

**RAJALAKSHMI ENGINEERING COLLEGE**  
**AN AUTONOMOUS INSTITUTION**  
**AFFILIATED TO ANNA UNIVERSITY**  
**RAJALAKSHMI NAGAR, THANDALAM,**  
**CHENNAI-602105**



**RAJALAKSHMI**  
**ENGINEERING COLLEGE**  
An AUTONOMOUS Institution  
Affiliated to ANNA UNIVERSITY, Chennai

**DEPARTMENT OF COMPUTER SCIENCE**  
**AND ENGINEERING**

**CS19641 COMPILER DESIGN LABORATORY**  
**ACADEMIC YEAR: 2024-2025 (EVEN)**

**INDEX**

<b>Exp No</b>	<b>Date</b>	<b>Name of the Experiment</b>	<b>Mark/ Viva</b>	<b>Signature</b>
<b>1</b>		Develop A Simple C Program To Demonstrate A Basic String Operations		
<b>2</b>		Develop A C Program To Analyse A Given C Code Snippet And Recognize Different Tokens, Including Keyword, Identifiers, Operator, Delimiter And Special Symbols		
<b>3</b>		Develop A Lexical Analyser To Recognize A Few Patterns In C. (Ex. Identifiers, Constants, Comments, And Operators, Etc.) Using Lex Tool.		
<b>4</b>		Design And Implement A Desk Calculator Using The Lex Tool.		
<b>5</b>		Recognize A Valid Variable Which Starts With A Letter Followed By Any Number Of Letters Or Digits Using Lex And YACC		
<b>6</b>		Evaluate The Expression That Takes Digits, *, + Using Lex And YACC Reframe It.		
<b>7</b>		Recognize A Valid Control Structures Syntax Of C Language (For Loop, While Loop, If-Else, If-Else-If, Switch Case, Etc.		
<b>8</b>		Generate Three Address Code For A Simple Program Using Lex And YACC.		
<b>9</b>		Develop The Back-End Of A Compiler That Takes Three-Address Code (TAC) As Input And Generates Corresponding 8086 Assembly Language Code As Output.		
<b>10</b>		Generate Three Address Codes For A Given Expression (Arithmetic Expression, Flow Of Control).		
<b>11</b>		Implement Code Optimization Techniques Like Dead Code And Common Expression Elimination.		
<b>12</b>		Implement Code Optimization Techniques – Copy Propagation.		

**Hardware requirements:PC****Software requirements:C Compiler, Editplus (FLEX)**

LOKESHWAR S (220701146)

**Reference annexure 1 for installation**

**EXP NO:1**

**DATE:**

**DEVELOP A SIMPLE C PROGRAM TO DEMONSTRATE A BASIC STRING OPERATIONS**

**Questions**

**1. Input and Output**

- **Question:** Modify the program to take a string input from the user and display it in uppercase.
- **Hint:** Use the toupper function from <ctype.h> to convert characters to uppercase.

**2. String Length**

- **Question:** Write a C program to check if a given substring exists within a string without using the strstr() function. If the substring is found, print its starting index; otherwise, print "Substring not found."

**3. String Comparison**

- **Question:** Extend the program to compare two strings entered by the user and print whether they are the same.
- **Hint:** Use the strcmp function from <string.h> for comparison.

**4. Remove Spaces**

- **Question:** Write a program to remove all spaces from a string entered by the user.
- **Hint:** Use a loop to copy non-space characters to a new string.

**5. Frequency of Characters**

- **Question:** Modify the program to calculate the frequency of each character in the string.
- **Hint:** Use an array of size 256 to store the count of each ASCII character.

**6. Concatenate Strings**

- **Question:** Extend the program to concatenate two strings entered by the user.
- **Hint:** Use the strcat function from <string.h>.

**7. Replace a Character**

- **Question:** Write a program to replace all occurrences of a specific character in the string with another character.
- **Hint:** Traverse the string and replace the character conditionally in a loop.

**AIM:**

To write a C program that takes a string input from the user and converts all its characters to uppercase using the toupper() function from the library.

**ALGORITHM:**

1. Start
2. Declare a character array str to store the input string.
3. Prompt the user to enter a string.
4. Use fgets() to read the string input from the user.
5. Check if the last character is a newline (\n) and replace it with \0 (null terminator).
6. Loop through each character of the string:
7. Use toupper() to convert each character to uppercase.
8. Store the converted character back in the string.
9. Print the modified uppercase string. End

**PROGRAM:**

```
#include <stdio.h>
#include <ctype.h>

int main() {
    char str[100];

    printf("Enter a string: ");
    scanf("%s", str);

    for (int i = 0; str[i] != '\0'; i++) {
        str[i] = toupper(str[i]);
    }

    printf("Uppercase String: %s\n", str);
    return 0;
}
```

LOKESHWAR S (220701146)

**OUTPUT:**

```
Enter a string: hello  
Uppercase String: HELLO
```

**AIM:**

To write a C program that checks whether a given substring exists within a string without using the strstr() function. If found, print its starting index; otherwise, print "Substring not found."

**ALGORITHM:**

1. Start
2. Declare two character arrays: one for the main string and one for the substring.
3. Take input for both strings from the user.
4. Compute the lengths of both strings.
5. Loop through the main string and check for a match with the substring: o Compare characters one by one. o If a match is found, print the starting index and exit.
6. If no match is found, print "Substring not found."
7. End

**PROGRAM:**

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

int findSubstring(char str[], char substr[]) {
    int len1 = strlen(str);
    int len2 = strlen(substr);

    for (int i = 0; i <= len1 - len2; i++) {
        int j;
        for (j = 0; j < len2; j++) {
            if (str[i + j] != substr[j]) {
                break;
            }
        }
        if (j == len2) {
            return i;
        }
    }
    return -1;
}
```

LOKESHWAR S (220701146)

```
int main() {
    char str[100], substr[50];

    printf("Enter the main string: ");
    scanf("%s", str);

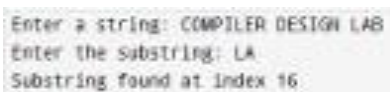
    printf("Enter the substring: ");
    scanf("%s", substr);

    int index = findSubstring(str, substr);

    if (index != -1)
        printf("Substring found at index %d\n", index);
    else
        printf("Substring not found\n");

    return 0;
}
```

### OUTPUT:

A screenshot of a terminal window showing the output of the program. It displays three lines of text: "Enter a string: COMPILER DESIGN LAB", "Enter the substring: LA", and "Substring found at index 16".

```
Enter a string: COMPILER DESIGN LAB
Enter the substring: LA
Substring found at index 16
```



### **AIM:**

To write a C program that compares two strings entered by the user and determines whether they are the same.

### **ALGORITHM:**

1. Start
2. Declare two character arrays to store the strings.
3. Take input for both strings from the user.
4. Use strcmp() to compare the two strings.
5. If the result is 0, print "Strings are the same."
6. Otherwise, print "Strings are different."
7. End

### **PROGRAM:**

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

int main() {
    char str1[100], str2[100];

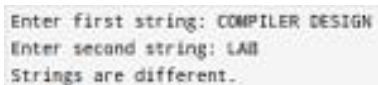
    printf("Enter first string: ");
    scanf("%s", str1);

    printf("Enter second string: ");
    scanf("%s", str2);

    if (strcmp(str1, str2) == 0)
        printf("Strings are the same.\n");
    else
        printf("Strings are different.\n");

    return 0;
}
```

### **OUTPUT:**

A screenshot of a terminal window showing the output of the C program. It displays three lines of text: "Enter first string: COMPILER DESIGN", "Enter second string: LAI", and "Strings are different.".

```
Enter first string: COMPILER DESIGN
Enter second string: LAI
Strings are different.
```

**AIM:**

To write a C program that removes all spaces from a string entered by the user.

**ALGORITHM:**

1. Start
2. Declare a character array for input.
3. Take string input from the user.
4. Traverse the string: o Copy only non-space characters to a new position in the array.
5. Print the modified string.
6. End

**PROGRAM:**

```
#include <stdio.h>

int main() {
    char str[100], result[100];
    int i, j = 0;

    printf("Enter a string: ");
    scanf("%[^\\n]", str);

    for (i = 0; str[i] != '\\0'; i++) {
        if (str[i] != ' ') {
            result[j++] = str[i];
        }
    }
    result[j] = '\\0';

    printf("String without spaces: %s\\n", result);
    return 0;
}
```

**OUTPUT:**

```
Enter a string: COMPILER DESIGN
String without spaces: COMPILERDESIGN
```

**AIM:**

To write a C program that calculates the frequency of each character in a given string.

**ALGORITHM:**

1. Start
2. Declare a character array for input.
3. Declare an integer array freq[256] initialized to 0 (for ASCII character frequencies).
4. Take string input from the user.
5. Traverse the string: o Increment the frequency count for each character.
6. Print characters with their respective frequencies.
7. End

**PROGRAM:**

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

int main() {
    char str[100];
    int freq[256] = {0};

    printf("Enter a string: ");
    scanf("%s", str);

    for (int i = 0; str[i] != '\0'; i++) {
        freq[(unsigned char)str[i]]++;
    }

    printf("Character frequencies:\n");
    for (int i = 0; i < 256; i++) {
        if (freq[i] > 0) {
            printf("%c = %d\n", i, freq[i]);
        }
    }
    return 0;
}
```

### OUTPUT:

```
Enter a string: Compiler design
Character Frequencies:
' ' : 1
' ' : 2
'C' : 1
'd' : 1
'e' : 2
'g' : 1
'i' : 2
'l' : 1
'm' : 1
'n' : 1
'o' : 1
'p' : 1
'r' : 1
's' : 1
```

**AIM:**

To write a C program that concatenates two strings entered by the user.

**ALGORITHM:**

1. Start
2. Declare two character arrays for input. 3
- . Take input for both strings.
4. Use strcat() to concatenate the second string to the first.
5. Print the concatenated result.
6. End

**PROGRAM:**

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

int main() {
    char str1[100], str2[100];

    printf("Enter first string: ");
    scanf("%s", str1);

    printf("Enter second string: ");
    scanf("%s", str2);

    strcat(str1, str2);

    printf("Concatenated String: %s\n", str1);
    return 0;
}
```

**OUTPUT:**

```
Enter a string: COMPILER DESIGN
String without spaces: COMPILERDESIGN
```

**AIM:**

To write a C program that replaces all occurrences of a specific character in a string with another character.

**ALGORITHM:**

1. Start
2. Declare a character array for input.
3. Take string input from the user.
4. Take input for the character to replace and its replacement.
5. Traverse the string: o Replace occurrences of the old character with the new one.
6. Print the modified string.
7. End

**PROGRAM:**

```
#include <stdio.h>

int main() {
    char str[100], oldChar, newChar;

    printf("Enter a string: ");
    scanf("%s", str);

    printf("Enter the character to replace: ");
    scanf(" %c", &oldChar);

    printf("Enter the new character: ");
    scanf(" %c", &newChar);

    for (int i = 0; str[i] != '\0'; i++) {
        if (str[i] == oldChar) {
            str[i] = newChar;
        }
    }

    printf("Modified String: %s\n", str);
    return 0;
}
```

**OUTPUT:**

```
Enter a string: compiler design
Enter character to replace: de
Enter new character: Modified string: compiler
esign
```

<b>Implementation</b>	
<b>Output/Signature</b>	

**RESULT:**

Thus the above program takes a string input, calculates and displays its length, copies and prints the string, concatenates it with a second input string, and finally compares both strings to check if they are the same or different.

**EXP NO: 02**

**DATE:**

**DEVELOP A C PROGRAM TO ANALYSE A GIVEN C CODE SNIPPET AND RECOGNIZE DIFFERENT TOKENS, INCLUDING KEYWORD, IDENTIFIERS, OPERATOR, DELIMITER AND SPECIAL SYMBOLS**

**AIM:**

To develop a C program that analyses a given C code snippet and recognizes different tokens, including keywords, identifiers, operators, delimiter and special symbols.

**ALGORITHM:**

- **Start**
- Take a C code snippet as input from the user or a file.
- Initialize necessary arrays and variables for keywords, identifiers, operators, and special symbols.
- Tokenize the input string using spaces, newlines, and other delimiters.
- For each token:
  - Check if it is a **keyword** (compare with a predefined list of C keywords).
  - Check if it is an **identifier** (valid variable/function name that doesn't match a keyword).
  - Check if it is an **operator** (e.g., +, -, \*, /, ==, &&).
  - Check if it is a **special symbol** (e.g., {, }, (, ), ;, ,).
- Print the categorized tokens.
- **End**

**PROGRAM:**

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main() {
    char input[100];
    char *str[] = {"int", "float", "long", "double", "printf"};
    int i=0, j=0, iskeyword=0;
    scanf("%[^END]s", input);
```



```
for(i=0;i<4;i++){
    int flag=1;
    for(j=0;str[i][j]!='\0';j++){
        if(input[j]!=str[i][j]){
            flag=0;
            break;
        }
    }
    if(flag) {
        iskeyword = 1;
        printf("%s is a keyword\n", str[i]);
        break;
    }
}
```

```
int start = j;
while(input[start]!='\0'){
    if(isalpha(input[start])){
        printf("%c",input[start]);
        start++;
        while(isalnum(input[start]) || input[start]=='_'){
            printf("%c",input[start]);
            start++;
        }
        printf(" is a identifier\n");
    }else if(isdigit(input[start])){
        printf("%c",input[start]);
        start++;
        while(isdigit(input[start])){
            printf("%c",input[start]);
            start++;
        }
        printf(" is a constant\n");
    }else if(input[start]=='.' || input[start]==';'){
        printf("%c is a delimiter\n",input[start]);
    }
}
```

```

        start++;
    }else if(input[start]=='+' ||input[start]=='-' || input[start]=='*' || input[start]=='/' || input[start]=='%' ||
input[start]=='=' ){
        printf("%c is a operator\n",input[start]);
        start++;
    }else if(input[start]=='(' ||input[start]==')' || input[start]=='{' || input[start]=='}' || input[start]=='[' ||
input[start]==']' ){
        printf("%c is a Symbol\n",input[start]);
        start++;
    }else{
        start++;
    }
}
return 0;
}

```

### OUTPUT:

```

Enter a C code snippet:
int main() {
    int a = 5, b = 10;
    float c = a + b;
    if (c > 10) {
        printf("Result: %f", c);
    }
    return 0;
}

Recognized Tokens:
Keyword: int
Identifier: main()
Special Symbol: {

```

<b>Implementation</b>	
<b>Output/Signature</b>	

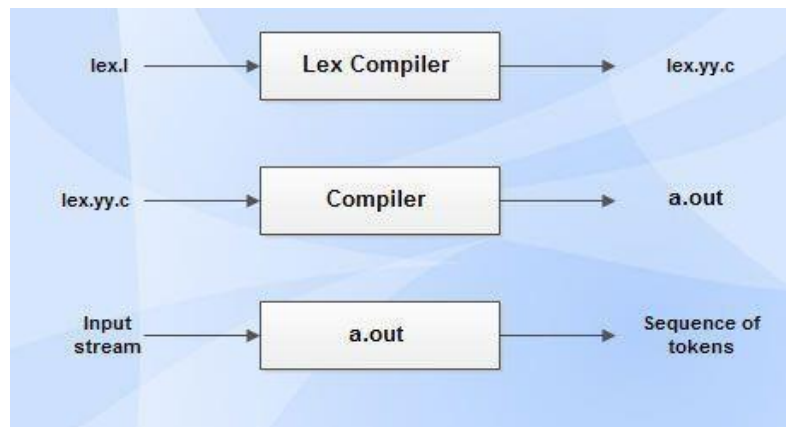
### RESULT :

Thus the above program reads a C code snippet, tokenizes it using space, tab, and newline as delimiters, classifies each token as a keyword, identifier, operator, or special symbol based on predefined lists, and prints the recognized tokens along with their types

## STUDY OF LEX TOOL

### LEX:

Lex is a tool in lexical analysis phase to recognize tokens using regular expression. Lex tool itself is a lex compiler.



- lex.l is an input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms lex.l to a C program known as lex.yy.c.
- lex.yy.c is compiled by the C compiler to a file called a.out.
- The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.
- yylval is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token.
- The attribute value can be numeric code, pointer to symbol table or nothing.
- Another tool for lexical analyzer generation is Flex.

### STRUCTURE OF LEX PROGRAMS:

Lex program will be in following form

declarations

%%

translation rules

%%

auxiliary functions

**Declarations:** This section includes declaration of variables, constants and regular definitions.

**Translation rules:** It contains regular expressions and code segments.

Form : Pattern { Action }

Pattern is a regular expression or regular definition.

Action refers to segments of code.

Patterns for tokens in the grammar

```
- digit → [0-9]
  digits → digit*
  number → digits ( . digits )? ( E [+-]? digits )?
  letter → [A-Za-z]
  id → letter ( letter | digit )*
  if → if
  then → then
  else → else
  relop → < | > | <= | >= | = | <>
- ws → (blank | tab | newline)+
```

```
% { LT, LE, EQ, NE, GT, GE, IF,
    THEN, ELSE, ID, NUMBER, RELOP
% }
delim    [ \t\n]
ws       {delim}+
letter   [A-Za-z]
digit    [0-9]
id       {letter}{(letter | digit)*}
number   {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
% %
{ws}     {}
if       { return(IF); }
then     { return(THEN); }
else     { return(ELSE); }
```

**Auxiliary functions:** This section holds additional functions which are used in actions. These functions are compiled separately and loaded with lexical analyzer. Lexical analyzer produced by lex starts its process by reading one character at a time until a valid match for a pattern is found. Once a match is found, the associated action takes place to produce token. The token is then given to parser for further processing.

## CONFLICT RESOLUTION IN LEX:

Conflict arises when several prefixes of input matches one or more patterns. This can be resolved by the following:

- Always prefer a longer prefix than a shorter prefix.
- If two or more patterns are matched for the longest prefix, then the first pattern listed in lex program is preferred

### **yylex()**

a function implementing the lexical analyzer and returning the token matched

### **yytext**

a global pointer variable pointing to the lexeme matched

### **yylen**

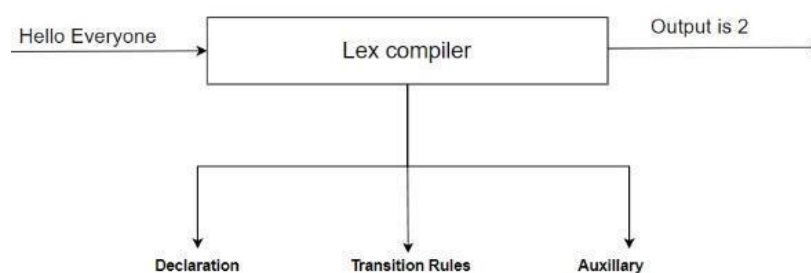
a global variable giving the length of the lexeme matched

### **yyval**

an external global variable storing the attribute of the token

### **yywrap:**

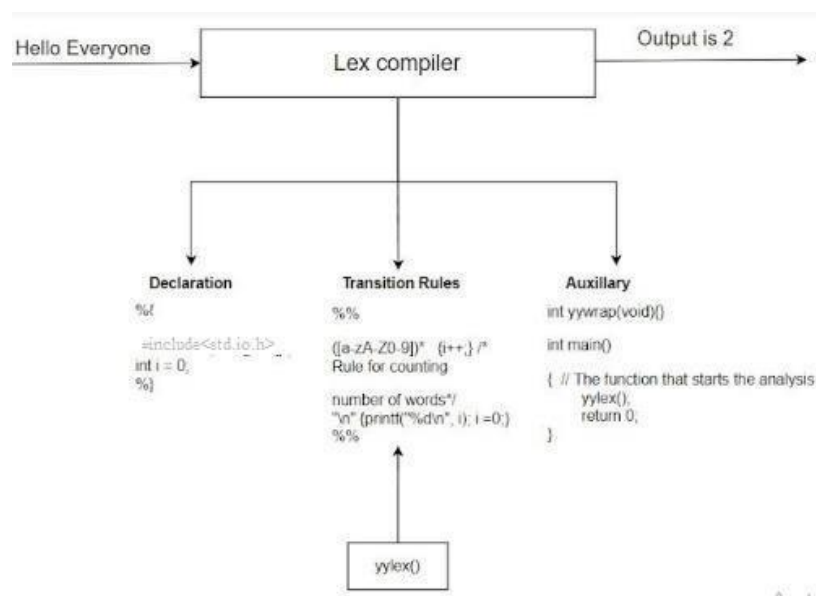
Function yywrap is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required.



Declaration-Has header files and initialization of variables

Translation rules-write the rules for counting the words

Auxillary- has yylex() that calls the translation rules



Simple lex program:

```
/*lex program to count number of words*/
% {
#include<stdio.h>
#include<string.h>
int i = 0;
% }

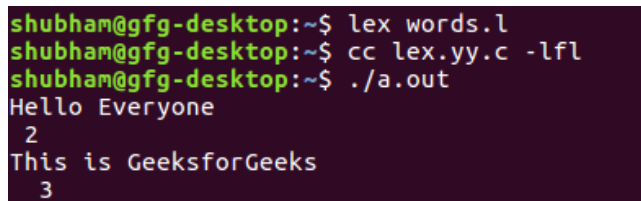
/* Rules Section*/
%%
([a-zA-Z0-9])* {i++;} /* Rule for counting
                        number of words*/

"\n" {printf("%d\n", i); i = 0;}
%%

int yywrap(void){ }

int main()
{
    // The function that starts the analysis
    yylex();

    return 0;
}
```



```
shubham@gfg-desktop:~$ lex words.l
shubham@gfg-desktop:~$ cc lex.yy.c -lfl
shubham@gfg-desktop:~$ ./a.out
Hello Everyone
2
This is GeeksforGeeks
3
```

**EXP NO: 03**

**DATE:**

**DEVELOP A LEXICAL ANALYSER TO RECOGNIZE A FEW PATTERNS IN C.  
(EX.IDENTIFIERS, CONSTANTS, COMMENTS, AND OPERATORS, ETC.) USING  
LEX TOOL.**

**AIM:**

To develop a Lexical Analyzer using the LEX tool that recognizes different tokens in a given C program snippet, including Identifier, Constants, Comments, Operators, Keywords, Special Symbols.

**ALGORITHM:**

- **Start**
- Define token patterns in **LEX** for:
  - **Keywords** (e.g., int, float, if, else)
  - **Identifiers** (variable/function names)
  - **Constants** (integer and floating-point numbers)
  - **Operators** (+, -, =, ==, !=, \*, /)
  - **Comments** (// single-line, /\* multi-line \*/)
  - **Special Symbols** ({, }, (, ), ;, ,)
- Read input source code.
- Match the code tokens using LEX rules.
- Print each recognized token with its type.
  - **End**

**PROGRAM:**

```
% {  
    #include <stdio.h>  
    #include <stdlib.h>  
    #include <string.h>  
    #include <stddef.h>  
% }  
  
%%  
"int"|"float"|"if"|"else"    { printf("KEYWORD: %s\n", yytext); }  
[a-zA-Z_][a-zA-Z0-9_]*      { printf("IDENTIFIER: %s\n", yytext); }  
[0-9]+                      { printf("INTEGER CONSTANT: %s\n", yytext); }
```

```
[0-9]*\.[0-9]+      { printf("FLOAT CONSTANT: %s\n", yytext); }
\\.*                { printf("SINGLE-LINE COMMENT\n"); }
\\*([^\*]\\*+[^/*])*\*/  { printf("MULTI-LINE COMMENT\n"); }
\+|\-|\*|\/|\%|==|!=    { printf("OPERATOR: %s\n", yytext); }
[\{\}\(\)\;\,]        { printf("SPECIAL SYMBOL: %s\n", yytext); }
[ \t\n]              { }
```

%%

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

```
int main() {
    yylex();
    return 0;
}
```



## OUTPUT:

```
lex lexer.l
cc lex.yy.c -o lexer
./a.out
Sample Input
int main() {
int a = 10;
float b = 20.5;
/* This is a multi-line comment */
if (a > b) {
a = a + b;
}
return 0;
}
```

```
Keyword: int
Identifier: main
Special Symbol: (
Special Symbol: )
Special Symbol: {
Keyword: int
Identifier: a
Operator: =
Constant: 10
Special Symbol: ;
Keyword: float
Identifier: b
Operator: =
Constant: 20.5
Special Symbol: ;
Multi-line Comment: /* This is a multi-line comment */
Keyword: if
Special Symbol: (
Identifier: a
Operator: >
Identifier: b
Special Symbol: )
Special Symbol: {
Identifier: a
```

<b>Implementation</b>	
<b>Output/Signature</b>	

**RESULT:**

Thus the above program reads a C code snippet, tokenizes it using LEX rules, recognizes and categorizes keywords, identifiers, constants, operators, comments, and special symbols, and then displays each token along with its type.

**EXP NO: 04**

**DATE:**

## **DESIGN AND IMPLEMENT A DESK CALCULATOR USING THE LEX TOOL**

### **Problem Statement**

Recognizes whether a given arithmetic expression is valid, using the operators +, -, \*, and /. The program should ensure that the expression follows basic arithmetic syntax rules (e.g., proper placement of operators, operands, and parentheses).

### **AIM:**

To design and implement a Desk Calculator using the LEX tool, which validates arithmetic expressions containing +, -, \*, /, numbers, and parentheses. The program ensures that the expression follows correct arithmetic syntax rules.

### **ALGORITHM:**

- **Start**
- Define token patterns in **LEX** for:
  - **Numbers** (integer and floating-point)
  - **Operators** (+, -, \*, /)
  - **Parentheses** ( (, ) )
  - **Whitespace** (to ignore spaces and tabs)
- Read an arithmetic expression as input.
- Use **LEX rules** to identify and validate tokens.
- If an **invalid token** is encountered, print an error message.
- If the expression is valid, print "Valid arithmetic expression."
- **End**

### **PROGRAM:**

```
% {  
#include <stdio.h>  
#include <stdlib.h>  
% }  
  
%%  
[0-9]+ { printf("NUMBER: %s\n", yytext); }  
[+\\-*/] { printf("OPERATOR: %s\n", yytext); }  
[n] { printf("NEWLINE\n"); }  
[ \\t] { /* Ignore whitespace */ }
```

```
.    { printf("INVALID CHARACTER: %s\n", yytext); }  
%%
```

```
int main() {  
    printf("Enter an expression: ");  
    yylex();  
    return 0;  
}
```

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

#### OUTPUT :

```
lex calculator.l  
cc lex.yy.c -o  
calculator  
./a.out
```

```
3 + 5 * (2 - 8)  
Number: 3  
Operator: +  
Number: 5  
Operator: *  
Left Parenthesis: (  
Number: 2  
Operator: -  
Number: 8  
Right Parenthesis: )  
Valid arithmetic expression.
```

Implementation	
Output/Signature	

#### RESULT:

Thus the above program reads an arithmetic expression, tokenizes it using LEX rules, and validates the syntax by recognizing numbers, operators (+, -, \*, /), and parentheses. If the expression is valid, it prints "Valid arithmetic expression." Otherwise, it detects and reports invalid tokens

## STUDY OF YACC TOOL

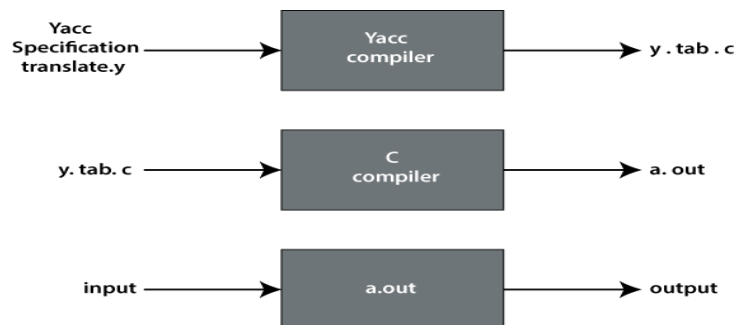
### YACC TOOL:

YACC is known as Yet Another Compiler Compiler. It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar. The input of YACC is the rule or grammar, and the output is a C program. If we have a file `translate.y` that consists of YACC specification, then the UNIX system command is:

### YACC `translate.y`

This command converts the file `translate.y` into a C file `y.tab.c`. It represents an LALR parser prepared in C with some other user's prepared C routines. By compiling `y.tab.c` along with the `ly` library, we will get the desired object program `a.out` that performs the operation defined by the original YACC program.

The construction of translation using YACC is illustrated in the figure below:



### STRUCTURE OF YACC:

Declarations

%%

Translation rules

%%

Supporting C rules

C declarations	<pre>%{ #include &lt;stdio.h&gt; %}</pre>
yacc declarations	<pre>%token NAME NUMBER %%</pre>
Grammar rules	<pre>statement: NAME '=' expression           expression           { printf("= %d\n", \$1); }         ;  expression: expression '+' NUMBER { \$\$ = \$1 + \$3; }           expression '-' NUMBER { \$\$ = \$1 - \$3; }           NUMBER                 { \$\$ = \$1; }         ; %%</pre>
Additional C code	<pre>int yyerror(char *s) {     fprintf(stderr, "%s\n", s);     return 0; }  int main(void) {     yyparse();     return 0; }</pre>

**Declarations Part:** This part of YACC has two sections; both are optional. The first section has ordinary C declarations, which is delimited by %{ and %}. Any temporary variable used by the second and third sections will be kept in this part. Declaration of grammar tokens also comes in the declaration part. This part defined the tokens that can be used in the later parts of a YACC specification.

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ID NUM
%start expr
```

**Terminal** points to `%token ID NUM`

**Start Symbol** points to `%start expr`

**Translation Rule Part:** After the first %% pair in the YACC specification part, we place the translation rules. Every rule has a grammar production and the associated semantic action.

A set of productions:

$$\langle \text{head} \rangle \Rightarrow \langle \text{body} \rangle_1 \mid \langle \text{body} \rangle_2 \mid \dots \mid \langle \text{body} \rangle_n$$

would be written in YACC as

```
<head> : <body>1    {<semantic action>1}
      | <body>2    {<semantic action>2}
      | .....
      | <body>n    {<semantic action>n}
      ;
```

Grammar rule section

```
expr  : expr '+' term
      | term
      ;
term   : term '*' factor
      | factor
      ;
factor : '(' expr ')'
      | ID
      | NUM
```

In a YACC production, an unquoted string of letters and digits that are not considered tokens is treated as non-terminals.

The semantic action of YACC is a set of C statements. In a semantic action, the symbol \$\$ is considered to be an attribute value associated with the head's non-terminal. While \$i is considered as the value associated with the grammar production of the body. If we have left only with associated production, the semantic action will be performed. The value of \$\$ is computed in terms of \$i's by semantic action.

**Supporting C–Rules:** It is the last part of the YACC specification and should provide a lexical analyzer named **yylex()**. These produced tokens have the token's name and are associated with its attribute value. Whenever any token like DIGIT is returned, the returned token name should have been declared in the first part of the YACC specification.

The attribute value which is associated with a token will communicate to the parser through a variable called **yylval**. This variable is defined by a YACC.

Whenever YACC reports that there is a conflict in parsing-action, we should have to create and consult the file **y.output** to see why this conflict in the parsing-action has arisen and to see whether the conflict has been resolved smoothly or not.

**EXP NO: 05**

**DATE:**

**RECOGNIZE A VALID VARIABLE WHICH STARTS WITH A LETTER  
FOLLOWED BY ANY NUMBER OF LETTERS OR DIGITS USING LEX AND  
YACC**

**Problem Statement:**

Recognizes a valid variable name. The variable name must start with a letter (either uppercase or lowercase) and can be followed by any number of letters or digits. The program should validate whether a given string adheres to this naming convention.

**AIM:**

To develop a **LEX and YACC program** that recognizes a **valid variable name** in C programming, which:

- Starts with a **letter** (a-z or A-Z)
- Followed by **any number of letters or digits** (a-z, A-Z, 0-9, \_)
- **Does not allow** invalid characters (e.g., 123abc, @var, x!y)

**ALGORITHM:**

**Step 1:** A Yacc source program has three parts as follows: Declarations %% translation rules  
%% supporting C routines

**Step 2:** Declarations Section: This section contains entries that:

Include standard I/O header file.

Define global variables.

Define the list rule as the place to start processing.

Define the tokens used by the parser.

**Step 3:** Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.

**Step 4:** Programs Section: The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the yacc library when processing this file.



Main- The required main program that calls the yyparse subroutine to start the program.

yyerror(s) -This error-handling subroutine only prints a syntax error message.

yywrap -The wrap-up subroutine that returns a value of 1 when the end of input occurs. The calc.lex file contains include statements for standard input and output, as programmer file information if we use the -d flag with the yacc command. The y.tab.h file contains definitions for

the tokens that the parser program uses.

**Step 5:**calc.lex contains the rules to generate these tokens from the input stream.

### PROGRAM:

Lex.l

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

int yywrap(void) {
    return 1;
}

%}

%%

[a-zA-Z_][a-zA-Z0-9_]* { return IDENTIFIER; }
\n                { return 0; }
.                  { return yytext[0]; }

%%
```

Yac.y

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

extern char *yytext;
int yylex();
int yyerror(char *msg);
```

LOKESHWAR S (220701146)

```
% }

%token IDENTIFIER

%%

variable: IDENTIFIER { printf("Valid variable name: %s\n", yytext); }
;

%%

int main() {
    printf("Enter a variable name:\n");
    yyparse();
    return 0;
}

int yyerror(char *msg) {
    printf("Error: %s\n", msg);
    return 0;
}
```

### OUTPUT :

```
yacc -d parser.y
lex lexer.l
cc lex.yy.c y.tab.c -o var_checker
./a.out

Enter a variable name: myVar1
Valid variable: myVar1
Enter a variable name: Hello123
Valid variable: Hello123
```

Implementation	
Output/Signature	

### RESULT:

Thus the above program reads an input string, checks whether it follows the rules for a valid variable name, and produces the following output.

**EXP NO: 06**

**DATE:**

**EVALUATE THE EXPRESSION THAT TAKES DIGITS, \*, + USING LEX AND YACC**

**AIM:**

To design and implement a **LEX and YACC program** that evaluates arithmetic expressions containing **digits, +, and \*** while following operator precedence rules.

**ALGORITHM:**

- Using the flex tool, create lex and yacc files.
- In the definition section of the lex file, declare the required header files along with an external integer variable yylval.
- In the rule section, if the regex pertains to digit convert it into integer and store yylval. Return the number.
- In the user definition section, define the function yywrap()
- In the definition section of the yacc file, declare the required header files along with the flag variables set to zero. Then define a token as number along with left as '+', '-', 'or', '\*', '/', '%' or '(' )'
- In the rules section, create an arithmetic expression as E. Print the result and return zero.
- Define the following:
  - E: E '+' E (add)
  - E: E '-' E (sub)
  - E: E '\*' E (mul)
  - E: E '/' E (div)
- If it is a single number return the number.
- In driver code, get the input through yyparse(); which is also called as main function.
- Declare yyerror() to handle invalid expressions and exceptions.
- Build lex and yacc files and compile.

**PROGRAM:**

Digits.l

```
% {
#include "digits.tab.h"
extern int yylval;
% }

%%

[0-9]+    { yylval = atoi(yytext); return NUMBER; }
[ \t\n]   ; // Skip whitespace
.         { return yytext[0]; }
```

%%

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

Digits.y

```
%{  
#include <stdio.h>  
#include <stdlib.h>  
int yylex();  
void yyerror(char *msg); // Declared as void to match definition  
%}
```

%token NUMBER

%left '+' '-'

%left '\*' '/'

%%

```
S: E {  
    printf("Result = %d\n", $1);  
    return 0;  
};
```

```
E: E '+' E { $$ = $1 + $3; }  
  | E '-' E { $$ = $1 - $3; }  
  | E '*' E { $$ = $1 * $3; }  
  | E '/' E {  
      if ($3 == 0) {  
          printf("Error: Division by zero\n");  
          exit(1);  
      }  
      $$ = $1 / $3;  
  }  
  | '(' E ')' { $$ = $2; }  
  | NUMBER { $$ = $1; }  
;
```

%%

```
int main() {  
    printf("Enter an arithmetic expression:\n");  
    yyparse();  
    return 0;  
}
```

LOKESHWAR S (220701146)

```
}  
  
void yyerror(char *msg) {  
    printf("Syntax Error: %s\n", msg);  
}
```

### OUTPUT :

```
lex expr.l  
yacc -d expr.y  
gcc lex.yy.c y.tab.c -o expr_eval  
./expr_eval  
Enter an arithmetic expression: 3 + 5 * 2  
Result: 13
```

Implementation	
Output/Signature	

### RESULT:

Thus the above program to evaluate the expression that takes digits, \*, + using lex and yacc is been implemented and executed successfully based on the precedence.

**EXP NO: 07**

**DATE:**

**RECOGNIZE A VALID CONTROL STRUCTURES SYNTAX OF C LANGUAGE  
(FOR LOOP, WHILE LOOP, IF-ELSE, IF-ELSE-IF, SWITCH CASE, ETC.,**

**AIM:**

To design and implement a LEX and YACC program that recognizes the syntax of common control structures in C programming, including:

For loop

- While loop
- If-else
- If-else-if
- Switch-case

**ALGORITHM:**

LEX (Lexical Analyzer)

1. Start
2. Define token patterns for:
  - Keywords (e.g., if, else, for, while, switch, case)
  - Identifiers (variable names)
  - Operators (arithmetic and relational)
  - Parentheses ((), {}, etc.)
  - Semicolon (;)
3. Pass recognized tokens to YACC for syntax validation.
4. End

YACC (Syntax Analyzer)

1. Start
2. Define grammar rules for:
  - For loop: for(initialization; condition; increment) { ... }
  - While loop: while(condition) { ... }
  - If-else: if(condition) { ... } else { ... }
  - If-else-if: if(condition) { ... } else if(condition) { ... } else { ... }
  - Switch-case: switch(expression) { case value: ... default: ... }
3. Parse the input expression and validate the syntax of the control structures.
4. Print appropriate messages for valid or invalid control structure syntax.
5. End

## PROGRAM:

Control.1

```
%{
#include "control.tab.h"
#include <string.h>
#include <stdlib.h>
%}

%%

"if"      { return IF; }
"else"    { return ELSE; }
"while"   { return WHILE; }
"for"     { return FOR; }

"<="      { return LE; }
">="      { return GE; }
"=="      { return EQ; }
"!="      { return NE; }

[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); return ID; }
[0-9]+      { yylval.str = strdup(yytext); return NUM; }

"="       { return '='; }
"+"       { return '+'; }
"-"       { return '-'; }
"*"       { return '*'; }
"/"       { return '/'; }
"<"       { return '<'; }
">"       { return '>'; }

"("       { return '('; }
")"       { return ')'; }
"{"       { return '{'; }
"}"       { return '}'; }
";"       { return ';'; }

[ \t\n]+   { /* ignore whitespace */ }
.          { return yytext[0]; }
%%

int yywrap() { return 1; }

control.y
```

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

void yyerror(const char* s);
int yylex();
extern FILE* yyin;
% }

%union {
    char* str;
}

%token <str> ID NUM
%token IF ELSE WHILE FOR
%token LE GE EQ NE

%left '<' '>' LE GE EQ NE
%left '+' '-'
%left '*' '/'

%%

program:
    stmt_list
    ;

stmt_list:
    stmt
    | stmt_list stmt
    ;

stmt:
    expr ';' { /* Expression statement - skip message */ }
    | IF '(' expr ')' stmt { printf("IF condition works\n"); }
    | IF '(' expr ')' stmt ELSE stmt { printf("IF-ELSE condition works\n"); }
    | WHILE '(' expr ')' stmt { printf("WHILE loop works\n"); }
    | FOR '(' expr ';' expr ';' expr ')' stmt { printf("FOR loop works\n"); }
    | '{' stmt_list '}' { /* Compound statement - no message needed */ }
    ;

expr:
    ID '=' expr
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
```



```
| expr '/' expr
| expr '<' expr
| expr '>' expr
| expr LE expr
| expr GE expr
| expr EQ expr
| expr NE expr
| ID
| NUM
;
%%
```

```
void yyerror(const char* s) {
    printf("Syntax Error: %s\n", s);
}

int main() {
    printf("Enter a C control structure (Ctrl+Z to stop):\n");
    yyin = stdin; // set to standard input
    yyparse();
    return 0;
}
```

### OUTPUT :

```
yacc -d control_structures.y
lex control_structures.l
gcc lex.yy.c y.tab.c -o control_validator
./control_validator
```

```
if (a > b) {
    // statements
} else {
    // statements
}
```

```
for (int i = 0; i < 10; i++) {
    // statements
}
```

<b>Implementation</b>	
<b>Output/Signature</b>	

### RESULT:

Thus the above program to recognize a valid control structures syntax of c language (for loop, while loop, if-else, if-else-if, switch case as been implemented and executed successfully with LEX and YACC.

**EXP NO: 08**

**DATE:**

## **GENERATE THREE ADDRESS CODE FOR A SIMPLE PROGRAM USING LEX AND YACC**

### **AIM:**

To design and implement a **LEX and YACC** program that generates **three-address code (TAC)** for a simple arithmetic expression or program. The program will:

- Recognize **expressions** like addition, subtraction, multiplication, and division.
- Generate **three-address code** that represents the operations in a way that could be directly translated into assembly code or intermediate code for a compiler.

### **ALGORITHM:**

1. Lexical Analysis (LEX) Phase:

**Input:** A string containing an arithmetic expression (e.g.,  $a = b + c * d$ ).

**Output:** A stream of tokens such as identifiers (variables), numbers (constants), operators, and special characters (like =, :, (), etc.).

#### **1. Define the Token Patterns:**

- **ID:** Identifiers (variables) are strings starting with a letter and followed by letters or digits (e.g., a, b, result).
- **NUMBER:** Constants (e.g., 1, 5, 100).
- **OPERATOR:** Arithmetic operators (+, -, \*, /).
- **ASSIGNMENT:** Assignment operator (=).
- **PARENTHESIS:** Parentheses for grouping (( and )).
- **WHITESPACE:** Spaces, tabs, and newline characters (which should be ignored).

#### **2. Write Regular Expressions for the Tokens:**

- ID -> [a-zA-Z\_][a-zA-Z0-9\_]\*
- NUMBER -> [0-9]+
- OPERATOR -> [\+|\-|\\*|/]
- ASSIGN -> "="
- PAREN -> [\(|\)]
- WHITESPACE -> [\t\n]+ (skip whitespace)

#### **3. Action on Tokens:**

- When a token is matched, pass it to **YACC** using yylval to store the token values.

2. Syntax Analysis and TAC Generation (YACC) Phase:

**Input:** Tokens provided by the **LEX** lexical analyzer.

**Output:** Three-address code for the given arithmetic expression.

**1. Define Grammar Rules:**

○ **Assignment:**

```
bash
CopyEdit
statement: ID '=' expr
```

This means an expression is assigned to a variable.

○ **Expressions:**

```
bash
CopyEdit
expr: expr OPERATOR expr
```

An expression can be another expression with an operator (+, -, \*, /).

```
bash
CopyEdit
expr: NUMBER
expr: ID
expr: '(' expr ')'
```

**2. Three-Address Code Generation:**

- For every arithmetic operation, generate a temporary variable (e.g., t1, t2, etc.) to hold intermediate results.
- For  $a = b + c$ , generate:

```
ini
CopyEdit
t1 = b + c
a = t1
```

- For  $a = b * c + d$ , generate:

```
ini
CopyEdit
t1 = b * c
t2 = t1 + d
a = t2
```

**3. Temporary Variable Management:**

- Keep a counter (temp\_count) for generating unique temporary variable names (t0, t1, t2, ...).
- Each time a new operation is encountered, increment the temp\_count to generate a new temporary variable.

**4. Rule Actions:**

- When a rule is matched (e.g.,  $\text{expr OPERATOR expr}$ ), generate the TAC and assign temporary variables for intermediate results.

Detailed Algorithm:

1. **Initialize Lexical Analyzer:**
  - Define the token patterns for ID, NUMBER, OPERATOR, ASSIGN, PAREN, and WHITESPACE.
2. **Define the Syntax Grammar:**
  - Define grammar rules for:
    - **Assignments:** ID = expr
    - **Expressions:** expr -> expr OPERATOR expr, expr -> NUMBER, expr -> ID, expr -> (expr)
3. **Token Matching:**
  - **LEX:** Match input characters against the defined regular expressions for tokens.
  - **YACC:** Use the tokens to parse and apply grammar rules.
4. **TAC Generation:**
  - **For Assignment:**
    - Upon parsing ID = expr, generate a temporary variable for the result of expr and assign it to the variable ID.
  - **For Arithmetic Operations:**
    - For each operator (e.g., +, -, \*, /), generate temporary variables for intermediate calculations.
5. **Output TAC:**
  - Print the generated three-address code, with each expression and its intermediate results represented by temporary variables.

## PROGRAM:

3address.l

```
% {
#include "3address.tab.h"
#include <string.h>
#include <stdlib.h>
% }

ID    [a-zA-Z_][a-zA-Z0-9_]*
NUM   [0-9]+

%%

{ID}  { yylval.str = strdup(yytext); return ID; }
{NUM} { yylval.str = strdup(yytext); return NUM; }
"="   { return '='; }
";"   { return ';'; }
"("   { return '('; }
")"   { return ')'; }
"+"   { return '+'; }
```

```
"-" { return '-'; }
"*" { return '*'; }
"/" { return '/'; }
[ \t\n] ; // skip whitespace

%%

int yywrap() {
    return 1;
}

3address.y
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int tempCount = 0;

char* createTemp() {
    char* temp = (char*)malloc(10);
    sprintf(temp, "t%d", tempCount++);
    return temp;
}

void yyerror(const char* s);
int yylex();
%}

%union {
    char* str;
}

%token <str> ID NUM
%type <str> expr

%left '+' '-'
%left '*' '/'

%%

stmt:
    ID '=' expr ';' {
        printf("%s = %s\n", $1, $3);
    }
    ;

expr:
    expr '+' expr {
```

```
        char* temp = createTemp();
        printf("%s = %s + %s\n", temp, $1, $3);
        $$ = temp;
    }
| expr '-' expr {
    char* temp = createTemp();
    printf("%s = %s - %s\n", temp, $1, $3);
    $$ = temp;
}
| expr '*' expr {
    char* temp = createTemp();
    printf("%s = %s * %s\n", temp, $1, $3);
    $$ = temp;
}
| expr '/' expr {
    char* temp = createTemp();
    printf("%s = %s / %s\n", temp, $1, $3);
    $$ = temp;
}
| '(' expr ')' {
    $$ = $2;
}
| ID {
    $$ = strdup($1);
}
| NUM {
    $$ = strdup($1);
}
;

%%

void yyerror(const char* s) {
    printf("Syntax Error: %s\n", s);
}

int main() {
    printf("Enter an arithmetic expression :\n");
    yyparse();
    return 0;
}
```

**OUTPUT :**

```
yacc -d expr.y
lex expr.l
gcc y.tab.c lex.yy.c -o expr_parser
./expr_parser
a = b * c + d;
t0 = b * c
t1 = t0 + d
a = t1
```

<b>Implementation</b>	
<b>Output/Signature</b>	

**RESULT:**

Thus the process effectively tokenizes the input, parses it according to defined grammar rules, and generates the corresponding Three-Address Code, facilitating further compilation or interpretation stages.

**EXP NO: 09**

**DATE:**

**DEVELOP THE BACK-END OF A COMPILER THAT TAKES THREE-ADDRESS CODE (TAC) AS INPUT AND GENERATES CORRESPONDING 8086 ASSEMBLY LANGUAGE CODE AS OUTPUT.**

**AIM:**

To design and implement the back-end of a compiler that takes three-address code (TAC) as input and produces 8086 assembly language code as output. The three-address code is an intermediate representation used by compilers to break down expressions and operations, while the 8086 assembly code is a machine-level representation of the program that can be executed by a processor.

**ALGORITHM:**

1. Parse the Three-Address Code (TAC):

**Input:** Three-Address Code, which is an intermediate representation. For example:

```
t0 = b + c
t1 = t0 * d
a = t1
```

**Output:** 8086 assembly language code. For example:

```
MOV AX, [b]      ; Load b into AX
ADD AX, [c]      ; Add c to AX
MOV [t0], AX     ; Store result in t0
```

---

2. Process Each TAC Instruction:

**1. Initialize Registers:**

- Set up the registers in 8086 assembly (e.g., AX, BX, CX, etc.) for storing intermediate results and final outputs.
- Maintain a temporary register counter for naming temporary variables in TAC (e.g., t0, t1).

**2. For each TAC instruction, based on its operation:**

- Identify the components: operands and operator.
  - Choose an appropriate register (AX, BX, etc.) for storing intermediate results.
  - If the operation involves multiple operands or temporary variables, map them to registers.
- 

3. Translating TAC to 8086 Assembly:

- **Addition/Subtraction (e.g.,  $t0 = b + c$ ):**
  - Load operands into registers and perform the operation:



```
MOV AX, [b]      ; Load b into AX
ADD AX, [c]      ; Add c to AX
MOV [t0], AX     ; Store result in t0
```

- **Multiplication (e.g.,  $t1 = t0 * d$ ):**

- Load operands into registers and perform the operation:

```
MOV AX, [t0]     ; Load t0 into AX
MOV BX, [d]      ; Load d into BX
MUL BX           ; Multiply AX by BX (result in AX)
MOV [t1], AX     ; Store result in t1
```

- **Assignment (e.g.,  $a = t1$ ):**

- Move the value from a temporary variable to the target variable:

```
MOV [a], [t1]    ; Move value of t1 into a
```

- **Division (e.g.,  $t2 = b / c$ ):**

- Division is a bit more complex due to the 8086's limitations with the DIV instruction. For example, the result might need to be stored in AX or DX:AX (if it's a 32-bit result):

```
MOV AX, [b]      ; Load b into AX
MOV DX, 0        ; Clear DX (important for division)
MOV BX, [c]      ; Load c into BX
DIV BX           ; AX = AX / BX (quotient in AX, remainder in DX)
MOV [t2], AX     ; Store quotient in t2
```

#### 4. Manage Memory and Registers:

- **Variables:** Variables like a, b, c are stored in memory, so you will use memory addressing modes such as [variable\_name] to access them.
- **Temporary Variables:** Temporary variables like t0, t1, t2, etc., are stored in registers (AX, BX, etc.) or memory if there are more variables than registers available.

---

#### 5. Handle Control Flow (Optional):

If the TAC contains control structures (such as loops, if-else statements, or function calls), you will need to generate labels and jump instructions in 8086 assembly.

- **If Statements:** For example, if  $(x > 0)$  {  $y = 1$ ; } could generate:

```
MOV AX, [x]
CMP AX, 0
JG positive_case ; Jump if greater
JMP end_if
```

### PROGRAM:

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

#define MAX_LINES 100
#define MAX_LEN 100

int main() {
    char tac[MAX_LINES][MAX_LEN];
    int count = 0;

    printf("Enter TAC instructions (Ctrl+Z to stop):\n");

    // Read all input first
    while (fgets(tac[count], sizeof(tac[count]), stdin)) {
        tac[count][strcspn(tac[count], "\n")] = '\0'; // Remove newline
        count++;
    }

    printf("\n--- 8086 Assembly Output ---\n");

    // Process each line after input is complete
    for (int i = 0; i < count; i++) {
        char lhs[20], op1[20], op2[20], op;
        if (sscanf(tac[i], "%s = %s %c %s", lhs, op1, &op, op2) == 4) {
            if (op == '+') {
                printf("MOV AX, [%s]\n", op1);
                printf("ADD AX, [%s]\n", op2);
                printf("MOV [%s], AX\n\n", lhs);
            } else if (op == '-') {
                printf("MOV AX, [%s]\n", op1);
                printf("SUB AX, [%s]\n", op2);
                printf("MOV [%s], AX\n\n", lhs);
            } else if (op == '*') {
                printf("MOV AX, [%s]\n", op1);
                printf("MOV BX, [%s]\n", op2);
                printf("MUL BX\n");
                printf("MOV [%s], AX\n\n", lhs);
            } else if (op == '/') {
                printf("MOV AX, [%s]\n", op1);
                printf("MOV DX, 0\n");
                printf("MOV BX, [%s]\n", op2);
                printf("DIV BX\n");
                printf("MOV [%s], AX\n\n", lhs);
            }
        } else if (sscanf(tac[i], "%s = %s", lhs, op1) == 2) {
            printf("MOV AX, [%s]\n", op1);
        }
    }
}
```

```
        printf("MOV [%s], AX\n\n", lhs);
    } else {
        printf("; Unsupported TAC format: %s\n\n", tac[i]);
    }
}

return 0;
}
```

## OUTPUT :

```
MOV AX, [b]
ADD AX, [c]
MOV [t0], AX

MOV AX, [t0]
MOV BX, [d]
MUL BX
MOV [t1], AX

MOV AX, [t1]
MOV [a], AX
```

<b>Implementation</b>	
<b>Output/Signature</b>	

## RESULT:

Thus the above example provides a foundational approach to converting TAC to 8086 assembly using C. For a complete compiler back-end, you would need to handle additional aspects such as register allocation, memory management, and more complex control flow constructs.

## STUDY OF CODE OPTIMIZATION

### CODE OPTIMIZATION:

The process of code optimization involves

- Eliminating the unwanted code lines
- Rearranging the statements of the code

### CODE OPTIMIZATION TECHNIQUES:

#### 1. Compile Time Evaluation

Two techniques that falls under compile time evaluation are-

##### A) Constant Folding

In this technique,

- As the name suggests, it involves folding the constants.
- The expressions that contain the operands having constant values at compile time are evaluated.
- Those expressions are then replaced with their respective results.

**Example:**

$$\text{Circumference of Circle} = (22/7) \times \text{Diameter}$$

Here,

- This technique evaluates the expression  $22/7$  at compile time.
- The expression is then replaced with its result 3.14.
- This saves the time at run time.

##### B) Constant Propagation

In this technique,

- If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.
- The condition is that the value of variable must not get alter in between.

**Example:**

$$\text{pi} = 3.14$$

$$\text{radius} = 10$$

$$\text{Area of circle} = \text{pi} \times \text{radius} \times \text{radius}$$

Here,

- This technique substitutes the value of variables 'pi' and 'radius' at compile time.
- It then evaluates the expression  $3.14 \times 10 \times 10$ .
- The expression is then replaced with its result 314.
- This saves the time at run time.

## 2. Common Sub-Expression Elimination

The expression that has been already computed before and appears again in the code for computation is called as **Common Sub-Expression**.

In this technique,

- As the name suggests, it involves eliminating the common sub expressions.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

**Example:**

Code Before Optimization	Code After Optimization
$S1 = 4 \times i$ $S2 = a[S1]$ $S3 = 4 \times j$ $S4 = 4 \times i$ // <b>Redundant Expression</b> $S5 = n$ $S6 = b[S4] + S5$	$S1 = 4 \times i$ $S2 = a[S1]$ $S3 = 4 \times j$ $S5 = n$ $S6 = b[S1] + S5$

## 3. Code Movement

In this technique,

- As the name suggests, it involves movement of the code.
- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
- Such a code unnecessarily gets execute again and again with each iteration of the loop.
- This leads to the wastage of time at run time.

**Example:**

Code Before Optimization	Code After Optimization
<pre>for ( int j = 0 ; j &lt; n ; j ++ ) {     x = y + z ; }</pre>	<pre>x = y + z ; for ( int j = 0 ; j &lt; n ; j ++ ) { }</pre>

<pre>a[j] = 6 x j; }</pre>	<pre>a[j] = 6 x j; }</pre>
----------------------------	----------------------------

#### 4. Dead Code Elimination

In this technique,

- As the name suggests, it involves eliminating the dead code.
- The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

**Example:**

Code Before Optimization	Code After Optimization
<pre>i = 0 ; if (i == 1) { a = x + 5 ; }</pre>	<pre>i = 0 ;</pre>

#### 5. Strength Reduction

In this technique,

- As the name suggests, it involves reducing the strength of expressions.
- This technique replaces the expensive and costly operators with the simple and cheaper ones.

**Example:**

Code Before Optimization	Code After Optimization
<pre>B = A x 2</pre>	<pre>B = A + A</pre>

Here,

- The expression “A x 2” is replaced with the expression “A + A”.

- T her than that of addition operator.  
h  
i  
s  
i  
s  
b  
e  
c  
a  
u  
s  
e  
t  
h  
e  
c  
o  
s  
t  
o  
f  
m  
u  
l  
t  
i  
p  
l  
i  
c  
a  
t  
i  
o  
n  
o  
p  
e  
r  
a  
t  
o  
r  
i  
s  
h  
i  
g

**EXP NO : 10**

**DATE :**

**GENERATE THREE ADDRESS CODES FOR A GIVEN EXPRESSION  
(ARITHMETIC EXPRESSION, FLOW OF CONTROL)**

**AIM:**

The aim is to generate Three-Address Code (TAC) for a given arithmetic expression and flow of control (e.g., if-else, loops). TAC is an intermediate representation used in compilers to simplify the task of code generation. It consists of simple instructions that make it easier to translate into machine-level code.

For example, for an arithmetic expression  $a = b + c * d$ , the TAC would break it down into simpler steps, using temporary variables to hold intermediate results.

**ALGORITHM:**

- The expression is read from the file using a file pointer
- Each string is read and the total no. of strings in the file is calculated.
- Each string is compared with an operator; if any operator is seen then the previous string and next string are concatenated and stored in a first temporary value and the three address code expression is printed
- Suppose if another operand is seen then the first temporary value is concatenated to the next string using the operator and the expression is printed.
- The final temporary value is replaced to the left operand value.

**PROGRAM:**

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

int tempCount = 1;

// Function to generate a temporary variable name
void newTemp(char *temp) {
    sprintf(temp, "t%d", tempCount++);
}

// Function to generate TAC for arithmetic expressions like a = b + c * d
void generateTACForExpression(char expr[]) {
    char lhs[20], rhs[100];
    char op1[20], op2[20], result[20], operator;
    char temp1[10], temp2[10];
    int i = 0, j = 0, len = strlen(expr);

    // Split LHS and RHS
```



```

char *equal = strchr(expr, '=');
if (!equal) {
    printf("; Invalid expression: %s\n", expr);
    return;
}

strncpy(lhs, expr, equal - expr);
lhs[equal - expr] = '\0';
strcpy(rhs, equal + 1);

// Remove spaces
char rhs_clean[100];
for (int k = 0; rhs[k]; k++) {
    if (!isspace(rhs[k])) rhs_clean[j++] = rhs[k];
}
rhs_clean[j] = '\0';

// Handle binary operators: +, -, *, /
// We'll scan from right to left to handle precedence (e.g., * before +)
char *opPtr = NULL;
if ((opPtr = strrchr(rhs_clean, '*')) ||
    (opPtr = strrchr(rhs_clean, '/')) ||
    (opPtr = strrchr(rhs_clean, '+')) ||
    (opPtr = strrchr(rhs_clean, '-'))) {

    operator = *opPtr;
    *opPtr = '\0';
    strcpy(op1, rhs_clean);
    strcpy(op2, opPtr + 1);

    newTemp(temp1);
    printf("%s = %s %c %s\n", temp1, op1, operator, op2);
    printf("%s = %s\n", lhs, temp1);
} else {
    // Just direct assignment
    printf("%s = %s\n", lhs, rhs_clean);
}
}

// Function to generate TAC for if/while (very simple form)
void generateTACForControl(char line[]) {
    char cond[50], label1[10], label2[10];
    static int labelCount = 1;

    if (strstr(line, "if") != NULL) {
        sscanf(line, "if (%[^)])", cond);
        sprintf(label1, "L%d", labelCount++);
        sprintf(label2, "L%d", labelCount++);
        printf("if not %s goto %s\n", cond, label1);
    }
}

```

```

    printf(" ; [true block statements]\n");
    printf("goto %s\n", label2);
    printf("%s:\n", label1);
    printf(" ; [else block statements]\n");
    printf("%s:\n", label2);
} else if (strstr(line, "while") != NULL) {
    sscanf(line, "while (%[^)])", cond);
    sprintf(label1, "L%d", labelCount++);
    sprintf(label2, "L%d", labelCount++);
    printf("%s:\n", label1);
    printf("if not %s goto %s\n", cond, label2);
    printf(" ; [loop body statements]\n");
    printf("goto %s\n", label1);
    printf("%s:\n", label2);
} else {
    printf("; Unknown control statement: %s\n", line);
}
}

int main() {
    FILE *fp;
    char line[100];

    fp = fopen("input.txt", "r");
    if (fp == NULL) {
        printf("Error opening input.txt\n");
        return 1;
    }

    printf("--- Three Address Code ---\n");

    while (fgets(line, sizeof(line), fp)) {
        // Remove newline
        line[strcspn(line, "\n")] = '\0';

        if (strstr(line, "if") || strstr(line, "while")) {
            generateTACForControl(line);
        } else {
            generateTACForExpression(line);
        }
    }

    fclose(fp);
    return 0;
}

```

### OUTPUT :

```
* TAC for arithmetic expression:
  t1 = b + c
  a = t1

TAC for if-else statement:
if a < b goto L1
goto L2
L1: x = 1
goto L3
L2: x = 2
L3:

TAC for while loop:
L1: if a >= b goto L2
x = x + 1
goto L1
L2:
```

Implementation	
Output/Signature	

### RESULT:

Thus the above program is the simplified example and a complete implementation and it would need to handle more complex expressions, nested control structures, and ensure proper parsing of the input.

**EXP NO : 11**

**DATE :**

**IMPLEMENT CODE OPTIMIZATION TECHNIQUES LIKE DEAD CODE AND  
COMMON EXPRESSION ELIMINATION**

**AIM:**

The aim is to implement code optimization techniques such as Dead Code Elimination (DCE) and Common Subexpression Elimination (CSE) on an intermediate representation of a program (such as Three-Address Code (TAC)). These optimization techniques help reduce the size of the code, improve runtime performance, and eliminate redundant computations during the compilation process.

**ALGORITHM:**

- Start
- Create the input file which contains three address code.
- Open the file in read mode.
- If the file pointer returns NULL, exit the program else go to 5.
- Scan the input symbol from left to right.
- Store the first expression in a string.
- Compare the string with the other expressions in the file.
- If there is a match, remove the expression from the input file.
- Perform these steps 5-8 for all the input symbols in the file.
- Scan the input symbol from the file from left to right.
- Get the operand before the operator from the three address code.
- Check whether the operand is used in any other expression in the three address code.
- If the operand is not used, then eliminate the complete expression from the three address code else go to 14.
- Perform steps 11 to 13 for all the operands in the three address code till end of the file is reached.
- Stop.

**PROGRAM:**

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

#define MAX 100

typedef struct {
    char lhs[20], rhs[50];
} TAC;
```

```

int isUsed(TAC tac[], int total, char *var, int current) {
    for (int i = current + 1; i < total; i++) {
        if (strstr(tac[i].rhs, var)) return 1;
    }
    return 0;
}

void replaceVar(char *src, char *oldVar, char *newVar) {
    char buffer[100] = "";
    char *pos = src, *match;
    while ((match = strstr(pos, oldVar)) != NULL) {
        strncat(buffer, pos, match - pos);
        strcat(buffer, newVar);
        pos = match + strlen(oldVar);
    }
    strcat(buffer, pos);
    strcpy(src, buffer);
}

int main() {
    FILE *fp;
    TAC tac[MAX];
    char line[100], *lhs, *rhs;
    int count = 0;

    // Open input file
    fp = fopen("input.txt", "r");
    if (!fp) {
        printf("Error: Could not open 'input.txt'\n");
        return 1;
    }

    // Read input file
    while (fgets(line, sizeof(line), fp)) {
        line[strcspn(line, "\n")] = 0;
        lhs = strtok(line, "=");
        rhs = strtok(NULL, "\n");
        if (lhs && rhs) {
            strcpy(tac[count].lhs, lhs);
            strcpy(tac[count].rhs, rhs);
            count++;
        }
    }
    fclose(fp);

    // Step 1: Common Subexpression Elimination (CSE)
    for (int i = 0; i < count; i++) {
        for (int j = i + 1; j < count; j++) {
            if (strcmp(tac[i].rhs, tac[j].rhs) == 0) {

```

```

        replaceVar(tac[j + 1].rhs, tac[j].lhs, tac[i].lhs);
        strcpy(tac[j].lhs, "");
        strcpy(tac[j].rhs, "");
    }
}

// Step 2: Copy Propagation
for (int i = 0; i < count; i++) {
    if (strchr(tac[i].rhs, '+') == NULL && strchr(tac[i].rhs, '-') == NULL &&
        strchr(tac[i].rhs, '*') == NULL && strchr(tac[i].rhs, '/') == NULL) {
        // rhs is a direct copy
        for (int j = i + 1; j < count; j++) {
            replaceVar(tac[j].rhs, tac[i].lhs, tac[i].rhs);
        }
        // mark line as empty
        strcpy(tac[i].lhs, "");
        strcpy(tac[i].rhs, "");
    }
}

// Step 3: Dead Code Elimination
for (int i = 0; i < count; i++) {
    if (tac[i].lhs[0] != '\0' && !isUsed(tac, count, tac[i].lhs, i)) {
        strcpy(tac[i].lhs, "");
        strcpy(tac[i].rhs, "");
    }
}

// Print Optimized Code
printf("\nOptimized Code:\n -----\n");
for (int i = 0; i < count; i++) {
    if (tac[i].lhs[0] != '\0') {
        printf("%s=%s\n", tac[i].lhs, tac[i].rhs);
    }
}

return 0;
}

```

**OUTPUT :**

```
Optimized Three-Address Code:  
t1 = a + b  
t3 = t1 * c  
t4 = t2 * c
```

<b>Implementation</b>	
<b>Output/Signature</b>	

**RESULT:**

Thus The Above Program To Implement Code Optimization Techniques Like Dead Code And Common Expression Elimination Is Executed And Implemented Successfully.

**EXP NO : 12**

**DATE :**

**IMPLEMENT CODE OPTIMIZATION TECHNIQUES  
COPY PROPAGATION**

**AIM:**

The aim is to implement code optimization techniques like Dead Code Elimination (DCE) and Common Subexpression Elimination (CSE) to improve the efficiency and performance of a program. These techniques are applied to intermediate code (e.g., Three-Address Code or TAC) during the compilation process.

**ALGORITHM:**

- The desired header files are declared.
- The two file pointers are initialized one for reading the C program from the file and one for writing the converted program with constant folding
- The file is read and checked if there are any digits or operands present.
- If there is, then the evaluations are to be computed in switch case and stored.
- Copy the stored data to another file.
- Print the copied data file.

**PROGRAM:**

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

#define MAX 100

typedef struct {
    char lhs[10];
    char op1[10];
    char op[5];
    char op2[10];
} Instruction;

int is_copy_instruction(Instruction *ins) {
    return strcmp(ins->op, "=") == 0 && strlen(ins->op2) == 0;
}

void copy_propagation(Instruction ins[], int count) {
    for (int i = 0; i < count; i++) {
        if (is_copy_instruction(&ins[i])) {
            char from[10], to[10];
            strcpy(to, ins[i].lhs);
            strcpy(from, ins[i].op1);
```



```
        for (int j = i + 1; j < count; j++) {
            if (strcmp(ins[j].op1, to) == 0)
                strcpy(ins[j].op1, from);
            if (strcmp(ins[j].op2, to) == 0)
                strcpy(ins[j].op2, from);
        }
    }
}

int main() {
    FILE *fin = fopen("input.txt", "r");

    if (!fin) {
        printf("Error opening input.txt\n");
        return 1;
    }

    Instruction ins[MAX];
    int count = 0;

    char line[100];
    while (fgets(line, sizeof(line), fin)) {
        // Skip blank lines
        if (strlen(line) <= 1) continue;

        Instruction temp;
        temp.op[0] = '\0';
        temp.op2[0] = '\0';

        int tokens = sscanf(line, "%s = %s %s %s", temp.lhs, temp.op1, temp.op, temp.op2);

        if (tokens == 2) {
            // It's a copy statement like: a = b
            strcpy(temp.op, "=");
            temp.op2[0] = '\0';
        } else if (tokens != 4) {
            printf("Invalid line: %s\n", line);
            continue;
        }

        ins[count++] = temp;
    }

    fclose(fin);

    // Perform copy propagation
    copy_propagation(ins, count);
}
```

```
// Print optimized code
printf("\nOptimized Code (Copy Propagation Only):\n\n");
for (int i = 0; i < count; i++) {
    if (strcmp(ins[i].op, "=") == 0 && strlen(ins[i].op2) == 0)
        printf("%s = %s\n", ins[i].lhs, ins[i].op1);
    else
        printf("%s = %s %s %s\n", ins[i].lhs, ins[i].op1, ins[i].op, ins[i].op2);
}

return 0;
}
```

### OUTPUT :

```
Enter statements (e.g., a = b or c = a + d). Enter 'END' to finish:
A=B+C+D
C=B+S+k
END

Optimized code:
A = B+C+D
C = B+S+k
```

Implementation	
Output/Signature	

### RESULT:

Thus the above to implement code optimization techniques for copy propagation is executed successfully.

## Annexure 1

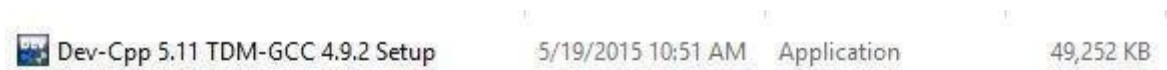
Download editplus from the following link:

[https://drive.google.com/file/d/0B9D4jOdpRzZHNTVraV9rX280R0E/view?resourcekey=0-wKxkIdA7Eqh0j2Jj\\_u87ow](https://drive.google.com/file/d/0B9D4jOdpRzZHNTVraV9rX280R0E/view?resourcekey=0-wKxkIdA7Eqh0j2Jj_u87ow)

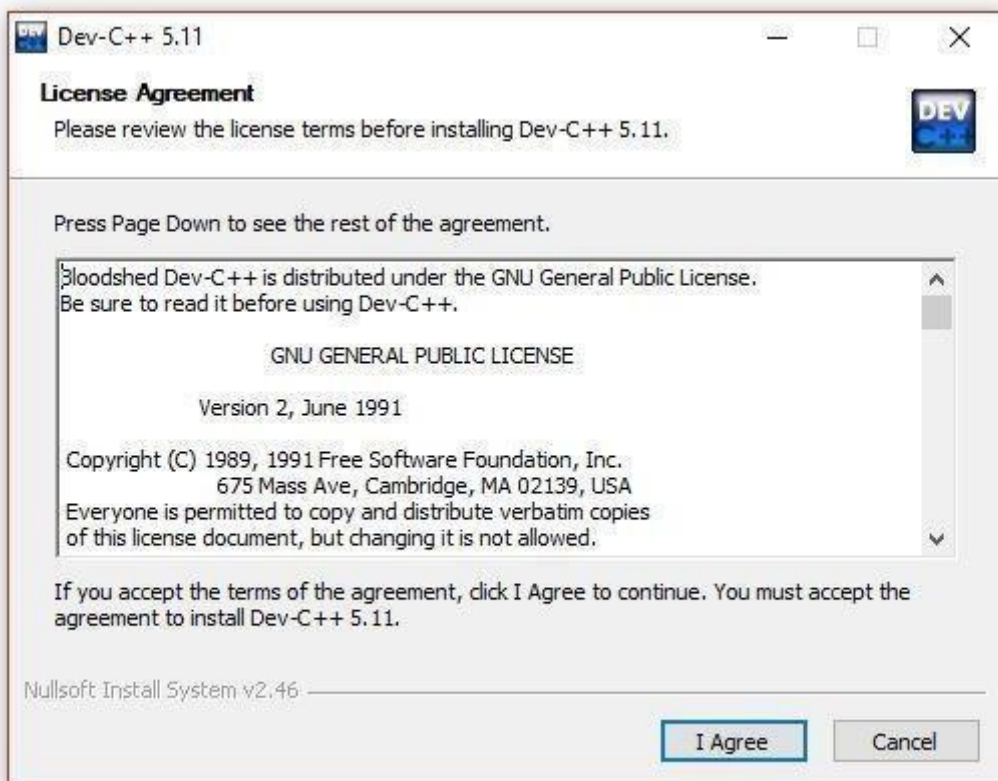
Note: Make sure that C compiler is installed.

*Install the setup:*

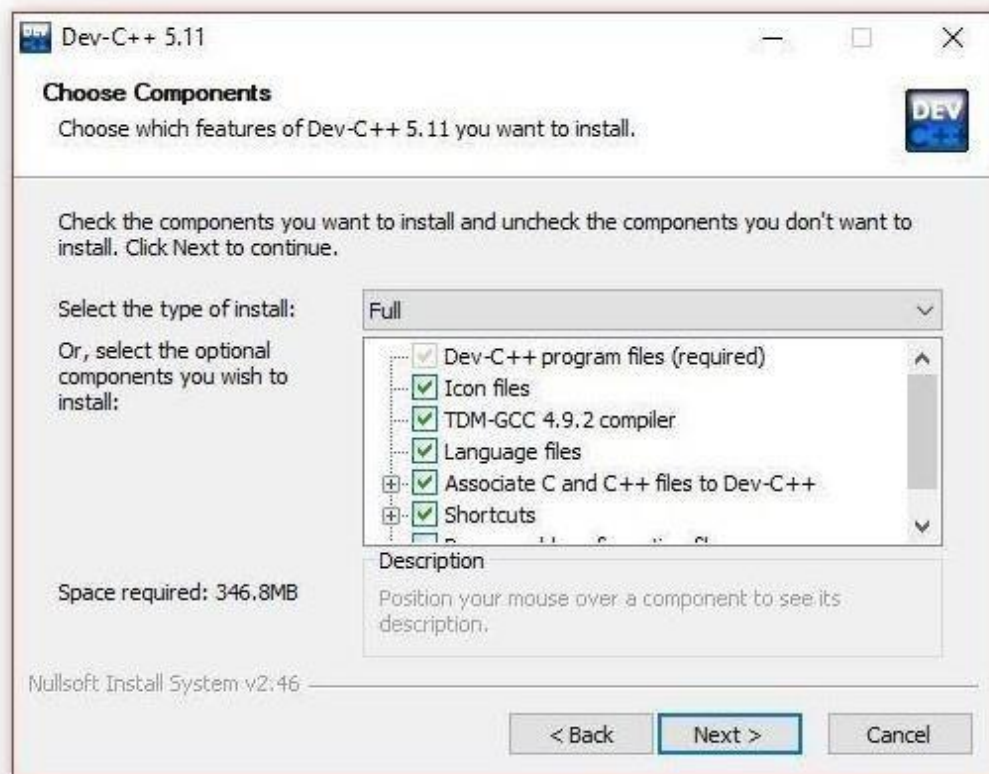
After downloading the setup double click on it, it extracts the package from the setup and asks to select the language. So select the language as per your choice and press the ok button.



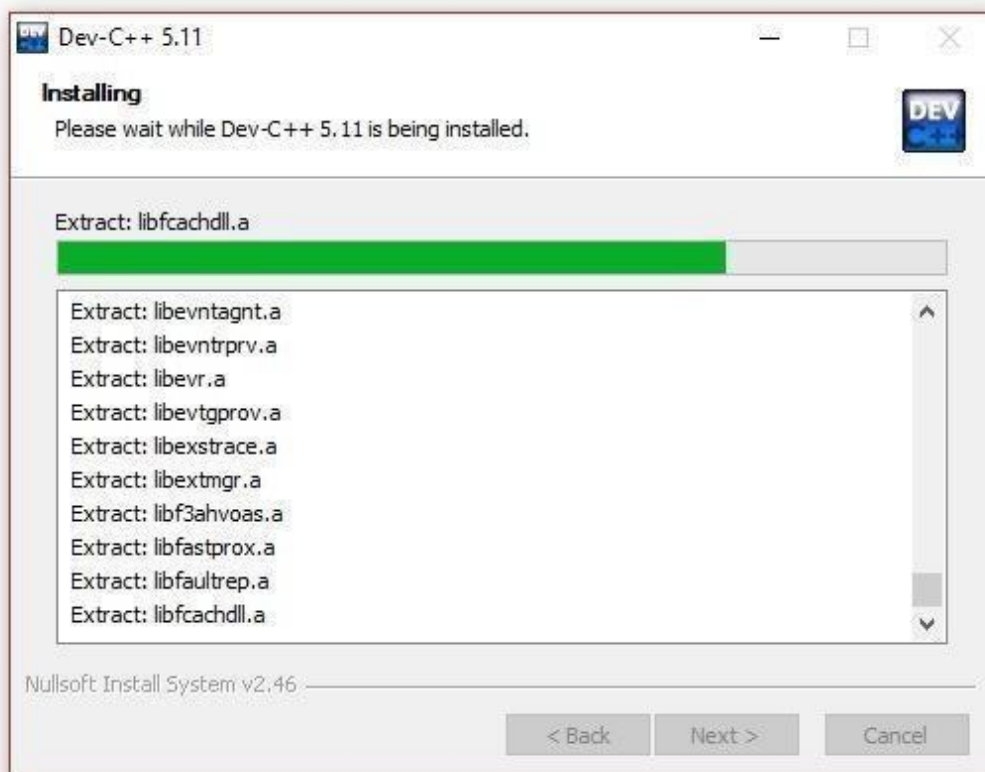
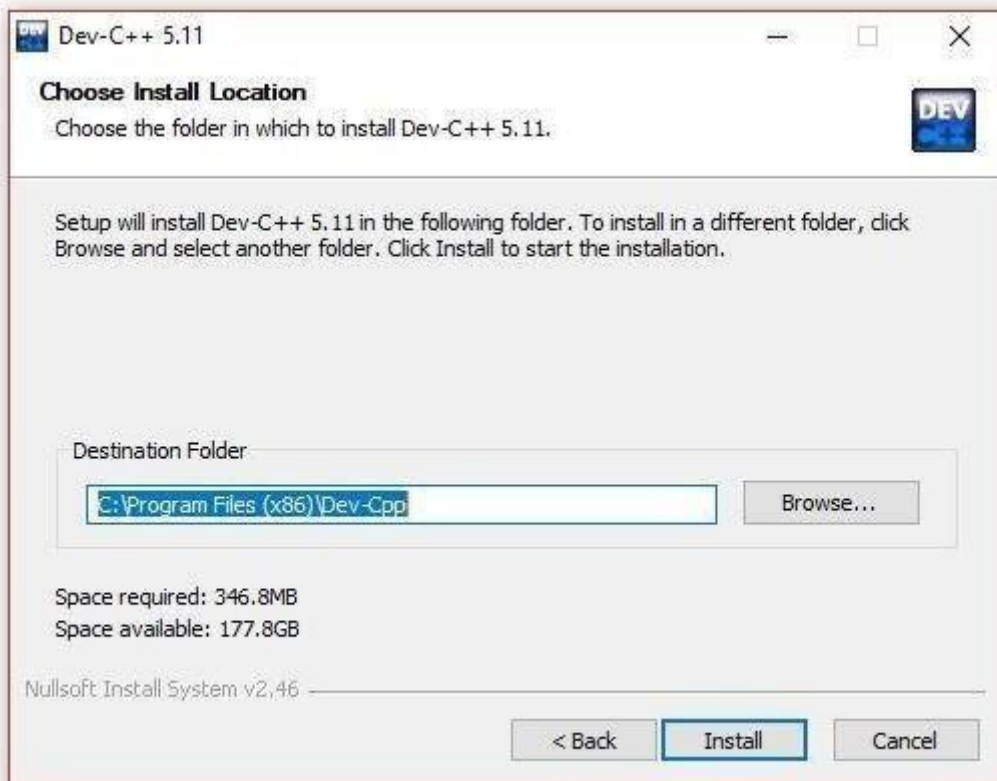
Accept the license agreement.



In the next prompt window, it will ask you to select the package you want to install. Here you don't need to anything just press the Next button.



Select the destination folder (keep it to default) and press the install button. It will take a few minutes.



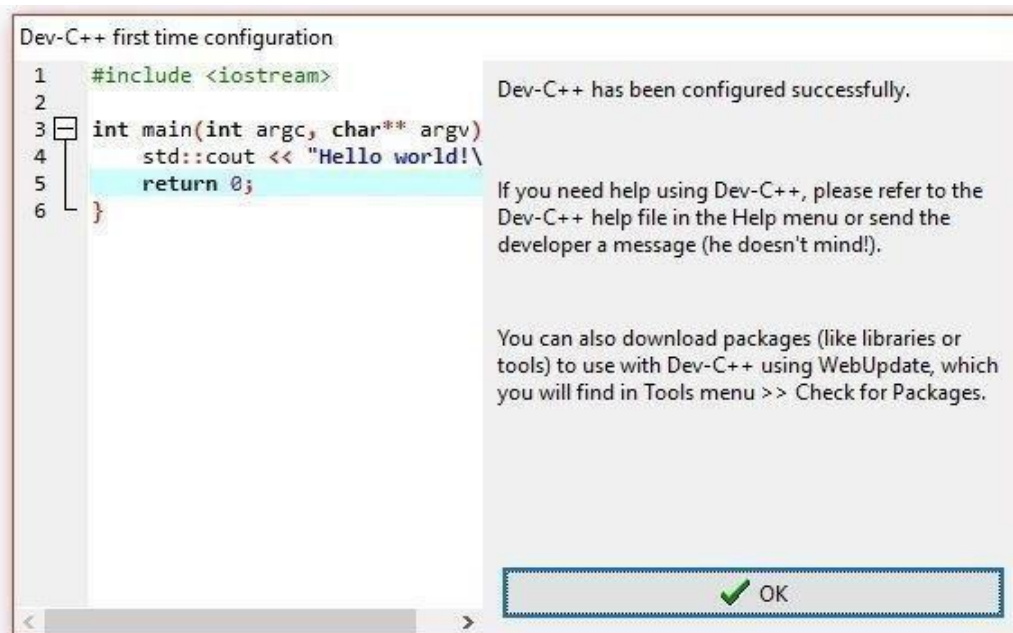
After completing the installation, Dev-C++ will prompt a window with a finish button. Just press the finish button.



After pressing the finish button it asks for language and fonts. If you want to change the language and fonts, then you can change.



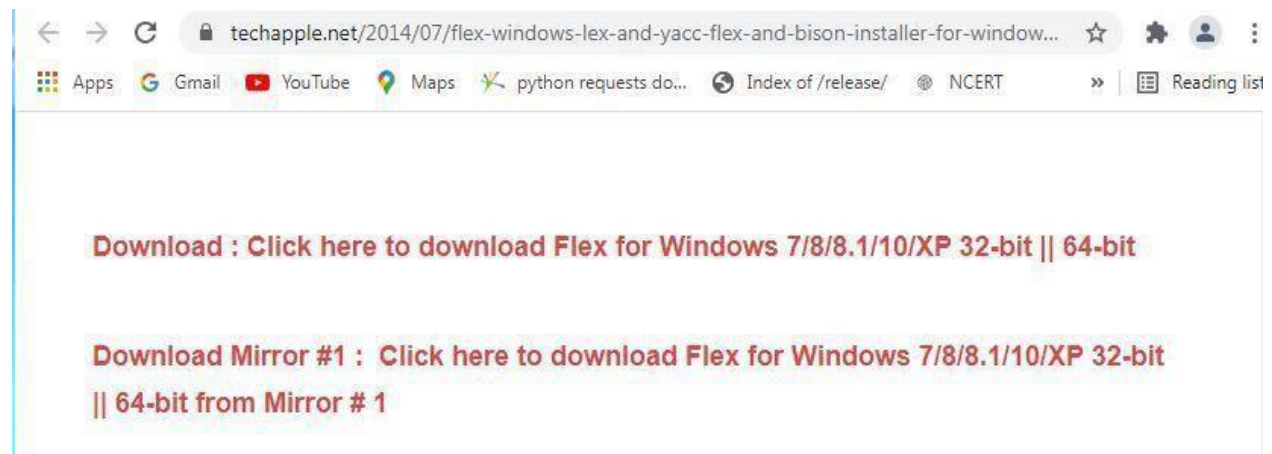




## Flex - Lex and Yacc

Step 1 : Install from the above URL

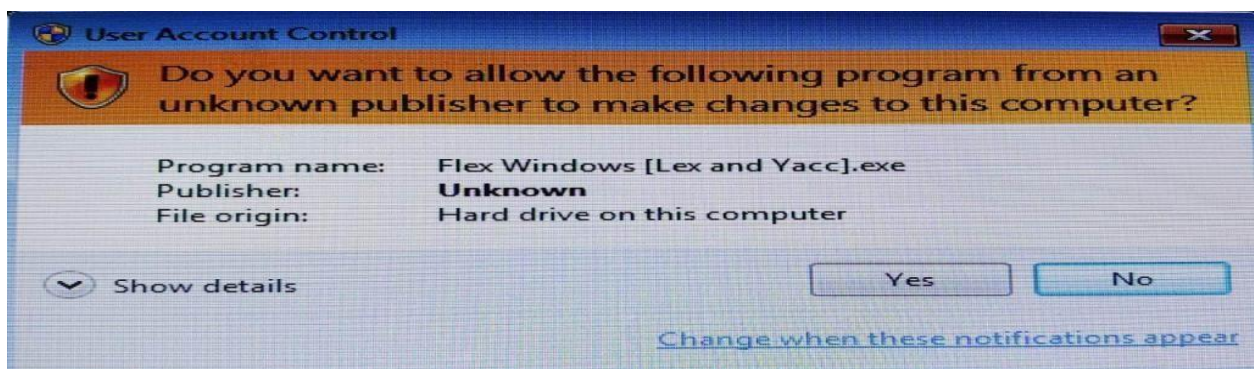
Step 2: On this page, all the features and minimum requirement of the system to install flex is given. Here the download link of the flex program for Windows XP, 7, 8, etc is given. Click on the download link, downloading of the executable file will start shortly. It is a small 30.19 MB file that will hardly take a minute.



Step 3: Now check for the executable file in downloads in your system and run it.



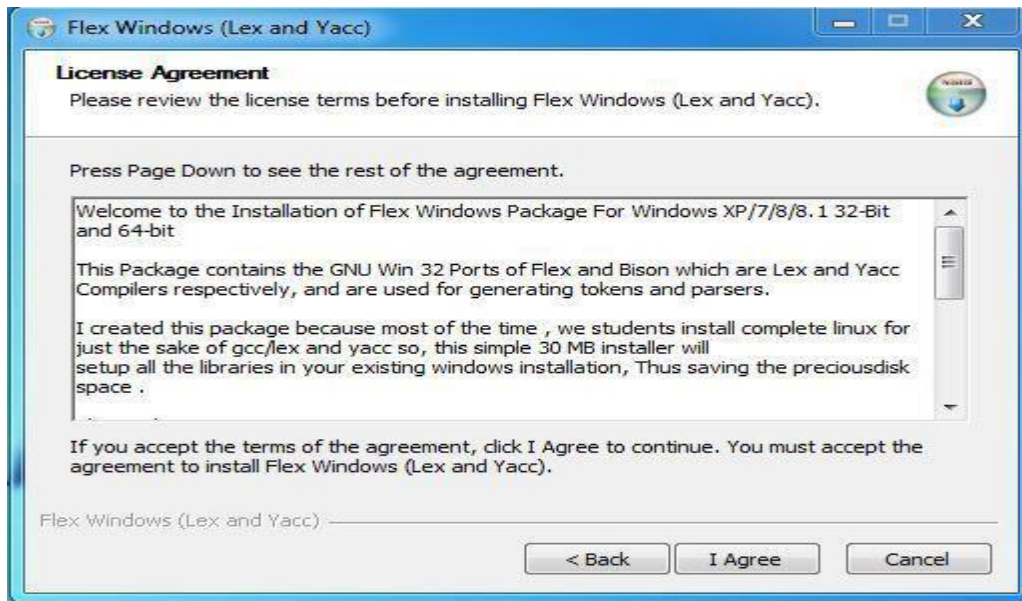
Step 4: It will prompt confirmation to make changes to your system. Click on Yes.



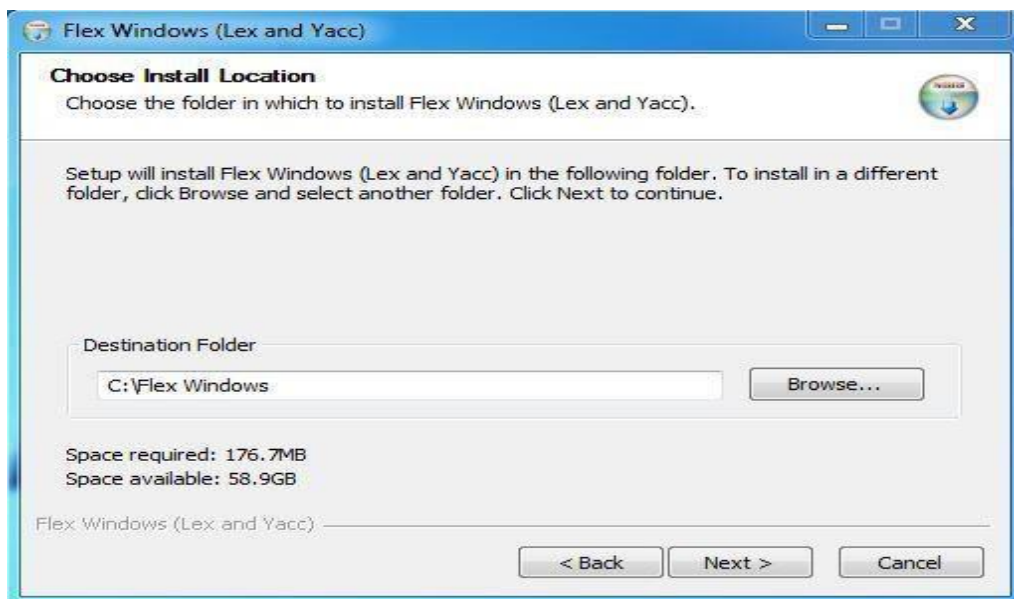
Step 5: Setup screen will appear, click on Next.



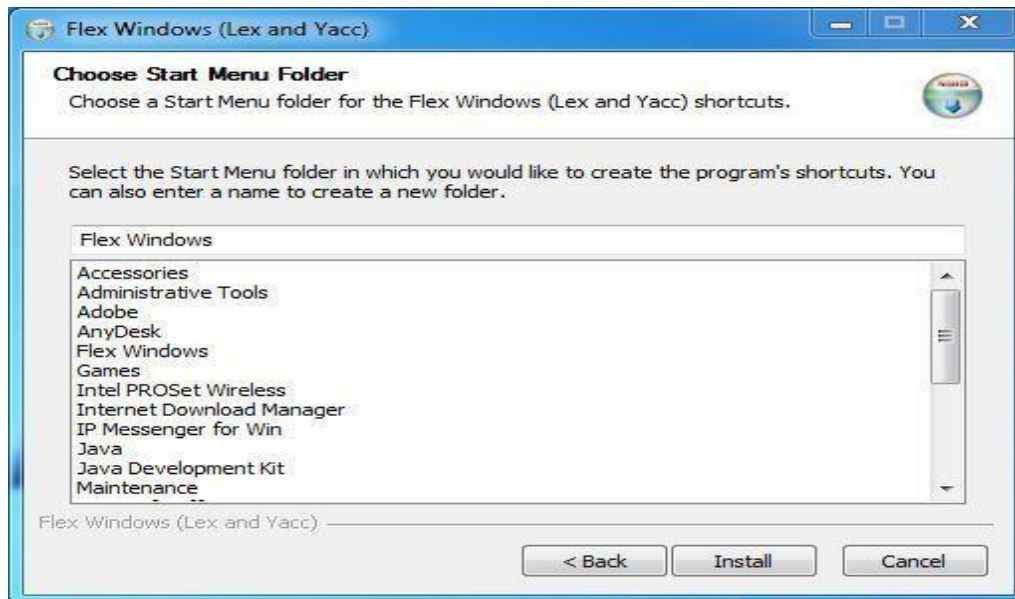
Step 6: The next screen will be of License Agreement, click on I Agree.



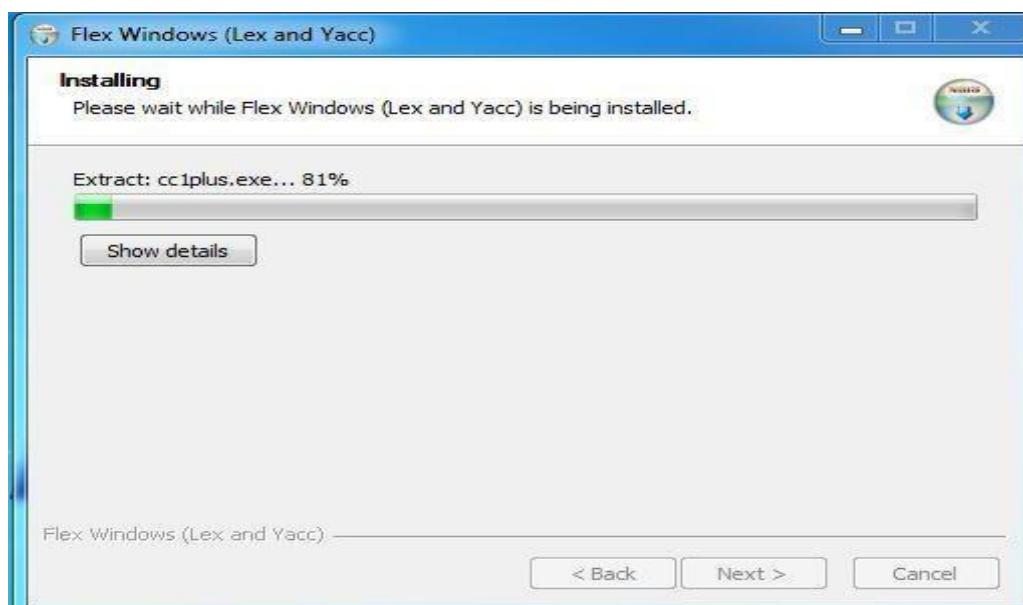
Step 7: The next screen will be of installing location so choose the drive which will have sufficient memory space for installation. It needed only a memory space of 176.7 MB.



Step 8: Next screen will be of choosing the Start menu folder so don't do anything just click on the Next Button.



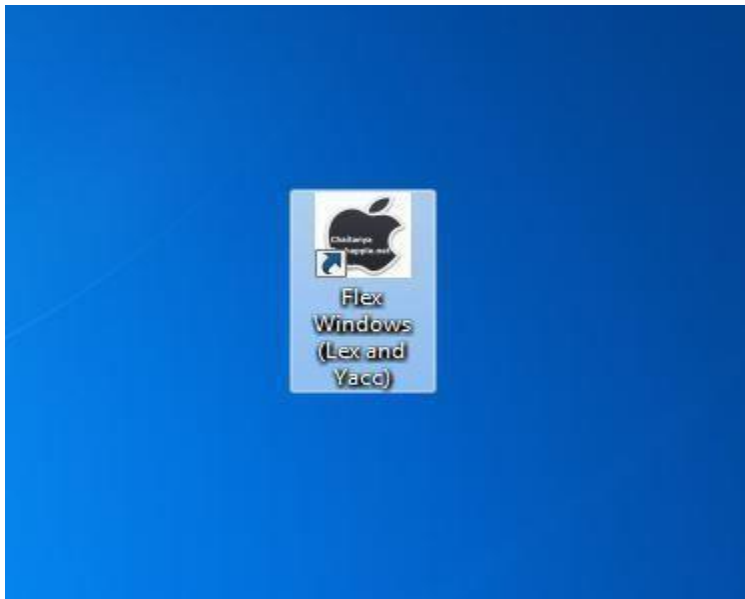
Step 9: After this installation process will start and will hardly take a minute to complete the installation.



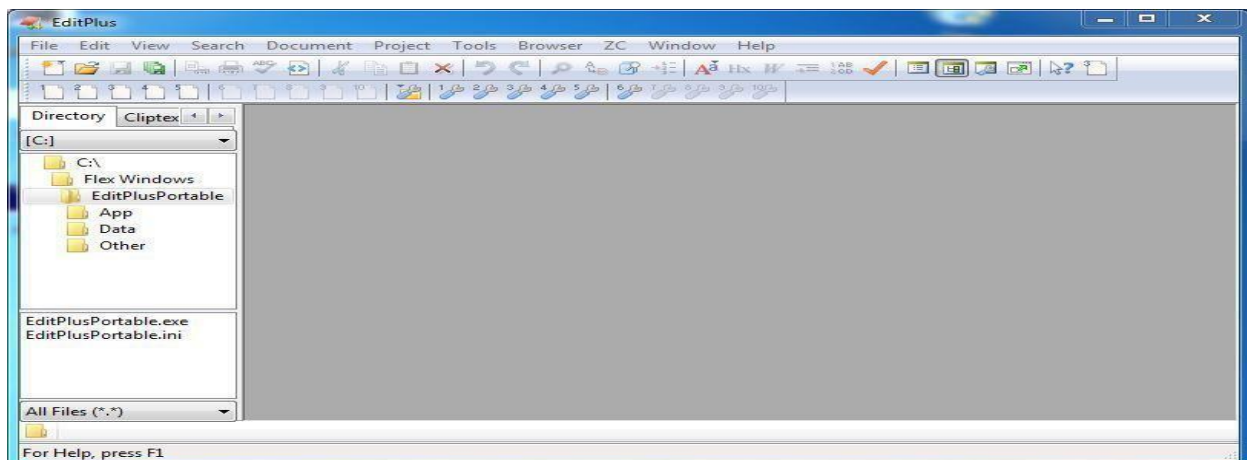
Step 10: Click on Finish after the installation process is complete. Keep the tick mark on the checkbox if you want to run Flex now if not then uncheck it.



Step 11: Flex Windows is successfully installed on the system and an icon is created on the desktop



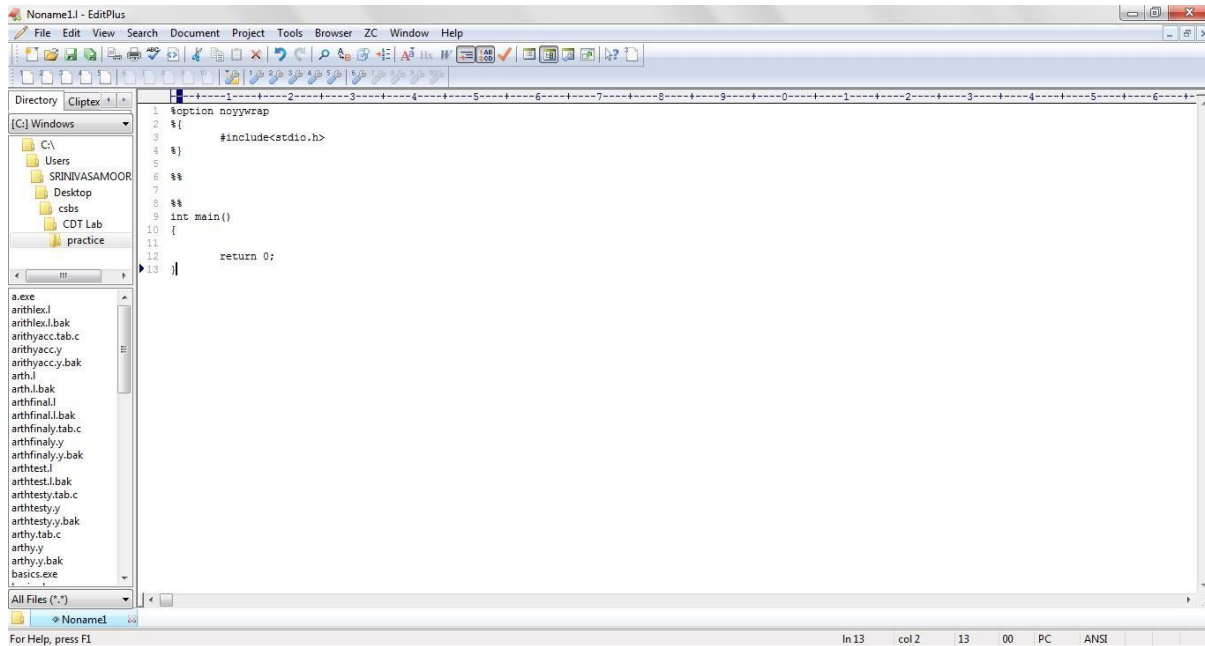
Step 12: Run the software and see the interface.



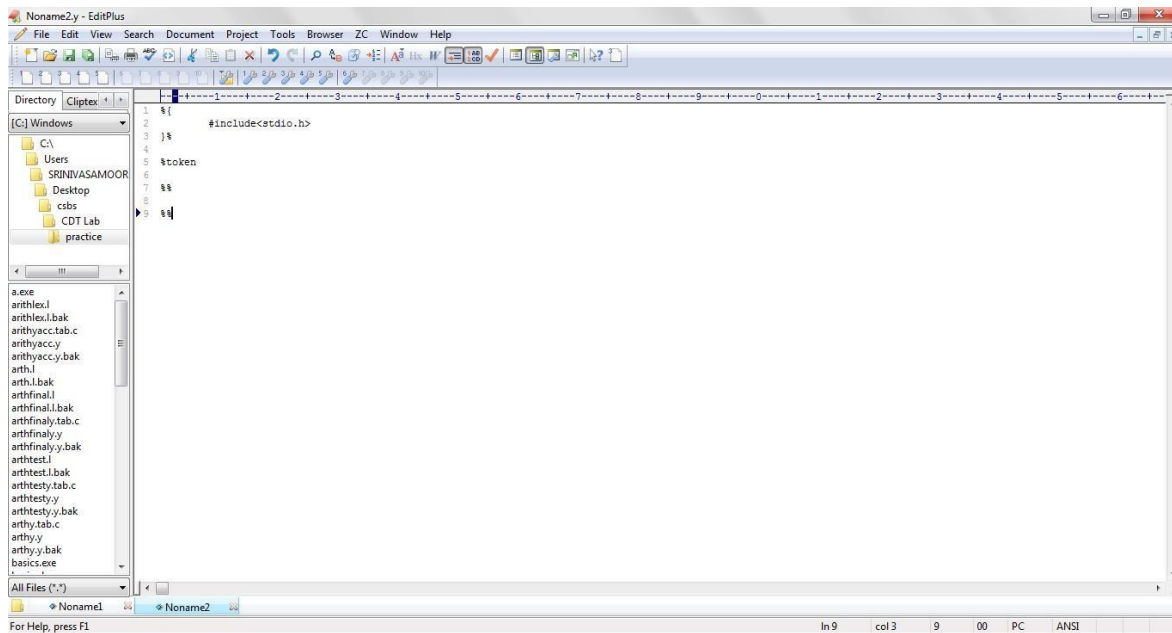
Flex is now installed on the system



To create a new file file>new>lex then the file would be Noname1.l then we can save and use lex file



To create a new yacc file file>new>yacc/bison then the file would be Noname2.y then we can save and use yacc file



Instructions to Run the Code :

### LEX WORKING

```
pro1.l" [New] 0L, 0C written
est69@hdc17001:~$ vim pro1.l
```

Vim filename.l

1. Press i in keyboard then start to type the code

Procedure :

2. After typing your code save as Press ESC :wq

3. lex filename.l or lex filename.c

4. cc lex.yy.c

5. ./a.out

### YACC Working

vim filename.l (For LEX CODE)

vim filename1.y (For YACC CODE)

lex filename.l

Yacc -d filename 1.y

Cc lex.yy.c y.tab.c

./a.out

LOKESHWAR S (220701146)



LOKESHWAR S (220701146)

LOKESHWAR S (220701146)

LOKESHWAR S (220701146)

LOKESHWAR S (220701146)