Compiler Design Lab

CIC-351

Student Name: Sameer Sharma

Roll No: 02314802723

Group: 5C1

Faculty Name: Ms. Garima Gupta

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Rubrics for Lab Assessment:

Rubrics		10 Marks			POs and PSOs Covered	
		0 Marks	1 Marks	2 Marks	PO	PSO
R1	Is able to identify and define the objective of the given problem?	No	Partially	Completely	PO1, PO2	PSO1, PSO2
R2	Is proposed design/procedure/algorithm solves the problem?	No	Partially	Completely	PO1,PO2, PO3	PSO1, PSO2
R3	Has the understanding of the tool/programming language to implement the proposed solution?	No	Partially	Completely	PO1,PO3, PO5	PSO1, PSO2
R4	Are the result(s) verified using sufficient test data to support the conclusions?	No	Partially	Completely	PO2,PO4, PO5	PSO2
R5	Individuality of submission?	No	Partially	Completely	PO8, PO12	PSO1, PSO3

S.No	Experiment	R1	R2	R3	R4	R5	Total Marks (10)	Remarks	Faculty Signature
		Is able to identify and define the objective of the given problem?	Is proposed design /procedure /algorithm solves the problem?	Has the understanding of the tool/programming language to implement the proposed solution?	Are the result(s) verified using sufficient test data to support the conclusions?	Individuality of submission?			
		2 Marks	**2 Marks**	**2 Marks**	**2 Marks**	**2 Marks**			
1									
2									
3									
4									
5									
6									
7									

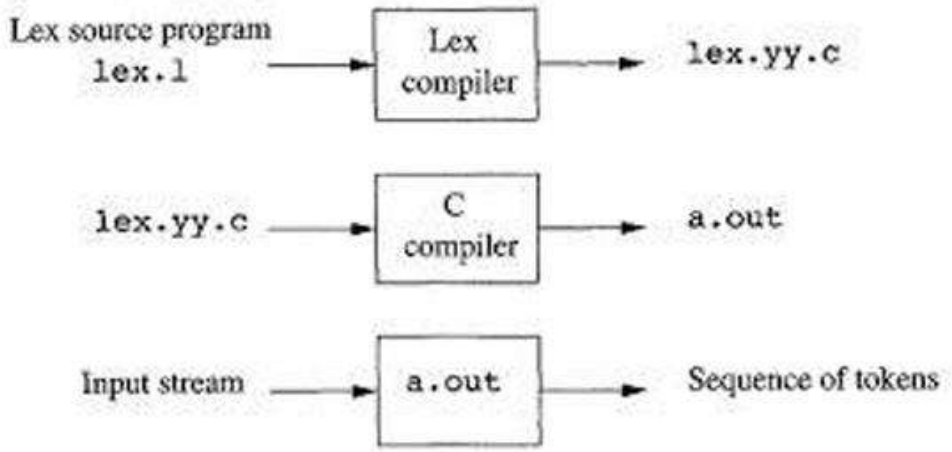
EXPERIMENT – 1

Aim:

Theory:

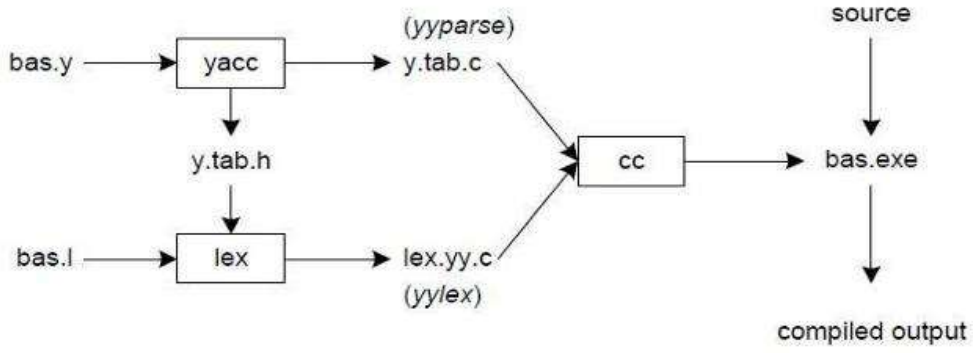
INTRODUCTION TO LEX AND YACC

We code patterns and input them to lex. It will read the patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on patterns written in the input file, and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing. The translation using Lex is shown below.



THE TRANSLATION USING LEX

When the lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and location of the variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index. We code a grammar and input it to Yacc. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical structure the tokens. The translation using Yacc as shown below. The below figure illustrate combined use and the file naming convention used by lex and yacc, such as “bas.y” for yacc compiler and “bas.l” for lex compiler.



LEX AND YACC COMPILERS

First, we need to specify all pattern matching rules for lex (bas.l) and grammar rules for yacc (bas.y). Commands to create our compiler, bas.exe, are listed below:

yacc –d # create y.tab.h, y.tab.c

bas.y lex bas.l # create lex.yy.c

gcc -o bas y.tab.c lex.yy.c -ll name 'bas.exe') # compile/link (output file

Yacc reads the grammar descriptions in bas.y and generates a syntax analyzer (parser), that includes function yyparse, in file y.tab.c. Included in file bas.y are token declarations. The –d option causes yacc to generate definitions for tokens and place them in file y.tab.h. Lex reads the pattern descriptions in bas.l, includes file y.tab.h, and generates a lexical analyzer, that includes function yylex, in file lex.yy.c. Finally, the lexer and parser are compiled and linked together to form the executable, bas.exe. From main, we call yyparse to run the compiler. Function yyparse automatically calls yylex to obtain each token.

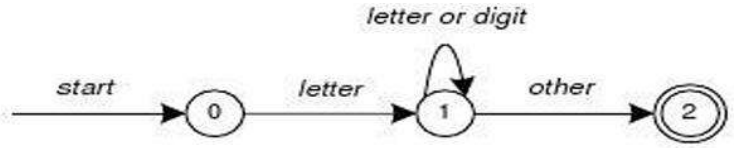
More about Lex:

The first phase in a compiler reads the input source and converts strings in the source to tokens. Using regular expressions, we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser.

The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

Letter (letter | digit)*

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state, and one or more final or accepting states.



FINITE STATE AUTOMATION

This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimic an FSA. Regular expressions in lex are composed of meta characters (Table 1). Pattern-matching examples are shown in Table 2. Within a character class, normal operators lose their meaning.

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

Table 1: Pattern Matching Primitives

Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	one of: a, b, c
[a-z]	any letter, a-z
[a\-z]	one of: a, -, z
[-az]	one of: -, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a, b
[a^b]	one of: a, ^, b
[a|b]	one of: a, |, b
a|b	one of: a, b

Table 2: Pattern Matching Examples

Two operators allowed in a character class are the hyphen (“-”) and circumflex (“^”). When used between two characters, the hyphen represents a range of characters. The circumflex,. If two patterns match the same string, the longest match wins. In case both matches are the same length, then the first pattern listed is used.

Lex Format:

... definitions ...

%%

... rules ...

%%

... subroutines ...

Input to Lex is divided into three sections, with %% dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file. %% Input is copied to output, one character at a time. The first %% is always required, as there must always be a rules section. However, if we don’t specify any rules, then the default action is to match everything and copy it to output.

Defaults for input and output are stdin and stdout, respectively. Here is the same example, with defaults explicitly coded.

```
%%

/* match everything except newline */ .

 ECHO;

/* match newline */

 \n ECHO;

 %%

int yywrap(void) {

 return 1;

}

int main(void) { yylex();

return 0; }
```

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by whitespace (space, tab or newline), and an optional action associated with the pattern. The action may be a single C statement, or multiple C statements enclosed in braces. Anything not starting in column one is copied exactly to the generated C file (Comments can copied from lex file to C file). In this example there are two patterns, "." and "\n", with an ECHO action associated for each pattern. Several macros and variables are predefined by lex. ECHO is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, ECHO is defined as

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

Variable yytext is a pointer to the matched string (NULL-terminated), and yyleng is the length of the matched string. Variable yyout is the output file, and defaults to stdout. Function yywrap is called by lex when input is exhausted. Return 1 if you are done, or 0 if more processing is required. Every C program requires a main function. In this case, we simply call yylex, the main entry-point for lex. Some implementations of lex include copies of main and yywrap in a library, eliminating the need to code them explicitly.

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

Table 3: Lex Predefined Variables

yywrap():

This function is called when end of file(or input) is encountered. If it return 1, the parsing stops. So this can be used to parse multiple files. Code can be written in third section, which will allow multiple file to be parsed. yylineno: Provides current line number information.

Sample Program: Lexical Analyzer for C programming language

```
letter [a-zA-Z]

digit [0-9]

%%

{digit}+("E"("+"|"-")?{digit}+)?

printf("\n%s\tis real number",yytext);

{digit}+"."{digit}+("E"("+"|"-")?{digit}+)? printf("\n%s\tis a float

number",yytext);

"if"|"else"|"int"|"float"|"switch"|"case"|"struct"|"char"|"return"|"for"|"do"|"whil

e"|"void"|"printf"|"scanf"

printf("\n%s\tis a keyword",yytext);

"\t"|"\b"|"\\n"|"\\t"|"\\a"|"\\b"|"\a" printf("\n%s\tis an escape

sequences",yytext);

{letter}({letter}|{digit})*

printf("\n%s\tis an identifier",yytext);

"["|"]"|"{"|"}"|"("|")"|"#"|"'"|"'"|"\"|"\\"|";"|"," printf("\n%s\t is a special

character",yytext);

"&&"|"<"|">"|"<="|">="|"+"|"="|"-"|"?"|"/"|"||"|"*"|"&"|"%" printf("\n%s\t is

an operator",yytext);

"%d"|"%s"|"%c"|"%f"|"%e" printf("\n%s\tis a format specifier",yytext);

%%

int yywrap()

{

   return 1;}

int main(int argc, char *argv[])

{

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

   yylex();
```

```
    fclose(yyin);
    return 0;
}
```

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it find more than one match, it take the one matching the most text. If it find two or more matches of the same length, the rule listed first in the flex input file is chosen.

Once the match is determined which satisfying one of the regular expression or rule, the text corresponding to the match (token) is made available in the global character pointer yytext and its length in the global integer yyleng. The action corresponding to the matched pattern is then executed.

YACC (Yet Another Compiler Compiler)

ACC (Yet Another Compiler Compiler) YACC is an LALR parser generator. The first version of YACC was created by S.C Johnson. Yacc translate a yacc file into a C file y.tab.c using LALR method

A Yacc source program has three parts:

Declarations

%%

Translation rules

%%

Supporting C- routines

The Declarations parts:

There are two optional sections in the declarations part of the Yacc program. In the first section put ordinary C declarations, delimited by %{ and %}. Here we place any temporaries used by the translation rules or procedures of the second and third sections.

Also in the declarations part are declarations of grammar tokens. Each grammar rule defines a symbol in terms of: Other symbols (Non terminals) and tokens (terminals) which come from the lexical analyzer.

Terminal symbols are of three types:

Named token: Defined via the %token identifier. By convention these are all uppercase

Character token: As same as character constant written in c language, Eg: + is a character

Constant: Literal string constant. Like C string constant Eg: constant. “abc” is string

The Lexer returns named tokens. Non terminals are usually represented by using lower case letters. A few Yacc specific declarations which begins with a % sign in Yacc specification file

1) %union. It defines the stack type for the parser. It is union of various data/ structures/ objects

2) %token. These are terminals return by the yylex function to the yacc. %token NUM NUM is a named token

3) %type. The type of non-terminal symbol in the grammar can be specified by using this rule.

4) %nonassoc. Specifies that there is no associativity of a terminal symbol

%nonassoc ´<´ no associativity

5) %left. Specifies that there is left associativity of a terminal symbol

%left ´+´ ´ - ´ make + and – be the same precedence and left associative

6) %right. Specifies that there is right associativity of a terminal symbol

%right ´+´ ´- ´ make + and – be the same precedence and right associative

7) %start. Specifies that the L.H.S non terminal symbol of production rule which should be taken as the starting point of the grammar rules.

8) %prec. Change the precedence level associated with a particular rule to that of the following token name or literal. %prec <terminal>

%left ´+´ ´ - ´

%right ´*´ ´/´

The above two statements declare the associativity and precedence exist among the tokens. The precedence of the token increases from top to bottom, so the bottom listed rule has the highest precedence compared to the rule listed above it. Here * and / have the higher precedence than the addition and subtraction.

Translation rules part:

After the first %% put the translation rules. Each rule consists of a grammar production and the associated semantic action. Consider this production

<Left_side> → <alt 1>| <alt 2>| ..... |<alt n>

would be written in yacc as

<left_side>

:

|

<alt 1> { semantic action 1}

<alt 2> { semantic action 2}

.....

|

;

<alt n> { semantic action n}

In Yacc production, quoted single character ´ c ´ is taken to be the terminal c and unquoted strings letter and digits not declare to be token to be taken as nonterminals. The Yacc semantic action is a sequence of C statements, In a semantic action the symbol $$ refers to the attribute value associated

with non terminals on the left, while $i refers to the value associated with the ith grammar symbol(terminal or non terminals) on the right. The semantic action is performed whenever we reduce by the associated production, So normally the semantic action computes a value for $$ in terms of the $i´s.

Example: Here the non-terminal term in the first production is the third grammar symbol on the right, while ´+´ is the second terminal symbol. We have omitted the semantic action for the second production, since copying the value is the default action for productions with a single grammar symbol on the right.

($$= $1) is the default action.

Supporting C routines part:

The third part of the yacc specification consists of supporting C- routines. A lexical analyzer by the name yylex() must be provided . Other procedures such as error recover routine may be added as necessary. The lexical analyzer yylex() produce pair consists of a token and its associated attribute value. The attribute value associated with a token is communicated to the parser through a yacc defined variable yylval.

How to compile Yacc and Lex files

1. Write a parser for a given grammar in a .y file

2. Write the lexical analyzer to process input and pass tokens to the parser when it needed, save the file as .l file

3. write error handler routine(like yyerror())

4. Compile .y file using the command yacc -d <filename.y> then it generate two file y.tab.c amd y.tab.h. If you are using bison, it generate <filename>.tab.c and <filename>.tab.h respectively

5. Compile lex file using the command flex <filename.l> it generate lex.yy.c

6. Compile the c file generated by the yacc compiler using gcc y.tab.c -ll it will generate .exe file a.out

7. Run the executable file and give the necessary input

History of lex and Yacc

Bison is descended from yacc, a parser generator written between 1975 and 1978 by Stephen C. Johnson at Bell Labs. As its name, short for “yet another compiler compiler,” suggests, many people were writing parser generators at the time. Johnson’s tool combined a firm theoretical foundation from parsing work by D. E. Knuth, which made its parsers extremely reliable, and a convenient input syntax. These made it extremely popular among users of Unix systems, although the restrictive license under which Unix was distributed at the time limited its use outside of academia and the Bell System. In about 1985, Bob Corbett, a graduate student at the University of California, Berkeley, re-implemented yacc using somewhat improved internal algorithms, which evolved into Berkeley yacc. Since his version was faster than Bell’s yacc Krrish Gupta 011 and was distributed under the flexible Berkeley license, it quickly became the most popular version of yacc. Richard Stallman of the Free Software Foundation (FSF) adapted Corbett’s work for use in the GNU project, where it has grown to include a vast number of new features as it has evolved into the current version of bison. Bison is now maintained as a project of the FSF and is distributed under the GNU Public License. In 1975, Mike Lesk and summer intern Eric Schmidt wrote lex, a lexical analyzer generator, with most of the

programming being done by Schmidt. They saw it both as a standalone tool and as a companion to Johnson's yacc. Lex also became quite popular, despite being relatively slow and buggy. (Schmidt nonetheless went on to have a fairly successful career in the computer industry where he is now the CEO of Google.) In about 1987, Vern Paxson of the Lawrence Berkeley Lab took a version of lex written in rat for (an extended Fortran popular at the time) and translated it into C, calling it flex, for "Fast Lexical Analyzer Generator." Since it was faster and more reliable than AT&T lex and, like Berkeley yacc, available under the Berkeley license, it has completely supplanted the original lex. Flex is now a SourceForge project, still under the Berkeley license. A flex program consists of three sections, separated by %% lines. The first section contains declarations and option settings. The second section is a list of patterns and actions, and the third section is C code that is copied to the generated scanner, usually small routines related to the code in the actions. In the declaration section, code inside of %{ and %} is copied through verbatim near the beginning of the generated C source file. In this case it just sets up variables for lines, words, and characters.

The commands for executing the LEX program are:

1. lex abc.l(abc is the file name)

2. cc lex.yy.c - efl

3. ./a.out

Practice Questions :

1. Write a program to construct a small calculator that can add, subtract, multiply and divide in LEX.

```
%
{
   int op = 0, i;
   float a, b;
   %
}
dig[0 - 9] + | ([0 - 9] *)"."([0 - 9] +)
                  add "+" sub "-" mul "*" div "/" pow "^" ln \n
               % %
/* digi() is a user defined function */
{
   dig
}
{
   digi();
}
```

```
{
   add
}
{
   op = 1;
}
{
   sub
}
{
   op = 2;
}
{
   mul
}
{
   op = 3;
}
{
   div
}
{
   op = 4;
}
{
   pow
}
{
   op = 5;
}
{
```

```
    ln
}
{
    printf("\n The Answer :%f\n\n", a);
}
% %
    digi()
{
    if (op == 0)
        /* atof() is used to convert - the ASCII input to float */
        a = atof(yytext);
    else
    {
        b = atof(yytext);
        switch (op)
        {
        case 1:
            a = a + b;
            break;
        case 2:
            a = a - b;
            break;
        case 3:
            a = a * b;
            break;
        case 4:
            a = a / b;
            break;
         case 5:
            for (i = a; b > 1; b--)
                a = a * i;
```

```
            break;
        }
        op = 0;
    }
}
main(int argv, char *argc[])
{
    yylex();
}
yywrap()
{
    return 1;
}
```

```
pc@pc:/media/pc/Shared/Compiler Lab$ ./a.out
5+5
 The Answer :10.000000
3+3
 The Answer :6.000000
3-3
 The Answer :0.000000
8/8
 The Answer :1.000000
6*22
 The Answer :132.000000
```

2. Write a program to match a string in LEX.

```
% {
#include <stdio.h>
     % } %
   % hello
{
   printf("Matched HELLO!\n");
}
```

```
[a - zA - Z] + { printf("Unmatched word: %s\n", yytext); }
.|\n{/* ignore other characters */} % %
      int main()
{
   yylex();
   return 0;
}
```

```
hello
 The Answer : Matched HELLO!
world
 The Answer : Unmatched word: world
hi
 The Answer : Unmatched word: hi
```

3. Write a LEX program to count the number of vowels and consonants in a given string.

```
%
{
#include <stdio.h>
   int vowels = 0, consonants = 0;
   %
}
% %
   [aeiouAEIOU]
{ vowels++; }
      [a - zA - Z]
{
   consonants++;
}
.|\n % %
{ /* ignore everything else */
}
int main()
{
   printf("Enter a string: ");
```

```
    yylex();
    printf("\n The Answer : Number of Vowels = %d\n", vowels);
    printf(" The Answer : Number of Consonants = %d\n",
        consonants);
    return 0;
}
```

```
Enter a string: OpenAI ChatGPT
 The Answer : Number of Vowels = 5
 The Answer : Number of Consonants = 8
```

Experiment – 2

Aim:- Write a program to check whether a string belong to the grammar or not.

a) S -> aS
S -> Sb
S -> ab
String of the form : aab

b) S -> aSa
S -> bSb
S -> a
S -> b
The language generated is : All odd length palindromes

c) S -> aSbb
S -> abb
The language generated is : $a^n b^{2n}$, where n>1

d) S -> aSb
S -> ab
The language generated is : $a^n b^n$, where n>0

Theory:- Context-Free Grammar (CFG)

A Context-Free Grammar (CFG) is a type of formal grammar used to describe the syntax of programming languages and formal languages. CFGs are widely studied in the field of Automata Theory and Compiler Design.

A CFG is defined as a 4-tuple:
$G=(V,\Sigma,R,S)$
where,

- V = Set of variables (non-terminals)
- Σ = Set of terminals (alphabet of the language)
- R = Set of production rules (of the form $A \rightarrow \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$)
- S = Start symbol (one of the variables from V)

Applications of CFG

- Compiler Design: CFGs are used to define the syntax of programming languages.
- Parsing: CFGs form the basis of parsing algorithms used by compilers.
- Natural Language Processing (NLP): Used to describe sentence structures in human languages.
- Formal Verification: Helps in describing and verifying system behaviors.

Example -
S -> aSb | ε
Check if "aabb" belongs:

- Derivation: $S \rightarrow aSb \rightarrow aaSbb \rightarrow aa\varepsilon bb \rightarrow aabb$
- "aabb" belongs to the grammar.

Code:-

```
#include <iostream>
using namespace std;

// Grammar (a): Strings generated are of form (a^n)b(a^m), ending with 'b'
bool grammarA(string s) {
    int n = s.size();
    if (n < 2){
        return false;
    }
    if (s[n - 1] != 'b'){
        return false;
    }
    int i = 0;
    while (i < n && s[i] == 'a'){
        i++;
    }
    while (i < n - 1 && s[i] == 'b'){
        i++;
    }
    return i == n - 1;
}

// Grammar (b): All odd length palindromes
bool grammarB(string s) {
    int n = s.size();
    if (n % 2 == 0){
        return false;
    }
    for (int i = 0; i < n / 2; i++){
        if (s[i] != s[n - 1 - i]){
            return false;
        }
    }
    return true;
}

// Grammar (c): Language = a^n b^(2n), n > 1
bool grammarC(string s) {
    int n = s.size();
    int i = 0;
    while (i < n && s[i] == 'a'){
        i++;
    }
    int countA = i;
    int countB = 0;
    while (i < n && s[i] == 'b'){
        countB++;
        i++;
```

```
    }
    return (i == n && countA > 1 && countB == 2 * countA);
}

// Grammar (d): Language = a^n b^n, n > 0
bool grammarD(string s) {
    int n = s.size();
    int i = 0;
    while (i < n && s[i] == 'a'){
        i++;
    }
    int countA = i;
    int countB = 0;
    while (i < n && s[i] == 'b'){
        countB++;
        i++;
    }
    return (i == n && countA > 0 && countA == countB);
}

int main() {
    int choice;
    string str;

    cout << "Select Grammar to check:\n";
    cout << "1. Grammar (a)\n";
    cout << "2. Grammar (b)\n";
    cout << "3. Grammar (c)\n";
    cout << "4. Grammar (d)\n";
    cout << "Enter choice: ";
    cin >> choice;
    cout << "Enter string: ";
    cin >> str;

    bool result = false;
    switch (choice) {
        case 1: result = grammarA(str); break;
        case 2: result = grammarB(str); break;
        case 3: result = grammarC(str); break;
        case 4: result = grammarD(str); break;
        default: cout << "Invalid choice\n"; return 0;
    }

    if (result)
        cout << "The string belongs to the selected grammar.\n";
    else
        cout << "The string does NOT belong to the selected grammar.\n";

    return 0;
}
```

Output:-

```
 if ($?) { g++ languages.cpp -o languages } ; if ($?) { .\languages }
Select Grammar to check:
1. Grammar (a)
2. Grammar (b)
3. Grammar (c)
4. Grammar (d)
Enter choice: 1
Enter string: aaabbb
The string belongs to the selected grammar.
PS D:\Vivek\College\CODES\compiler design> cd "d:\Vivek\College\CODES\compiler design\" ;
 if ($?) { g++ languages.cpp -o languages } ; if ($?) { .\languages }
Select Grammar to check:
1. Grammar (a)
2. Grammar (b)
3. Grammar (c)
4. Grammar (d)
Enter choice: 2
Enter string: aababababaa
The string belongs to the selected grammar.
```

```
 if ($?) { g++ languages.cpp -o languages } ; if ($?) { .\languages }
Select Grammar to check:
1. Grammar (a)
2. Grammar (b)
3. Grammar (c)
3. Grammar (c)
3. Grammar (c)
3. Grammar (c)
3. Grammar (c)
3. Grammar (c)
4. Grammar (d)
Enter choice: 3
Enter string: aaabbbbbb
The string belongs to the selected grammar.
```

```
 if ($?) { g++ languages.cpp -o languages } ; if ($?) { .\languages }
Select Grammar to check:
1. Grammar (a)
2. Grammar (b)
3. Grammar (c)
4. Grammar (d)
Enter choice: 4
Enter string: aabb
The string belongs to the selected grammar.
```

Experiment – 3

Aim:- Write a program to check whether a string includes a keyword or not.

Theory:-

- In programming languages like C++, certain words are **reserved keywords** (e.g., int, while, return).
- These keywords have predefined meanings and cannot be used as identifiers (like variable names).
- During **lexical analysis** (the first phase of a compiler), the source code is broken into tokens, and each token is checked against a list of keywords.
- This process ensures the compiler can distinguish between keywords and user-defined identifiers.

Example -

int sum = a + b;

During lexical analysis, the line is split into tokens:

"int", "sum", "=", "a", "+", "b", ";"

The token "int" is matched against the list of C++ keywords and recognized as a keyword.

Other tokens like "sum" and "a" are identified as identifiers.

Code:-

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
bool isKeyword(string str) {
   string keywords[] = {
      "alignas", "alignof", "and", "and_eq", "asm", "auto", "bitand", "bitor",
      "bool", "break", "case", "catch", "char", "char16_t", "char32_t", "class",
      "compl", "const", "constexpr", "const_cast", "continue", "decltype",
      "default", "delete", "do", "double", "dynamic_cast", "else", "enum",
      "explicit", "export", "extern", "false", "float", "for", "friend", "goto",
      "if", "inline", "int", "long", "mutable", "namespace", "new", "noexcept",
      "not", "not_eq", "nullptr", "operator", "or", "or_eq", "private",
      "protected", "public", "register", "reinterpret_cast", "return", "short",
```

```
        "signed", "sizeof", "static", "static_assert", "static_cast", "struct",
        "switch", "template", "this", "thread_local", "throw", "true", "try",
        "typedef", "typeid", "typename", "union", "unsigned", "using", "virtual",
        "void", "volatile", "wchar_t", "while", "xor", "xor_eq"
    };

    int n = sizeof(keywords) / sizeof(keywords[0]);
    for (int i = 0; i < n; i++) {
        if (str == keywords[i])
            return true;
    }
    return false;
}

int main() {
    string paragraph;
    cout << "Enter a paragraph: ";
    getline(cin, paragraph);
    stringstream ss(paragraph);
    string word;
    bool found = false;
    while (ss >> word) {
        if (isKeyword(word)) {
            cout << word << " is a C++ keyword.\n";
            found = true;
        }
    }
    if (!found)
        cout << "No C++ keywords found in the paragraph.\n";
    return 0;
}
```

Output:-

```
Enter a paragraph: This program will check if int or while or return is a
    keyword in my text.
if is a C++ keyword.
int is a C++ keyword.
or is a C++ keyword.
while is a C++ keyword.
or is a C++ keyword.
return is a C++ keyword.
```

Experiment – 4

Aim:- Write a program to remove left recursion from a Grammar.

Theory:-

- Left recursion occurs when a non-terminal refers to itself as the first symbol in its production, e.g.,

  $A \rightarrow A\alpha \mid \beta$

- This causes problems for top-down parsers (like recursive descent) due to infinite recursion.
- Left recursion is removed by transforming into:

  $A \rightarrow \beta A'$

  $A' \rightarrow \alpha A' \mid \varepsilon$

Example -

Given Grammar:

E -> E+T | T

After Removing Left Recursion:

E -> T E'

E' -> +T E' | ε

This new grammar avoids infinite recursion and is suitable for predictive parsing.

Code:-

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    int n;
    cout << "Enter number of productions: ";
    cin >> n;
    cin.ignore();
    for (int p = 0; p < n; p++) {
        string production;
        cout << "\nEnter production " << p + 1 << " (example: E->E+T|T): ";
        getline(cin, production);
```

```
string nonTerminal = production.substr(0, 1);
string rhs = production.substr(production.find("->") + 2);
string alpha[20], beta[20];
int alphaCount = 0, betaCount = 0;
string temp = "";
for (char c : rhs) {
    if (c == '|') {
        if (temp[0] == nonTerminal[0])  // left recursive
            alpha[alphaCount++] = temp.substr(1);
        else
            beta[betaCount++] = temp;
        temp = "";
    } else {
        temp += c;
    }
}
if (!temp.empty()) {
    if (temp[0] == nonTerminal[0])
        alpha[alphaCount++] = temp.substr(1);
    else
        beta[betaCount++] = temp;
}
cout << "\nResult after processing production " << p + 1 << ":\n";
if (alphaCount == 0) {
    cout << "No Left Recursion found in " << production << ".\n";
} else {
    cout << "After removing Left Recursion:\n";
    cout << nonTerminal << " -> ";
    for (int i = 0; i < betaCount; i++) {
        cout << beta[i] << nonTerminal << "' ";
    }
    cout << endl;
    cout << nonTerminal << "' -> ";
    for (int i = 0; i < alphaCount; i++) {
```

```
            cout << alpha[i] << nonTerminal << "' ";
        }
        cout << "ε\n";
    }
  }
  return 0;
}
```

Output:-

```
Enter number of productions: 2

Enter production 1 (example: E->E+T|T): E->E+T|T

Result after processing production 1:
After removing Left Recursion:
E -> TE'
E' -> +TE' ε

Enter production 2 (example: E->E+T|T): A->Aa|b

Result after processing production 2:
After removing Left Recursion:
A -> bA'
A' -> aA' ε
```

Experiment – 5

Aim:- Write a program to perform left factoring on a Grammar.

Theory:-

- Left factoring is used when multiple productions of a non-terminal share a common prefix, making parsing decisions ambiguous.
- It rewrites the grammar to factor out the common part:

  A -> αβ1 | αβ2

  → A -> αA'

  A' -> β1 | β2
- This helps predictive parsers (LL(1)) decide which production to use by looking ahead one symbol.

Example -

- Given Grammar:

  A -> aB | aC
- After Left Factoring:

  A -> a A'

  A' -> B | C

Now the parser first reads 'a' and then decides between B or C, removing ambiguity.

Code:-

```
#include <iostream>
#include <cstring>
using namespace std;

const int MAX_PROD = 10;
const int MAX_LEN = 100;

int splitProductions(const char input[], char productions[MAX_PROD][MAX_LEN]) {
    int count = 0;
    int len = strlen(input);
    int j = 0;
    for (int i = 0; i <= len; i++) {
```

```
        if (input[i] == '|' || input[i] == '\0') {
            productions[count][j] = '\0';
            count++;
            j = 0;
        } else {
            productions[count][j++] = input[i];
        }
    }
    return count;
}

int commonPrefixLength(const char s1[], const char s2[]) {
    int i = 0;
    while (s1[i] != '\0' && s2[i] != '\0' && s1[i] == s2[i])
        i++;
    return i;
}

void leftFactoring(const char nonTerminal[], char productions[MAX_PROD][MAX_LEN],
int n) {
    bool factored = false;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            int prefixLen = commonPrefixLength(productions[i], productions[j]);
            if (prefixLen > 0) {
                factored = true;
                cout << nonTerminal << " -> " << productions[i][0] << nonTerminal << "'" <<
endl;
                cout << nonTerminal << "' -> ";
                bool first = true;
                for (int k = 0; k < n; k++) {
                    int currentPrefixLen = commonPrefixLength(productions[i], productions[k]);
                    if (currentPrefixLen >= prefixLen) {
                        if (!first) cout << " | ";
```

```
                if (strlen(productions[k] + prefixLen) == 0)
                    cout << "ε";
                else
                    cout << (productions[k] + prefixLen);
                first = false;
            }
        }
        cout << endl;
        return;
    }
  }
  cout << nonTerminal << " -> ";
  for (int i = 0; i < n; i++) {
    cout << productions[i];
    if (i != n - 1)
      cout << " | ";
  }
  cout << endl;
}

int main() {
  char inputLine[MAX_LEN];
  char nonTerminal[10];
  char rhs[MAX_LEN];
  char productions[MAX_PROD][MAX_LEN];
  cout << "Enter production in format (e.g., A -> ab|ac|b): ";
  cin.getline(inputLine, MAX_LEN);
  char *arrowPos = strstr(inputLine, "->");
  if (arrowPos == NULL) {
    cout << "Invalid input format." << endl;
    return 1;
  }
  int ntLength = arrowPos - inputLine;
```

```
    strncpy(nonTerminal, inputLine, ntLength - 1);
    nonTerminal[ntLength - 1] = '\0';
    strcpy(rhs, arrowPos + 2);
    int start = 0;
    while (rhs[start] == ' ') start++
    char cleanRhs[MAX_LEN];
    int idx = 0;
    for (int i = start; i < strlen(rhs); i++)
        cleanRhs[idx++] = rhs[i];
    cleanRhs[idx] = '\0';
    int prodCount = splitProductions(cleanRhs, productions);
    cout << "\nAfter Left Factoring:\n";
    leftFactoring(nonTerminal, productions, prodCount);
    return 0;
}
```

Output:-

```
Enter production in format (e.g., A -> ab|ac|b): A -> ab|ac|b

After Left Factoring:
A -> aA'
A' -> b | c
```

PROGRAM 5(B)

AIM: To implement Minimum Spanning Tree and analyse its time complexity (Kruskal and Prim's algorithms).

THEORY:

A minimum spanning tree (MST) connects all vertices in a weighted graph with the minimum total edge weight.
Two common algorithms are Kruskal's and Prim's algorithms.
Kruskal's algorithm sorts all edges and adds them one by one while avoiding cycles.
Prim's algorithm grows the spanning tree starting from an arbitrary vertex by adding the minimum weight edge at each step.

CODE:

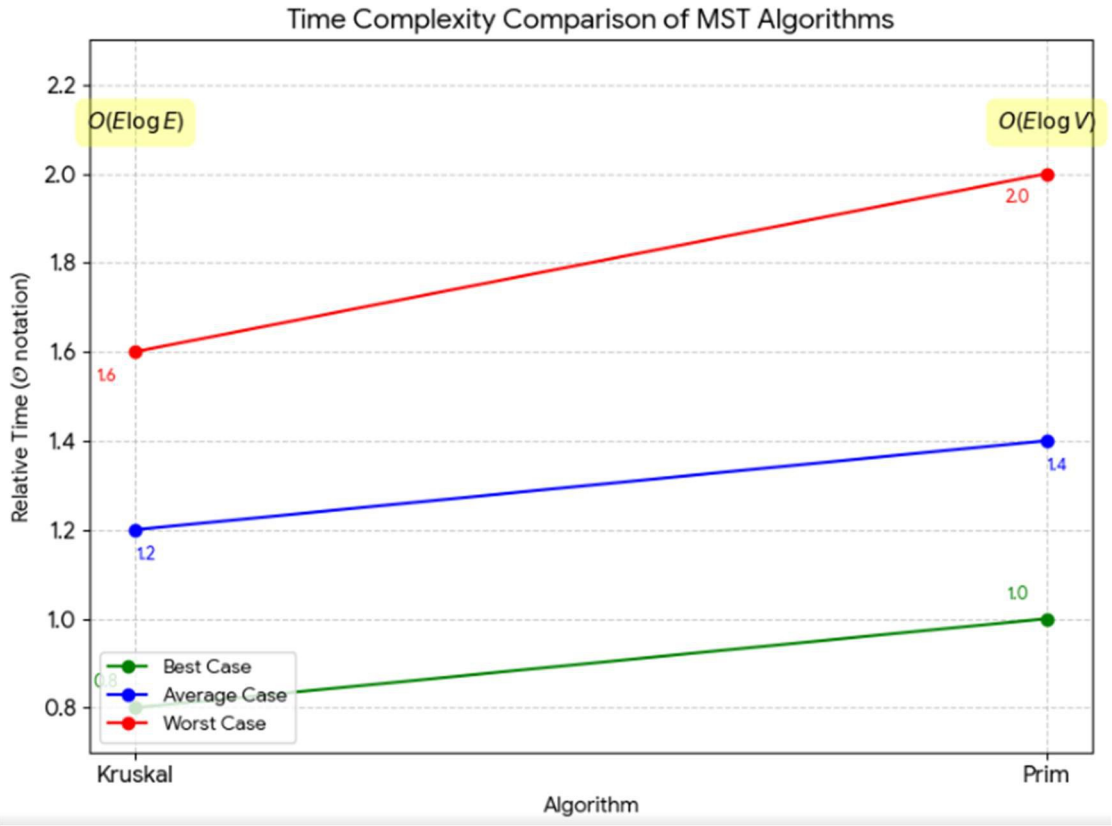
```
#include <iostream>

#include <vector>

#include <algorithm>

#include <climits>

using namespace std;

struct Edge {

    int u, v, w;

    bool operator<(Edge const& other) {

        return w < other.w;

    }

};

int find(vector<int>& parent, int i) {

    if (parent[i] == i) return i;

    return parent[i] = find(parent, parent[i]);

}


void unite(vector<int>& parent, vector<int>& rank, int u, int v) {

    u = find(parent, u);
```

```
    v = find(parent, v);
    if (u != v) {
        if (rank[u] < rank[v]) parent[u] = v;
        else if (rank[u] > rank[v]) parent[v] = u;
        else parent[v] = u, rank[u]++;
    }
}

void kruskalMST(vector<Edge>& edges, int V) {
    sort(edges.begin(), edges.end());
    vector<int> parent(V), rank(V, 0);
    for (int i = 0; i < V; i++) parent[i] = i;
    int cost = 0;
    cout << "Edges in MST (Kruskal):\n";
    for (auto& e : edges) {
        if (find(parent, e.u) != find(parent, e.v)) {
            cost += e.w;
            cout << e.u << " - " << e.v << " : " << e.w << endl;
            unite(parent, rank, e.u, e.v);
        }
    }
    cout << "Total cost: " << cost << endl;
}

void primMST(vector<vector<int>>& graph, int V) {
    vector<int> key(V, INT_MAX), parent(V, -1);
    vector<bool> inMST(V, false);
    key[0] = 0;
```

```
    for (int count = 0; count < V - 1; count++) {
        int u = -1;
        for (int i = 0; i < V; i++)
            if (!inMST[i] && (u == -1 || key[i] < key[u])) u = i;

        inMST[u] = true;
        for (int v = 0; v < V; v++) {
            if (graph[u][v] && !inMST[v] && graph[u][v] < key[v]) {
                key[v] = graph[u][v];
                parent[v] = u;
            }
        }
    }

    cout << "Edges in MST (Prim):\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << " : " << graph[i][parent[i]] << endl;
}

int main() {
    int V = 4;
    vector<Edge> edges = {{0,1,10}, {0,2,6}, {0,3,5}, {1,3,15}, {2,3,4}};
    kruskalMST(edges, V);

    vector<vector<int>> graph = {
        {0, 10, 6, 5},
        {10, 0, 0, 15},
```

```
        {6, 0, 0, 4},
        {5, 15, 4, 0}
    };
    primMST(graph, V);
    return 0;
}
```

OUTPUT:



PROGRAM 6

AIM: Implement and analyse the time complexity of Heap Sort.

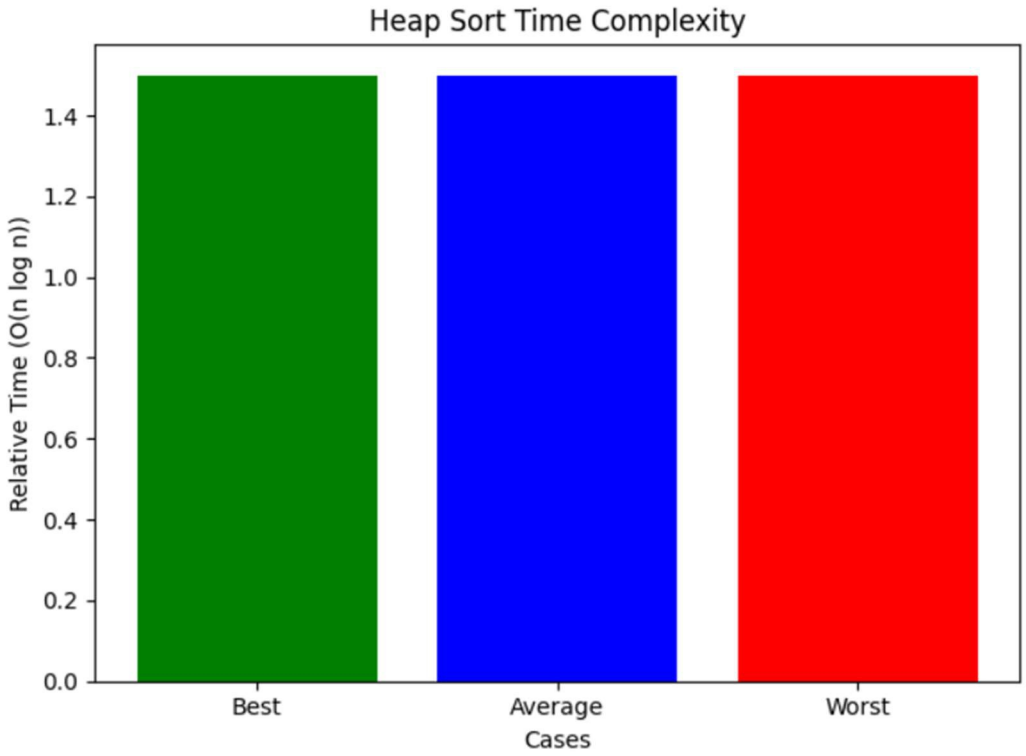
THEORY:

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure.
It first builds a max-heap and then repeatedly extracts the maximum element to sort the array.
Heap Sort has O(n log n) time complexity for all cases.

CODE:

```
#include <iostream>
using namespace std;

void heapify(int arr[], int n, int i) {
    int largest = i, l = 2*i + 1, r = 2*i + 2;
    if (l < n && arr[l] > arr[largest]) largest = l;
    if (r < n && arr[r] > arr[largest]) largest = r;
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    for (int i = n/2 - 1; i >= 0; i--) heapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = 6;
    heapSort(arr, n);
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) cout << arr[i] << " ";
    return 0;
}
```

OUTPUT:


Heap Sort Time Complexity
Relative Time (O(n log n))
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
Best
Average
Worst
Cases

PROGRAM 7

AIM: To implement Dijkstra's and Bellman-Ford algorithms and analyse their time complexities.

THEORY:

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph with non-negative weights.
Bellman-Ford algorithm also finds shortest paths but can handle negative edge weights.
Dijkstra's algorithm has O(V^2) or O(E + V log V) with a priority queue, while Bellman-Ford has O(VE) complexity.

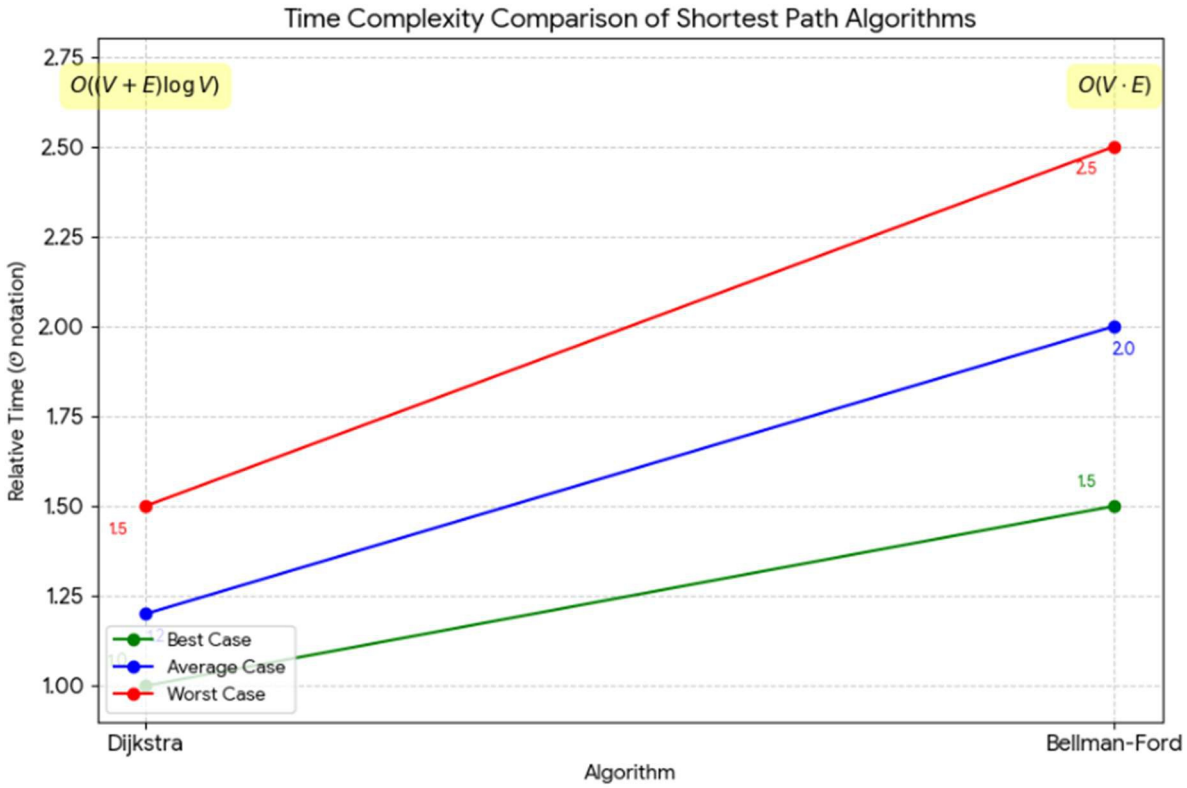
CODE:

```
#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;


void dijkstra(int V, vector<vector<pair<int,int>>>& adj, int src) {

    vector<int> dist(V, INT_MAX);

    dist[src] = 0;

    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;

    pq.push({0, src});


    while (!pq.empty()) {

        int u = pq.top().second;

        pq.pop();


        for (auto [v, w] : adj[u]) {

            if (dist[u] + w < dist[v]) {

                dist[v] = dist[u] + w;

                pq.push({dist[v], v});
```

```
            }
        }
    }

    cout << "Dijkstra's Shortest Distances:\n";
    for (int i = 0; i < V; i++)
        cout << i << " : " << dist[i] << endl;
}
void bellmanFord(int V, vector<vector<int>>& edges, int src) {
    vector<int> dist(V, INT_MAX);
    dist[src] = 0;
    for (int i = 0; i < V - 1; i++)
        for (auto e : edges)
            if (dist[e[0]] != INT_MAX && dist[e[0]] + e[2] < dist[e[1]])
                dist[e[1]] = dist[e[0]] + e[2];

    cout << "Bellman-Ford Shortest Distances:\n";
    for (int i = 0; i < V; i++)
        cout << i << " : " << dist[i] << endl;

int main() {
    int V = 5;
    vector<vector<pair<int,int>>> adj(V);
    adj[0] = {{1, 2}, {2, 4}};
    adj[1] = {{2, 1}, {3, 7}};
    adj[2] = {{4, 3}};
    adj[3] = {{4, 1}};
    dijkstra(V, adj, 0);
```

```
    vector<vector<int>> edges = {{0,1,2},{0,2,4},{1,2,1},{1,3,7},{2,4,3},{3,4,1}};
    bellmanFord(V, edges, 0);
    return 0;
}
```

OUTPUT:



PROGRAM 8

AIM: Implement N Queen's problem using Backtracking.

THEORY:

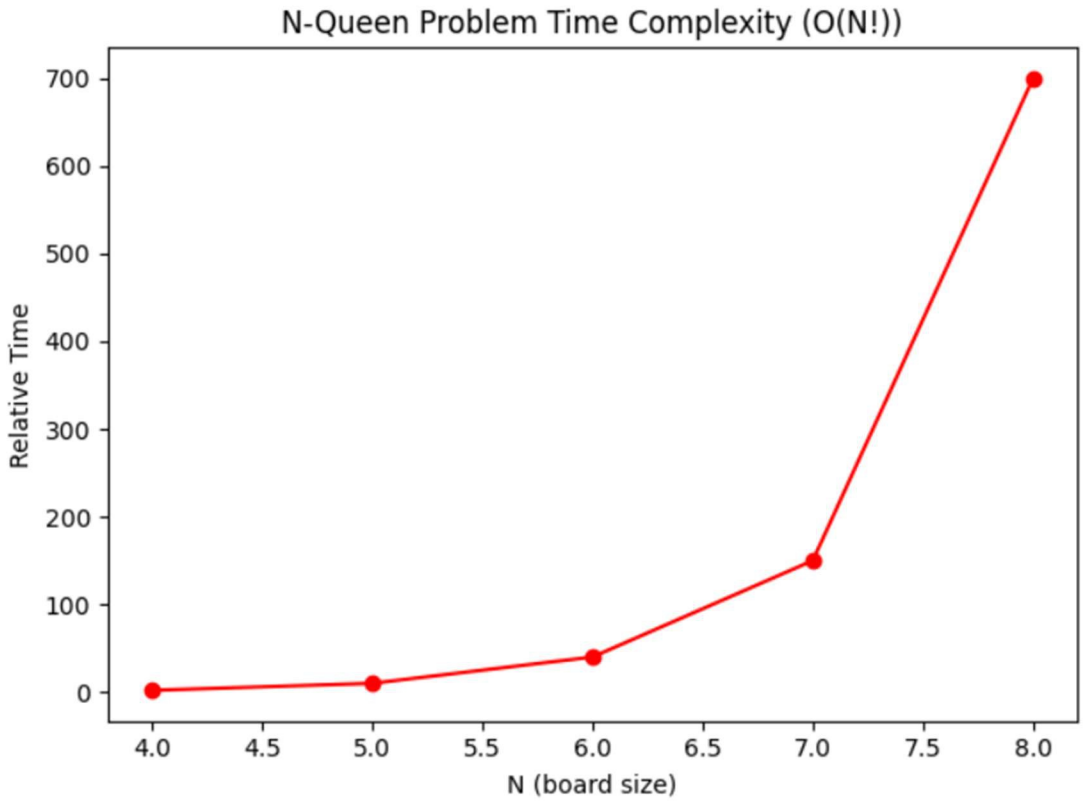
The N Queen's problem aims to place N queens on an N×N chessboard such that no two queens threaten each other.
A backtracking algorithm incrementally places queens column by column and backtracks when no valid position is found.

CODE:

```
#include <iostream>
using namespace std;

#define N 4

void printSolution(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            cout << board[i][j] << " ";
        cout << endl;
    }
}

bool isSafe(int board[N][N], int row, int col) {
    for (int i = 0; i < col; i++)
        if (board[row][i]) return false;
    for (int i=row,j=col; i>=0 && j>=0; i--,j--)
        if (board[i][j]) return false;
    for (int i=row,j=col; i<N && j>=0; i++,j--)
        if (board[i][j]) return false;
    return true;
}

bool solveNQUtil(int board[N][N], int col) {
    if (col >= N) return true;
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            if (solveNQUtil(board, col + 1)) return true;
            board[i][col] = 0;
        }
    }
    return false;
```

```
}

bool solveNQ() {
    int board[N][N] = {0};
    IF (!SOLVENQUTIL(BOARD, 0)) {
        cout << "Solution does not exist";
        return false;
    }
    printSolution(board);
    return true;
}

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

OUTPUT:



PROGRAM 9(A)

AIM: To implement Matrix Chain Multiplication and analyse its time complexity.

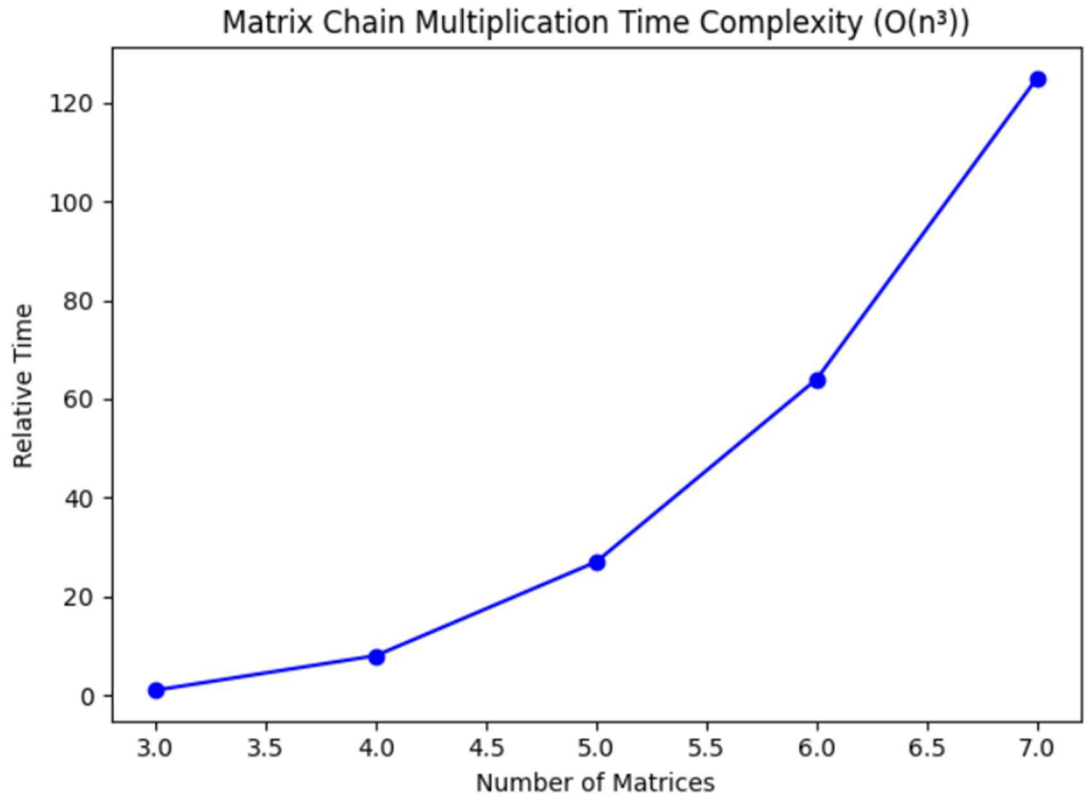
THEORY:

Matrix Chain Multiplication problem determines the most efficient way to multiply a chain of matrices.
Dynamic programming is used to find the minimum number of scalar multiplications needed.
The time complexity is O(n^3).

CODE:

```
#include <iostream>
#include <climits>
using namespace std;

int matrixChainOrder(int p[], int n) {
    int m[n][n];
    for (int i = 1; i < n; i++) m[i][i] = 0;
    for (int L = 2; L < n; L++) {
        for (int i = 1; i < n - L + 1; i++) {
            int j = i + L - 1;
            m[i][j] = INT_MAX;
            for (int k = i; k <= j - 1; k++) {
                int q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j]) m[i][j] = q;
            }
        }
    }
    return m[1][n-1];
}

int main() {
    int arr[] = {1, 2, 3, 4};
    int n = 4;
    cout << "Minimum number of multiplications is " << matrixChainOrder(arr, n);
    return 0;
}
```

OUTPUT:


Matrix Chain Multiplication Time Complexity (O(n³))
Relative Time
Number of Matrices
0
20
40
60
80
100
120
3.0
3.5
4.0
4.5
5.0
5.5
6.0
6.5
7.0

PROGRAM 9(B)

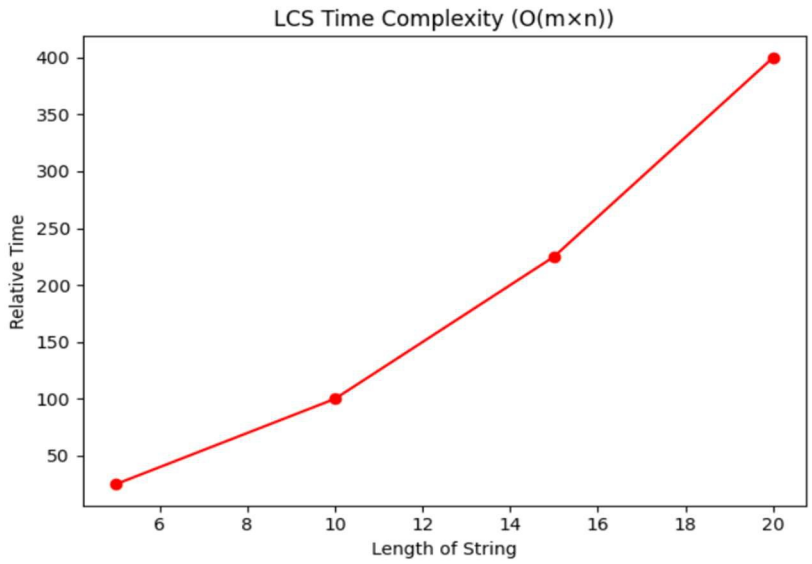
AIM: To implement Longest Common Subsequence (LCS) and analyse its time complexity.

THEORY: The Longest Common Subsequence (LCS) problem finds the longest sequence present in both given sequences in the same order.
Dynamic programming is used to compute LCS in O(mn) time complexity.

CODE:

```
#include <iostream>
#include <string>
using namespace std;

int lcs(string X, string Y, int m, int n) {
   int L[m+1][n+1];
   for (int i=0; i<=m; i++) {
      for (int j=0; j<=n; j++) {
         if (i==0 || j==0) L[i][j] = 0;
         else if (X[i-1] == Y[j-1]) L[i][j] = L[i-1][j-1] + 1;
         else L[i][j] = max(L[i-1][j], L[i][j-1]);
      }
   }
   return L[m][n];
}

int main() {
   string X = "AGGTAB", Y = "GXTXAYB";
   cout << "Length of LCS is " << lcs(X, Y, X.length(), Y.length());
   return 0;
}
```

OUTPUT:



PROGRAM 10

AIM: To implement Naïve String Matching, Rabin-Karp, and Knuth-Morris-Pratt (KMP) algorithms and analyse their time complexities.

THEORY:

String Matching algorithms are used to find occurrences of a pattern in a text.
Naïve algorithm checks for the pattern at all positions.
Rabin-Karp uses hashing for efficiency, and KMP preprocesses the pattern to skip unnecessary comparisons.
Time complexities are: Naïve - O((n−m+1)m), Rabin-Karp - O(n+m), and KMP - O(n+m).

CODE:

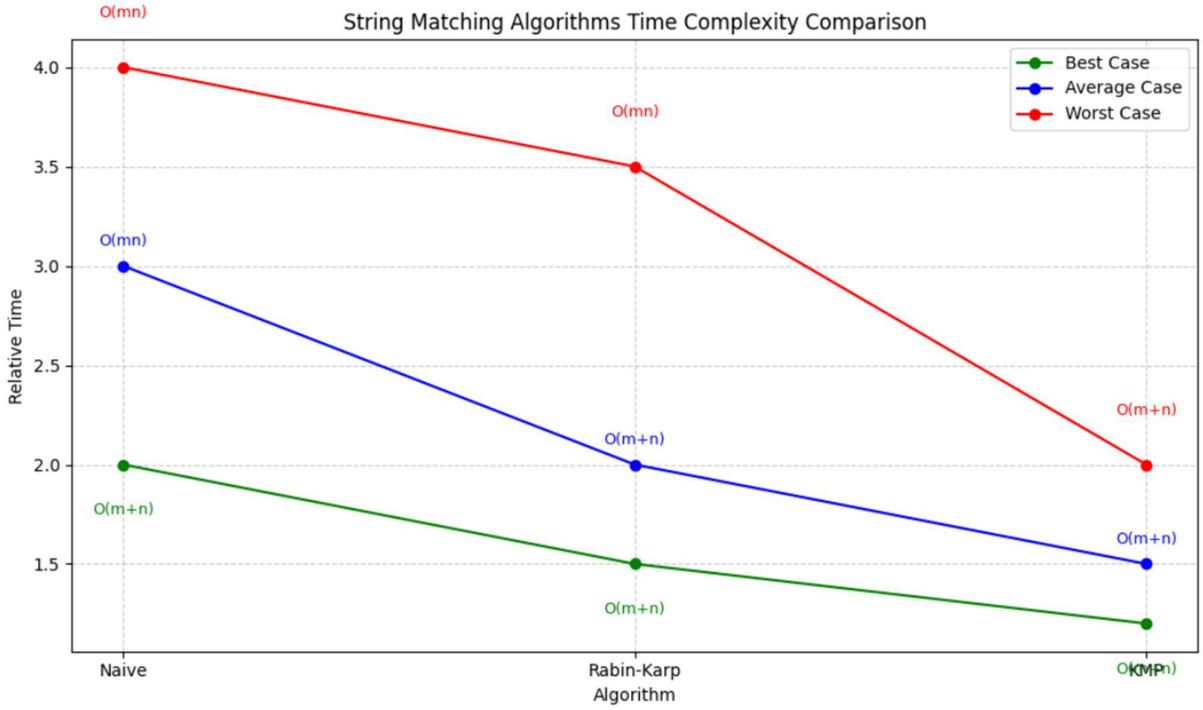
```
#include <iostream>

#include <string>

using namespace std;

void naiveSearch(string txt, string pat) {

    int n = txt.size(), m = pat.size();

    for (int i = 0; i <= n - m; i++) {

        int j;

        for (j = 0; j < m; j++)

            if (txt[i + j] != pat[j]) break;

        if (j == m) cout << "Pattern found at index " << i << endl;

    }

}

void rabinKarp(string txt, string pat, int q) {

    int n = txt.size(), m = pat.size();

    int p = 0, t = 0, h = 1;

    for (int i = 0; i < m - 1; i++)

        h = (h * d) % q;


    for (int i = 0; i < m; i++) {
```

```
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }

    for (int i = 0; i <= n - m; i++) {
        if (p == t) {
            int j;
            for (j = 0; j < m; j++)
                if (txt[i + j] != pat[j]) break;
            if (j == m) cout << "Pattern found at index " << i << endl;
        }
        if (i < n - m) {
            t = (d * (t - txt[i] * h) + txt[i + m]) % q;
            if (t < 0) t += q;
        }
    }
}
void computeLPS(string pat, int m, int* lps) {
    int len = 0;
    lps[0] = 0;
    int i = 1;
    while (i < m) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0)
```

```
            len = lps[len - 1];
        else
            lps[i++] = 0;
      }
   }
}

void KMPSearch(string pat, string txt) {
   int m = pat.size(), n = txt.size();
   int lps[m];
   computeLPS(pat, m, lps);
   int i = 0, j = 0;
   while (i < n) {
      if (pat[j] == txt[i]) i++, j++;
      if (j == m) {
         cout << "Pattern found at index " << i - j << endl;
         j = lps[j - 1];
      } else if (i < n && pat[j] != txt[i]) {
         if (j != 0) j = lps[j - 1];
         else i++;
      }
   }
}
int main() {
   string txt = "AABAACAADAABAABA";
   string pat = "AABA";
   cout << "Naive Search:\n";
   naiveSearch(txt, pat);
```

```
    cout << "\nRabin-Karp:\n";
    rabinKarp(txt, pat, 101);
    cout << "\nKMP Algorithm:\n";
    KMPSearch(pat, txt);
    return 0;
}
```

OUTPUT:



S.No	Experiment	R1	R2	R3	R4	R5	Total Marks (10)	Remarks	Faculty Signature
		Is able to identify and define the objective of the given problem?	Is proposed design /procedure /algorithm solves the problem?	Has the understanding of the tool/programming language to implement the proposed solution?	Are the result(s) verified using sufficient test data to support the conclusions?	Individuality of submission?			
		2 Marks	**2 Marks**	**2 Marks**	**2 Marks**	**2 Marks**			
1									
2									
3									
4									
5									
6									
7									

EXPERIMENT-11

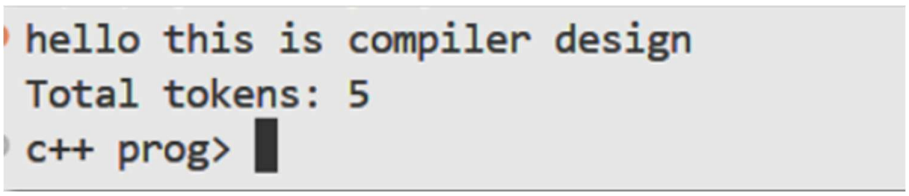
AIM: Write a program to count the number of tokens in a given string.

THEORY: Tokenization is the process of breaking source code or input text into meaningful elements called *tokens*.
In compiler design, the lexical analyzer identifies keywords, identifiers, constants, operators, and delimiters.
Each token helps the parser recognize the structure of the input according to grammar rules.

CODE:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    string s;
    getline(cin, s);
    stringstream ss(s);
    string token;
    int count = 0;
    while (ss >> token) count++;
    cout << "Total tokens: " << count << endl;
}
```

OUTPUT:



EXPERIMENT-12

AIM: Write a Lex and Yacc program to create a calculator that can evaluate an arithmetic expression.

THEORY: Lex performs lexical analysis, converting sequences of characters into tokens such as numbers and operators.
Yacc performs syntax analysis using grammar rules to compute the result of an arithmetic expression.
Together they implement a simple expression parser and evaluator.

CODE:

calc.l

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

%%
[0-9]+          { yylval = atoi(yytext); return NUMBER; }
[+\-*/()]       { return yytext[0]; }
[ \t]+          ;
\n              { return '\n'; }
.               { return yytext[0]; }
%%

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

calc.y

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

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

%token NUMBER
%left '+' '-'
%left '*' '/'

%%
input:
     /* empty */
   | input line
   ;
```

```
line:
      E '\n'   { printf("Result = %d\n", $1); }
    ;

E : E '+' E    { $$ = $1 + $3; }
  | E '-' E    { $$ = $1 - $3; }
  | E '*' E    { $$ = $1 * $3; }
  | E '/' E    { $$ = $1 / $3; }
  | '(' E ')'  { $$ = $2; }
  | NUMBER     { $$ = $1; }
  ;
%%

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

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

OUTPUT:

```
Enter expression:
5+3*2
Result = 11
```

EXPERIMENT-13

AIM: Write a Lex and Yacc program with error handling during parsing.

THEORY: Yacc provides error recovery using the reserved token error.
When a syntax error is detected, Yacc skips input until it finds a symbol that allows parsing to resume, avoiding abrupt termination.

CODE:

errorparser.l

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

%%
[0-9]+          { yylval = atoi(yytext); return NUMBER; }
[+\-*/()]       { return yytext[0]; }
[ \t]+          ;     /* Ignore whitespace */
\n              { return '\n'; }
.               { return yytext[0]; }
%%

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

errorparser.y

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

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

%token NUMBER
%left '+' '-'
%left '*' '/'

%%
input:
     /* empty */
   | input line
   ;

line:
     expr '\n'       { printf("Valid Expression\n"); }
   | error '\n'      { printf("Syntax Error (recovered)\n"); yyerrok; }
```

```
    ;

expr:
      expr '+' expr   { $$ = $1 + $3; }
    | expr '-' expr   { $$ = $1 - $3; }
    | expr '*' expr   { $$ = $1 * $3; }
    | expr '/' expr   { $$ = $1 / $3; }
    | '(' expr ')'    { $$ = $2; }
    | NUMBER          { $$ = $1; }
    ;
%%

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

int main(void){
    printf("Enter expressions (press Ctrl+C to exit):\n");
    yyparse();
    return 0;
}
```

OUTPUT:

```
Enter expressions (press Ctrl+C to exit):
5+3*2
Valid Expression
```

EXPERIMENT-14

AIM: Implement intermediate code generation for given expressions.

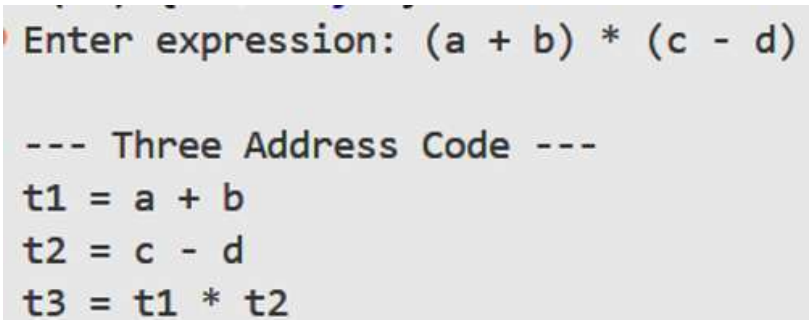
THEORY: Intermediate Code Generation converts high-level source code into an intermediate form like Three-Address Code (TAC).
It simplifies optimization and machine code generation while remaining platform-independent.

CODE:

```
#include <bits/stdc++.h>
using namespace std;

struct TAC {
   string result, arg1, op, arg2;
};

int tempCount = 1;
string newTemp() { return "t" + to_string(tempCount++); }

int precedence(char c) {
   if (c == '+' || c == '-') return 1;
   if (c == '*' || c == '/') return 2;
   return 0;
}

vector<TAC> generateTAC(string exp) {
   stack<char> op;
   stack<string> val;
   vector<TAC> code;

   for (int i = 0; i < exp.size(); i++) {
      if (isspace(exp[i])) continue;
      if (isalnum(exp[i])) {
         string operand(1, exp[i]);
         val.push(operand);
      } else if (exp[i] == '(')
         op.push('(');
      else if (exp[i] == ')') {
         while (!op.empty() && op.top() != '(') {
            string b = val.top(); val.pop();
            string a = val.top(); val.pop();
            string t = newTemp();
            code.push_back({t, a, string(1, op.top()), b});
            op.pop();
            val.push(t);
         }
         op.pop();
      } else {
```

```
            while (!op.empty() && precedence(op.top()) >= precedence(exp[i])) {
                string b = val.top(); val.pop();
                string a = val.top(); val.pop();
                string t = newTemp();
                code.push_back({t, a, string(1, op.top()), b});
                op.pop();
                val.push(t);
            }
            op.push(exp[i]);
        }
    }

    while (!op.empty()) {
        string b = val.top(); val.pop();
        string a = val.top(); val.pop();
        string t = newTemp();
        code.push_back({t, a, string(1, op.top()), b});
        op.pop();
        val.push(t);
    }

    return code;
}

int main() {
    string exp;
    cout << "Enter expression: ";
    getline(cin, exp);

    vector<TAC> code = generateTAC(exp);

    cout << "\n--- Three Address Code ---\n";
    for (auto &c : code)
        cout << c.result << " = " << c.arg1 << " " << c.op << " " << c.arg2 << "\n";
}
```

OUTPUT:



EXPERIMENT-15

AIM: Parse the expression ((a + b) / c) * (d - e) and construct its syntax tree.

THEORY: A syntax tree (parse tree) is a hierarchical representation of program structure according to the grammar rules of the language.
Each interior node represents an operator; each leaf node represents an operand.

CODE:

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    string val;
    Node *left, *right;
    Node(string v) : val(v), left(NULL), right(NULL) {}
};

Node* buildSyntaxTree() {
    Node* a = new Node("a");
    Node* b = new Node("b");
    Node* c = new Node("c");
    Node* d = new Node("d");
    Node* e = new Node("e");

    Node* plus = new Node("+");
    plus->left = a;
    plus->right = b;

    Node* divide = new Node("/");
    divide->left = plus;
    divide->right = c;

    Node* minus = new Node("-");
    minus->left = d;
    minus->right = e;

    Node* multiply = new Node("*");
    multiply->left = divide;
    multiply->right = minus;

    return multiply;     // root of the tree
}

void printTree(Node* root, int depth = 0) {
    if (!root) return;
    for (int i = 0; i < depth; i++) cout << "    ";
    cout << root->val << "\n";
```

```
    printTree(root->left, depth + 1);
    printTree(root->right, depth + 1);
}

int main() {
    Node* root = buildSyntaxTree();
    cout << "Syntax Tree for expression ((a + b) / c) * (d - e):\n";
    printTree(root);
    return 0;
}
```

OUTPUT:

```
Syntax Tree for expression ((a + b) / c) * (d - e):
*
   /
      +
         a
         b
      c
   -
      d
      e
c++ prog>
```

EXPERIMENT-16

AIM: Construct a recursive descent parser for an arithmetic expression.

THEORY:

A recursive descent parser is a top-down parser built from a set of mutually recursive functions, where each function corresponds to one non-terminal in the grammar.

Each function:

- Consumes tokens from the input string,
- Matches grammar rules, and
- Reports syntax errors if the input doesn't follow the grammar.

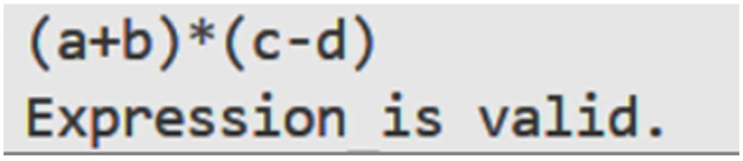
CODE:

```
#include <bits/stdc++.h>
using namespace std;

string input;
int pos = 0;

void E();
void Eprime();
void T();
void Tprime();
void F();

void error() {
   cout << "Syntax Error at position " << pos << endl;
   exit(0);
}

char lookahead() {
   if (pos < input.size()) return input[pos];
   return '$';
}

void match(char expected) {
   if (lookahead() == expected)
      pos++;
   else
      error();
}

void E() {
   T();
   Eprime();
}
```

```
void Eprime() {
    if (lookahead() == '+') {
        match('+');
        T();
        Eprime();
    } else if (lookahead() == '-') {
        match('-');
        T();
        Eprime();
    }
}

void T() {
    F();
    Tprime();
}

void Tprime() {
    if (lookahead() == '*') {
        match('*');
        F();
        Tprime();
    } else if (lookahead() == '/') {
        match('/');
        F();
        Tprime();
    }
}

void F() {
    if (lookahead() == '(') {
        match('(');
        E();
        match(')');
    } else if (isalpha(lookahead())) {
        match(lookahead());
    } else {
        error();
    }
}

int main() {
    cin >> input;
    E();
    if (lookahead() == '$' || pos == input.size())
        cout << "Expression is valid.\n";
    else
```

```
        cout << "Invalid expression.\n";
}
```

OUTPUT:



EXPERIMENT-17

AIM: Design and implement a symbol table using a hash table.

THEORY: A Symbol Table is a data structure used by a compiler to store information about identifiers such as variables, constants, functions, and classes.
It maps each identifier to its associated attributes (like type, scope, and memory location).
The table supports efficient insertion, lookup, and deletion operations that occur during different compiler phases.

Purpose in Compiler Design

1. Lexical Analysis: Detects new identifiers and inserts them into the table.
2. Syntax Analysis: Checks consistency of variable names.
3. Semantic Analysis: Verifies type compatibility and scope.
4. Code Generation: Retrieves address and type information for machine code output.

CODE:

```
#include <bits/stdc++.h>
using namespace std;

struct Symbol {
   string name, type;
   int scope;
};

class SymbolTable {
   vector<list<Symbol>> table;
   int size;
   hash<string> hashFunc;

public:
   SymbolTable(int s = 101) {
      size = s;
      table.resize(size);
   }

   int hashIndex(const string &key) {
      return hashFunc(key) % size;
   }

   void insert(const string &name, const string &type, int scope) {
      int idx = hashIndex(name);
      for (auto &sym : table[idx])
         if (sym.name == name && sym.scope == scope)
            return;
      table[idx].push_back({name, type, scope});
   }
```

```
    Symbol* lookup(const string &name) {
        int idx = hashIndex(name);
        for (auto &sym : table[idx])
            if (sym.name == name)
                return &sym;
        return nullptr;
    }

    void remove(const string &name, int scope) {
        int idx = hashIndex(name);
        auto &lst = table[idx];
        for (auto it = lst.begin(); it != lst.end(); ++it) {
            if (it->name == name && it->scope == scope) {
                lst.erase(it);
                return;
            }
        }
    }

    void display() {
        for (int i = 0; i < size; i++) {
            if (!table[i].empty()) {
                cout << "Bucket " << i << ": ";
                for (auto &sym : table[i])
                    cout << "(" << sym.name << ", " << sym.type << ", " << sym.scope << ") ";
                cout << "\n";
            }
        }
    }
};

int main() {
    SymbolTable st;
    st.insert("x", "int", 0);
    st.insert("y", "float", 0);
    st.insert("x", "int", 1);
    st.display();
    Symbol* s = st.lookup("x");
    if (s) cout << "Found: " << s->name << " " << s->type << " scope " << s->scope << "\n";
    st.remove("x", 1);
    st.display();
}
```

OUTPUT:

```
Bucket 35: (x, int, 0) (x, int, 1)
Bucket 81: (y, float, 0)
Found: x int scope 0
Bucket 35: (x, int, 0)
Bucket 81: (y, float, 0)
c++ prog>
```

EXPERIMENT-18

AIM: Apply loop-invariant code motion to optimize a loop.

```
for (i = 0; i < n; i++) {
x = 10 * a;
y[i] = x + b[i];
}
```

THEORY: Loop-Invariant Code Motion (LICM) is a compiler optimization technique that moves computations that produce the same result in every iteration of a loop outside the loop.
This reduces redundant computations, improving execution efficiency.

CODE:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
   int n = 5, a = 3;
   int b[] = {1, 2, 3, 4, 5}, y[5];
   int x = 10 * a;
   for (int i = 0; i < n; i++) {
      y[i] = x + b[i];
   }
   cout << "Optimized Loop Output:\n";
   for (int i = 0; i < n; i++) cout << y[i] << " ";
   cout << endl;
}
```

OUTPUT:

```
Optimized Loop Output:
31 32 33 34 35
c++ prog>
```