

1.

AIM:

Implementation of Language recognizer for set of all strings over input alphabet  $\Sigma=\{a,b\}$  containing even number of a's and even number of b's.

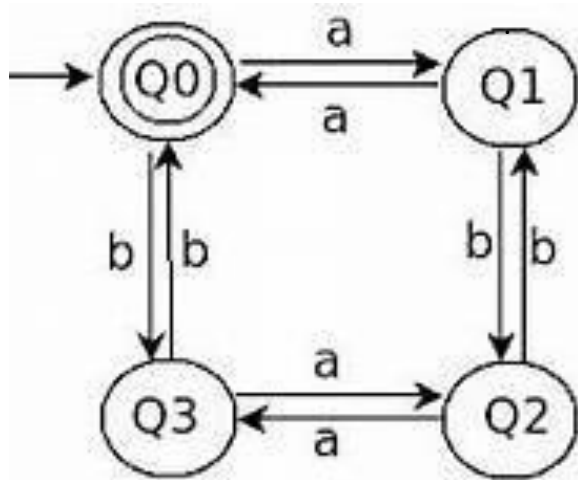
Description:

The acceptable strings of the language are  $\epsilon$ (Null string), aa, bb, abba, babbab

etc. Deterministic Finite Automata for the given language is given below:

DFA  $M=(Q,\Sigma,\delta,Q_0,F)$  Where  
 $Q$ =Set of all states  $=\{Q_0,Q_1,Q_2,Q_3\}$   
 $\Sigma$ =Input Alphabet $=\{a,b\}$ ,  
Start state is  $Q_0$   
 $F$ =Set of all final States $=\{Q_0\}$

And the transitions are defined  
in the transition diagram



---

Algorithm: Language recognizer

Input:

*input //input string*

Output:

Algorithm prints a message

“String accepted”: If the input is acceptable by the  
language, “String not accepted” otherwise,

“Invalid token”: If the input string contains symbols other than input alphabet.

---

Method:

```
state=0 //initial state
while((current=input[i++])!='\0'){
    switch(state)
        case 0: if(current=='a')    state=1;
                else if(current=='b') state=2;
                else
                    Print "Invalid token" ; exit;
        case 1: if(current=='a')    state=0;
                else if(current=='b') state=3;
                else
                    Print "Invalid token" ; exit;
        case 2: if(current=='a')    state=3;
                else if(current=='b') state=0;
                else
                    Print "Invalid token" ; exit;
        case 3: if(current=='a')    state=2;
                else if(current=='b') state=1;
                else
                    Print "Invalid token" ; exit;
    end switch
end while
//Print output
if(state==0)
    Print "String accepted"
else
    Print "String not accepted"
```

---

## C Code

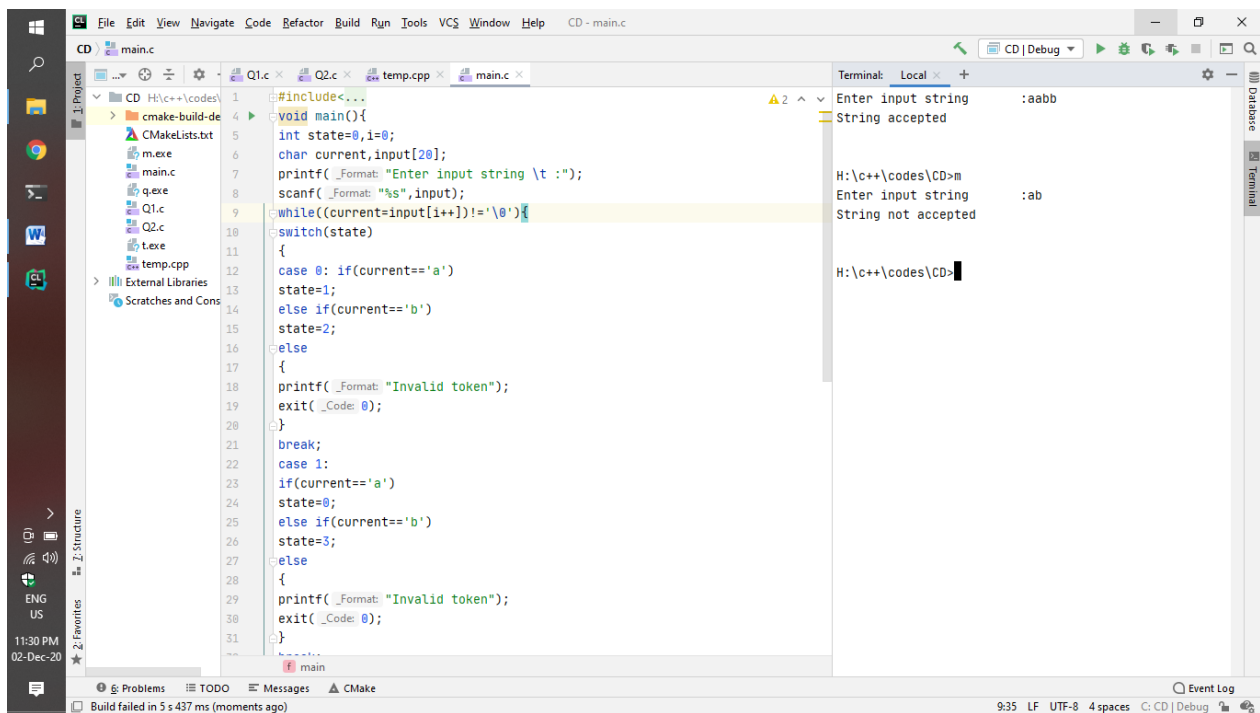
```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void main() {
    int state=0,i=0;
    char current,input[20];
    printf("Enter input string \t :");
    scanf("%s",input);
    while( (current=input[i++]) !='\0' ) {
        switch(state)
        {
            case 0: if(current=='a')
                    state=1;
                    else if(current=='b')
                    state=2;
                    else
                    {
                        printf("Invalid token");
                        exit(0);
                    }
                    break;
            case 1:
                    if(current=='a')
                    state=0;
                    else if(current=='b')
                    state=3;
                    else
                    {
                        printf("Invalid token");
```

```

        exit(0);
    }
    break;
case 2: if(current=='a')
        state=3;
    else if(current=='b') state=0;
    else
    {
        printf("Invalid token");
        exit(0);
    }
    break;
case 3: if(current=='a') state=2;
    else if(current=='b') state=1;
    else
    {
        printf("Invalid token");
        exit(0);
    }
    break;
}
}
if(state==0)
    printf("String accepted\n\n"); else
    printf("String not accepted\n\n");
}

```

OUTPUT:



2.

AIM:

Implementation of Language recognizer for set of all strings ending with two symbols of same type.

DESCRIPTION:

The acceptable strings of the language are Null string, aa, bb, abb, baa, abaa, abbaabb ....

Deterministic Finite Automata for the given language is

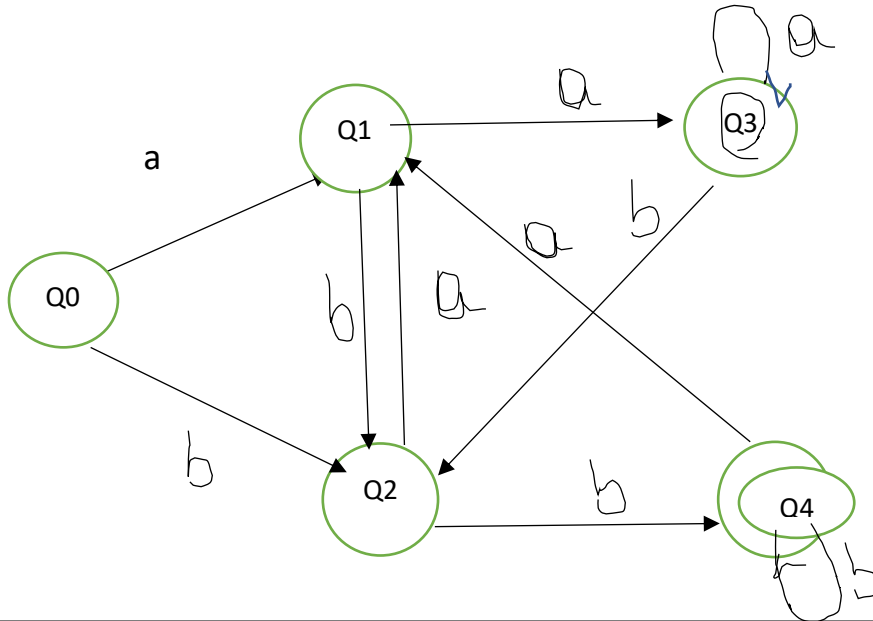
DFA  $M=(Q, \Sigma, \delta, Q_0, F)$  where

$Q$ =set of all states=  $\{Q_0, Q_1, Q_2, Q_3, Q_4\}$

$\Sigma$ =input Alphabet=  $\{a, b\}$

Start state is  $Q_0$

$F$ =set of all final states= $\{Q_3, Q_4\}$



---

## Algorithm: Language recognizer

### Input:

*input* //input string

### Output:

Algorithm prints a message

“String accepted”: If the input is acceptable by the language,

“String not accepted” otherwise,

“Invalid token”: If the input string contains symbols other than input alphabet.

---

### Method:

```
state=0 //initial state
while((current=input[i++])!='\0'){
    switch(state)
        case 0: if(current=='a')    state=1;
                 else if(current=='b') state=2;
                 else
                     Print "Invalid token" ; exit;
        case 1: if(current=='a')    state=3;
                 else if(current=='b') state=2;
                 else
                     Print "Invalid token" ; exit;
        case 2: if(current=='a')    state=1;
```

```

        else if(current=='b')    state=4;
        else
            Print "Invalid token" ; exit;
    case 3: if(current=='a')        state=3;
        else if(current=='b')    state=2;
        else
            Print "Invalid token" ; exit;
    end switch
end while
//Print output
if(state==0)
    Print "String accepted"
else
    Print "String not accepted"

```

---

### C-PROGRAM

```

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void main() {
    int state=0, i=0;
    char token, input[20];
    printf("Enter input string \t :");
    scanf("%s", input);
    while((token=input[i++]) != '\0') {
        switch(state)
        {
            case 0: if(token=='a')
                    state=1;
                    else if(token=='b')
                    state=2;
                    else
                    {
                        printf("Invalid token");
                        exit(0);
                    }
                    break;
            case 1: if(token=='a')
                    state=3;
                    else if(token=='b')
                    state=2;
                    else
                    {
                        printf("Invalid token");
                        exit(0);
                    }
                    break;
            case 2:
                    if(token=='a')
                    state=1;
                    else if(token=='b')
                    state=4;
                    else
                    {
                        printf("Invalid token");
                        exit(0);
                    }
                    break;
            case 3:
                    if(token=='a')
                    state=3;
                    else if(token=='b')

```

```

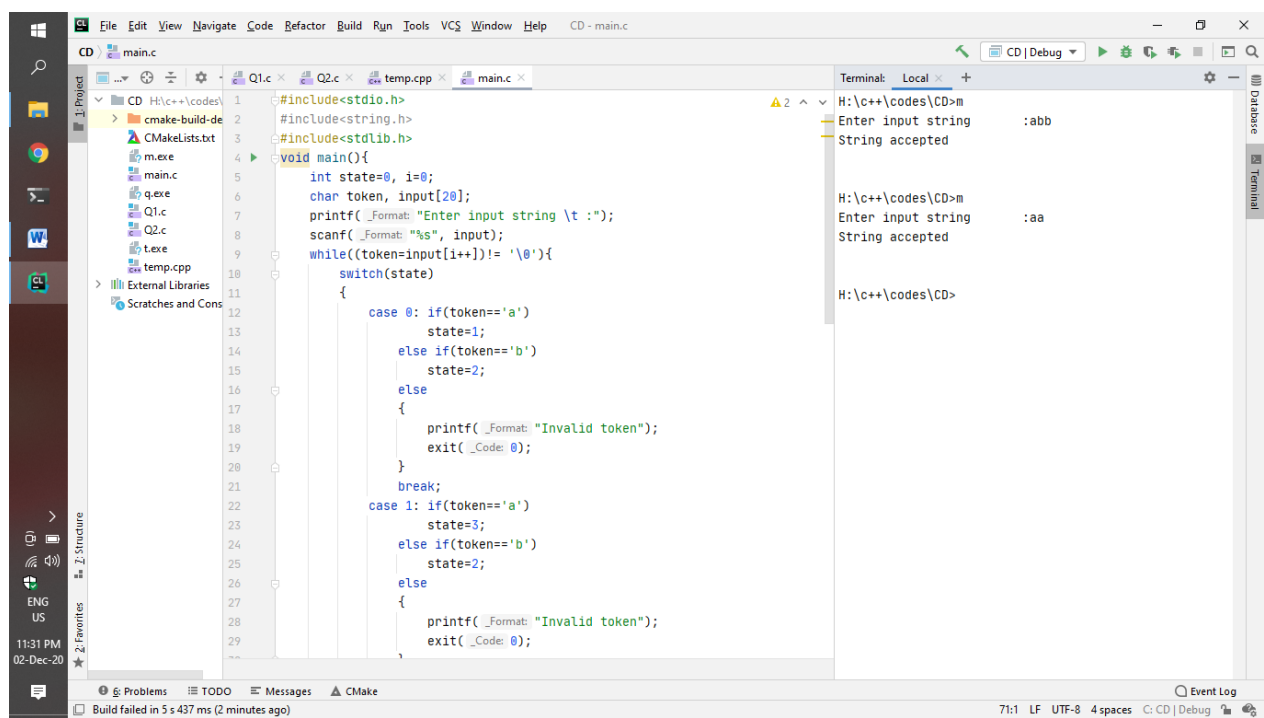
        state=2;
    else
    {
        printf("Invalid token");
        exit(0);
    }
    break;
case 4:
    if(token=='a')
        state=1;
    else if(token=='b')
        state=4;
    else
    {
        printf("Invalid token"); exit(0);
    }
    break;
}
}
if(state==3||state==4)
    printf("String accepted\n\n");
else
    printf(" String not accepted\n\n");
}

```

OUTPUT:

Test cases:

<u>Input</u>	<u>Expected Output</u>
aa	String accepted
bb	String accepted
abab	String not accepted
abcaa	Invalid token



## WEEK-2:

### Problem 2.a

Aim:

Identification of consonants and vowels

Code:

```
%{  
void display(int);  
%}  
%%  
[a|e|i|o|u|A|E|I|O|U]{  
int flag=1;  
display(flag);  
return;  
}  
[a-z A-Z(^a|e|i|o|u|A|E|I|O|U)] {  
int flag=0;  
display(flag);  
return;  
}  
%%  
void display (int flag)  
{  
if(flag==1)  
printf("%s is a vowel",yytext);  
else  
printf("%s is consonant",yytext);  
}  
main()  
{  
printf("enter a alphabet to check if it is vowel or consonant");  
yylex();  
}
```

OUTPUT:

enter a alphabet to check if it is vowel or consonant A

A is vowel

enter a alphabet to check if it is vowel or consonant B

B is consonent

Program 2.b:

Aim: Count number of vowels and consonants

Algorithm:

1. Define a string.
2. Convert the string to lowercase so that comparisons can be reduced. Else we need to compare with capital (**A, E, I, O, U**).
3. If any character in string matches with vowels (**a, e, i, o, u**) then increment the vcount by 1.
4. If any character lies between 'a' and 'z' except vowels, then increment the count for ccount by 1.
5. Print both the counts.

**Code:**

```
%{  
  
    int vow_count=0;  
  
    int const_count =0;  
  
}%  
  
%%  
  
[aeiouAEIOU] {vow_count++;}  
  
[a-zA-Z] {const_count++;}  
  
%%  
  
int yywrap(){}  
  
int main()  
  
{  
  
    printf("Enter the string of vowels and consonents:");  
  
    yylex();  
  
    printf("Number of vowels are: %d\n", vow_count);  
  
    printf("Number of consonants are: %d\n", const_count);  
  
    return 0;
```



}

## OUTPUT

```
C:\Users\hp\Desktop\flex>lex vowel1.1
C:\Users\hp\Desktop\flex>gcc lex.yy.c
C:\Users\hp\Desktop\flex>a.exe
Enter the string of vowels and consonants:djfkd
hai
Number of vowels are: 2
Number of consonants are: 6
```

Program 2c:

Aim:

Count the number of Lines in given input.

Algorithm:

```
begin
num_lines=0
num_chars=0
if new line
then num_lines++
end
```

LOGIC:

Read each character from the text file :

- Is it a capital letter in English? [A-Z] : increment capital letter count by 1.
- Is it a small letter in English? [a-z] : increment small letter count by 1
- Is it [0-9]? increment digit count by 1.
- All other characters (like '!', '@', '&') are counted as special characters

- How to count the number of lines? we simply count the encounters of '\n' <newline> character.that's all!!
- To count the number of words we count white spaces and tab character(of course, newline characters too..)
- 

### Code:

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

int lines=0, words=0,s_letters=0,c_letters=0, num=0, spl_char=0,total=0;
%}

%%

\n { lines++; words++;}

[\t ' '] words++;

[A-Z] c_letters++;

[a-z] s_letters++;

[0-9] num++;

. spl_char++;

%%

main(void)
{
yyin= fopen("myfile.txt","r");
yylex();

total=s_letters+c_letters+num+spl_char;

printf(" This File contains ...");

printf("\n\t%d lines", lines);

printf("\n\t%d words",words);

printf("\n\t%d small letters", s_letters);
```

```

printf("\n\t%d capital letters",c_letters);

printf("\n\t%d digits", num);

printf("\n\t%d special characters",spl_char);

printf("\n\tIn total %d characters.\n",total);

}

int yywrap()

{

return(1);

}

```

#### Sample Output:

```

C:\Users\hp\Desktop\flex>lex vowel2.1
C:\Users\hp\Desktop\flex>gcc lex.yy.c
C:\Users\hp\Desktop\flex>a.exe
This File contains ...
    4 lines
    4 words
   19 small letters
    0 capital letters
    1 digits
    0 special characters
In total 20 characters.

```

#### Program 2d:

##### AIM:

Recognize strings ending with 00

DFA in LEX code which accepts strings ending with 00

Prerequisite: Designing Finite Automata

Problem: Design a LEX code to construct a DFA which accepts the language: all the strings ending with "11" over inputs '0' and '1'.

Examples:

**Input:** 100100

**Output:** Accepted

**Input:** 100101

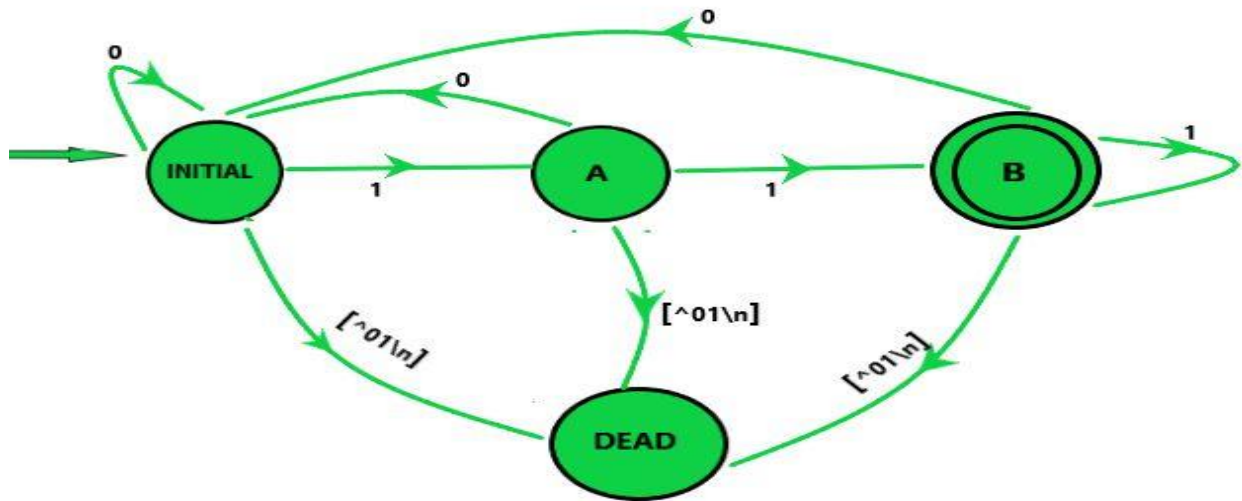
**Output:** Not Accepted

**Input:** asdf

**Output:** Invalid

Approach:

LEX provides us with an INITIAL state by default. So in order to make a DFA, use this as the initial state of the DFA. Now we define three more states A, B and DEAD where DEAD state would be use if encounter a wrong or invalid input. When user input invalid character, move to DEAD state and print message "INVALID" and if input string ends at state B then display a message "Accepted". If input string ends at state INITIAL and A then display a message "Not Accepted".



**Lex code:**

```

%{
%}

%s A B DEAD

%%

<INITIAL>0 BEGIN A;
<INITIAL>1 BEGIN INITIAL;
<INITIAL>[^01\n] BEGIN DEAD;
<INITIAL>\n BEGIN INITIAL; {printf("Not Accepted\n");}

<A>0 BEGIN B;
<A>1 BEGIN INITIAL;
<A>[^01\n] BEGIN DEAD;
<A>\n BEGIN INITIAL; {printf("Not Accepted\n");}

<B>0 BEGIN B;
<B>1 BEGIN INITIAL;
<B>[^01\n] BEGIN DEAD;
<B>\n BEGIN INITIAL; {printf("Accepted\n");}

<DEAD>[^\\n] BEGIN DEAD;
<DEAD>\n BEGIN INITIAL; {printf("Invalid\n");}

```

```

%%

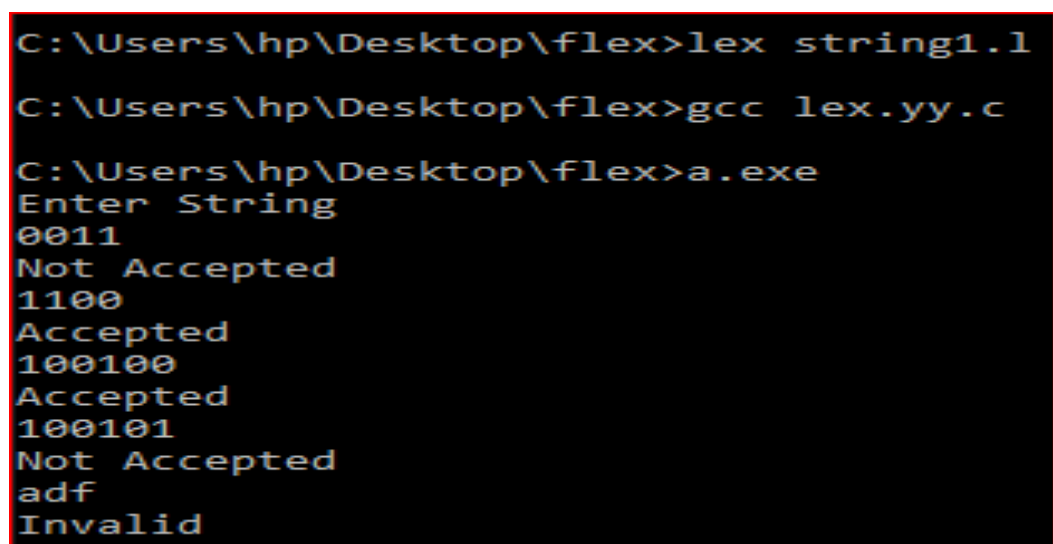
int main()
{
    printf("Enter String\n");

    yylex();
}

int yywrap()
{
    return 1;
}

```

Output:



```

C:\Users\hp\Desktop\flex>lex string1.l
C:\Users\hp\Desktop\flex>gcc lex.yy.c
C:\Users\hp\Desktop\flex>a.exe
Enter String
0011
Not Accepted
1100
Accepted
100100
Accepted
100101
Not Accepted
adf
Invalid

```

---

AIM:

Write a Program to Design Lexical Analyzer by using LEX Tool.

**Description:** The main motto of Lexical Analyzer program is identifying the strings as keywords, identifiers and symbols as Operators or special characters.

We have 7 phases in Compiler Design. Among those 7 phases, the first phase is Lexical Analysis. So, We are designing the Lexical Analyzer for the Lexical Analysis phase in Compiler Design.

**What is LEX Tool:** Lex is a computer program that generates lexical analyzers ("scanners" or "lexers"). Lex is commonly used with the yacc parser generator. Lex, originally written by Mike Lesk and Eric Schmidt and described in 1975, is the standard lexical analyzer generator on many Unix systems, and an equivalent tool is specified as part of the POSIX standard. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

Here, to write this program we have to follow some structure i.e, we need 3 sections to write the program, those are: I. Declaration Section

ii. Transition rules Section

iii. Auxiliary Function Section

Each Section ends with the symbol `“%%”` (A pair of percentages)

### **Declaration Section:**

The declarations section consists of two parts, regular definitions and auxiliary declarations. LEX allows the use of short-hands and extensions to regular expressions for the regular definitions. The auxiliary declarations are copied as such by LEX to the output `lex.yy.c` file.

### **Example**

```
%{  
  
#include int global_variable;    //Auxiliary declarations  
  
%}  
  
number [0-9]+                    //Regular definitions  
  
op [-|+|*|/|^|=]  
  
%%  
  
/* Rules */  
  
%%  
  
/* Auxiliary functions */
```

A regular definition in LEX is of the form : `D R` where `D` is the symbol representing the regular expression `R`. The auxiliary declarations (which are optional) are written in C language and are enclosed within `'%{'` and `'%}'`. It is generally used to declare functions, include header files, or define global variables and constants.

### **Transition rules Section:**

Rules in a LEX program consists of two parts:

- i. The pattern to be matched
- ii. The corresponding action to be executed

### **Example:**

```

/* Declarations */

%%

{number} {printf(" number");}

{op} {printf(" operator");}

%%

/* Auxiliary functions */

```

The pattern to be matched is specified as a regular expression

LEX obtains the regular expressions of the symbols number and op from the declarations section and generates code into a function yylex() in the lex.yy.c file. This function checks the input stream for the first match to one of the patterns specified and executes code in the action part corresponding to the pattern.

### Auxiliary functions:

LEX generates C code for the rules specified in the Rules section and places this code into a single function called yylex(). (To be discussed in detail later). In addition to this LEX generated code, the programmer may wish to add his own code to the lex.yy.c file. The auxiliary functions section allows the programmer to achieve this.

### Example:

```

/* Declarations */

%%

/* Rules */

%%

int main()
{
    yylex();
    return 1;
}

```

The C code in the auxiliary section and the declarations in the declaration section are copied as such to the lex.yy.c file.

### Lex Code:

```

digit [0-9]*

id [a-zA-Z][a-zA-Z0-9]*

```

```
num [0-9]*\.[0-9]*
```

```
%{
```

```
%}
```

```
%%
```

```
int |
```

```
float |
```

```
char |
```

```
double |
```

```
void |
```

```
main { printf("\n %s is keyword",yytext);}
```

```
"<=" {printf("\n %s is Relational operator Lessthan or Equal to",yytext);}
```

```
"<" {printf("\n %s is Relational operator Lessthan",yytext);}
```

```
">=" {printf("\n %s is Relational operator Greaterthan or Equal to",yytext);}
```

```
">" {printf("\n %s is Relational operator Greaterthan",yytext);}
```

```
"==" {printf("\n %s is Relational operator Equal to",yytext);}
```

```
"!=" {printf("\n %s is Relational operator Not Equal to",yytext);}
```

```
{id} { printf("\n %s is identifier",yytext); }
```

```
{num} { printf("\n %s is float",yytext);}
```

```
{digit} {printf("\n %s is digit",yytext);}
```

```
%%
```

```
int main()
```

```
{
```

```
yylex();
```

```
}
```

```
int yywrap()
```

```
{
```

```
return 1;
```

```
}
```

### Test cases for input and output:

I: 234 O: number

I: int O: keyword



l: 2>=1 number relation operator number

OUTPUT:

```
C:\Users\hp\Desktop\flex>lex exp3a.l
C:\Users\hp\Desktop\flex>gcc lex.yy.c
C:\Users\hp\Desktop\flex>a.exe
void

void is keyword
int

int is keyword
3>=2

3 is digit
>= is Relational operator Greaterthan or Equal to
2 is digit
5.5<6.1

5.5 is float
< is Relational operator Lessthan
6.1 is float
5==5

5 is digit
== is Relational operator Equal to
5 is digit
5!=3

5 is digit
!= is Relational operator Not Equal to
3 is digit
```

## 2. Dealing with comments

Lex code:

```
digit [0-9]*
id [a-zA-Z][a-zA-Z0-9]*
num [0-9]*\.[0-9]*

%{
int cnt=0,n=0,com=0,scom=0;

%}

%%

\n {scom=0;n++;}

"//" {scom=1;printf("\n single line comment\n\n");}

"/*" {com=1;printf("\n comment start\n");}

"*/" {com=0;printf("\n comment end\n");}

int |

float |
```

```

char |
double |
void |

main { if(!com&&!scom) printf(" \n %s is keyword",yytext);}

"<=" {if (!com&&!scom) printf("\n %s is Relational operator Lessthan or Equal to",yytext);}

"<" {if(!com&&!scom) printf("\n %s is Relational operator Lessthan",yytext);}

">=" {if(!com) printf("\n %s is Relational operator Greaterthan or Equal to",yytext);}

">" {if(!com&&!scom) printf("\n %s is Relational operator Greaterthan",yytext);}

"==" {if(!com&&!scom) printf("\n %s is Relational operator Equal to",yytext);}

"!=" {if (!com&&!scom) printf("\n %s is Relational operator Not Equal to",yytext);}

{id} { if(!com&&!scom) printf("\n %s is identifier",yytext); }

{num} { if(!com&&!scom) printf("\n %s is float",yytext);}

{digit} {if (!com&&!scom) printf("\n %s is digit",yytext);}

%%

int main()

{

yylex();

printf(" \n no of lines = %d\n",n);

return 0;

}

int yywrap()

{

return 1;

}

```

Sample Input and Output:

```

C:\Users\hp\Desktop\flex>lex exp3b.l
C:\Users\hp\Desktop\flex>gcc lex.yy.c
C:\Users\hp\Desktop\flex>a.exe
//swap two

single line comment

/*swap two

comment start
*/

comment end

```

## WEEK-4:

### AIM

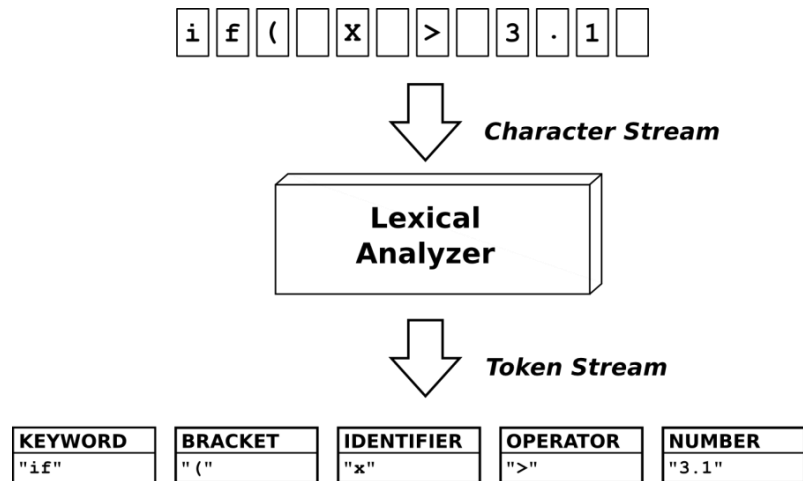
Design a Lexical analyzer for identifying different types of tokens used in C language. Make a note of the following constraints.

- Comment line and whitespace must be eliminated
- The reserved keywords such as if, else, for, int, return, etc., must be reported as invalid identifiers.
- C allows identifier names to begin with underscore character too.
- Print the number of tokens present in the input C program.

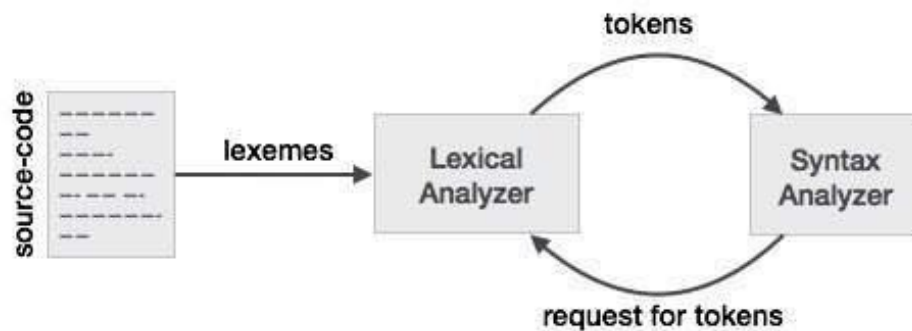
### DESCRIPTION ABOUT LEXICAL ANALYZER

- Lexical Analysis is the first phase of compiler also known as scanner. It converts the input program into a sequence of **Tokens**.
- **What is a token?** A lexical token is a sequence of characters that can be treated as a unit in the Regular Expression/grammar of the programming languages.
- **Example of tokens:**
  - ✓ Keywords - for, while, if etc.,
  - ✓ Identifier - Variable name, function name etc.,
  - ✓ Operators - '+', '/', '-', etc.,
  - ✓ Separators - ',', ';', etc.,
- **Example of Non-Tokens:**
  - ✓ Comments, preprocessor directive, macros, blanks, tabs, newline, etc.,
- Lexical analyzer (or scanner) is a program to recognize tokens (also called symbols) from an input source file (or source code). Each token is a meaningful character string, such as a number, an operator, or an identifier.
- Lexical analyzer converts lexemes into tokens i.e. it converts input source program into stream of tokens.
- It reads program text line by line and character by character
  - ✓ Removes comment line
  - ✓ Removes white space
- It reads program line by line so that we get error details with line number. • It also provides help in generating error message by providing row number and column number.

- **Work flow of lexical analyzer:**



- Its main task is to read input characters and produce as output a sequence of tokens that parser uses for syntax analysis.



# ALGORITHM

1. Get the space separated input C program
2. Read the input string left to right character by character for performing Tokenization  
.i.e. Dividing the program into valid tokens.
3. Check for identifiers by finding a string starts with an alphabet by using `isalpha()`, followed by alphabet or number or underscore.
4. Check for literal by finding a string constant enclosed with in double quotes.
5. Check for operators such as `+`, `-`, `*`, `/`
6. Check for delimiters by identifying special symbols such as `{`, `:`, `;`, `{`, `}`, `(`, `)`, `,`
7. Check for constants by identifying numbers by using `isdigit()` function
8. Remove white space characters.
9. Remove comments.
10. Print the various tokens by removing duplications.

## C-program

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
int CHECK_KEYWORD(char *);
int CHECK_IDENTIFIER(char *);
int CHECK_CONST(char *);
int CHECK_SPL(char *);
int CHECK_ARTH(char *);
int CHECK_RELATION(char *);
int CHECK_COMMENT(char *);
char Keywords[32][10] = {"auto", "double", "int", "struct", "break", "else",
"long", "switch",
                        "case", "enum", "register", "typedef", "char",
"extern", "return", "union",
                        "continue", "for", "signed", "void", "do", "if",
"static", "while", "default",
                        "goto", "sizeof", "volatile", "const", "float",
"short", "unsigned"};
char SPL[13] = {'$', '#', '&', '^', '{', '}', '[', ']', '?', ':', '(', ')',
';'};
char ARTH[6] = {'+', '-', '*', '/', '%'};
char RELATION1[2] = {'>', '<'};
char RELATION2[4][2] = {">=", "<=", "==", "!="};
int count = 0;
int main() {
    char get_data[20];
    printf("Enter Data : ");
    gets(get_data);
    int len = strlen(get_data);
    int i = 0;
    while (1) {
        char temp[20] = {'\0'};
        int j = 0;
        while (i-1 != len) {
            if (isspace(get_data[i]))
                break;
            temp[j] = get_data[i];
            j++;
            i++;
        }
        while (1) {
            if (CHECK_KEYWORD(temp))
                break;
            if (CHECK_IDENTIFIER(temp))
                break;
            if (CHECK_SPL(temp))
                break;
            if (CHECK_ARTH(temp))
                break;
            if (CHECK_RELATION(temp))
                break;
            if (CHECK_COMMENT(temp))
                break;
            if (CHECK_CONST(temp))
                break;
        }
        if (i++ >= len)
            break;
    }
    return 0;
}
int CHECK_KEYWORD(char data[]) {
    int i;
    for (i = 0; i < 32; i++) {
        if (strcmp(data, Keywords[i]) == 0) {
```

```

        printf("%s is a Keyword\n",data);
        return 1;
    }
}
return 0;
}
int CHECK_IDENTIFIER(char ptr[]) {
    if (isalpha(ptr[0]) || ptr[0] == '_' ) {
        printf("%s is a IDENTIFIER\n",ptr);
        return 1;
    }
    return 0;
}
int CHECK_CONST(char ptr[]) {
    int len = strlen(ptr), flag = 0;
    int i;
    for (i = 0; i < len; i++) {
        if (ptr[i] == '.' || ptr[i] == 'E' || ptr[i] == 'e') {
            flag = 1;
            break;
        }
    }
    if (flag) {
        printf("%s is a floating number\n",ptr);
        return 1;
    } else {
        printf("%s is a Integer\n",ptr);
        return 1;
    }
}
int CHECK_SPL(char ptr[]) {
    if (strlen(ptr) > 1)
        return 0;
    int i;
    for (i = 0; i < 13; i++) {
        if (ptr[0] == SPL[i]) {
            printf("%c is Special Character\n",ptr[0]);
            return 1;
        }
    }
    return 0;
}
int CHECK_ARTH(char ptr[]) {
    if (strlen(ptr) > 1)
        return 0;
    int i;
    for (i = 0; i < 6; i++) {
        if (ptr[0] == ARTH[i]) {
            printf("%c is Arithmetic Operator\n",ptr[0]);
            return 1;
        }
    }
    return 0;
}
int CHECK_RELATION(char ptr[]) {
    int i;
    int len = strlen(ptr);
    if (len == 1) {
        for (i = 0; i < 2; i++) {
            if (ptr[0] == RELATION1[i]) {
                printf("%s is Relational Operator\n",ptr);
                return 1;
            }
        }
    } else {
        for (i = 0; i < 4; i++) {
            if (ptr[0] == RELATION2[i][0] && ptr[1] == RELATION2[i][1]) {
                printf("%s is Relational Operator\n",ptr);
                return 1;
            }
        }
    }
}

```

```

    }
}
return 0;
}
int CHECK_COMMENT(char ptr[]) {
    if (ptr[0] == '/' && ptr[1] == '/') {
        count = count -1;
        return 1; }
    return 0;
}

```

## OUTPUT

The screenshot shows a C++ IDE with a project named 'CD'. The main.c file contains functions for checking arithmetic and relational operators. The terminal output shows the program's execution for the input 'a + b'.

```

H:\c++\codes\CD>n
Enter Data : a + b
a is a IDENTIFIER
+ is Arithmetic Operator
b is a IDENTIFIER

H:\c++\codes\CD>

```

## WEEK-5

**Aim:** Write a Program to Design Recursive Descent Parser

**Description:** We have completed the Lexical Analysis phase. Presently, we are in the Syntax Analysis phase. Syntax Analysis is one of the phases in Compiler Design. Syntax Analysis is also known as “Parsing”.

**Parsing:** Parsing or **syntactic analysis** is the process of analysing a string of symbols, either in natural language or in computer languages, conforming to the rules of formal grammar. The term parsing comes from Latin pars (orationis), meaning part (of speech).

*“A parser is a software component that takes input data (frequently text) and builds a data structure – often some kind of parse tree, abstract syntax tree or other hierarchical structure – giving a structural representation of the input, checking for correct syntax in the process.”*

The task of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. This can be done in essentially two ways:

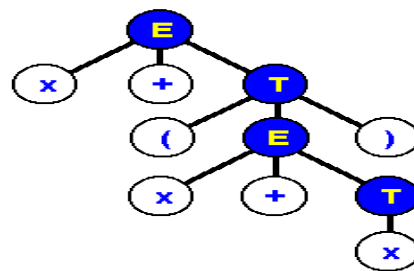
**Top-down parsing-** Top-down parsing can be viewed as an attempt to find leftmost derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal

grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules.

**Bottom-up parsing** - A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers. Another term used for this type of parser is Shift-Reduce parsing. **LL parsers** and **Recursive-Descent Parser** are examples of top-down parsers which cannot accommodate left recursive production rules.

**Recursive-Descent Parser:** A recursive descent parser is a kind of top-down parser built from a set of mutually recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the productions of the grammar.

Example:

$$\begin{aligned} E &\rightarrow x+T \\ T &\rightarrow (E) \\ T &\rightarrow x \end{aligned}$$


/\* Recursive Descent Parser for the Expression Grammar:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid i \end{aligned}$$

C-PROGRAM

```

#include<stdio.h>
#include<string.h>
int E(),Edash(),T(),Tdash(),F();
char *ip;
char string[50];
int main()
{
    printf("Enter the string\n");
    scanf("%s",string);

```



```

ip=string;
printf("\n\nInput\tAction\n-----\n");
if(E()) {
    printf("\n-----\n");
    printf("\n String is successfully parsed\n");
}
else{
    printf("\n-----\n");
    printf("Error in parsing String\n");
}
}
int E()
{
    printf("%s\tE->TE' \n",ip);
    if(T())
    {
        if(Edash())
        {
            return 1;
        }
        else
            return 0;
    }
    else
        return 0;
}
int Edash()
{
    if(*ip=='+')
    {
        printf("%s\tE'->+TE' \n",ip);
        ip++;
        if(T())
        {
            if(Edash())
            {
                return 1;
            }
            else
                return 0;
        }
        else
            return 0;
    }
    else
    {
        printf("%s\tE'->^ \n",ip);
        return 1;
    }
}
int T()
{
    printf("%s\tT->FT' \n",ip);
    if(F())
    {
        if(Tdash())
        {
            return 1;
        }
        else
            return 0;
    }
    else
        return 0;
}
int Tdash()
{
    if(*ip=='*')
    {

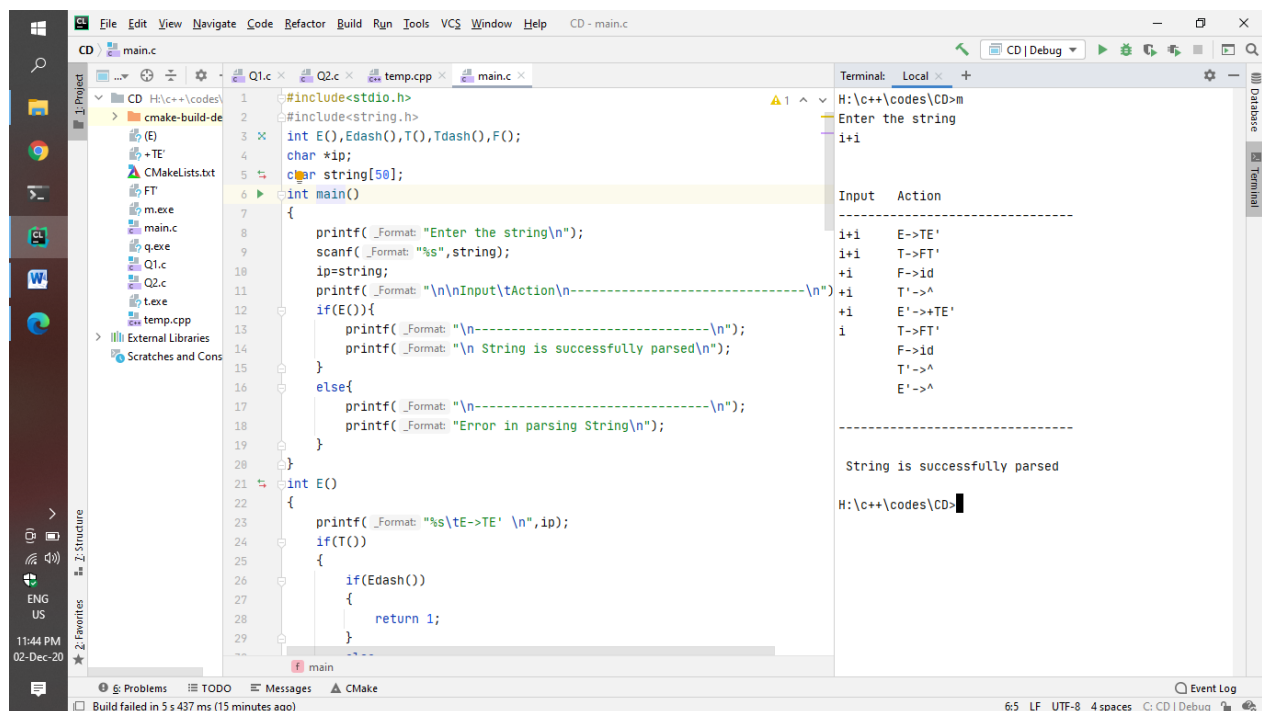
```

```

        printf("%s\tT'->*FT' \n",ip);
        ip++;
        if(F())
        {
            if(Tdash())
            {
                return 1;
            }
            else
                return 0;
        }
        else
            return 0;
    }
    else
    {
        printf("%s\tT'->^ \n",ip);
        return 1;
    }
}
int F()
{
    if(*ip=='(')
    {
        printf("%s\tF->(E) \n",ip);
        ip++;
        if(E())
        {
            if(*ip==')')
            {
                ip++;
                return 0;
            }
            else
                return 0;
        }
        else
            return 0;
    }
    else if(*ip=='i')
    {
        ip++;
        printf("%s\tF->id \n",ip);
        return 1;
    }
    else
        return 0;
}

```

**OUTPUT:**



## WEEK-6

### AIM:

Write a C program for the computation of FIRST and FOLLOW for a given CFG

### DESCRIPTION:

The functions follow and follow first are both involved in the calculation of the Follow Set of a given Non-Terminal. The follow set of the start symbol will always contain "\$". Now the calculation of Follow falls under three broad cases :

- If a Non-Terminal on the R.H.S. of any production is followed immediately by a Terminal then it can immediately be included in the Follow set of that Non-Terminal.
  - If a Non-Terminal on the R.H.S. of any production is followed immediately by a Non-Terminal, then the First Set of that new Non-Terminal gets included on the follow set of our original Non-Terminal. In case encountered an epsilon i.e. " #" then, move on to the next symbol in the production.
- Note :** " #" is never included in the Follow set of any Non-Terminal.
- If reached the end of a production while calculating follow, then the Follow set of that non-terminal will include the Follow set of the Non-Terminal on the L.H.S. of that production. This can easily be implemented by recursion.

Assumptions :

1. Epsilon is represented by '#'.
2. Productions are of the form A=B, where 'A' is a single Non-Terminal and 'B' can be any combination of Terminals and Non-Terminals.
3. L.H.S. of the first production rule is the start symbol.
4. Grammar is not left recursive.
5. Each production of a non terminal is entered on a different line.
6. Only Upper Case letters are Non-Terminals and everything else is a terminal.
7. Do not use '!' or '\$' as they are reserved for special purposes.

### Explanation :

Store the grammar on a 2D character array **production**. **findfirst** function is for calculating the first of any non terminal. Calculation of **first** falls under two broad cases :

- If the first symbol in the R.H.S of the production is a Terminal then it can directly be included in the first set.
- If the first symbol in the R.H.S of the production is a Non-Terminal then call the findfirst function again on that Non-Terminal. To handle these cases like Recursion is the best possible solution. Here again, if the First of the new Non-Terminal contains an epsilon then we have to move to the next symbol of the original production which can again be a Terminal or a Non-Terminal.

### C-PROGRAM:

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

#define input(x) scanf("%d",&x)
#define and &&
#define or ||

void FRIST(void);
void FOLLOW(void);
void _frist_(char [], char);
void _result_(char Result[],char val);
void _follow_(char);
void frist(char);
void _result(char c);

int numOfProductionrules;
char fristdata[10][10], temp_frist_rules[10];
int n, m = 0, p, i = 0, j = 0;
char followdata[10][10],followResult[10], temp_follow_rules[10];

int main(void) {
    int op;
    printf("\n1.FRIST\n2.FOLLOW");
    printf("\nchoose your option : ");
    input(op);

    if (op == 1)
        FRIST();
    else if (op == 2)
        FOLLOW();
    else {
        printf("YOU choose wrong option bye!");
        exit(0);
    }

    return 0;
}

void FRIST(void) {
    int i,l = 0, k=0;
    char f;
    char result[20];
    printf("How many number of productions ? :");
    input(numOfProductionrules);

    for(i = 0; i < numOfProductionrules; i++) {
        printf("Enter productions Number %d : ",i+1);
        scanf("%s",&fristdata[i]);

        int flag;
        printf("if production doesn't exists enter it 1 else any number : ");
        input(flag);

        if (flag == 1)
            temp_frist_rules[k++] = fristdata[i][0];
```



```

        for (k = 0; subResult[k] != '\0'; k++)
            _result_(Result, subResult[k]);

        for ( k = 0; subResult[k] != '\0'; k++)
            if (subResult[k] == '$') {
                foundEpsilon = 1;
                break;
            }

        if (!foundEpsilon)
            break;
        j++;
    }
}

return ;
}

void _result_(char Result[], char val)
{
    int k;
    for (k=0; Result[k] != '\0'; k++)
        if ( Result[k] == val)
            return;
    Result[k] = val;
    Result[k+1] = '\0';
}

void _follow_(char c) {
    if ( followdata[0][0] == c)
        _result('$');

    for (int i = 0; i < n; i++){
        for (int j = 2; j < strlen(followdata[i]); j++){
            if (followdata[i][j] == c) {
                if (followdata[i][j+1] != '\0')
                    frist(followdata[i][j+1]);

                if(followdata[i][j+1] == '\0' and c != followdata[i][0])
                    _follow_(followdata[i][0]);
            }
        }
    }
}

void frist(char c) {
    if (!(isupper(c)))
        _result(c);

    for (int k = 0; k < n; k++) {
        if (followdata[k][0] == c) {
            if (followdata[k][2] == '$')
                _follow_(followdata[i][0]);

            else if(islower(followdata[k][2]))
                _result(followdata[k][2]);
            else
                frist(followdata[k][2]);
        }
    }
}

void _result(char c){
    for (int i = 0; i <= m; i++)
        if (followResult[i]==c)
            return;
}

```

```

        followResult[m++] = c;
    }

```

## OUTPUT:

```

input
Enter the no of productions:
5
Enter the productions:
s=AbCd
A=Cf
A=a
C=gE
E=h
Enter the elemets whose fisrt & follow is to be found:s
First(s)={sga}
Follow(s)={$}
Continue (0/1)?1
Enter the elemets whose fisrt & follow is to be found:A
First(A)={ga}
Follow(A)={b}
Continue (0/1)?1
Enter the elemets whose fisrt & follow is to be found:C
First(C)={g}
Follow(C)={df}
Continue (0/1)?1
Enter the elemets whose fisrt & follow is to be found:E
First(E)={h}
Follow(E)={df}
Continue (0/1)?0

...Program finished with exit code 0
Press ENTER to exit console.

```

## WEEK-7:

### AIM:

Implement a Predictive Parser for the Expression Grammar:

$E \rightarrow TE'$   
 $E' \rightarrow +TE' \mid \epsilon$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT' \mid \epsilon$   
 $F \rightarrow (E) \mid i$

### C-PROGRAM:

```

#include<stdio.h>
#include<stdlib.h>
void pop(),push(char ),display();
char stack[100]="\0";
char input[100];
int top=-1;
char *ip;
void main()
{
    printf(" enter the input string followed by $ \n");
    scanf("%s",input);
    ip=input;
    push('$');
    push('E');
    printf("STACK\t INPUT \t ACTION\n");
    printf("-----\t ----- \t ----- \n");
    printf("\n%s\t%s\tSHIFT",stack,ip);
    while(stack[top]!='$')
    {
        if(stack[top]=='+' || stack[top]=='*' || stack[top]=='i' ||
stack[top]=='(' || stack[top]==')' || stack[top]=='$')
        {
            if(stack[top]==*ip)
            {
                pop();
                ip++;
            }
        }
    }
}

```

```

        printf("\n%s\t%s\tSHIFT", stack, ip);
        printf("\n");
    }
    else
    {
        printf("\n error ");
        exit(0);
    }
}
else if(stack[top]=='E' && (*ip=='(' || *ip=='i'))
{
    pop();
    push('E');
    push('T');
    printf("\n%s\t%s\t", stack, ip);
    printf("REDUCE BY E->TE'\n");
}
else if(stack[top]=='E' && (*ip==')' || *ip=='$'))
{
    pop();
    printf("\n%s\t%s\t", stack, ip);
    printf("REDUCE BY E'->^\n");
}
else if(stack[top]=='E' && *ip=='+' )
{
    pop();
    push('E');
    push('T');
    push('+');
    printf("\n%s\t%s\t", stack, ip);
    printf("REDUCE BY E'->+TE'\n");
}
else if(stack[top]=='T' && (*ip=='(' || *ip=='i'))
{
    pop();
    push('T');
    push('F');
    printf("\n%s\t%s\t", stack, ip);
    printf("REDUCE BY T->FT'\n");
}
else if(stack[top]=='T' && (*ip==')' || *ip=='$' || *ip=='+'))
{
    pop();
    printf("\n%s\t%s\t", stack, ip);
    printf("REDUCE BY E->TE'\n");
}
else if(stack[top]=='T' && *ip=='*')
{
    pop();
    push('T');
    push('F');
    push('*');
    printf("\n%s\t%s\t", stack, ip);
    printf("REDUCE BY T'->*FT'\n");
}
else if(stack[top]=='F' && *ip=='(' )
{
    pop();
    push('(');
    push('E');
    push('(');
    printf("\n%s\t%s\t", stack, ip);
    printf("REDUCE BY F->(E)\n");
}
else if(stack[top]=='F' && *ip=='i')
{
    pop();
    push('i');
    printf("\n%s\t%s\t", stack, ip);

```



```

        printf("REDUCE BY F->id\n");
    }
    else
    {
        printf("\n error");
        exit(0);
    }
}
printf("\n%s\t%s\t",stack,ip);
printf(" Accept\n\n\n");
}
void push(char c)
{
    top++;
    stack[top]=c;
}
void pop()
{
    stack[top]='\0';
    top--;
}
#include<stdio.h>
#include<stdlib.h>
void pop(),push(char ),display();
char stack[100]="\0";
char input[100];
int top=-1;
char *ip;
void main()
{
    printf(" enter the input string followed by $ \n");
    scanf("%s",input);
    ip=input;
    push('$');
    push('E');
    printf("STACK\t INPUT \t ACTION\n");
    printf("-----\t ----- \t ----- \n");
    printf("\n%s\t%s\tSHIFT",stack,ip);
    while(stack[top]!='$')
    {
        if(stack[top]=='+' || stack[top]=='*' || stack[top]=='i' ||
stack[top]=='(' || stack[top]==')' || stack[top]=='$')
        {
            if(stack[top]==*ip)
            {
                pop();
                ip++;
                printf("\n%s\t%s\tSHIFT",stack,ip);
                printf("\n");
            }
            else
            {
                printf("\n error ");
                exit(0);
            }
        }
        else if(stack[top]=='E' && (*ip=='(' || *ip=='i'))
        {
            pop();
            push('E');
            push('T');
            printf("\n%s\t%s\t",stack,ip);
            printf("REDUCE BY E->TE'\n");
        }
        else if(stack[top]=='E' && (*ip==')' || *ip=='$'))
        {
            pop();
            printf("\n%s\t%s\t",stack,ip);
            printf("REDUCE BY E'->^\n");
        }
    }
}

```

```

    }
    else if(stack[top]=='E' && *ip=='+' )
    {
        pop();
        push('E');
        push('T');
        push('+');
        printf("\n%s\t%s\t",stack,ip);
        printf("REDUCE BY E->TE'\n");
    }
    else if(stack[top]=='T' && (*ip=='(' || *ip=='i'))
    {
        pop();
        push('T');
        push('F');
        printf("\n%s\t%s\t",stack,ip);
        printf("REDUCE BY T->FT'\n");
    }
    else if(stack[top]=='T' && (*ip==')' || *ip=='$' || *ip=='+'))
    {
        pop();
        printf("\n%s\t%s\t",stack,ip);
        printf("REDUCE BY E->TE'\n");
    }
    else if(stack[top]=='T' && *ip=='*')
    {
        pop();
        push('T');
        push('F');
        push('*');
        printf("\n%s\t%s\t",stack,ip);
        printf("REDUCE BY T->*FT'\n");
    }
    else if(stack[top]=='F' && *ip=='(' )
    {
        pop();
        push('(');
        push('E');
        push('(');
        printf("\n%s\t%s\t",stack,ip);
        printf("REDUCE BY F->(E)\n");
    }
    else if(stack[top]=='F' && *ip=='i')
    {
        pop();
        push('i');
        printf("\n%s\t%s\t",stack,ip);
        printf("REDUCE BY F->id\n");
    }
    else
    {
        printf("\n error");
        exit(0);
    }
}
printf("\n%s\t%s\t",stack,ip);
printf(" Accept\n\n\n");
}
void push(char c)
{
    top++;
    stack[top]=c;
}
void pop()
{
    stack[top]='\0';
    top--;
}

```

EXPLANATION:

## OUTPUT:

enter the input string followed by \$

d+d\$

STACK	INPUT	ACTION
-------	-------	--------

-----	-----	-----
-------	-------	-------

\$E	d+d\$	SHIFT
-----	-------	-------

\$ET	d+d\$	REDUCE BY E->TE'
------	-------	------------------

\$ETF	d+d\$	REDUCE BY T->FT'
-------	-------	------------------

\$ETd	d+d\$	REDUCE BY F->d
-------	-------	----------------

\$ET	+d\$	SHIFT
------	------	-------

\$E	+d\$	REDUCE BY E->TE'
-----	------	------------------

\$ET+	+d\$	REDUCE BY E'->+TE'
-------	------	--------------------

\$ET	d\$	SHIFT
------	-----	-------

\$ETF	d\$	REDUCE BY T->FT'
-------	-----	------------------

\$ETd	d\$	REDUCE BY F->d
-------	-----	----------------

\$ET	\$	SHIFT
------	----	-------

\$E	\$	REDUCE BY E->TE'
-----	----	------------------

\$	\$	REDUCE BY E'->^
----	----	-----------------

\$	\$	Accept
----	----	--------

Output

Clear

```
/tmp/igMXjc87i0.o
enter the input string followed by $
d+d$
STACK    INPUT    ACTION
-----
$E d+d$   SHIFT
$ET d+d$   REDUCE BY E->TE'

$ETF d+d$   REDUCE BY T->FT'

$ETd d+d$   REDUCE BY F->d

$ET +d$ SHIFT

$E +d$ REDUCE BY E->TE'

$ET+ +d$ REDUCE BY E'->+TE'

$ET d$ SHIFT

$ETF d$ REDUCE BY T->FT'

$ETd d$ REDUCE BY F->d

$ET $ SHIFT

$E $ REDUCE BY E->TE'

$ $ REDUCE BY E'->^

$ $ Accept
```

## WEEK-8:

AIM:

### Implementation of Shift Reducing Parser

In other words, it is a process of “reducing” (opposite of deriving a symbol using a production rule) a string  $w$  to the start symbol of a grammar.

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner to reach the start symbol. The image given below depicts the bottom-up parsers available.

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

- **Shift step:** The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
- **Reduce step:** When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

Source Code:

```

#include<stdio.h>
#include<stdlib.h>
void pop(),push(char ),display();
char stack[100]="\0";
char inputbuffer[100];
int top=-1;
char *ip;
void main()
{
    printf("E->E+E\n");
    printf("E->E*E\n");
    printf("E->(E)\n");
    printf("E->d\n");
    printf(" enter the input string followed by $ \n");
    scanf("%s",inputbuffer);
    ip=inputbuffer;
    push('$');
    printf("STACK\t BUFFER \t ACTION\n");
    printf("-----\t ----- \t ----- \n");
    display();
    do
    {
        if((stack[top]=='E' && stack[top-1]=='$') && (*(ip)=='$'))
            break;
        if(stack[top]=='$')
        {
            push(*ip);
            ip++;
            printf("Shift");
        }
        else if(stack[top]=='d')
        {
            display();
            pop();
            push('E');
            printf("Reduce E->d\n");
        }
        else if(stack[top]=='E' && stack[top-1]=='+' && stack[top-2]=='E'&&
*ip!='*')
        {
            display();
            pop();
            pop();
            pop();
            push('E');
            printf("Reduce E->E+E");
        }
        else if(stack[top]=='E' && stack[top-1]=='*' && stack[top-2]=='E')
        {
            display();
            pop();
            pop();
            pop();
            push('E');
            printf("Reduce E->E*E");
        }
        else if(stack[top]==')' && stack[top-1]=='E' && stack[top-2]=='(')
        {
            display();
            pop();
            pop();
            pop();
            push('E');
            printf("Reduce E->(E)");
        }
        else
        {
            display();
            push(*ip);
        }
    }
}

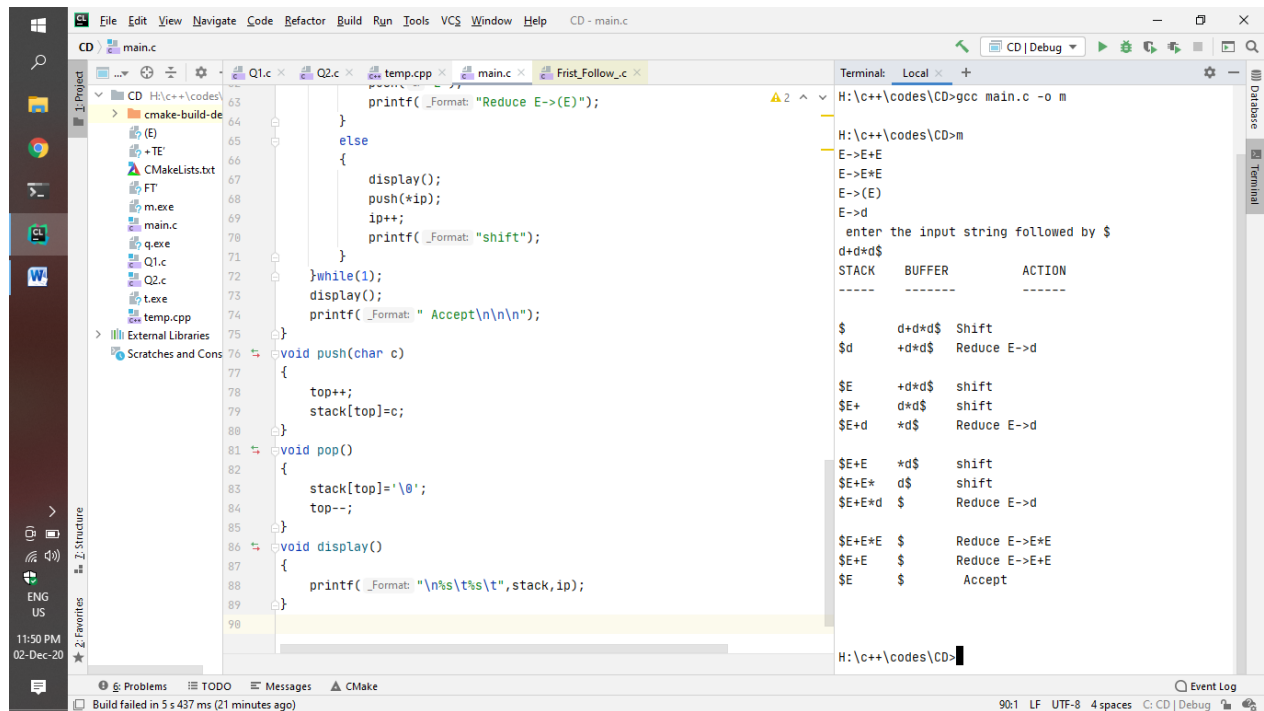
```

```

        ip++;
        printf("shift");
    }
}while(1);
display();
printf(" Accept\n\n\n");
}
void push(char c)
{
    top++;
    stack[top]=c;
}
void pop()
{
    stack[top]='\0';
    top--;
}
void display()
{
    printf("\n%s\t%s\t",stack,ip);
}

```

### OUTPUT:



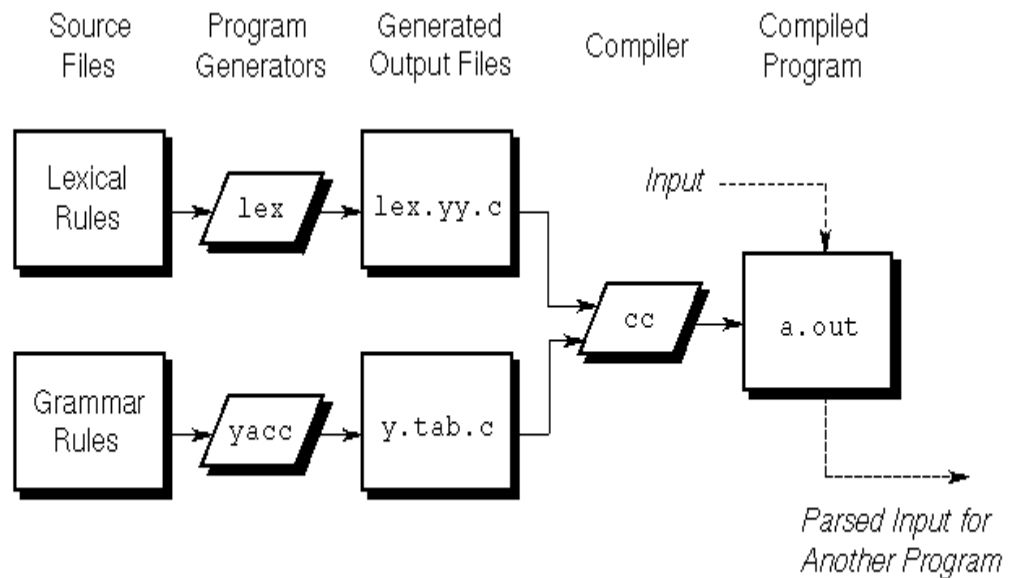
### WEEK-9

**AIM:** Implement LALR parser using LEX and YACC for the following Grammar:

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$



ZK-0455U-R

### parser.l

```

%{
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {yylval=atoi(yytext);
return DIGIT;
}
[\t] ;
\n return 0;
. return yytext[0];
%%

```

Compilation of lex:

C:\Users\IDRBT-RF7\Desktop\flex>lex a.l

### Parser1.y

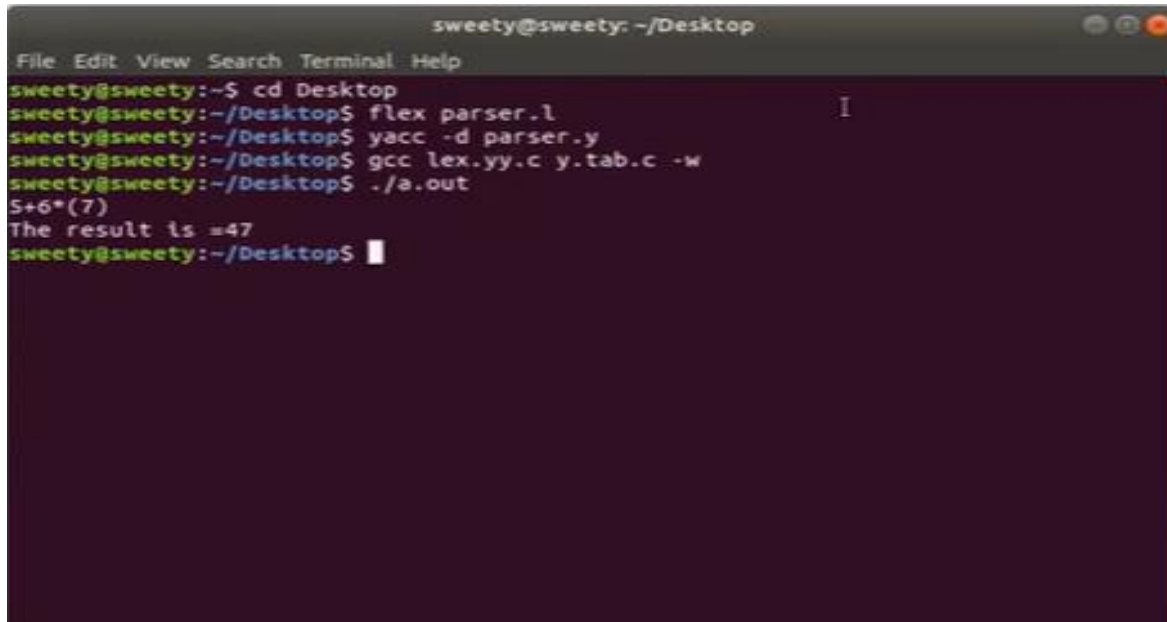
```

%{
#include<stdio.h>
%}
%token DIGIT
%%
S: E { printf("The result is %d\n", $1); }
;
E: E+'T' { $$ = $1 + $3; }
| T { $$ = $1; }
;
T: T*'F' { $$ = $1 * $3; }
| F { $$ = $1; }
;
F: '('E')' { $$ = $2; }

```

```
| DIGIT {$$ = $1;}
;
%%
main()
{
  yyparse();
}
yyerror(char *s)
{
  printf("%s",s);
}
```

OUTPUT:



```
sweety@sweety: ~/Desktop
File Edit View Search Terminal Help
sweety@sweety:~$ cd Desktop
sweety@sweety:~/Desktop$ flex parser.l
sweety@sweety:~/Desktop$ yacc -d parser.y
sweety@sweety:~/Desktop$ gcc lex.yy.c y.tab.c -w
sweety@sweety:~/Desktop$ ./a.out
S+6*(7)
The result is =47
sweety@sweety:~/Desktop$
```

## WEEK-10

AIM:

Implementation of Intermediate code generator a. Quadruple Generation

Declarations:

A variable or procedure has to be declared before it can be used. Declaration involves allocation of space in memory and entry of type and name in the symbol table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.

Taking the whole program as a collection of procedures and sub-procedures, it becomes possible to declare all the names local to the procedure. Memory allocation is done in a consecutive manner and names are allocated to memory in the sequence they are declared in the program. We use offset variable and set it to zero {offset = 0} that denote the base address.

The source programming language and the target machine architecture may vary in the way names are stored, so relative addressing is used. While the first name is allocated memory starting from the memory location 0 {offset=0}, the next name declared later, should be allocated memory next to the first one.

CODE:

Qual.l



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

%%

[a-z]([a-z]|[0-9])* { strcpy(yyval.exp,yytext);
return VAR;
}

\t;

\n return 0;

. return yytext[0];

%%
```

### **Quad.y**

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

struct quad
{
char op[5];
char arg1[10];
char arg2[10];
char result[10];
}QUAD[30];

int i=0,j;

%}

%union

{
char exp[10];
}

%token <exp> VAR
%type <exp> S E T F

%%

S: E { printf("\n There are %d quadrupls n",i);
printf("\n List of Quadruples are: \n");
```

```

for(j=0;j<i;j++)
printf("%s\t%s\t%s\t%s\n",QUAD[j].op,QUAD[j].arg1,QUAD[j].arg2,QUAD[j].result);
}
;
E: E+'T { printf("\n E ->E+T, $1=%s, $3=%s, $$=%s\n",$1,$3,$$);
strcpy(QUAD[i].op,"+");
strcpy(QUAD[i].arg1,$1);
strcpy(QUAD[i].arg2,$3);
strcpy(QUAD[i].result,$$);i++;
i++;
}
| T { printf("\n E -> T, $1=%s, $$=%s\n",$1,$$);}
;
T: T'*F { printf("\n T -> T*F, $1=%s, $3=%s, $$=%s\n",$1,$3,$$);
strcpy(QUAD[i].op,"*");
strcpy(QUAD[i].arg1,$1);
strcpy(QUAD[i].arg2,$3);
strcpy(QUAD[i].result,$$);
i++;
}
| F { printf("\n T -> F, $1=%s, $$=%s\n",$1,$$);}
;
F: VAR {printf("\n F ->VAR and $1=%s, $$=%s \n",$1,$$);}
;
%%
main()
{
yyparse();
}
int yywrap(){
return 1;
}
yyerror(char *s)
{
printf("%s",s); }

```

OUTPUT:

T -> T\*F, \$1=d, \$3=f,\$\$=d

E ->E+T, \$1=a, \$3=d,\$\$=a

F ->VAR and \$1=e, \$\$=e

T -> F, \$1=e, \$\$=e

E ->E+T, \$1=a, \$3=e,\$\$=a

There are 7 quadruples n

List of Quadruples are:

+	a	c	a
*	d	f	d
+	a	d	a
+	a	e	a



