

**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)**



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

**CURRICULUM AND SYLLABUS**  
**CHOICE BASED CREDIT SYSTEM**  
**B.E. COMPUTER SCIENCE AND ENGINEERING**  
**REGULATION 2019**

**Vision**

To promote highly ethical and innovative computer professionals through excellence in teaching, training and research.

**Mission**

- To produce globally competent professionals, motivated to learn the emerging technologies and to be innovative in solving real world problems.
- To promote research activities amongst the students and the members of faculty that could benefit the society.
- To impart moral and ethical values in their profession.

**PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)**

**PEO 1:** To equip students with essential background in computer science, basic electronics and applied mathematics.

**PEO 2:** To prepare students with fundamental knowledge in programming languages and tools and enable them to develop applications.

**PEO 3:** To encourage the research abilities and innovative project development in the field of networking, security, data mining, web technology, mobile communication and also emerging technologies for the cause of social benefit.

**PEO 4:** To develop professionally ethical individuals enhanced with analytical skills, communication skills and organizing ability to meet industry requirements.

## **PROGRAMME OUTCOMES (POs)**

**PO1: Engineering knowledge:** Apply the knowledge of Mathematics, Science, Engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

**PO2: Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

**PO3: Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

**PO 4: Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

**PO 5: Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

**PO 6: The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

**PO 7: Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

**PO 8: Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

**PO 9: Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

**PO10: Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

**PO11: Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team to manage project and in multidisciplinary environments.

**PO12: Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

## **PROGRAM SPECIFIC OUTCOMES (PSOs)**

A graduate of the Computer Science and Engineering Program will demonstrate

PSO 1: Foundation Skills: Ability to understand, analyze and develop computer programs in the areas related to algorithms, system software, web design, machine learning, data analytics, and networking for efficient design of computer-based systems of varying complexity. Familiarity and practical competence with a broad range of programming language and open source platforms.

PSO 2: Problem-Solving Skills: Ability to apply mathematical methodologies to solve computational task, model real world problem using appropriate data structure and suitable algorithm. To understand the standard practices and strategies in software project development, using open- ended programming environments to deliver a quality product.

PSO 3: Successful Progression: Ability to apply knowledge in various domains to identify research gaps and to provide solution to new ideas, inculcate passion towards higher studies, creating innovative career paths to be an entrepreneur and evolve as an ethically social responsible computer science professional.

Mapping of Program Educational Objectives (PEOs) and Program Outcomes (POs) with correlation levels (1: Slight, 2: Moderate, 3: Substantial, -: No correlation), based on the provided information:

### **PEO and PO Mapping Table**

PEOs/POs	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
PEO1	3	3	2	2	3	-	-	-	2	2	2	2	3	3	2
PEO2	2	2	3	2	3	-	-	-	2	2	2	2	3	3	2
PEO3	2	2	3	3	3	3	3	2	2	2	2	2	3	3	3
PEO4	2	2	2	2	2	2	2	3	3	3	3	2	2	2	3

Subject Code	Subject Name ( Lab oriented Theory Course)	Category	L	T	P	C
CSI9641	COMPILER DESIGN	PC	3	0	2	4

**Objectives:**

- Learn the various phases of a Compiler.
- Demonstrate the compiler construction tools.
- Analyze the various parsing techniques and different levels of translation.
- Understand intermediate code generation and run-time environment.
- Learn how to optimize and effectively incorporate in machine code generation.

<b>UNIT-I</b>	<b>INTRODUCTION TO COMPILERS</b>	<b>5</b>
Translators-Compilation and Interpretation-Language processors-The Structure of a Compiler-Compiler Construction Tools-Evolution of Programming Languages-Programming Language basics.		
<b>UNIT-II</b>	<b>LEXICAL ANALYSIS</b>	<b>9</b>
Role of the Lexical Analyzer-Input Buffering - Specification of Tokens - Recognition of Tokens - Finite Automata-NFA-DFA - Converting Regular Expression to Automata- Design of a Lexical Analyzer Generator- LEX..		
<b>UNIT-III</b>	<b>SYNTAX ANALYSIS</b>	<b>12</b>
Role of the Parser-Context Free Grammars-Ambiguity-Left Recursion-Left Factoring-Top Down Parsing-Recursive Descent Parsing-LL(1)Grammars-Non recursive Predictive Parsing-Error Recovery in Predictive Parsing-Bottom up Parsing-Shift Reduce Parsing-LR Parsing-SLR-Canonical LR-LALR Parser-YACC..		
<b>UNIT-IV</b>	<b>INTERMEDIATE CODE GENERATION</b>	<b>10</b>
Syntax directed Definitions-Construction of Syntax Tree- DAG - Three Address Code -Types and declarations-ControlFlow - Backpatching. Storage Organization-Stack allocation of space- Heap Management.		
<b>UNIT-V</b>	<b>CODE OPTIMIZATION AND CODE GENERATION</b>	<b>9</b>
Basic Blocks and Flow graphs- Optimization of Basic Blocks- Peephole Optimization-Principal sources of Optimization-Global Data Flow Analysis-Code Generation-Issues in Design of a Code Generator-A Simple Code Generator Algorithm.		
<b>Contact Hours</b>		<b>: 45</b>

**List of Experiments**

1	Develop a lexical analyzer to recognize tokens in C. (Ex. identifiers, constants, operators, keywords etc.)			
2	Design a Desk Calculator using LEX.			
3	Recognize an arithmetic expression using LEX and YACC.			
4	Evaluate expression that takes digits, *, + using YACC.			
5	Generate Three address codes for a given expression (arithmetic expression, flow of control).			
6	Implement Code Optimization Techniques like copy propagation, dead code elimination, Common sub expression elimination.			
7	Generate Target Code (Assembly language) for the given set of Three Address Code.			
		Contact Hours	:	30
		Total Contact Hours	:	75

**Course Outcomes:**

On completion of the course, the students will be able to

- Demonstrate the functioning of a Compiler.
- Analyse the local and global impact of translators.
- Develop language specifications using context free grammars (CFG).
- Apply the various optimization techniques.
- Generate a target code.

Text Book(s):	
1	Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, "Compilers – Principles, Techniques and Tools", Second Edition, Pearson Education, 2007.

Reference Book(s) / Web link(s):	
1	Randy Allen, Ken Kennedy, "Optimizing Compilers for Modern Architectures: A Dependence-based Approach", First Edition, Morgan Kaufmann Publishers, 2002.
2	Steven S. Muchnick, "Advanced Compiler Design and Implementation", First Edition, Morgan Kaufmann publishers, 2003.

3	D. Grune, H.E. Bal, C.J.H. Jacobs, K.G. Langendoen, "Modern Compiler Design", Wiley, 2008
4	Allen I. Holub, "Compiler Design in C", Prentice Hall of India, 2003.

**CO - PO – PSO matrices of course**

PO/PSO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO10	PO11	PO12	PSO 1	PSO 2	PSO 3
19641.0	-	-	1	-	1	-	-	-	-	-	-	-	1	-	-
19641.0	-	-	2	-	2	-	-	-	-	-	-	-	2	-	-
19641.0	-	-	2	-	2	-	-	-	-	-	-	-	2	-	-
19641.0	-	-	2	-	2	-	-	-	-	-	-	-	2	-	-
19641.0	-	-	3	-	2	-	-	-	-	-	-	-	2	-	-
Average lapping	-	-	1	-	1	-	-	-	-	-	-	-	1	-	-

Note: Enter correlation levels 1, 2 or 3 as defined below:

1: Slight (Low) 2: Moderate (Medium) 3: Substantial (High) If there is no correlation, put "-"

## INDEX

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

**Hardware requirements:PC**

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

**Reference annexure 1 for installation**

**EXP NO:**

**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 <ctype.h> library.

**ALGORITHM:**

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

**PROGRAM:**

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
int main() {
    char str[100];
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    size_t len = strlen(str);
    if (len > 0 && str[len - 1] == '\n') {
        str[len - 1] = '\0';
    }
    for (int i = 0; str[i] != '\0'; i++) {
        str[i] = toupper((unsigned char)str[i]);
    }
    printf("Uppercase String: %s\n", str);
    return 0;
}
```

**OUTPUT:**

main.c		Output
<pre>1 // Online C compiler to run C program online 2 #include &lt;stdio.h&gt; 3 #include &lt;ctype.h&gt; 4 #include &lt;string.h&gt; 5 int main() { 6     char str[100]; 7     printf("Enter a string: "); 8     fgets(str, sizeof(str), stdin); 9     size_t len = strlen(str); 10    if (len &gt; 0 &amp;&amp; str[len - 1] == '\n') { 11        str[len - 1] = '\0'; 12    } 13    for (int i = 0; str[i] != '\0'; i++) { 14        str[i] = toupper((unsigned char)str[i]); 15    } 16    printf("Uppercase String: %s\n", str); 17    return 0; 18 } 19</pre>	<div>Enter a string: hello Uppercase String: HELLO  === Code Execution Successful ===</div>	

## 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:

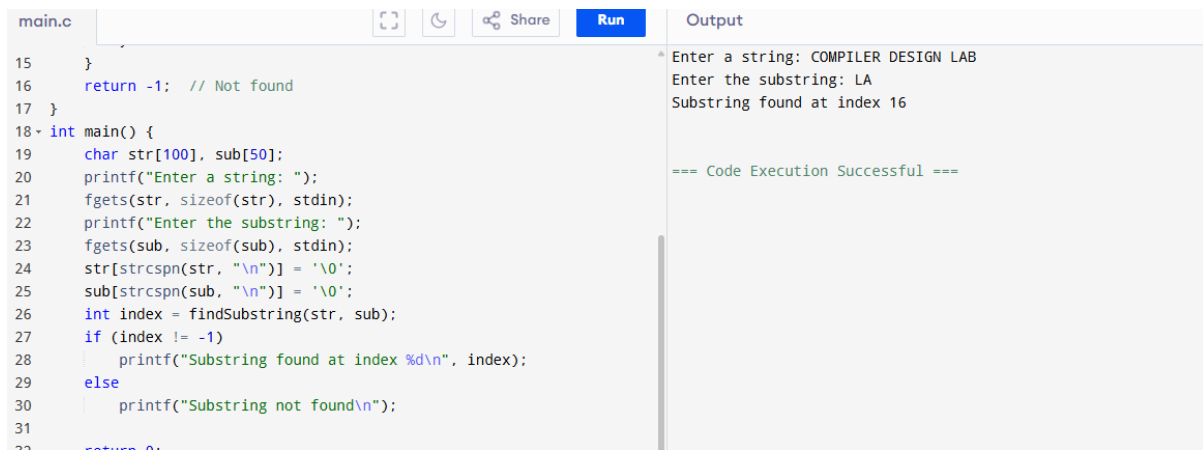
1. Start
2. Declare two character arrays: one for the main string and one for the substring.
3. Take input for both strings from the user.
4. Compute the lengths of both strings.
5. Loop through the main string and check for a match with the substring:
  - o Compare characters one by one.
  - o If a 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 sub[]) {
    int strLen = strlen(str), subLen = strlen(sub);
    for (int i = 0; i <= strLen - subLen; i++) {
        int j;
        for (j = 0; j < subLen; j++) {
            if (str[i + j] != sub[j]) {
                break;
            }
        }
        if (j == subLen) {
            return i; // Found at index i
        }
    }
    return -1; // Not found
}
int main() {
    char str[100], sub[50];
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    printf("Enter the substring: ");
    fgets(sub, sizeof(sub), stdin);
    str[strcspn(str, "\n")] = '\0';
    sub[strcspn(sub, "\n")] = '\0';
    int index = findSubstring(str, sub);
    if (index != -1)
        printf("Substring found at index %d\n", index);
    else
        printf("Substring not found\n");
}
```

```
    return 0;
}
```

## OUTPUT:



The screenshot shows a C code editor with a file named 'main.c'. The code defines a function 'findSubString' and a 'main' function. The 'main' function prompts the user to enter a string and a substring, then calls 'findSubString' to find the index of the substring. The output window shows the execution results for the input 'COMPILER DESIGN LAB' and substring 'LA', indicating the substring was found at index 16.

```
main.c
15 }
16 return -1; // Not found
17 }
18 int main() {
19     char str[100], sub[50];
20     printf("Enter a string: ");
21     fgets(str, sizeof(str), stdin);
22     printf("Enter the substring: ");
23     fgets(sub, sizeof(sub), stdin);
24     str[strcspn(str, "\n")] = '\0';
25     sub[strcspn(sub, "\n")] = '\0';
26     int index = findSubString(str, sub);
27     if (index != -1)
28         printf("Substring found at index %d\n", index);
29     else
30         printf("Substring not found\n");
31 }
32 return 0;
```

Output

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

=== Code Execution Successful ===
```

## 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: ");
    fgets(str1, sizeof(str1), stdin);
    printf("Enter second string: ");
    fgets(str2, sizeof(str2), stdin);
    str1[strcspn(str1, "\n")] = '\0';
    str2[strcspn(str2, "\n")] = '\0';
    if (strcmp(str1, str2) == 0)
        printf("Strings are the same.\n");
    else
        printf("Strings are different.\n");
    return 0;
}
```

## OUTPUT:

```
1  #include <stdio.h>
2  #include <string.h>
3~ int main() {
4      char str1[100], str2[100];
5      printf("Enter first string: ");
6      fgets(str1, sizeof(str1), stdin);
7      printf("Enter second string: ");
8      fgets(str2, sizeof(str2), stdin);
9      str1[strcspn(str1, "\n")] = '\0';
10     str2[strcspn(str2, "\n")] = '\0';
11     if (strcmp(str1, str2) == 0)
12         printf("Strings are the same.\n");
13     else
14         printf("Strings are different.\n");
15
16     return 0;
17 }
18
```

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

```
=== Code Execution Successful ===
```

## 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:
  - Copy only non-space characters to a new position in the array.
5. Print the modified string.
6. End

## PROGRAM:

```
#include <stdio.h>
void removeSpaces(char str[]) {
    int i, j = 0;
    for (i = 0; str[i] != '\0'; i++) {
        if (str[i] != ' ') {
            str[j++] = str[i];
        }
    }
    str[j] = '\0';
}
int main() {
    char str[100];
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    removeSpaces(str);
    printf("String without spaces: %s\n", str);
    return 0;
}
```

## OUTPUT:

```
1 #include <stdio.h>
2 void removeSpaces(char str[]) {
3     int i, j = 0;
4     for (i = 0; str[i] != '\0'; i++) {
5         if (str[i] != ' ') {
6             str[j++] = str[i];
7         }
8     }
9     str[j] = '\0';
10 }
11 int main() {
12     char str[100];
13     printf("Enter a string: ");
14     fgets(str, sizeof(str), stdin);
15     removeSpaces(str);
16     printf("String without spaces: %s\n", str);
17     return 0;
18 }
```

Enter a string: COMPILER DESIGN  
String without spaces: COMPILERDESIGN

=== Code Execution Successful ===

#include  
2- void rem  
3 int i  
4 for (i  
5 {  
6 {  
7 str[i]  
8 }  
9 str[i]  
10 }  
11 int main  
12 char  
13 printf  
14 fgets  
15 remov  
16 print  
17 return  
18 }  
Screen  
Auto

**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:
  - Increment the frequency count for each character.
6. Print characters with their respective frequencies.
7. End

**PROGRAM:**

```
#include <stdio.h>
#include <string.h>
void countFrequency(char str[]) {
    int freq[256] = {0};
    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]);
        }
    }
}
int main() {
    char str[100];
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    countFrequency(str);
    return 0;
}
```

**OUTPUT:**

```

3- void countFrequency(char str[]) {
4     int freq[256] = {0};
5-     for (int i = 0; str[i] != '\0'; i++) {
6         freq[(unsigned char)str[i]]++;
7     }
8     printf("Character Frequencies:\n");
9-     for (int i = 0; i < 256; i++) {
10-         if (freq[i] > 0) {
11             printf("%c : %d\n", i, freq[i]);
12         }
13     }
14 }
15- int main() {
16     char str[100];
17     printf("Enter a string: ");
18     fgets(str, sizeof(str), stdin);
19     countFrequency(str);
20     return 0;
21 }

```

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[50];
    printf("Enter first string: ");
    fgets(str1, sizeof(str1), stdin);
    printf("Enter second string: ");
    fgets(str2, sizeof(str2), stdin);
    str1[strcspn(str1, "\n")] = '\0';
    str2[strcspn(str2, "\n")] = '\0';
    strcat(str1, str2);
    printf("Concatenated string: %s\n", str1);
    return 0;
}
```

### OUTPUT:

```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[100], str2[50];
    printf("Enter first string: ");
    fgets(str1, sizeof(str1), stdin);
    printf("Enter second string: ");
    fgets(str2, sizeof(str2), stdin);
    str1[strcspn(str1, "\n")] = '\0';
    str2[strcspn(str2, "\n")] = '\0';
    strcat(str1, str2);
    printf("Concatenated string: %s\n", str1);
    return 0;
}
```

```
Enter first string: compiler
Enter second string: design
Concatenated string: compilerdesign
```

```
=== Code Execution Successful ===
```

**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:
  - Replace occurrences of the old character with the new one.
6. Print the modified string.
7. **End**

**PROGRAM:**

```
#include <stdio.h>
void replaceChar(char str[], char oldChar, char newChar) {
    for (int i = 0; str[i] != '\0'; i++) {
        if (str[i] == oldChar) {
            str[i] = newChar;
        }
    }
}
int main() {
    char str[100], oldChar, newChar;
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    printf("Enter character to replace: ");
    scanf("%c", &oldChar);
    getchar(); // Consume leftover newline character
    printf("Enter new character: ");
    scanf("%c", &newChar);
    replaceChar(str, oldChar, newChar);
    printf("Modified string: %s\n", str);
    return 0;
}
```

## OUTPUT:

```
    if (str[i] == oldChar) {
        str[i] = newChar;
    }
}

int main() {
    char str[100], oldChar, newChar;
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    printf("Enter character to replace: ");
    scanf("%c", &oldChar);
    getchar(); // Consume leftover newline character
    printf("Enter new character: ");
    scanf("%c", &newChar);
    replaceChar(str, oldChar, newChar);
    printf("Modified string: %s\n", str);
    return 0;
}
```

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

=== Code Execution Successful ===
```

Implementation	
Output/Signature	

## 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:**

**DATE:**

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

**AIM:**

To develop a C program that analyzes a given C code snippet and recognizes different tokens, including keywords, identifiers, operators, 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>
// List of C keywords
const char *keywords[] = {
    "int", "float", "char", "double", "if", "else", "for", "while",
    "do", "return", "void", "switch", "case", "break", "continue",
    "default", "struct", "typedef", "enum", "union", "static",
    "extern", "const", "sizeof", "goto", "volatile", "register"
};
const int num_keywords = sizeof(keywords) / sizeof(keywords[0]);
```

```

// List of C operators
const char *operators[] = {"+", "-", "*", "/", "=", "==", "!=", "<", ">", "<=", ">=", "&&", "||",
"++", "--"};

const int num_operators = sizeof(operators) / sizeof(operators[0]);

// List of special symbols
const char special_symbols[] = {';', '(', ')', '{', '}', '[', ']', ',', '#', '&', '|', ':', '"', '\\'};

// Function to check if a word is a keyword
int isKeyword(char *word) {
    for (int i = 0; i < num_keywords; i++) {
        if (strcmp(word, keywords[i]) == 0)
            return 1;
    }
    return 0;
}

// Function to check if a character is an operator
int isOperator(char *word) {
    for (int i = 0; i < num_operators; i++) {
        if (strcmp(word, operators[i]) == 0)
            return 1;
    }
    return 0;
}

// Function to check if a character is a special symbol
int isSpecialSymbol(char ch) {
    for (int i = 0; i < sizeof(special_symbols); i++) {
        if (ch == special_symbols[i])
            return 1;
    }
    return 0;
}

// Function to classify tokens

```

```

void analyzeTokens(char *code) {
    char *token = strtok(code, " \t\n"); // Tokenizing by spaces, tabs, and newlines
    printf("\nRecognized Tokens:\n");
    while (token != NULL) {
        if (isKeyword(token))
            printf("Keyword: %s\n", token);
        else if (isOperator(token))
            printf("Operator: %s\n", token);
        else if (isalpha(token[0]) || token[0] == '_') // Identifiers start with a letter or underscore
            printf("Identifier: %s\n", token);
        else if (isSpecialSymbol(token[0]))
            printf("Special Symbol: %s\n", token);
        else
            printf("Unknown Token: %s\n", token);
        token = strtok(NULL, " \t\n");
    }
}

// Main function
int main() {
    char code[500];
    printf("Enter a C code snippet:\n");
    fgets(code, sizeof(code), stdin);
    analyzeTokens(code);
    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>	

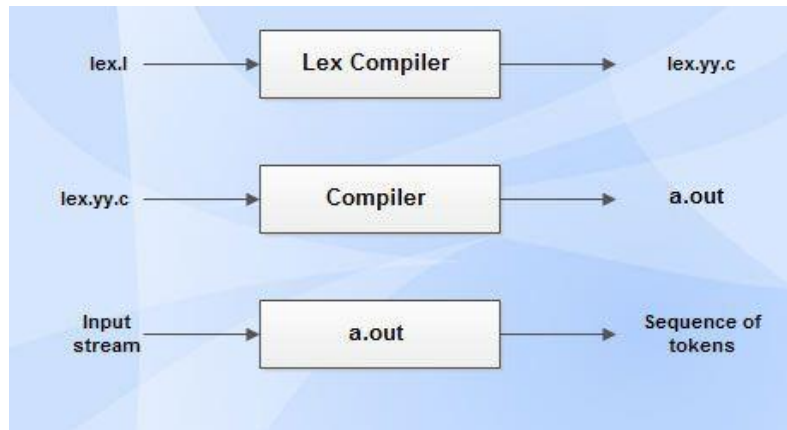
## RESULT :

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

```
%{ 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**

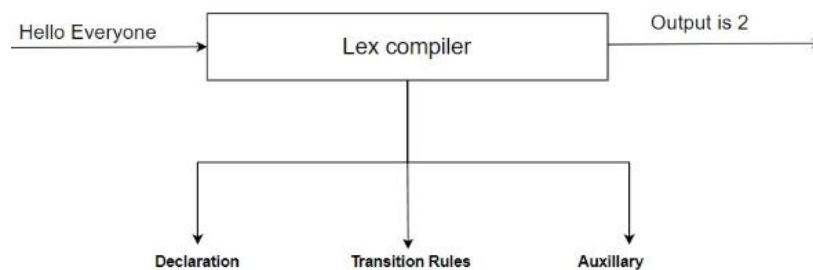
a global variable giving the length of the lexeme matched

### **yyval**

an external global variable storing the attribute of the token

### **yywrap:**

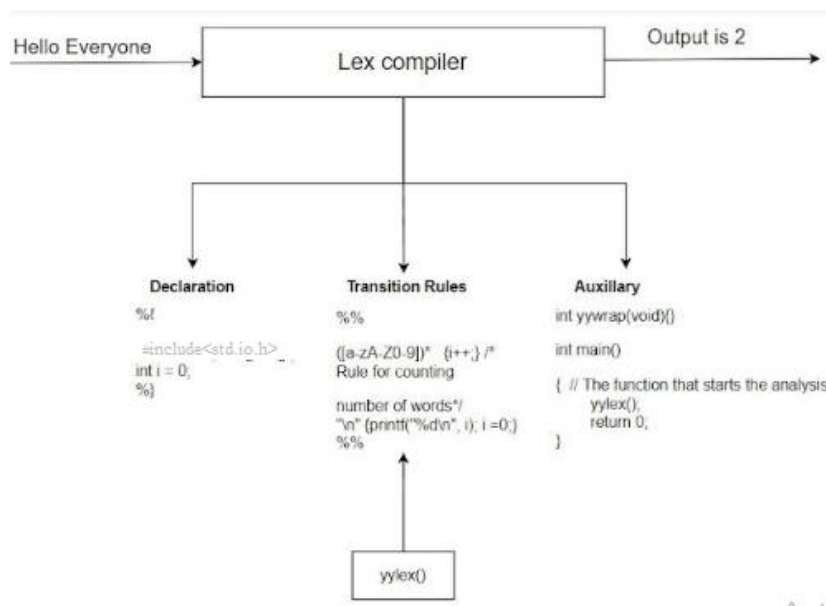
Function yywrap is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required.



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;

% }

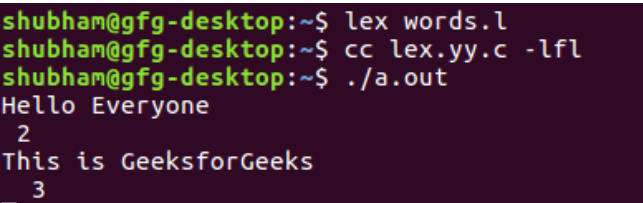
/* Rules Section*/
%%
([a-zA-Z0-9])* {i++;} /* Rule for counting
                      number of words*/

"\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:**

**DATE:**

**DEVELOP A LEXICAL ANALYZER 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>
% }
%option noyywrap
%%
// Keywords
"int"|"float"|"char"|"double"|"if"|"else"|"return"|"for"|"while"|"do" {
    printf("Keyword: %s\n", yytext);
}
// Identifiers (starting with a letter or underscore, followed by letters, digits, or underscores)
[a-zA-Z_][a-zA-Z0-9_]* {
    printf("Identifier: %s\n", yytext);
}
// Constants (integer and floating-point numbers)
[0-9]+(\.[0-9]+)? {
    printf("Constant: %s\n", yytext);
}
// Operators
"+"|"-"|"*"|"/"|"="|"=="|"!="|"<"|>"|"&&"|"||"|"++"|"--" {
    printf("Operator: %s\n", yytext);
}
```



```

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:**

**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>
int isValid = 1; // Flag to track if the expression is valid
% }
%option noyywrap
%%
// Numbers (integer and floating-point)
[0-9]+(\.[0-9]+)? {
    printf("Number: %s\n", yytext);
}

// Operators
"+"|"-"|"*"|"/" {
    printf("Operator: %s\n", yytext);
}

// Parentheses
"(" { printf("Left Parenthesis: %s\n", yytext); }
")" { printf("Right Parenthesis: %s\n", yytext); }
```

```

// Ignore spaces and tabs
[ \t]+ ;
// Invalid tokens
. {
    printf("Error: Invalid token '%s'\n", yytext);
    isValid = 0;
}
%%
int main() {
    printf("Enter an arithmetic expression:\n");
    yylex();
    if (isValid)
        printf("Valid arithmetic expression.\n");
    else
        printf("Invalid arithmetic expression.\n");

    return 0;
}

```

### 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



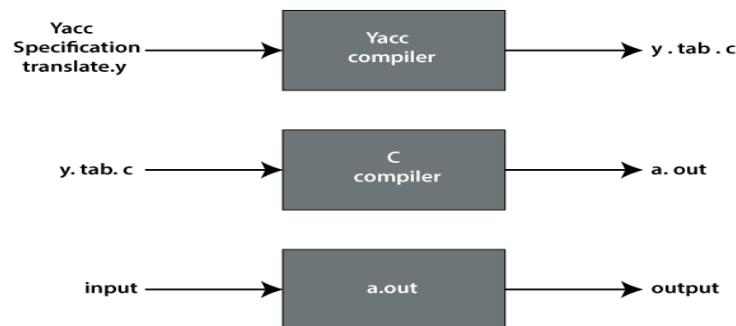
## YACC TOOL:

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

Start Symbol

**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:**

**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 PROGRAM**

```
% {  
#include "y.tab.h"  
% }  
%option noyywrap  
%%  
// Pattern for valid variable names  
[a-zA-Z][a-zA-Z0-9]* { return IDENTIFIER; }  
// Ignore whitespace  
[ \t\n] { /* Skip */ }  
  
. { return yytext[0]; }  
%%
```

### **YACC PROGRAM**

```
% {  
#include <stdio.h>  
void yyerror(const char *msg);  
% }  
%token IDENTIFIER  
%%  
stmt: IDENTIFIER { printf("Valid variable: %s\n", yytext); }  
;  
%%  
void yyerror(const char *msg) {  
    printf("Invalid variable\n");  
}
```

```
int main() {  
    printf("Enter a variable name: ");  
    yyparse();  
    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:**

**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:**

**LEX CODE :** expr.l

```
% {
#include "y.tab.h"
% }
%%
[0-9]+ {
    yylval = atoi(yytext);
    return NUMBER;
}
[+\n]   return yytext[0];
[*]     return yytext[0];
[ \t]   ; /* Ignore whitespace */

.       yyerror("Invalid character");
```

%%

**YACC Program :** expr.y



```

%{
#include <stdio.h>
#include <stdlib.h>
int yylex();
void yyerror(const char *s);
%}
%token NUMBER
%left '+' /* Lower precedence */
%left '*' /* Higher precedence */
%%
expression:
    expression '+' expression { $$ = $1 + $3; }
  | expression '*' expression { $$ = $1 * $3; }
  | NUMBER                    { $$ = $1; }
;
%%
int main() {
    printf("Enter an arithmetic expression:\n");
    yyparse();
    return 0;
}
void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

```

### 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

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

### 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:**

**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:**

LEX File (control\_structures.l):

```
% {
#include "y.tab.h"
% }
%%
"if"      { return IF; }
"else"    { return ELSE; }
"for"     { return FOR; }
"while"   { return WHILE; }
"switch"  { return SWITCH; }
"case"    { return CASE; }
[a-zA-Z_][a-zA-Z0-9_]* { return IDENTIFIER; }
"=="|"!="|<="|>="|"<|">" { return REL_OP; }
"+"|"-"|"*"|"/" { return ARITH_OP; }
"("       { return LPAREN; }
")"       { return RPAREN; }
"{"       { return LBRACE; }
"}"       { return RBRACE; }
";"       { return SEMICOLON; }
[ \t\n]   ; /* Ignore whitespace */
.         { printf("Invalid character: %s\n", yytext); }
%%
int yywrap() {
    return 1;
}
```

**YACC File (control\_structures.y)**

```
% {
#include <stdio.h>
#include <stdlib.h>
void yyerror(const char *s);
int yylex(void);
% }
%token IF ELSE FOR WHILE SWITCH CASE IDENTIFIER REL_OP ARITH_OP
%token LPAREN RPAREN LBRACE RBRACE SEMICOLON
%start program
%%
program:
    statement
    | program statement
    ;
statement:
    if_statement
    | for_loop
    | while_loop
    | switch_case
    ;
```

```

if_statement:
    IF LPAREN condition RPAREN LBRACE statements RBRACE
    | IF LPAREN condition RPAREN LBRACE statements RBRACE ELSE LBRACE
statements RBRACE
;
for_loop:
    FOR LPAREN assignment SEMICOLON condition SEMICOLON assignment RPAREN
LBRACE statements RBRACE
;
while_loop:
    WHILE LPAREN condition RPAREN LBRACE statements RBRACE
;

switch_case:
    SWITCH LPAREN expression RPAREN LBRACE case_statements RBRACE
;
case_statements:
    CASE expression COLON statements
    | case_statements CASE expression COLON statements
    | case_statements DEFAULT COLON statements
;
condition:
    IDENTIFIER REL_OP IDENTIFIER
    | IDENTIFIER REL_OP NUMBER
    | NUMBER REL_OP IDENTIFIER
    | NUMBER REL_OP NUMBER
;
assignment:
    IDENTIFIER '=' expression
;
expression:
    IDENTIFIER
    | NUMBER
    | expression ARITH_OP expression
;
statements:
    statement
    | statements statement
;
%%
void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main() {
    printf("Enter C control structures for validation:\n");
    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:**

**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., `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:

LEX file (expr.l)

```
% {
#include "y.tab.h"
% }

%%

[0-9]+ { yylval.str = strdup(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); return ID; }
[+\\-*/=()] { return yytext[0]; }
[ \\t\\n] { /* Ignore whitespace */ }
. { printf("Unexpected character: %s\\n", yytext); }
%%
```

YACC Program expr.y

```
% {
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int temp_count = 0;
```



```

char* new_temp() {
    char* temp = (char*)malloc(8);
    sprintf(temp, "t%d", temp_count++);
    return temp;
}

void emit(char* result, char* op1, char op, char* op2) {
    printf("%s = %s %c %s\n", result, op1, op, op2);
}

void emit_assign(char* id, char* expr) {
    printf("%s = %s\n", id, expr);
}
%}

%union {
    char* str;
}

%token <str> ID NUMBER
%type <str> expr term factor

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

%%
statement : ID '=' expr { emit_assign($1, $3); }
;

expr      : expr '+' term { $$ = new_temp(); emit($$, $1, '+', $3); }
          | expr '-' term { $$ = new_temp(); emit($$, $1, '-', $3); }
          | term          { $$ = $1; }
;

term      : term '*' factor { $$ = new_temp(); emit($$, $1, '*', $3); }
          | term '/' factor { $$ = new_temp(); emit($$, $1, '/', $3); }
          | factor         { $$ = $1; }
;

factor    : '(' expr ')' { $$ = $2; }
          | NUMBER      { $$ = $1; }
          | ID          { $$ = $1; }
;

%%

int main() {
    yyparse();
    return 0;
}

```

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

### 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:**

**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>

void generateAssembly(const char* tac) {
    char result[10], op1[10], op2[10];

    // Parse the TAC instruction
    sscanf(tac, "%s = %s %s %s", result, op1, op, op2);

    // Generate assembly code based on the operator
    if (strcmp(op, "+") == 0) {
        printf("MOV AX, [%s]\n", op1);
        printf("ADD AX, [%s]\n", op2);
        printf("MOV [%s], AX\n", result);
    } else if (strcmp(op, "-") == 0) {
        printf("MOV AX, [%s]\n", op1);
        printf("SUB AX, [%s]\n", op2);
        printf("MOV [%s], AX\n", result);
    } else if (strcmp(op, "*") == 0) {
        printf("MOV AX, [%s]\n", op1);
        printf("MOV BX, [%s]\n", op2);
        printf("MUL BX\n");
        printf("MOV [%s], AX\n", result);
    } else if (strcmp(op, "/") == 0) {
        printf("MOV AX, [%s]\n", op1);
        printf("MOV BX, [%s]\n", op2);
        printf("DIV BX\n");
        printf("MOV [%s], AX\n", result);
    } else {
        printf("Unsupported operation: %s\n", op);
    }
}

int main() {
    const char* tacInstructions[] = {
```

```

    "t0 = b + c",
    "t1 = t0 * d",
    "a = t1"
};

int numInstructions = sizeof(tacInstructions) / sizeof(tacInstructions[0]);
for (int i = 0; i < numInstructions; i++) {
    generateAssembly(tacInstructions[i]);
    printf("\n");
}
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
<pre>S1 = 4 x i S2 = a[S1] S3 = 4 x j S4 = 4 x i // <b>Redundant Expression</b> S5 = n S6 = b[S4] + S5</pre>	<pre>S1 = 4 x i S2 = a[S1] S3 = 4 x j S5 = n S6 = b[S1] + S5</pre>

## 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”.
- This is because the cost of multiplication operator is higher than that of addition operator.

**EXP NO :**

**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>
// Function to generate TAC for arithmetic expressions
void generateArithmeticTAC(const char* expr) {
    // Example: a = b + c * d
    // Expected TAC:
    // t1 = c * d
    // t2 = b + t1
    // a = t2

    char result[10], op1[10], op[2], op2[10];
    sscanf(expr, "%s = %s %s %s", result, op1, op, op2);

    if (strcmp(op, "+") == 0 || strcmp(op, "-") == 0) {
        printf("t1 = %s %s %s\n", op1, op, op2);
        printf("%s = t1\n", result);
    } else if (strcmp(op, "*") == 0 || strcmp(op, "/") == 0) {
        printf("t1 = %s %s %s\n", op1, op, op2);
        printf("%s = t1\n", result);
    } else {
        printf("Unsupported operation: %s\n", op);
    }
}
```

```

}

// Function to generate TAC for if-else statements
void generateIfElseTAC(const char* condition, const char* trueStmt, const char* falseStmt)
{
    // Example:
    // if (a < b) x = 1; else x = 2;
    // Expected TAC:
    // if a < b goto L1
    // goto L2
    // L1: x = 1
    // goto L3
    // L2: x = 2
    // L3:

    printf("if %s goto L1\n", condition);
    printf("goto L2\n");
    printf("L1: %s\n", trueStmt);
    printf("goto L3\n");
    printf("L2: %s\n", falseStmt);
    printf("L3:\n");
}

// Function to generate TAC for while loops
void generateWhileLoopTAC(const char* condition, const char* body) {
    // Example:
    // while (a < b) { x = x + 1; }
    // Expected TAC:
    // L1: if a >= b goto L2
    // x = x + 1
    // goto L1
    // L2:

    printf("L1: if %s goto L2\n", condition);
    printf("%s\n", body);
    printf("goto L1\n");
    printf("L2:\n");
}

int main() {
    // Example usage:
    printf("TAC for arithmetic expression:\n");
    generateArithmeticTAC("a = b + c");

    printf("\nTAC for if-else statement:\n");
    generateIfElseTAC("a < b", "x = 1", "x = 2");

    printf("\nTAC for while loop:\n");
    generateWhileLoopTAC("a >= b", "x = x + 1");
}

```

```
    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 :**

**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_CODE_LINES 100
#define MAX_LINE_LENGTH 100
#define MAX_VAR_LENGTH 20
typedef struct {
    char lhs[MAX_VAR_LENGTH];
    char op1[MAX_VAR_LENGTH];
    char operator;
    char op2[MAX_VAR_LENGTH];
    int isDead;
} TAC;
```

```

// Function to parse a TAC line
void parseTACLine(char *line, TAC *tac) {
    sscanf(line, "%s = %s %c %s", tac->lhs, tac->op1, &tac->operator, tac->op2);
    tac->isDead = 0;
}

// Function to perform Dead Code Elimination
void performDCE(TAC tac[], int n) {
    int used[MAX_CODE_LINES] = {0};

    // Mark variables that are used
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (strcmp(tac[i].lhs, tac[j].op1) == 0 || strcmp(tac[i].lhs, tac[j].op2) == 0) {
                used[i] = 1;
                break;
            }
        }
    }

    // Eliminate dead code
    for (int i = 0; i < n; i++) {
        if (!used[i]) {
            tac[i].isDead = 1;
        }
    }
}

// Function to perform Common Subexpression Elimination
void performCSE(TAC tac[], int n) {
    for (int i = 0; i < n; i++) {
        if (tac[i].isDead) continue;

        for (int j = i + 1; j < n; j++) {
            if (tac[j].isDead) continue;

            if (strcmp(tac[i].op1, tac[j].op1) == 0 &&
                strcmp(tac[i].op2, tac[j].op2) == 0 &&
                tac[i].operator == tac[j].operator) {
                // Replace the second occurrence with the first
                strcpy(tac[j].op1, tac[i].lhs);
                tac[j].operator = "\0";
                strcpy(tac[j].op2, "");
                tac[j].isDead = 1;
            }
        }
    }
}

// Function to print the optimized TAC

```

```

void printOptimizedTAC(TAC tac[], int n) {
    printf("Optimized Three-Address Code:\n");
    for (int i = 0; i < n; i++) {
        if (!tac[i].isDead) {
            printf("%s = %s", tac[i].lhs, tac[i].op1);
            if (tac[i].operator != '\0') {
                printf(" %c %s", tac[i].operator, tac[i].op2);
            }
            printf("\n");
        }
    }
}

```

```

int main() {
    char *code[] = {
        "t1 = a + b",
        "t2 = a + b",
        "t3 = t1 * c",
        "t4 = t2 * c",
        "d = t3 + t4",
        "e = t5 - t6"
    };

    int n = sizeof(code) / sizeof(code[0]);
    TAC tac[MAX_CODE_LINES];

    // Parse the TAC lines
    for (int i = 0; i < n; i++) {
        parseTACLine(code[i], &tac[i]);
    }

    // Perform Common Subexpression Elimination
    performCSE(tac, n);

    // Perform Dead Code Elimination
    performDCE(tac, n);

    // Print the optimized TAC
    printOptimizedTAC(tac, n);

    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 :**

**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 <string.h>
#include <stdlib.h>
#define MAX_LINES 100
#define MAX_LENGTH 50
typedef struct {
    char var[MAX_LENGTH];
    char value[MAX_LENGTH];
    int is_direct_assignment;
} Statement;
void apply_copy_propagation(Statement statements[], int count) {
    for (int i = 0; i < count; i++) {
        if (statements[i].is_direct_assignment) {
            char *lhs = statements[i].var;
            char *rhs = statements[i].value;
            for (int j = i + 1; j < count; j++) {
                if (statements[j].is_direct_assignment) {
                    if (strcmp(statements[j].value, lhs) == 0) {
                        strcpy(statements[j].value, rhs);
                    }
                } else {
                    char *pos = strstr(statements[j].value, lhs);
                    if (pos != NULL) {
                        char temp[MAX_LENGTH];
                        strcpy(temp, pos + strlen(lhs));
```

```

        *pos = '\0';
        strcat(statements[j].value, rhs);
        strcat(statements[j].value, temp);
    }
}
}
}
}

int main() {
    Statement statements[MAX_LINES];
    int count = 0;

    printf("Enter statements (e.g., a = b or c = a + d). Enter 'END' to finish:\n");
    char line[MAX_LENGTH];
    while (fgets(line, sizeof(line), stdin)) {
        if (strncmp(line, "END", 3) == 0) break;
        line[strcspn(line, "\n")] = 0; // Remove newline character

        char *equals = strchr(line, '=');
        if (equals != NULL) {
            *equals = '\0';
            strcpy(statements[count].var, line);
            strcpy(statements[count].value, equals + 1);
            statements[count].is_direct_assignment = (strchr(equals + 1, '+') == NULL &&
                                                         strchr(equals + 1, '-') == NULL &&
                                                         strchr(equals + 1, '*') == NULL &&
                                                         strchr(equals + 1, '/') == NULL);

            count++;
        }
    }

    apply_copy_propagation(statements, count);

    printf("\nOptimized code:\n");
    for (int i = 0; i < count; i++) {
        if (!(statements[i].is_direct_assignment && statements[i].value[0] == '\0')) {
            printf("%s = %s\n", statements[i].var, statements[i].value);
        }
    }

    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.

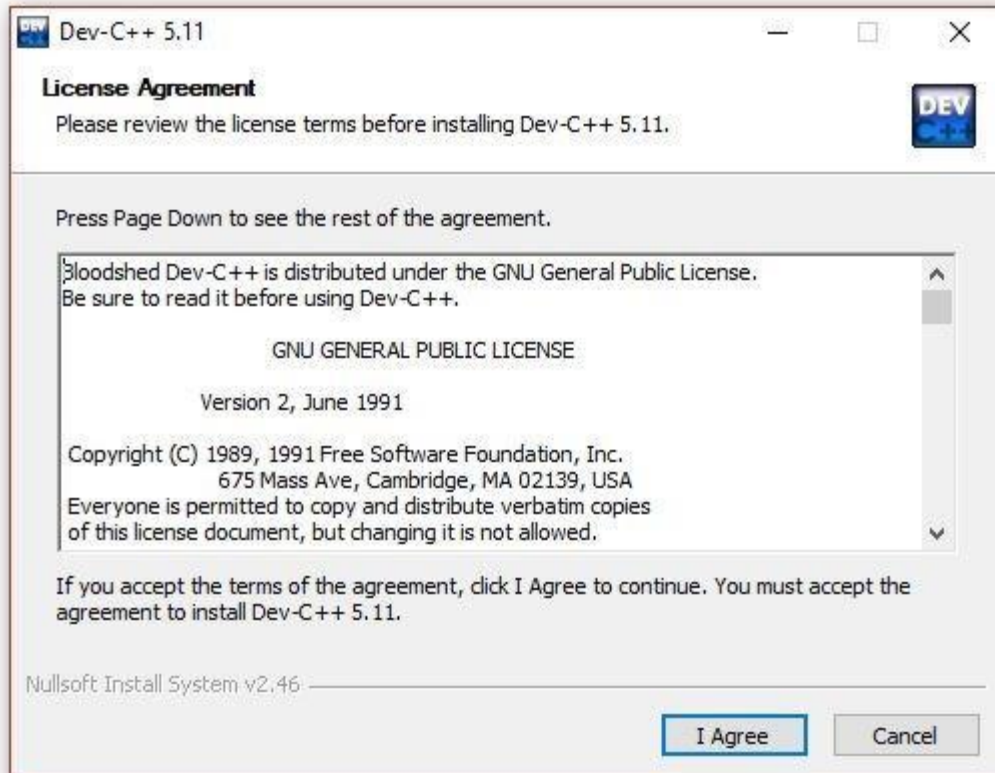
*Install the setup:*

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.

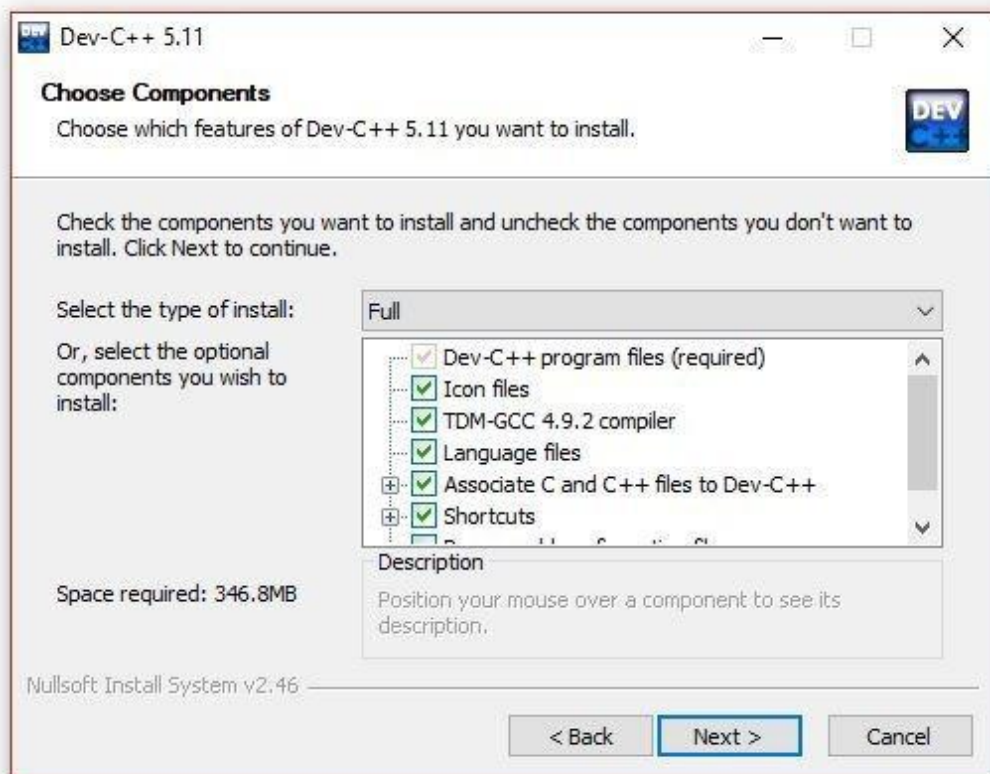
 Dev-Cpp 5.11 TDM-GCC 4.9.2 Setup	5/19/2015 10:51 AM	Application	49,252 KB
--	--------------------	-------------	-----------



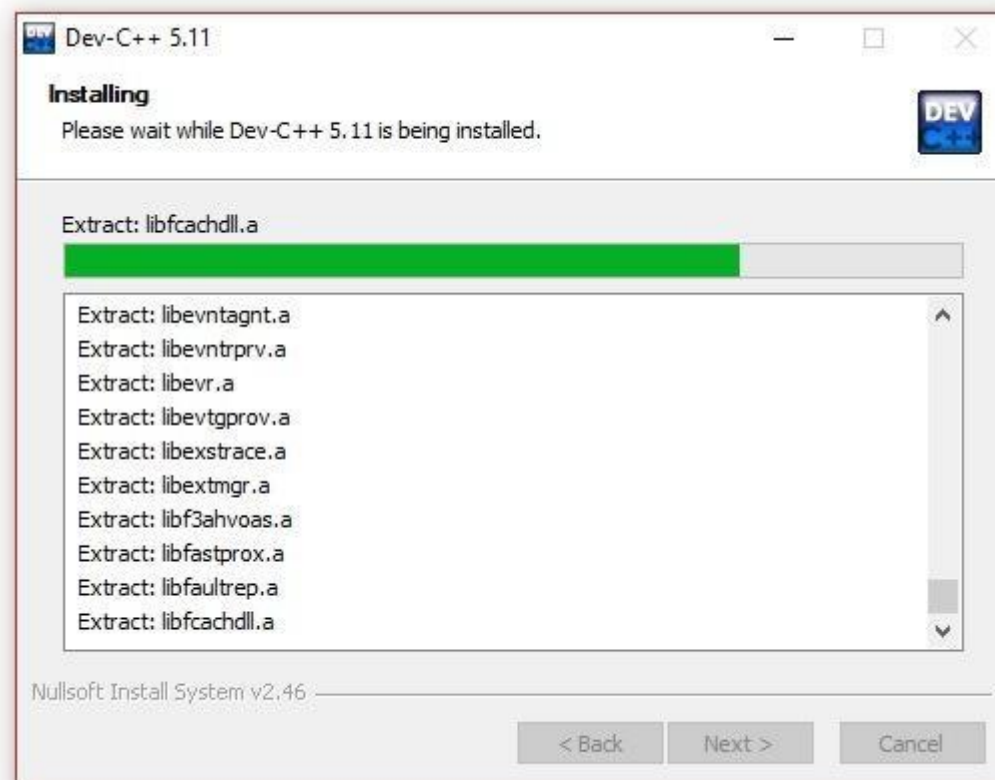
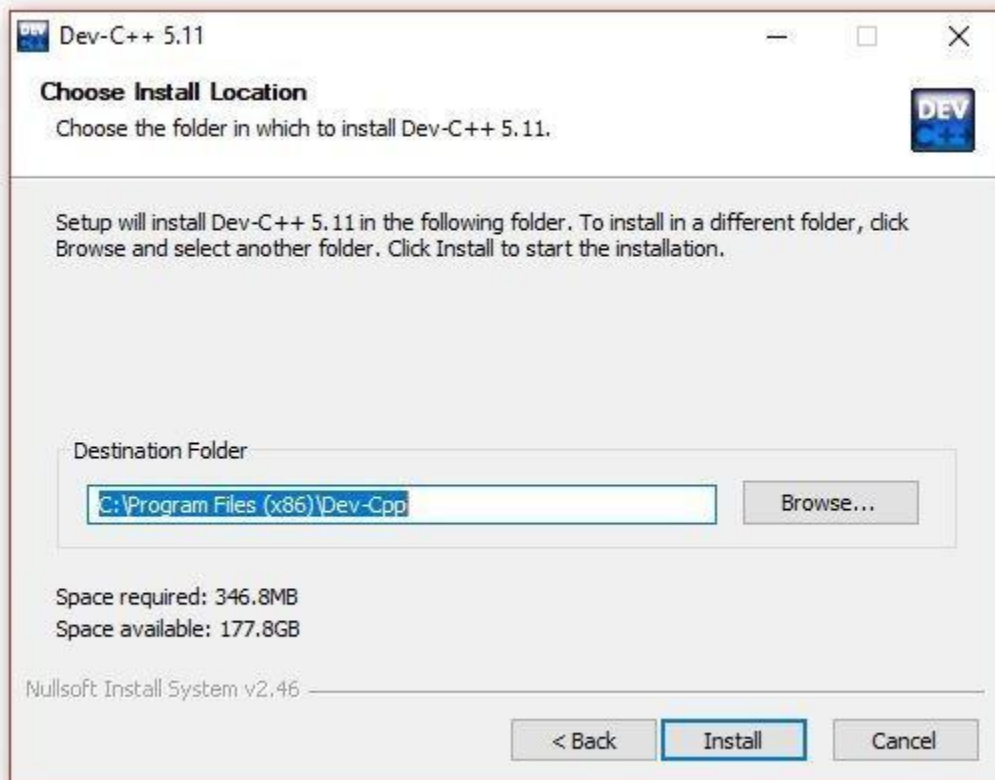
Accept the license agreement.



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.

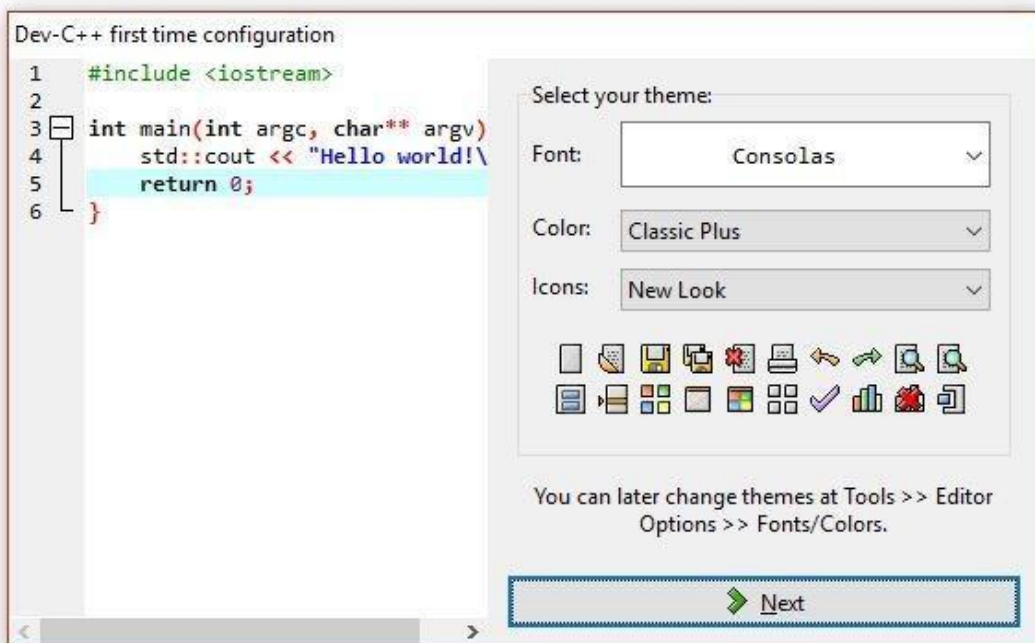
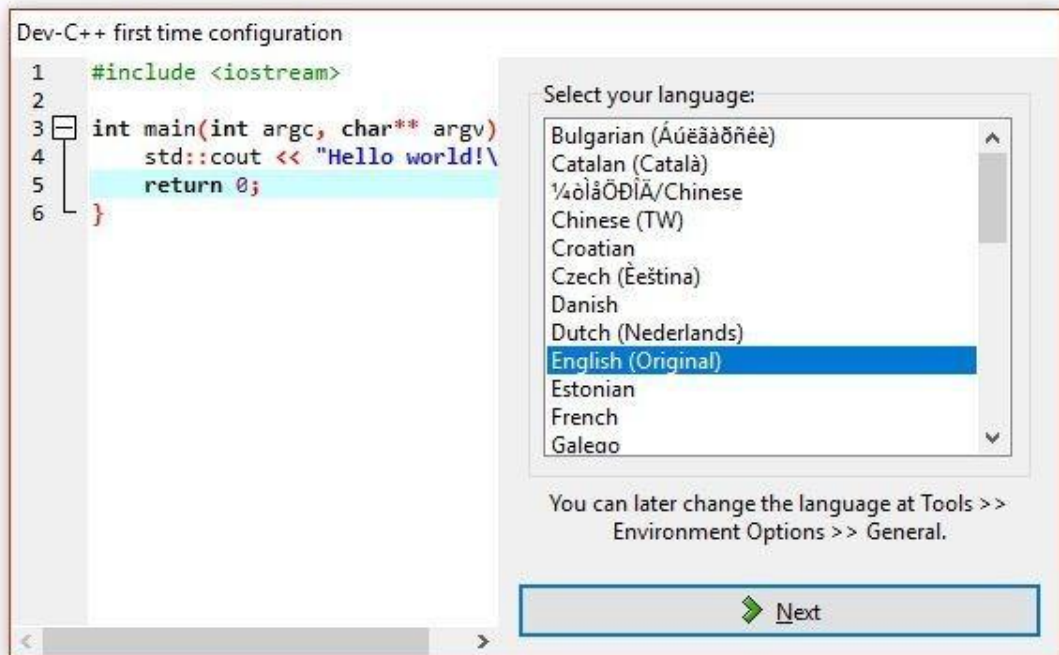


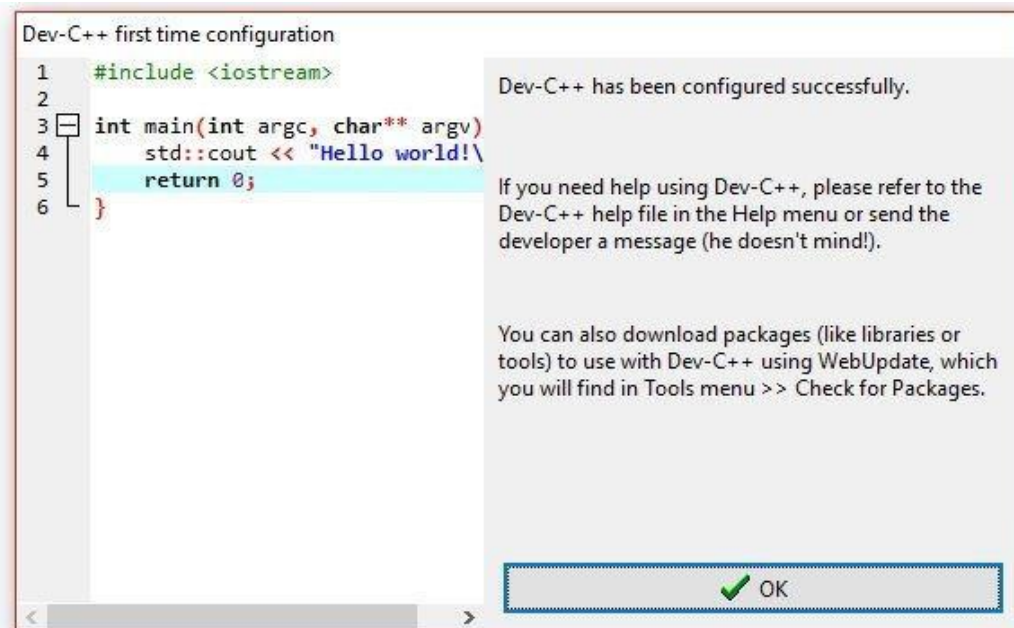
After completing the installation, Dev-C++ will prompt a window with a finish button. Just press the finish button.



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



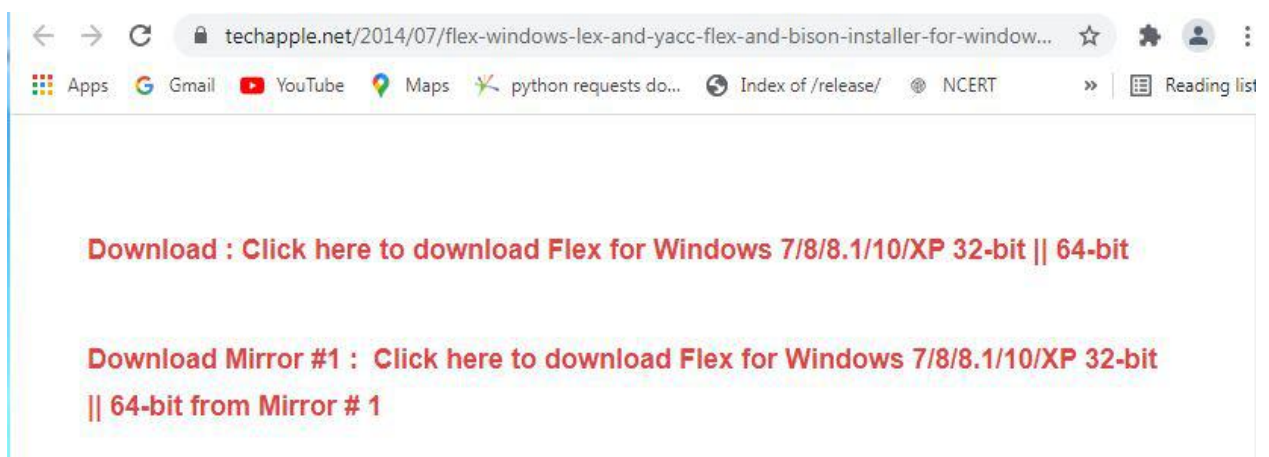




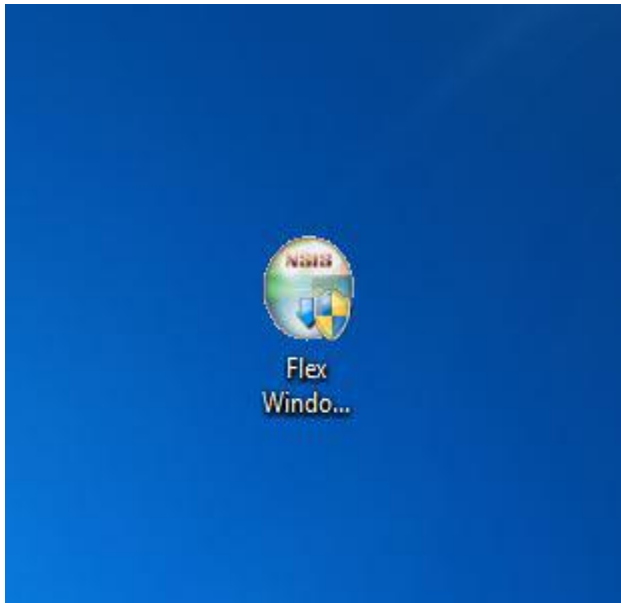
## Flex - Lex and Yacc

Step 1 : Install from the above URL

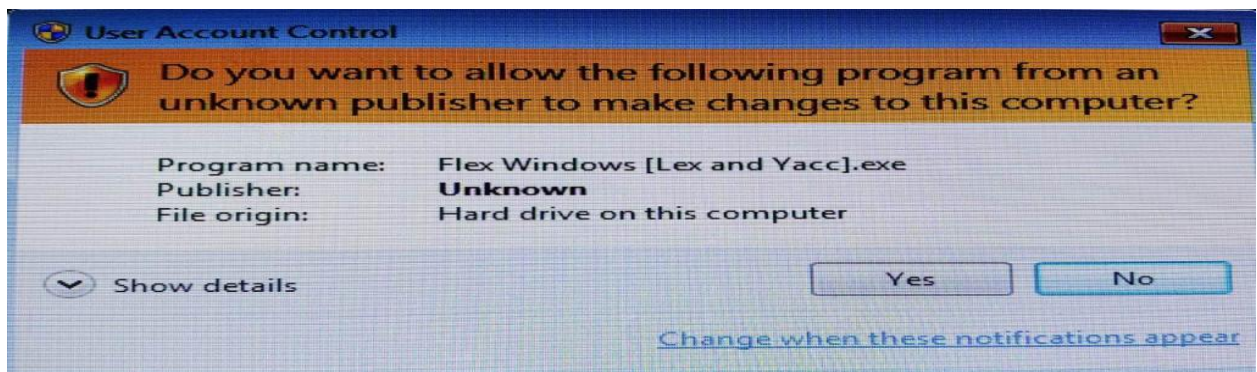
Step 2: On this page, all the features and minimum requirement of the system to install flex is given. Here the download link of the flex program for Windows XP, 7, 8, etc is given. Click on the download link, downloading of the executable file will start shortly. It is a small 30.19 MB file that will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



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



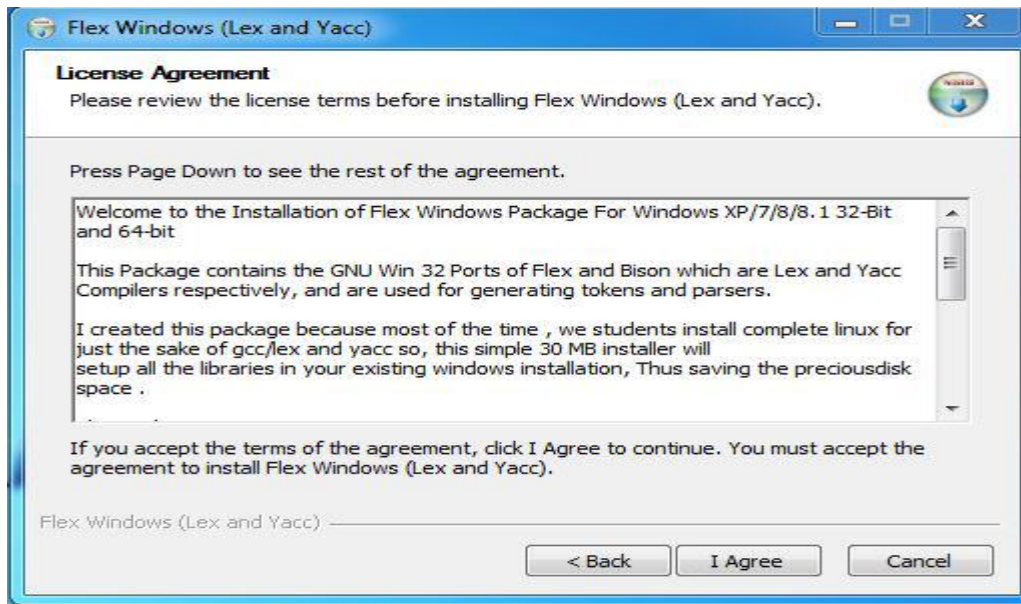
Step 4: It will prompt confirmation to make changes to your system. Click on Yes.



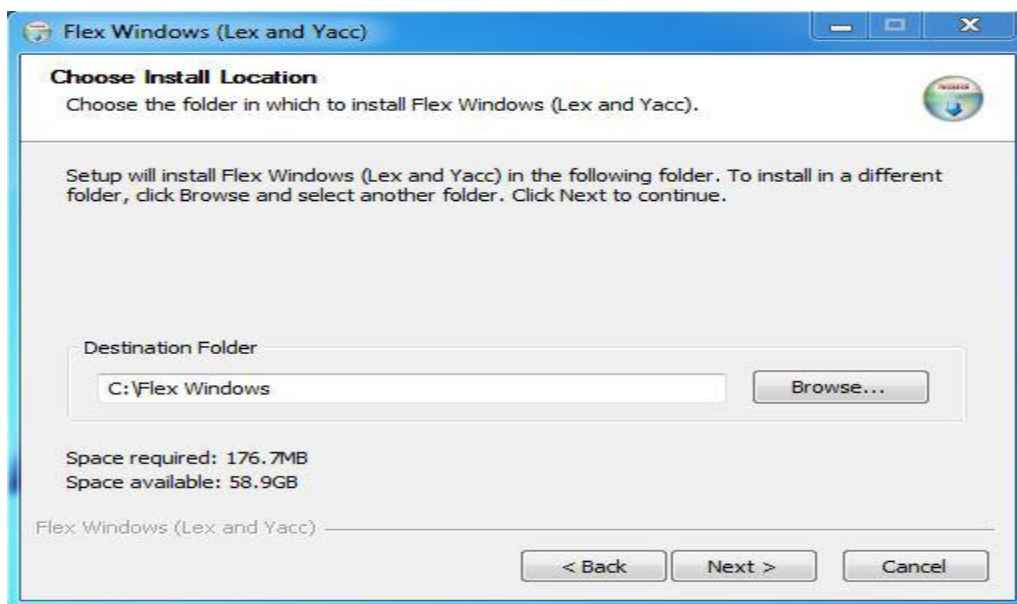
Step 5: Setup screen will appear, click on Next.



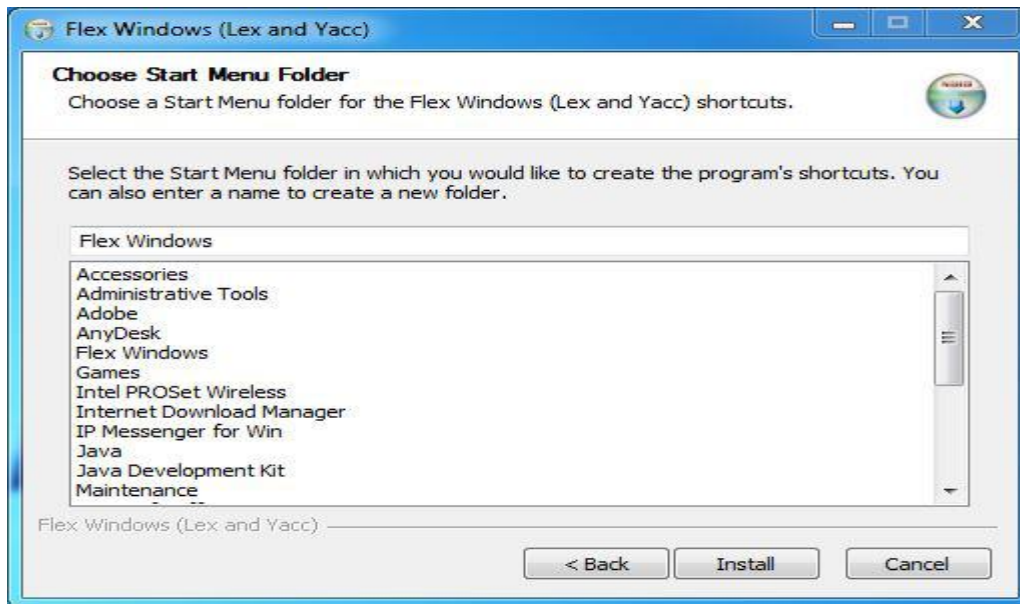
Step 6: The next screen will be of License Agreement, click on I Agree.



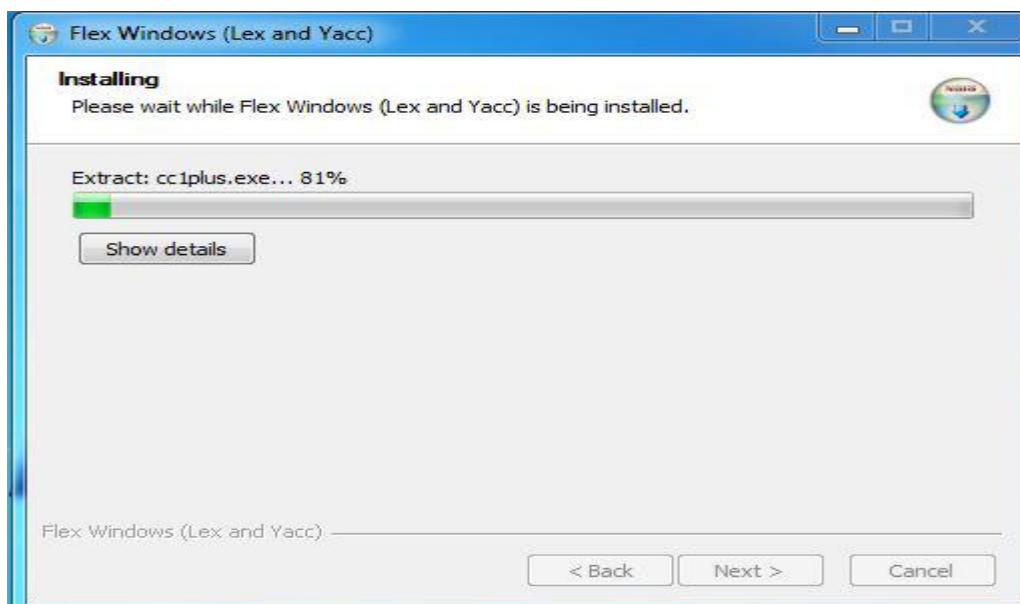
Step 7: The next screen will be of installing location so choose the drive which will have sufficient memory space for installation. It needed only a memory space of 176.7 MB.



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.

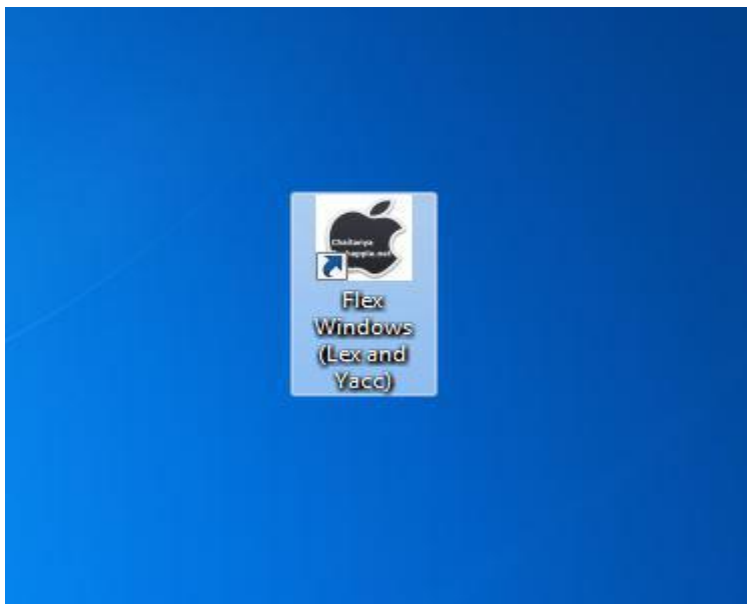


Step 10: Click on Finish after the installation process is complete. Keep the tick mark on the checkbox if you want to run Flex now if not then uncheck it.

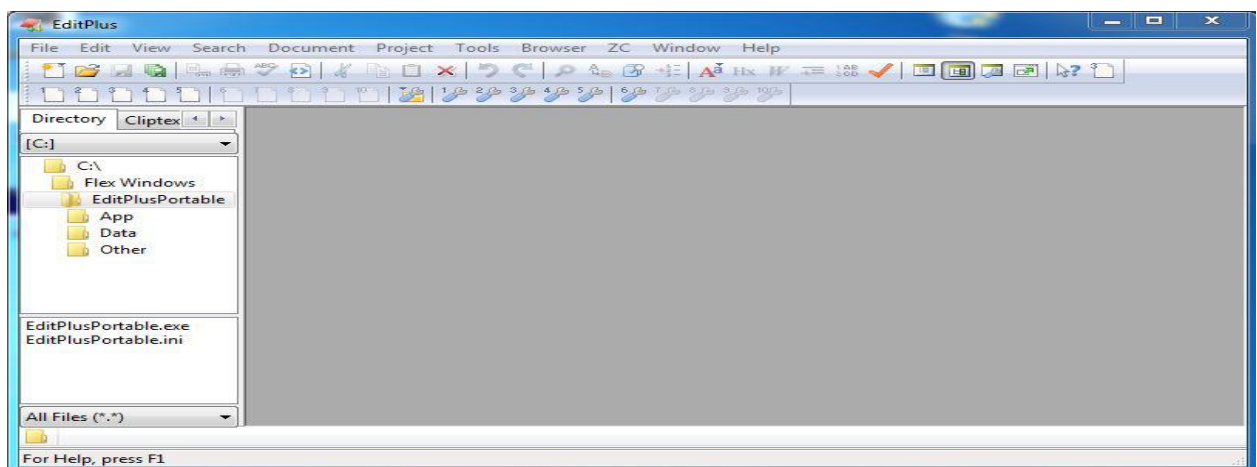




Step 11: Flex Windows is successfully installed on the system and an icon is created on the desktop

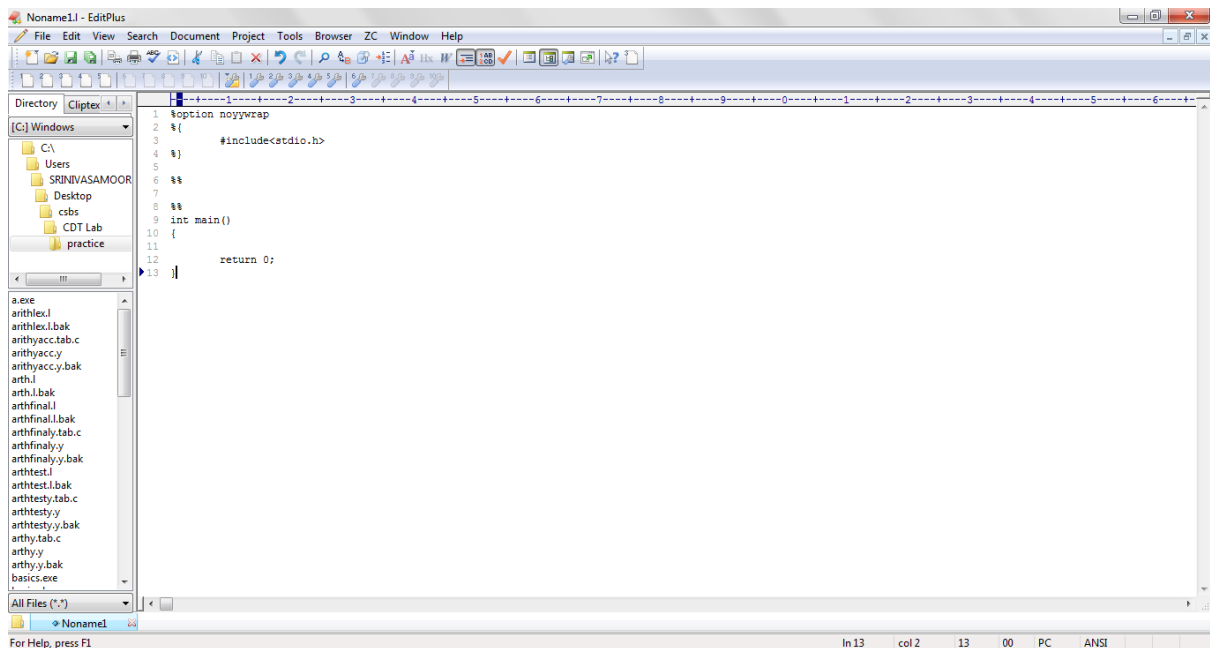


Step 12: Run the software and see the interface.



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

