {printf("%s \t- MULTI LINE COMMENT\n", yytext);}
[ \n\t] ;
; {printf("%s \t- SEMICOLON DELIMITER\n", yytext);}
, {printf("%s \t- COMMA DELIMITER\n", yytext);}
\{ {printf("%s \t- OPENING BRACES\n", yytext);}
\} {printf("%s \t- CLOSING BRACES\n", yytext);}
\( {printf("%s \t- OPENING BRACKETS\n", yytext);}
\) {printf("%s \t- CLOSING BRACKETS\n", yytext);}
\[ {printf("%s \t- SQUARE OPENING BRACKETS\n", yytext);}
\] {printf("%s \t- SQUARE CLOSING BRACKETS\n", yytext);}
\: {printf("%s \t- COLON DELIMITER\n", yytext);}
\\ {printf("%s \t- FSLASH\n", yytext);}
\. {printf("%s \t- DOT DELIMITER\n", yytext);}
auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|int|long|reg
ister|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volatile|while|main
/[\(|" "|\{|;|:|"\n"|"\t"] {printf("%s \t- KEYWORD\n", yytext); insertST(yytext, "KEYWORD");}
\"[^\n]*\"/[;|,|\)] {printf("%s \t- STRING CONSTANT\n", yytext); insertCT(yytext,"STRING
CONSTANT");}
\'[A-Z|a-z]\'/[;|,|\)|:] {printf("%s \t- Character CONSTANT\n", yytext);
insertCT(yytext,"Character CONSTANT");}
[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[ {printf("%s \t- ARRAY IDENTIFIER\n", yytext); insertST(yytext,
"IDENTIFIER");}

{operator}/[a-z]|[0-9]|;|" "|[A-Z]|\(|\"|\'|\)|\n|\t {printf("%s \t- OPERATOR\n", yytext);}
```

```
[1-9][0-9]*|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^] {printf("%s \t- NUMBER
CONSTANT\n", yytext); insertCT(yytext, "NUMBER CONSTANT");}
([0-9]*)\.([0-9]+)/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^] {printf("%s \t- Floating
CONSTANT\n", yytext); insertCT(yytext, "Floating CONSTANT");}
[A-Za-z_][A-Za-z_0-9]*/[" "|;|,|\(|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\.|\{|\^|\t] {printf("%s \t-
IDENTIFIER\n", yytext); insertST(yytext, "IDENTIFIER");}


(.?) {
                if(yytext[0]=='#')
                {
                printf("Error in Pre-Processor directive at line no. %d\n",yylineno);
        }
           else if(yytext[0]=='/')
           {
                printf("ERR_UNMATCHED_COMMENT at line no. %d\n",yylineno);
        }
        else if(yytext[0]=='"')
        {
                printf("ERR_INCOMPLETE_STRING at line no. %d\n",yylineno);
        }
        else
        {
                printf("ERROR at line no. %d\n",yylineno);
        }
        printf("%s\n", yytext);
        return 0;
}

%%

int main(int argc , char **argv){


printf("=============================================================
=====\n");

        int i;
        for (i=0;i<1001;i++){
                ST[i].length=0;
                CT[i].length=0;
        }

        yyin = fopen(argv[1],"r");
        yylex();

        printf("\n\nSYMBOL TABLE\n\n");
        printST();
        printf("\n\nCONSTANT TABLE\n\n");
        printCT();
}

int yywrap(){
   return 1;
}
```

- **Parser:(scanner.l):**

```
 %{
        #include <stdio.h>
        #include <string.h>
        #include "y.tab.h"

        #define ANSI_COLOR_RED          "\x1b[31m"
        #define ANSI_COLOR_GREEN        "\x1b[32m"
        #define ANSI_COLOR_YELLOW       "\x1b[33m"
        #define ANSI_COLOR_BLUE                 "\x1b[34m"
        #define ANSI_COLOR_MAGENTA"\x1b[35m"
        #define ANSI_COLOR_CYAN                 "\x1b[36m"
        #define ANSI_COLOR_RESET        "\x1b[0m"

        struct symboltable
        {
                char name[100];
                char class[100];
                char type[100];
                char value[100];
                int lineno;
                int length;
        }ST[1001];
```

```
struct constanttable
{
        char name[100];
        char type[100];
        int length;
}CT[1001];

int hash(char *str)
{
        int value = 0;
        for(int i = 0 ; i < strlen(str) ; i++)
        {
                value = 10*value + (str[i] - 'A');
                value = value % 1001;
                while(value < 0)
                        value = value + 1001;
        }
        return value;
}

int lookupST(char *str)
{
        int value = hash(str);
        if(ST[value].length == 0)
        {
                return 0;
        }
        else if(strcmp(ST[value].name,str)==0)
        {
                return 1;
        }
        else
        {
                for(int i = value + 1 ; i!=value ; i = (i+1)%1001)
                {
                        if(strcmp(ST[i].name,str)==0)
                        {
                                return 1;
                        }
                }
                return 0;
```

```
	}
}

int lookupCT(char *str)
{
	int value = hash(str);
	if(CT[value].length == 0)
		return 0;
	else if(strcmp(CT[value].name,str)==0)
		return 1;
	else
	{
		for(int i = value + 1 ; i!=value ; i = (i+1)%1001)
		{
			if(strcmp(CT[i].name,str)==0)
			{
				return 1;
			}
		}
		return 0;
	}
}

void insertST(char *str1, char *str2)
{
	if(lookupST(str1))
	{
		return;
	}
	else
	{
		int value = hash(str1);
		if(ST[value].length == 0)
		{
			strcpy(ST[value].name,str1);
			strcpy(ST[value].class,str2);
			ST[value].length = strlen(str1);
			insertSTline(str1,yylineno);
			return;
		}

		int pos = 0;
```

```
                    for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
                    {
                        if(ST[i].length == 0)
                        {
                            pos = i;
                            break;
                        }
                    }

                    strcpy(ST[pos].name,str1);
                    strcpy(ST[pos].class,str2);
                    ST[pos].length = strlen(str1);
                }
            }

            void insertSTtype(char *str1, char *str2)
            {
                for(int i = 0 ; i < 1001 ; i++)
                {
                    if(strcmp(ST[i].name,str1)==0)
                    {
                        strcpy(ST[i].type,str2);
                    }
                }
        }

            void insertSTvalue(char *str1, char *str2)
            {
                for(int i = 0 ; i < 1001 ; i++)
                {
                    if(strcmp(ST[i].name,str1)==0)
                    {
                        strcpy(ST[i].value,str2);
                    }
                }
            }

            void insertSTline(char *str1, int line)
            {
                for(int i = 0 ; i < 1001 ; i++)
                {
                        if(strcmp(ST[i].name,str1)==0)
                        {
                                ST[i].lineno = line;
                        } } }
        void insertCT(char *str1, char *str2)
        {
                if(lookupCT(str1))
                        return;
                else
                {
                        int value = hash(str1);
                        if(CT[value].length == 0)
                        {
                                strcpy(CT[value].name,str1);
                                strcpy(CT[value].type,str2);
                                CT[value].length = strlen(str1);
                                return;
                        }

                        int pos = 0;

                        for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
                        {
                                if(CT[i].length == 0)
                                {
                                        pos = i;
                                        break;
                                }       }

                        strcpy(CT[pos].name,str1);
                        strcpy(CT[pos].type,str2);
                        CT[pos].length = strlen(str1);
                } }

        void printST()
        {
                printf("%10s | %15s | %10s | %10s | %10s\n","SYMBOL", "CLASS",
"TYPE","VALUE", "LINE NO");
                for(int i=0;i<81;i++) {
                        printf("-");
                }
                printf("\n");
```

```
                    for(int i = 0 ; i < 1001 ; i++)
                    {
                              if(ST[i].length == 0)
                              {
                                        continue;
                              }
                              printf("%10s | %15s | %10s | %10s | %10d\n",ST[i].name, ST[i].class,
ST[i].type, ST[i].value, ST[i].lineno);
                    }    }

          void printCT()
          {
                    printf("%10s | %15s\n","NAME", "TYPE");
                    for(int i=0;i<81;i++) {
                              printf("-");
                    }
                    printf("\n");
                    for(int i = 0 ; i < 1001 ; i++)
                    {
                              if(CT[i].length == 0)
                                        continue;

                              printf("%10s | %15s\n",CT[i].name, CT[i].type);
                    }
          }
          char curid[20];
          char curtype[20];
          char curval[20];

%}

DE "define"
IN "include"

%%
\n        {yylineno++;}
([#][" "]*({IN})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|" "|"\t"] { }
([#][" "]*({DE})[" "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\/|" "|"\t"]                  { }
\/\/(.*)
                                                            { }
\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/
                              { }
```

```
[ \n\t] ;
";"                 { return(';'); }
","                 { return(','); }
("{")           { return('{'); }
("}")           { return('}'); }
"("                 { return('('); }
")"                 { return(')'); }
("["|"<:")      { return('['); }
("]"|":>")      { return(']'); }
":"                 { return(':'); }
"."                 { return('.'); }

"char"          { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return CHAR;}
"double"        { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return DOUBLE;}
"float"     { strcpy(curtype,yytext); insertST(yytext, "Keyword");return FLOAT;}
"while"         { insertST(yytext, "Keyword"); return WHILE;}
"do"            { insertST(yytext, "Keyword"); return DO;}
"int"           { strcpy(curtype,yytext); insertST(yytext, "Keyword");return INT;}
"return"        { insertST(yytext, "Keyword");  return RETURN;}
"sizeof"        { insertST(yytext, "Keyword");  return SIZEOF;}
"struct"        { strcpy(curtype,yytext); insertST(yytext, "Keyword");  return STRUCT;}
"union"         { strcpy(curtype,yytext); insertST(yytext, "Keyword");  return UNION;}
"void"          { strcpy(curtype,yytext); insertST(yytext, "Keyword");  return VOID;}

"++"            { return increment_operator; }
"--"            { return decrement_operator; }
"<<"            { return leftshift_operator; }
">>"            { return rightshift_operator; }
"<="            { return lessthan_assignment_operator; }
"<"                 { return lessthan_operator; }
">="            { return greaterthan_assignment_operator; }
">"                 { return greaterthan_operator; }
"=="            { return equality_operator; }
"!="            { return inequality_operator; }
"&&"            { return AND_operator; }
"||"            { return OR_operator; }
"^"                 { return caret_operator; }
"*="            { return multiplication_assignment_operator; }
"/="            { return division_assignment_operator; }
"%="            { return modulo_assignment_operator; }
"+="            { return addition_assignment_operator; }
"-="            { return subtraction_assignment_operator; }
```

```
"<<="                    { return leftshift_assignment_operator; }
">>="                    { return rightshift_assignment_operator; }
"&="                     { return AND_assignment_operator; }
"^="                     { return XOR_assignment_operator; }
"|="                     { return OR_assignment_operator; }
"&"                              { return amp_operator; }
"!"                              { return exclamation_operator; }
"~"                              { return tilde_operator; }
"-"                              { return subtract_operator; }
"+"                              { return add_operator; }
"*"                              { return multiplication_operator; }
"/"                              { return division_operator; }
"%"                              { return modulo_operator; }
"|"                              { return pipe_operator; }
\=                               { return assignment_operator;}

\"[^\n]*\"/[;|,|\)]                      {strcpy(curval,yytext); insertCT(yytext,"String Constant");
return string_constant;}
\'[A-Z|a-z]\'/[;|,|\)|:]         {strcpy(curval,yytext); insertCT(yytext,"Character Constant");
return character_constant;}
[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[           {strcpy(curid,yytext); insertST(yytext, "Array Identifier");
return identifier;}
[1-9][0-9]*|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^]          {strcpy(curval,yytext);
insertCT(yytext, "Number Constant"); return integer_constant;}
([0-9]*)\.([0-9]+)/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^]          {strcpy(curval,yytext);
insertCT(yytext, "Floating Constant"); return float_constant;}
[A-Za-z_][A-Za-z_0-9]* {strcpy(curid,yytext);insertST(yytext,"Identifier");  return identifier;}

(.?) {
                if(yytext[0]=='#')
                {
                        printf("Error in Pre-Processor directive at line no. %d\n",yylineno);
                }
                else if(yytext[0]=='/')
                {
                        printf("ERR_UNMATCHED_COMMENT at line no. %d\n",yylineno);
                }
                else if(yytext[0]=='"')
                {
                        printf("ERR_INCOMPLETE_STRING at line no. %d\n",yylineno);
                }
                else
```

```
                    {
                            printf("ERROR at line no. %d\n",yylineno);
                    }
                    printf("%s\n", yytext);
                    return 0;
}

%%
```

2. Phase II: (ICG.l):

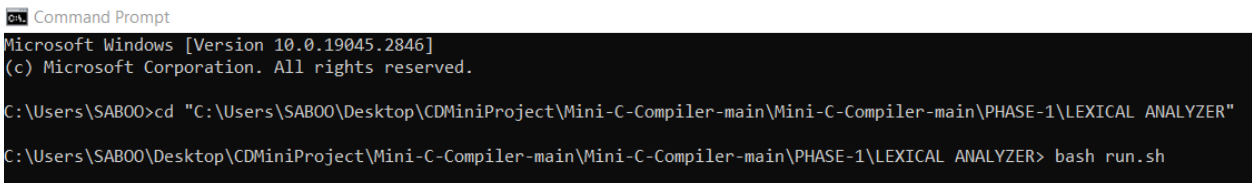
```
%{
#include"header.h"
#include <stdio.h>
#include "y.tab.h"
#include<string.h>

//#define YY_USER_ACTION yylloc.first_line = yylloc.last_line = yylineno;

extern void yyerror(const char *);  /* prints grammar violation message */

extern int sym_type(const char *);  /* returns type from symbol table */

#define sym_type(identifier) IDENTIFIER /* with no symbol table, fake it */

static void comment(void);

static int check_type(void);

int line = 0;

%}

%option yylineno

DIGIT   [0-9]
LETTER   [a-zA-Z_]
ALL   [a-zA-Z_0-9]
WS   [ \t\v\n\f]
CHARACTER \'[^\']\'
```

```
STRING \"[^\"]*\"


%%
	/* Unary operators */
"++" { return(INC_OP); }
"--" { return(DEC_OP); }

"/*" { comment(); }
"//"[^\n]* { /* Consume Comment */ }

	/* Data Types */
int { strcpy(yylval.string,yytext); return(INT); }
float { strcpy(yylval.string,yytext); return(FLOAT); }
char { strcpy(yylval.string,yytext); return(CHAR); }
void { strcpy(yylval.string,yytext); return(VOID); }
main { strcpy(yylval.string,yytext); return(MAIN); }
break       { strcpy(yylval.string,yytext); return(BREAK); }
switch      { strcpy(yylval.string,yytext); return(SWITCH); }
case        { strcpy(yylval.string,yytext); return(CASE); }
default     { strcpy(yylval.string,yytext); return(DEFAULT); }
return      { strcpy(yylval.string,yytext); return(RETURN); }

	/* User Defined Data Types */
struct { strcpy(yylval.string,yytext); return(STRUCT); }

	/* Headers */
"#include" { return PREPROC; }

	/* C Libraries */
"math.h" { return MATH; }
"stdio.h" { return STDIO; }
"string.h" {return STRING; }

	/* Control while */
"while" { return(WHILE); }

	/* User Defined Data Types, Identifiers */
{LETTER}{ALL}* { strcpy(yylval.string,yytext); return IDENTIFIER;}
{DIGIT}+ { strcpy(yylval.string,yytext); return INTEGER_LITERAL;}
{DIGIT}+\.{DIGIT}+ { strcpy(yylval.string,yytext); return FLOAT_LITERAL;}
\"{ALL}+(".h"|".c")\" {return HEADER_LITERAL;}
```

```
{CHARACTER} {return(CHARACTER_LITERAL);}
{LETTER}?\"(\\.|[^\\"])*\" {return(STRING_LITERAL); }

        /* Assignment Operators */
"+=" {return(ADD_ASSIGN); }
"-=" {return(SUB_ASSIGN); }
"*=" {return(MUL_ASSIGN); }
"/=" {return(DIV_ASSIGN); }
"%=" {return(MOD_ASSIGN); }

        /*Logical Operation*/
"&&" { strcpy(yylval.string,yytext);return(AND_LOG); }
"||" { strcpy(yylval.string,yytext);return(OR_LOG); }
"!" { strcpy(yylval.string,yytext);return(NOT); }

";" {return(';'); }
"{" {return('{'); }
"}" {return('}'); }
"," {return(','); }
":" {return(':'); }
"=" {return('='); }
"(" {return('('); }
")" {return(')'); }
("["|"<:") {return('['); }
("]"|":>") {return(']'); }
"." {return('.'); }
"&" {return('&'); }

        /*Relational op*/
"<" { strcpy(yylval.string,yytext);return(LT); }
">" { strcpy(yylval.string,yytext);return(GT); }
"<=" { strcpy(yylval.string,yytext); return(LE_OP); }
">=" { strcpy(yylval.string,yytext); return(GE_OP); }
"==" { strcpy(yylval.string,yytext); return(EQ_OP); }
"!=" { strcpy(yylval.string,yytext); return(NE_OP); }

        /* eval op */
"-" { strcpy(yylval.string,yytext);return(SUB); }
"+" { strcpy(yylval.string,yytext);return(ADD); }
"*" { strcpy(yylval.string,yytext);return(MUL); }
"/" { strcpy(yylval.string,yytext);return(DIV); }
"%" { strcpy(yylval.string,yytext);return(MOD); }
```

```
{WS}+ { ;/* whitespace separates tokens */}

. { return(yytext[0]); }
"\n" {yylval.ival = line++; printf("\n%d\n",line);}
%%

int yywrap(void)
{
return 1;
}

static void comment(void)
{
   int c;

   while ((c = input()) != 0)
      if (c == '*')
      {
         while ((c = input()) == '*')
            ;

         if (c == '/')
            return;

         if (c == 0)
            break;
      }
   yyerror("unterminated comment");
}
```

3. Execution Steps:



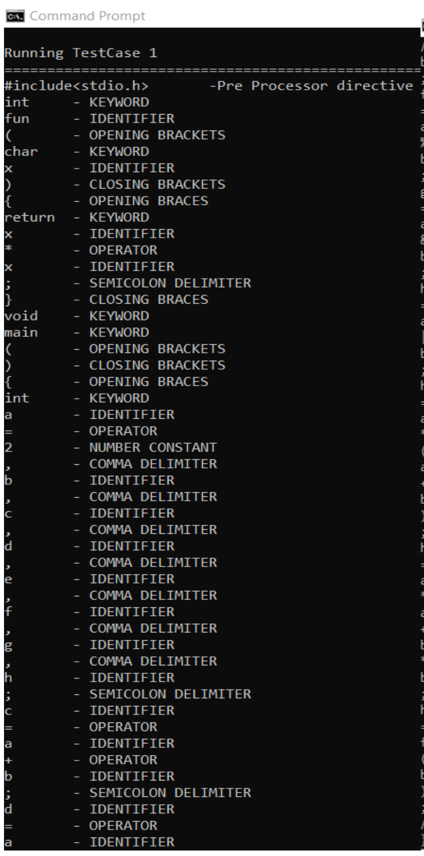
```
Command Prompt
Microsoft Windows [Version 10.0.19045.2846]
(c) Microsoft Corporation. All rights reserved.

C:\Users\SABOO>cd "C:\Users\SABOO\Desktop\CDMiniProject\Mini-C-Compiler-main\Mini-C-Compiler-main\PHASE-1\LEXICAL ANALYZER"

C:\Users\SABOO\Desktop\CDMiniProject\Mini-C-Compiler-main\Mini-C-Compiler-main\PHASE-1\LEXICAL ANALYZER> bash run.sh
```

a. cd <space> folder location in device
b. bash run.sh (according to PowerShell X86) – run command
c. press enter key

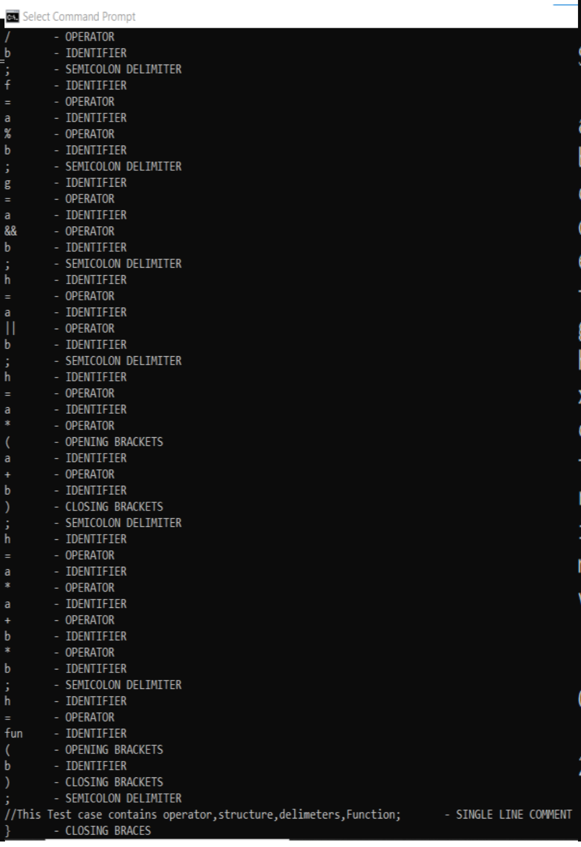
TEST CASES WITH SNAPSHOTS

- **Lexical Analyzer (test1.c)**

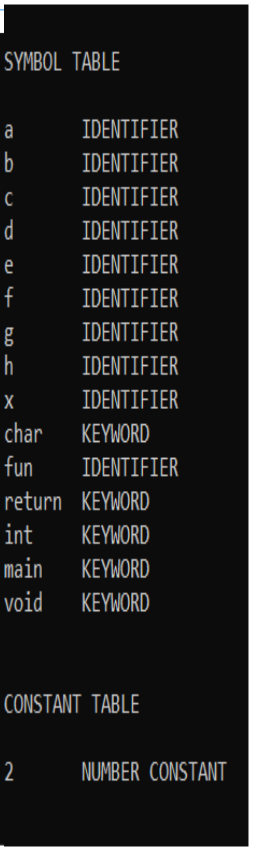
```
#include<stdio.h>
int fun(char x){
        return x*x;
}
void main(){
        int a=2,b,c,d,e,f,g,h;
        c=a+b;
        d=a*b;
        e=a/b;
        f=a%b;
        g=a&&b;
        h=a||b;
        h=a*(a+b);
        h=a*a+b*b;
        h=fun(b);
        //This Test case contains operator,structure,delimeters,Function;     }
```



```
Command Prompt

Running TestCase 1
================================================
#include<stdio.h>       -Pre Processor directive
int     - KEYWORD
fun     - IDENTIFIER
(       - OPENING BRACKETS
char    - KEYWORD
x       - IDENTIFIER
)       - CLOSING BRACKETS
{       - OPENING BRACES
return  - KEYWORD
x       - IDENTIFIER
*       - OPERATOR
x       - IDENTIFIER
;       - SEMICOLON DELIMITER
}       - CLOSING BRACES
void    - KEYWORD
main    - KEYWORD
(       - OPENING BRACKETS
)       - CLOSING BRACKETS
{       - OPENING BRACES
int     - KEYWORD
a       - IDENTIFIER
=       - OPERATOR
2       - NUMBER CONSTANT
,       - COMMA DELIMITER
b       - IDENTIFIER
,       - COMMA DELIMITER
c       - IDENTIFIER
,       - COMMA DELIMITER
d       - IDENTIFIER
,       - COMMA DELIMITER
e       - IDENTIFIER
,       - COMMA DELIMITER
f       - IDENTIFIER
,       - COMMA DELIMITER
g       - IDENTIFIER
,       - COMMA DELIMITER
h       - IDENTIFIER
;       - SEMICOLON DELIMITER
c       - IDENTIFIER
=       - OPERATOR
a       - IDENTIFIER
+       - OPERATOR
b       - IDENTIFIER
;       - SEMICOLON DELIMITER
d       - IDENTIFIER
=       - OPERATOR
a       - IDENTIFIER
```



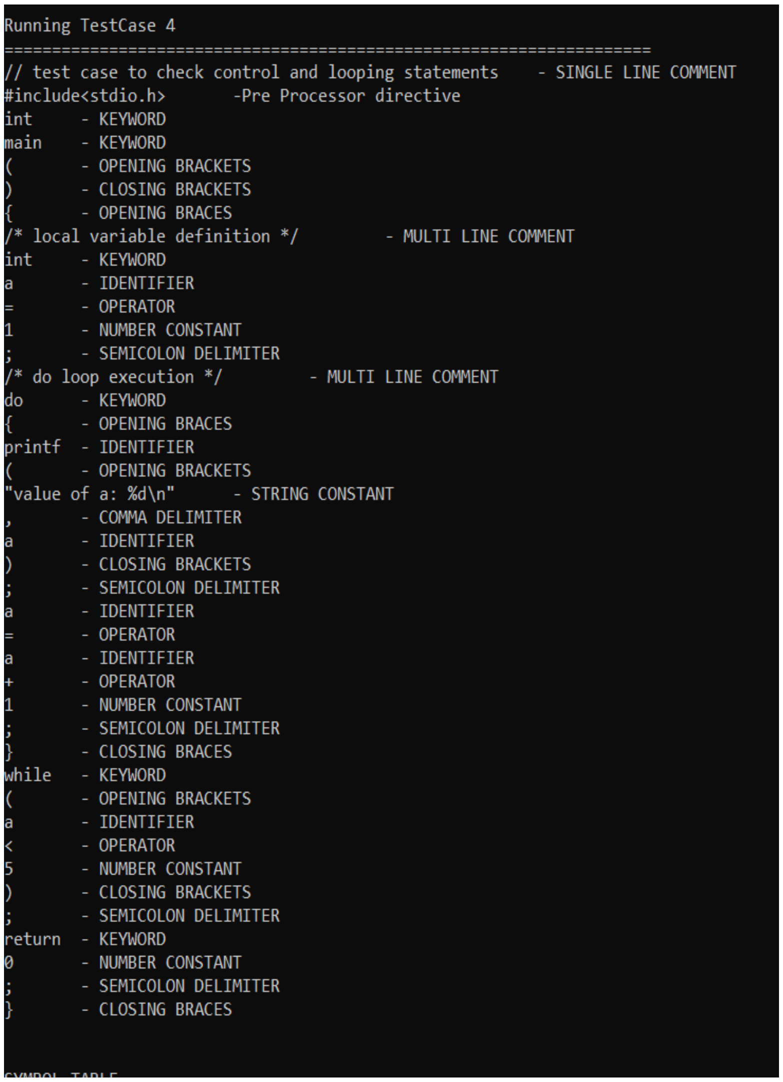
```
Select Command Prompt
/       - OPERATOR
b       - IDENTIFIER
;       - SEMICOLON DELIMITER
f       - IDENTIFIER
=       - OPERATOR
a       - IDENTIFIER
%       - OPERATOR
b       - IDENTIFIER
;       - SEMICOLON DELIMITER
g       - IDENTIFIER
=       - OPERATOR
a       - IDENTIFIER
&&      - OPERATOR
b       - IDENTIFIER
;       - SEMICOLON DELIMITER
h       - IDENTIFIER
=       - OPERATOR
a       - IDENTIFIER
||      - OPERATOR
b       - IDENTIFIER
;       - SEMICOLON DELIMITER
h       - IDENTIFIER
=       - OPERATOR
a       - IDENTIFIER
*       - OPERATOR
(       - OPENING BRACKETS
a       - IDENTIFIER
+       - OPERATOR
b       - IDENTIFIER
)       - CLOSING BRACKETS
;       - SEMICOLON DELIMITER
h       - IDENTIFIER
=       - OPERATOR
a       - IDENTIFIER
*       - OPERATOR
a       - IDENTIFIER
+       - OPERATOR
b       - IDENTIFIER
*       - OPERATOR
b       - IDENTIFIER
;       - SEMICOLON DELIMITER
h       - IDENTIFIER
=       - OPERATOR
fun     - IDENTIFIER
(       - OPENING BRACKETS
b       - IDENTIFIER
)       - CLOSING BRACKETS
;       - SEMICOLON DELIMITER
//This Test case contains operator,structure,delimeters,Function;       - SINGLE LINE COMMENT
}       - CLOSING BRACES
```



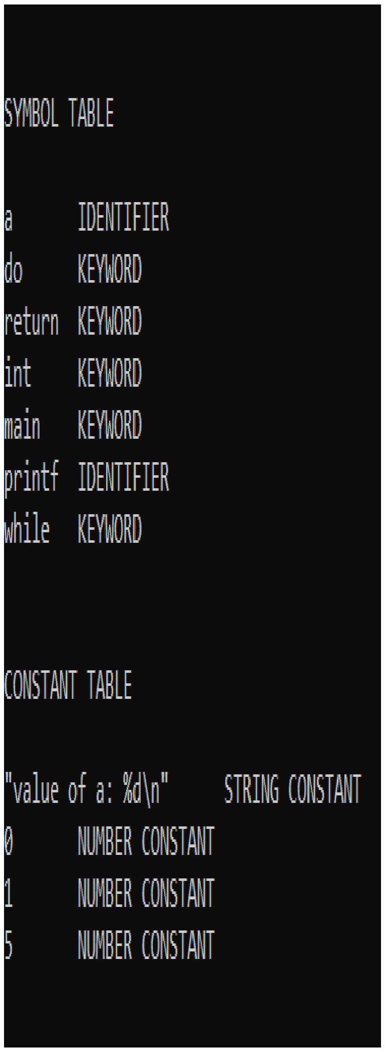
```
SYMBOL TABLE

a       IDENTIFIER
b       IDENTIFIER
c       IDENTIFIER
d       IDENTIFIER
e       IDENTIFIER
f       IDENTIFIER
g       IDENTIFIER
h       IDENTIFIER
x       IDENTIFIER
char    KEYWORD
fun     IDENTIFIER
return  KEYWORD
int     KEYWORD
main    KEYWORD
void    KEYWORD


CONSTANT TABLE

2       NUMBER CONSTANT
```

➢ Lexical Analyzer (test4.c)

```
// test case to check control and looping statements
#include<stdio.h>
int main () {
  /* local variable definition */
  int a = 1;
  /* do loop execution */
  do {
    printf("value of a: %d\n", a);
    a = a + 1;   }
  while ( a < 5 );
  return 0;
}
```



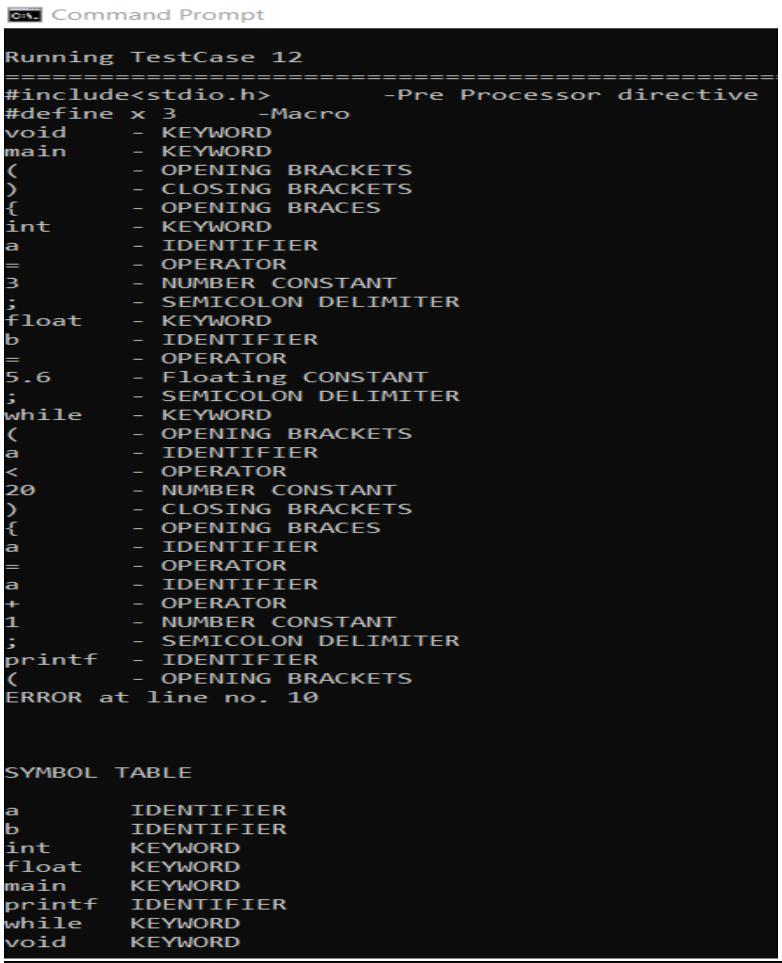
```
Running TestCase 4
=====================================================================
// test case to check control and looping statements    - SINGLE LINE COMMENT
#include<stdio.h>       -Pre Processor directive
int      - KEYWORD
main     - KEYWORD
(        - OPENING BRACKETS
)        - CLOSING BRACKETS
{        - OPENING BRACES
/* local variable definition */         - MULTI LINE COMMENT
int      - KEYWORD
a        - IDENTIFIER
=        - OPERATOR
1        - NUMBER CONSTANT
;        - SEMICOLON DELIMITER
/* do loop execution */         - MULTI LINE COMMENT
do       - KEYWORD
{        - OPENING BRACES
printf   - IDENTIFIER
(        - OPENING BRACKETS
"value of a: %d\n"      - STRING CONSTANT
,        - COMMA DELIMITER
a        - IDENTIFIER
)        - CLOSING BRACKETS
;        - SEMICOLON DELIMITER
a        - IDENTIFIER
=        - OPERATOR
a        - IDENTIFIER
+        - OPERATOR
1        - NUMBER CONSTANT
;        - SEMICOLON DELIMITER
}        - CLOSING BRACES
while    - KEYWORD
(        - OPENING BRACKETS
a        - IDENTIFIER
<        - OPERATOR
5        - NUMBER CONSTANT
)        - CLOSING BRACKETS
;        - SEMICOLON DELIMITER
return   - KEYWORD
0        - NUMBER CONSTANT
;        - SEMICOLON DELIMITER
}        - CLOSING BRACES


SYMBOL TABLE
```



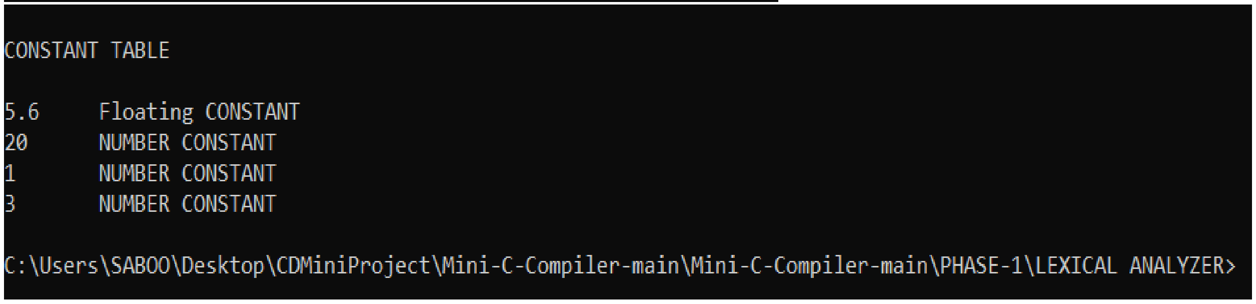
```
SYMBOL TABLE

a       IDENTIFIER
do      KEYWORD
return  KEYWORD
int     KEYWORD
main    KEYWORD
printf  IDENTIFIER
while   KEYWORD


CONSTANT TABLE

"value of a: %d\n"      STRING CONSTANT
0       NUMBER CONSTANT
1       NUMBER CONSTANT
5       NUMBER CONSTANT
```

➢ Lexical Analyzer (test12.c)

```
#include<stdio.h>
#define x 3
void main(){
        int a=3;
        float b=5.6;
        while(a<20){
                a = a + 1;
                printf("%d", a);        }
        a++;    }
```



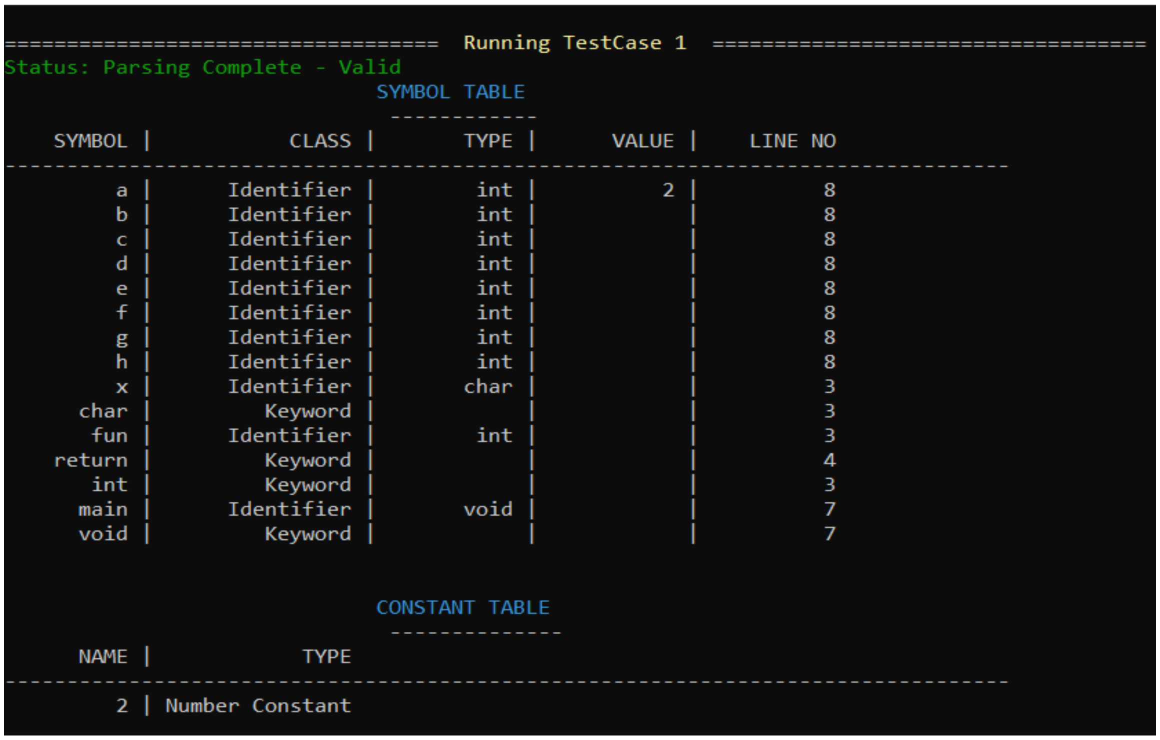
```
Command Prompt
Running TestCase 12
=================================================
#include<stdio.h>       -Pre Processor directive
#define x 3      -Macro
void      - KEYWORD
main      - KEYWORD
(         - OPENING BRACKETS
)         - CLOSING BRACKETS
{         - OPENING BRACES
int       - KEYWORD
a         - IDENTIFIER
=         - OPERATOR
3         - NUMBER CONSTANT
;         - SEMICOLON DELIMITER
float     - KEYWORD
b         - IDENTIFIER
=         - OPERATOR
5.6       - Floating CONSTANT
;         - SEMICOLON DELIMITER
while     - KEYWORD
(         - OPENING BRACKETS
a         - IDENTIFIER
<         - OPERATOR
20        - NUMBER CONSTANT
)         - CLOSING BRACKETS
{         - OPENING BRACES
a         - IDENTIFIER
=         - OPERATOR
a         - IDENTIFIER
+         - OPERATOR
1         - NUMBER CONSTANT
;         - SEMICOLON DELIMITER
printf    - IDENTIFIER
(         - OPENING BRACKETS
ERROR at line no. 10


SYMBOL TABLE

a       IDENTIFIER
b       IDENTIFIER
int     KEYWORD
float   KEYWORD
main    KEYWORD
printf  IDENTIFIER
while   KEYWORD
void    KEYWORD
```



```
CONSTANT TABLE

5.6     Floating CONSTANT
20      NUMBER CONSTANT
1       NUMBER CONSTANT
3       NUMBER CONSTANT

C:\Users\SABOO\Desktop\CDMiniProject\Mini-C-Compiler-main\Mini-C-Compiler-main\PHASE-1\LEXICAL ANALYZER>
```

➢ Parser (test1.c)

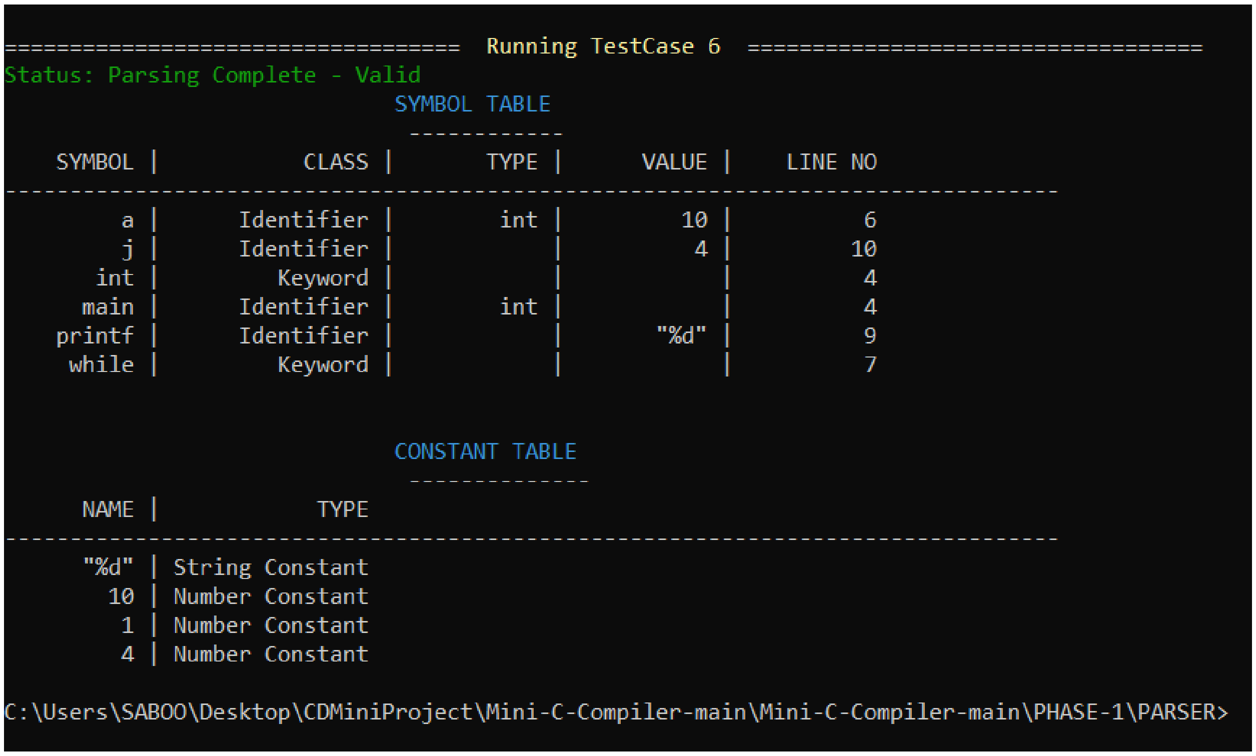
```
#include<stdio.h>

int fun(char x){

	return x*x;	}

void main(){

	int a=2,b,c,d,e,f,g,h;

	c=a+b;

	d=a*b;

	e=a/b;

	f=a%b;

	g=a&&b;

	h=a||b;

	h=a*(a+b);

	h=a*a+b*b;

	h=fun(b);

	//This Test case contains operator,structure,delimeters,Function;	}
```



```
===================================  Running TestCase 1  ===================================
Status: Parsing Complete - Valid
                              SYMBOL TABLE
                               ------------
   SYMBOL |            CLASS |        TYPE |       VALUE |     LINE NO
--------------------------------------------------------------------------------
        a |       Identifier |         int |           2 |           8
        b |       Identifier |         int |             |           8
        c |       Identifier |         int |             |           8
        d |       Identifier |         int |             |           8
        e |       Identifier |         int |             |           8
        f |       Identifier |         int |             |           8
        g |       Identifier |         int |             |           8
        h |       Identifier |         int |             |           8
        x |       Identifier |        char |             |           3
     char |          Keyword |             |             |           3
      fun |       Identifier |         int |             |           3
   return |          Keyword |             |             |           4
      int |          Keyword |             |             |           3
     main |       Identifier |        void |             |           7
     void |          Keyword |             |             |           7


                              CONSTANT TABLE
                               --------------
     NAME |             TYPE
--------------------------------------------------------------------------------
        2 | Number Constant
```

➢ Parser (test6.c)

```
//without error
#include<stdio.h>
#define x 3
int main()
{
    int a=4;
    while(a<10)
    {
        printf("%d",a);
        j=1;
        while(j<=4)
            j++;
    }
    }
```

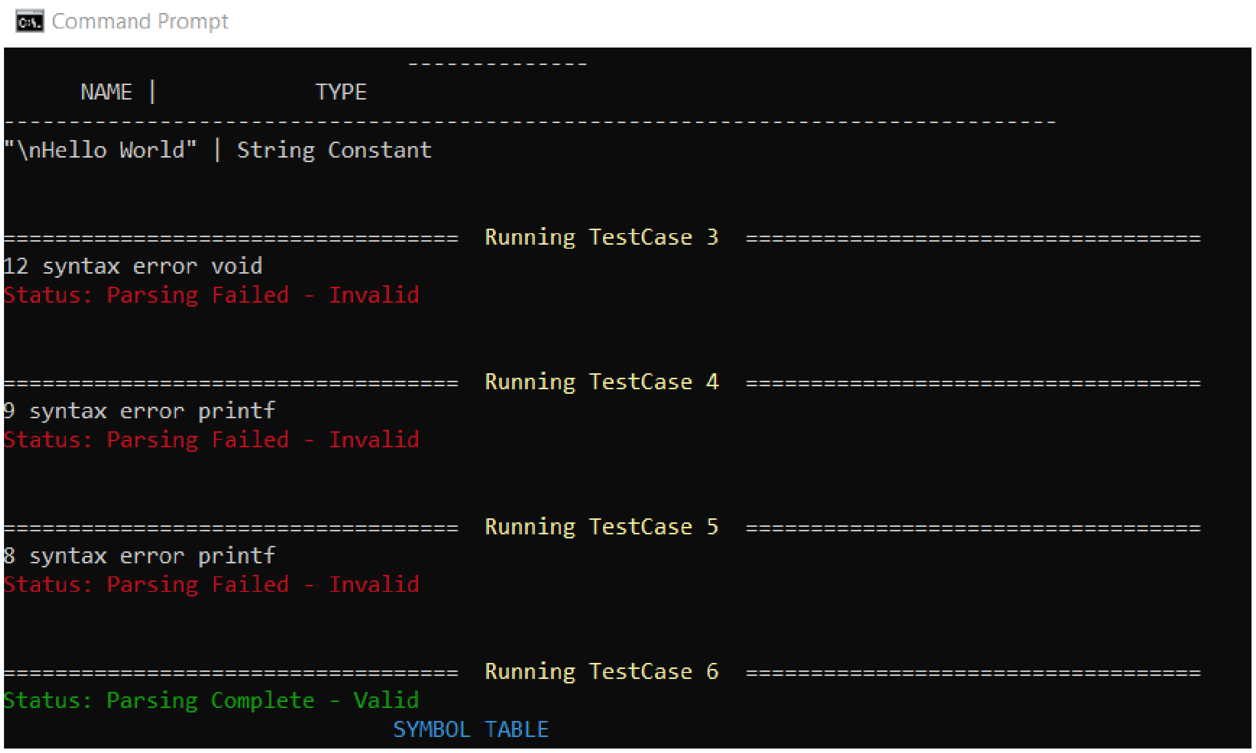


```
===================================  Running TestCase 6  ===================================
Status: Parsing Complete - Valid
                              SYMBOL TABLE
                               ------------
    SYMBOL |           CLASS |        TYPE |       VALUE |     LINE NO
--------------------------------------------------------------------------------
         a |      Identifier |         int |          10 |           6
         j |      Identifier |             |           4 |          10
       int |         Keyword |             |             |           4
      main |      Identifier |         int |             |           4
    printf |      Identifier |             |        "%d" |           9
     while |         Keyword |             |             |           7


                              CONSTANT TABLE
                               --------------
      NAME |            TYPE
--------------------------------------------------------------------------------
      "%d" | String Constant
        10 | Number Constant
         1 | Number Constant
         4 | Number Constant

C:\Users\SABOO\Desktop\CDMiniProject\Mini-C-Compiler-main\Mini-C-Compiler-main\PHASE-1\PARSER>
```

➢ PARSER (WITH-ERROR OUTPUT):

```
// with error
#include<stdio.h>
struct student{
   int a;
   char c;
};
union teacher{
   int q;
   int p;
}               //semicolon missing
void abc(){
   printf("\nHello World");
}
void main(){
   student *S;
   teacher *T;
   abc();
}

//with error - missing multiple semicolon
#include<stdio.h>
#define x 3
int main()
{
   //int c=4;
   int b=5
   /* multiline
   comment*/
   printf("%d",b);
      printf("this is for checking git commit");
   while(b<10)
   {
      printf("%d", b)
      b++;
   }
}
```

```
//with error - dangling else problem
#include<stdio.h>
#define x 3
int main()
{
    int a=4;
    if(a<10)
        printf("10");
    else
    {
        if(a<12)
            printf("11");
        else
            printf("All");
        else
            printf("error");
    }
}
```



Command Prompt

```
                                --------------
      NAME |              TYPE
----------------------------------------------------------------------------------
"\nHello World" | String Constant


===================================  Running TestCase 3  ===================================
12 syntax error void
Status: Parsing Failed - Invalid


===================================  Running TestCase 4  ===================================
9 syntax error printf
Status: Parsing Failed - Invalid


===================================  Running TestCase 5  ===================================
8 syntax error printf
Status: Parsing Failed - Invalid


===================================  Running TestCase 6  ===================================
Status: Parsing Complete - Valid
                              SYMBOL TABLE
```

➢ ICG Test Cases

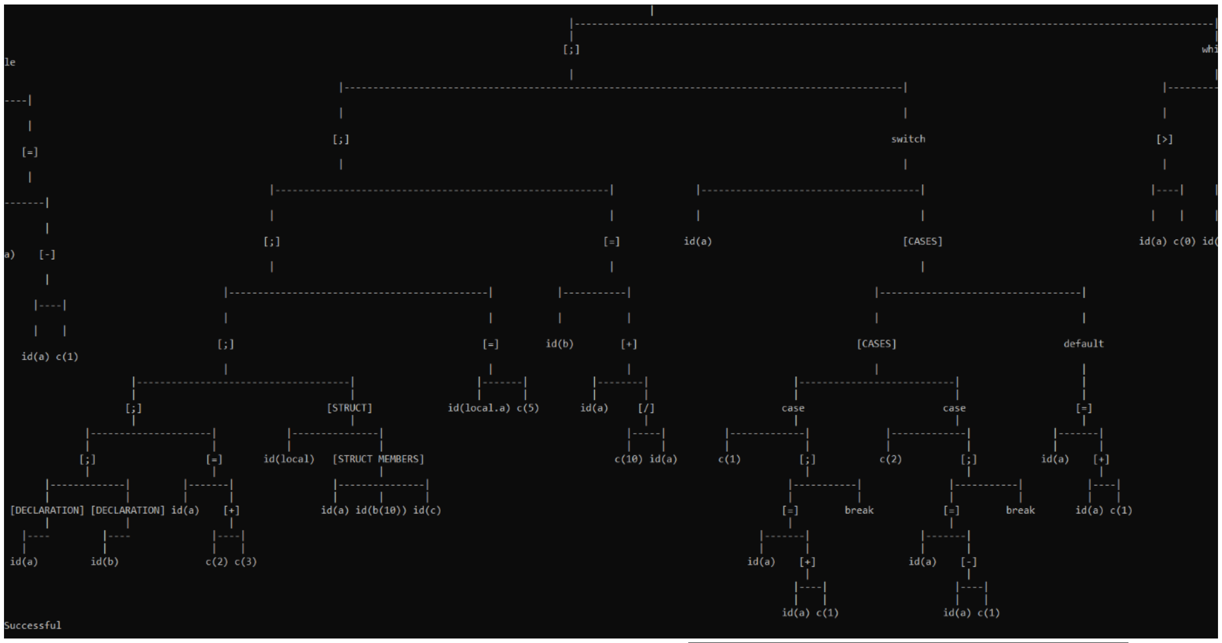
```
#include<stdio.h>

#include<string.h>

struct structa{

   int a;

   int b[10],c;

}structb;

int main(){

   int a;

   int b;

   a=2+3;

   struct structa local;

   local.a =5;

   b = a + 10/a;

   switch(a){

   case 1:a++;

   break;

   case 2: a--;

   break;

   default: a=a+1;        }

   while(a>0)    {

              a--;     } }
```

```
#include <stdio.h>

int main(){

         int a=5;

         int b=6;

         if(a<=7)
```

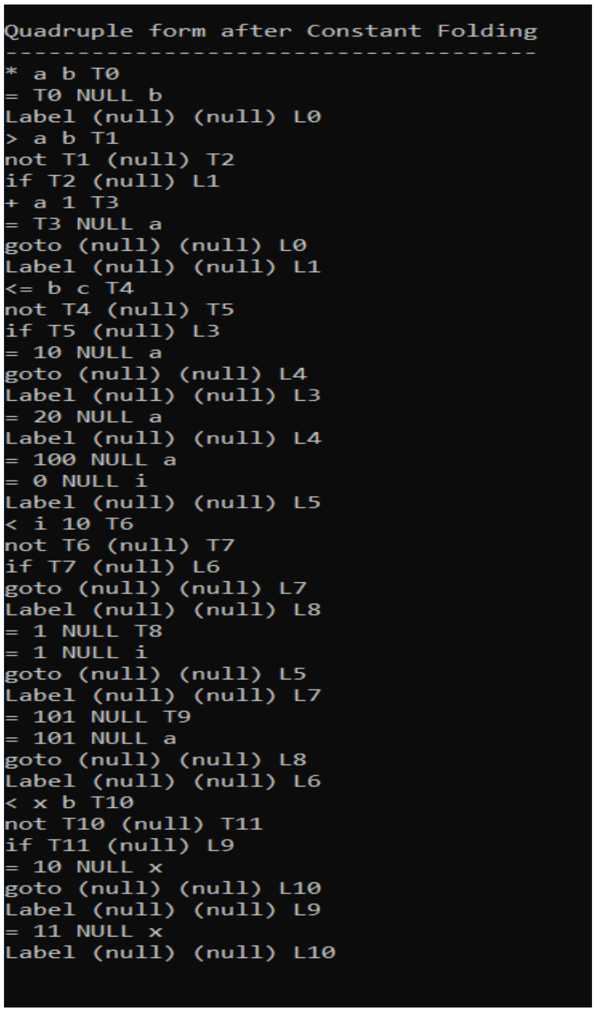
```
            b=b-4;
      else
            b=b+3;
      return 0;       }
```

```
#include<stdio.h>
void main(){
    int b = a*b;
    while( a > b ){
        a = a+1;     }
    if( b < = c ){
        a = 10;      }
    else{
        a = 20;      }
    a = 100;
      for(i=0;i<10;i = i+1){
          a = a+1;}
      (x < b) ? x = 10 : x=11;       }
```

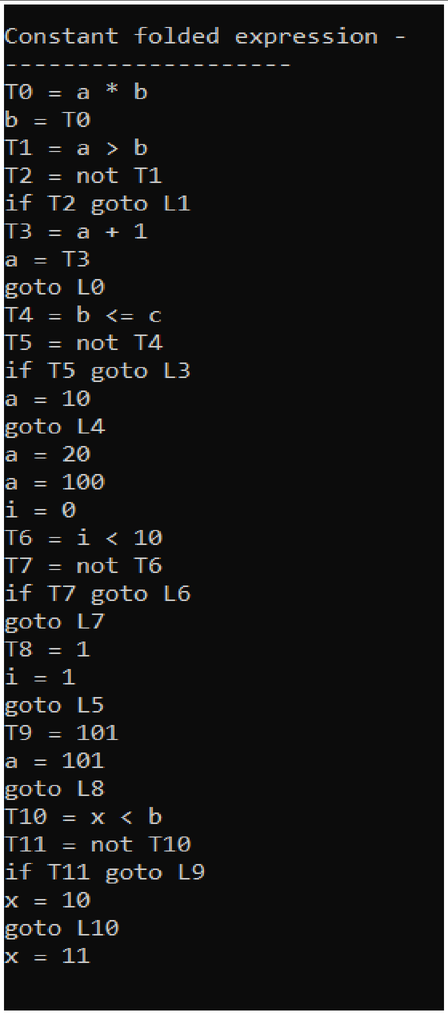
```
#include <stdio.h>
int main(){
  int d=3;
  int a[10];
  int x=a[d-2];
  return 0;    }
```



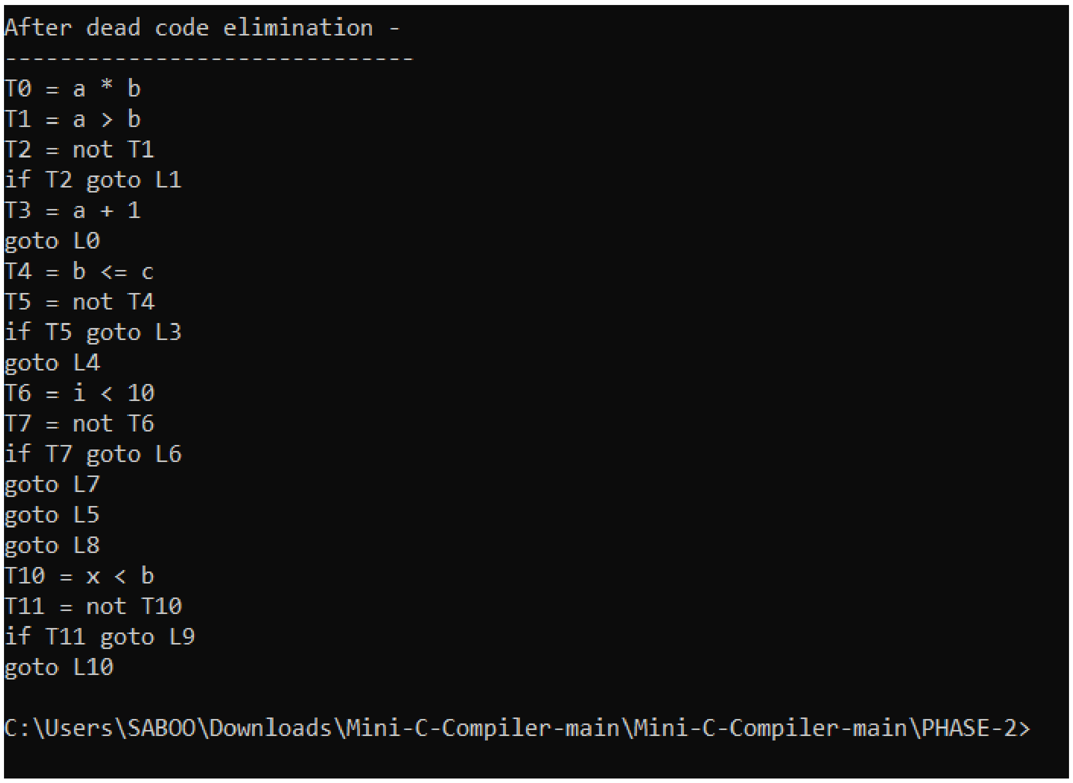
```
Successful
```

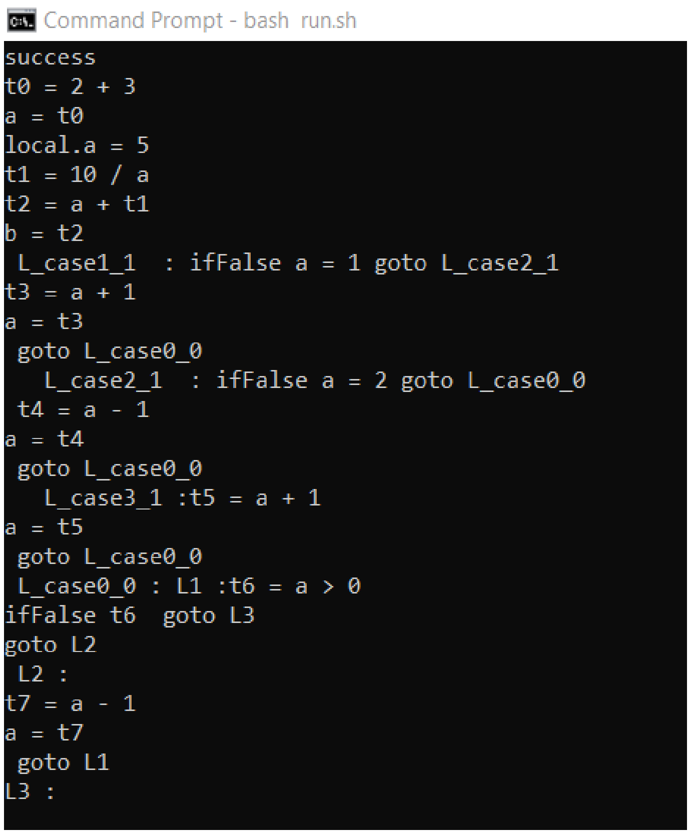


```
Quadruple form after Constant Folding
--------------------------------------
* a b T0
= T0 NULL b
Label (null) (null) L0
> a b T1
not T1 (null) T2
if T2 (null) L1
+ a 1 T3
= T3 NULL a
goto (null) (null) L0
Label (null) (null) L1
<= b c T4
not T4 (null) T5
if T5 (null) L3
= 10 NULL a
goto (null) (null) L4
Label (null) (null) L3
= 20 NULL a
Label (null) (null) L4
= 100 NULL a
= 0 NULL i
Label (null) (null) L5
< i 10 T6
not T6 (null) T7
if T7 (null) L6
goto (null) (null) L7
Label (null) (null) L8
= 1 NULL T8
= 1 NULL i
goto (null) (null) L5
Label (null) (null) L7
= 101 NULL T9
= 101 NULL a
goto (null) (null) L8
Label (null) (null) L6
< x b T10
not T10 (null) T11
if T11 (null) L9
= 10 NULL x
goto (null) (null) L10
Label (null) (null) L9
= 11 NULL x
Label (null) (null) L10
```



```
Constant folded expression -
--------------------
T0 = a * b
b = T0
T1 = a > b
T2 = not T1
if T2 goto L1
T3 = a + 1
a = T3
goto L0
T4 = b <= c
T5 = not T4
if T5 goto L3
a = 10
goto L4
a = 20
a = 100
i = 0
T6 = i < 10
T7 = not T6
if T7 goto L6
goto L7
T8 = 1
i = 1
goto L5
T9 = 101
a = 101
goto L8
T10 = x < b
T11 = not T10
if T11 goto L9
x = 10
goto L10
x = 11
```


```
After dead code elimination -
------------------------------
T0 = a * b
T1 = a > b
T2 = not T1
if T2 goto L1
T3 = a + 1
goto L0
T4 = b <= c
T5 = not T4
if T5 goto L3
goto L4
T6 = i < 10
T7 = not T6
if T7 goto L6
goto L7
goto L5
goto L8
T10 = x < b
T11 = not T10
if T11 goto L9
goto L10

C:\Users\SABOO\Downloads\Mini-C-Compiler-main\Mini-C-Compiler-main\PHASE-2>
```


Command Prompt - bash run.sh

```
success
t0 = 2 + 3
a = t0
local.a = 5
t1 = 10 / a
t2 = a + t1
b = t2
 L_case1_1  : ifFalse a = 1 goto L_case2_1
t3 = a + 1
a = t3
 goto L_case0_0
   L_case2_1  : ifFalse a = 2 goto L_case0_0
 t4 = a - 1
a = t4
 goto L_case0_0
   L_case3_1 :t5 = a + 1
a = t5
 goto L_case0_0
 L_case0_0 : L1 :t6 = a > 0
ifFalse t6  goto L3
goto L2
 L2 :
t7 = a - 1
a = t7
 goto L1
L3 :
```

CONCLUSION & FUTURE WORK

- The lexical analyser and the syntax analyser for a subset of C language, which include selection statements, compound statements, iteration statements, jumping statements, user defined functions and primary expressions is generated.
- It is important to note that conflicts (shift-reduce and reduce-reduce) may occur in case of syntax analyser if proper care is not taken while specifying the context-free grammar for the language. We should always specify unambiguous grammar for the parser to work properly.
- In a compiler the process of Intermediate code generation is independent of machine and the process of conversion of Intermediate code to target code is independent of language used.
- Making a full fledge compiler is a difficult as well as a tedious task.
- Thus, we have done the front end of compilation process (Mini Compiler) as our mini project.
- It includes 2 phases of compilation:
- lexical analysis
- syntax analysis
- Followed by intermediate code generation.
- Intermediate code generator generates three address codes for:
    1. Assignments
    2. Expressions
    3. Arrays
    4. conditional statements
    5. iterative statements

In future, we can extend it to support and generate three-address code for pointers, structures and functions.

- We have implemented the parser for only a subset of C language. The future work may include specifying the grammar for more pre-defined functions in C (like string functions, file read and write functions), go-to jump statements and enumerations.

REFERENCES

- geeksforgeeks.org
- tutorialspoint.com
- epaperpress.com
- youtube.com
- programiz.com
- researchgate.net