Jaipur Engineering College and Research Center, Jaipur

Department of Information Technology

LAB MANUAL

Session-2020-21

Lab Name	**:**	Compiler Design Lab
Lab Code	**:**	5IT4 - 22
Branch	**:**	Information Technology
Year/Semester	**:**	3rd Year/V

Prepared by:

Brijesh Kumar Singh
Assistant Professor
Department of IT

Jaipur Engineering College and Research Center, Jaipur

Department of Information Technology

Shri Ram Ki Nangal, Via-Vatika, Sitapura, Jaipur

(Rajasthan Technical University, KOTA)

INDEX

JECRC
JAIPUR ENGINEERING COLLEGE
AND RESEARCH CENTRE

MOTTO of JECRC

TEACH

TRAIN

&

TRANSFORM

For

- Contribution towards National Development
- Global Competencies among Students
- Incorporating a Value System
- Promotion to use of Technology
- Zeal for Excellence

Vision and Mission of Institute

Vision:

To become a renowned centre of outcome based learning, and work towards academic, professional, cultural and social enrichment of the lives of individuals and communities.

Mission:

- Focus on evaluation of learning outcomes and motivate students to inculcate research aptitude by project based learning.
- Identify areas of focus and provide platform to gain knowledge and solutions based on informed perception of Indian, regional and global needs.
- Offer opportunities for interaction between academia and industry.
- Develop human potential to its fullest extent so that intellectually capable and imaginatively gifted leaders can emerge in a range of professions.

Vision and Mission of Department of Information Technology

Vision:

To establish outcome based excellence in teaching, learning and commitment to support IT Industry.

Mission:

M1: To provide outcome based education.

M2: To provide fundamental & Intellectual knowledge with essential skills to meet current and future need of IT Industry across the globe.

M3: To inculcate the philosophy of continuous learning, ethical values & Social Responsibility.

Program Educational Objectives(PEOs)

1. To provide students with the fundamentals of Engineering Sciences with more emphasis in Information Technology by way of analyzing and exploiting engineering challenges.
2. To train students with good scientific and engineering knowledge so as to comprehend, analyze, design, and create novel products and solutions for the real life problems.
3. To inculcate professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, entrepreneurial thinking and an ability to relate engineering issues with social issues.
4. To provide students with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the self-motivated life-long learning needed for a successful professional career.
5. To prepare students to excel in Industry and Higher education by Educating Students along with High moral values and Knowledge

Program Outcomes (POs)

1. **Engineering Knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems in IT.
2. **Problem analysis:** Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences in IT.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations using IT.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions using IT.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations in IT.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice using IT.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development in IT.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice using IT.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings in IT.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project Management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage IT projects and in multidisciplinary environments.
12. **Life –long Learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological changes needed in IT.

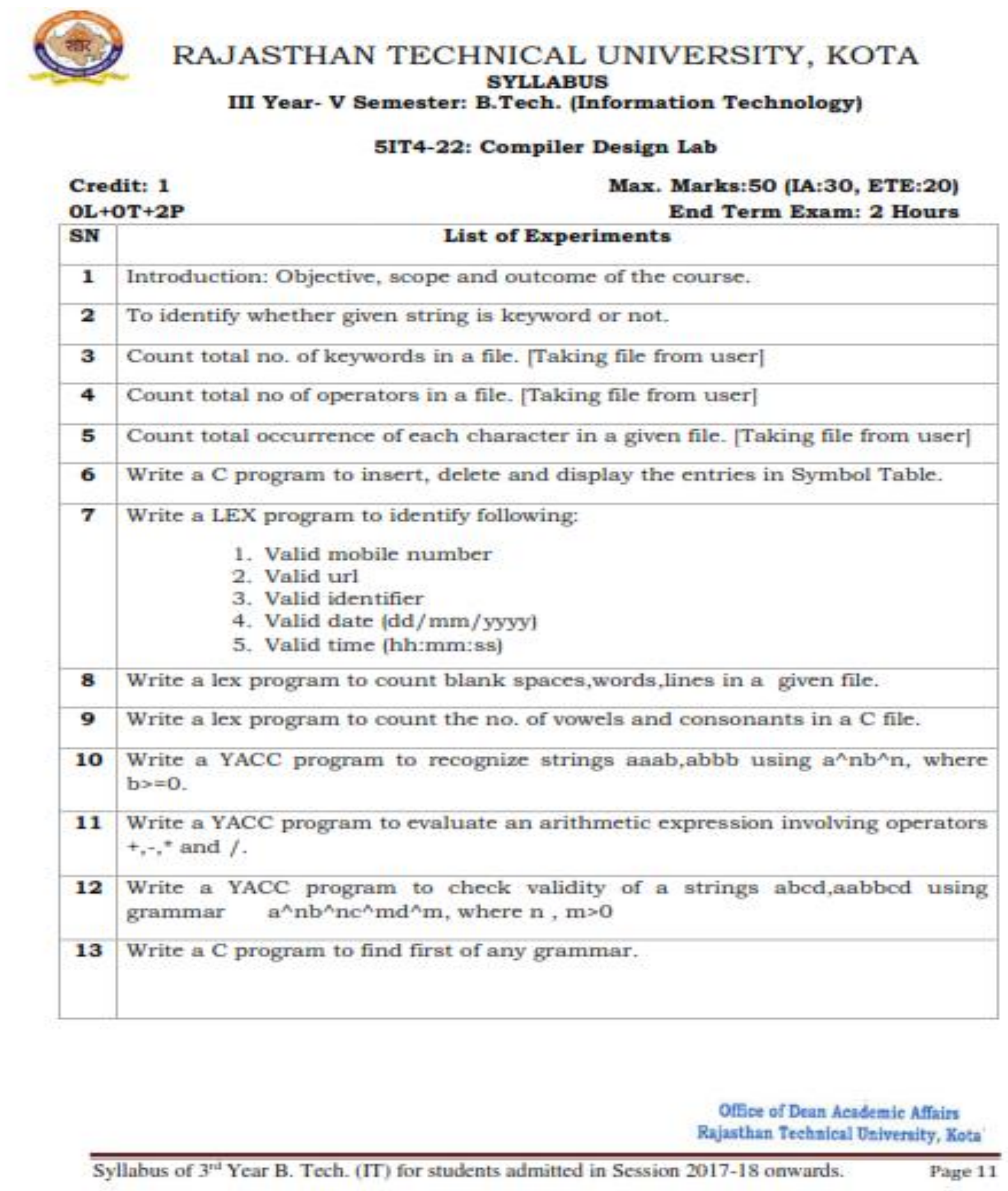
PSO of the Department:

PSO 1: Graduates of the program would be able to develop mobile and web based IT solutions for real time problems.

PSO 2: Graduates of the program would be able to apply the concepts of artificial intelligence, machine learning and deep learning.

RTU Syllabus with List of Experiments:

Objectives: At the end of the semester, the students should have clearly understood and implemented the following:



RAJASTHAN TECHNICAL UNIVERSITY, KOTA

SYLLABUS

III Year- V Semester: B.Tech. (Information Technology)

5IT4-22: Compiler Design Lab

Credit: 1 **Max. Marks:50 (IA:30, ETE:20)**

0L+0T+2P **End Term Exam: 2 Hours**

SN	List of Experiments
1	Introduction: Objective, scope and outcome of the course.
2	To identify whether given string is keyword or not.
3	Count total no. of keywords in a file. [Taking file from user]
4	Count total no of operators in a file. [Taking file from user]
5	Count total occurrence of each character in a given file. [Taking file from user]
6	Write a C program to insert, delete and display the entries in Symbol Table.
7	Write a LEX program to identify following: 1. Valid mobile number 2. Valid url 3. Valid identifier 4. Valid date (dd/mm/yyyy) 5. Valid time (hh:mm:ss)
8	Write a lex program to count blank spaces,words,lines in a given file.
9	Write a lex program to count the no. of vowels and consonants in a C file.
10	Write a YACC program to recognize strings aaab,abbb using a^nb^n, where b>=0.
11	Write a YACC program to evaluate an arithmetic expression involving operators +,-,* and /.
12	Write a YACC program to check validity of a strings abcd,aabbcd using grammar a^nb^nc^md^m, where n , m>0
13	Write a C program to find first of any grammar.

Office of Dean Academic Affairs
Rajasthan Technical University, Kota

Syllabus of 3rd Year B. Tech. (IT) for students admitted in Session 2017-18 onwards. Page 11

Software / Tools Used:

- Turbo C 4.5
- Lex / Jlex/ JFlex
- YACC / CUP / Bison / ANTLR

COURSE OUTCOMES

OBJECTIVE: This laboratory course is intended to make the students experiment on the basic techniques of compiler construction and tools that can used to perform syntax-directed translation of a high-level programming language into an executable code. Students will design and implement language processors in C by using tools to automate parts of the implementation process. This will provide deeper insights into the more advanced semantics aspects of programming languages, code generation, machine independent optimizations, dynamic memory allocation, and object orientation.

COURSE OUTCOMES

Graduates would be able:

CO1- Design Lexical analyzer for given language using C and LEX tools.

CO2- Design and convert BNF rules into YACC form to generate various parsers

CO3- Generate machine code from the intermediate code forms.

CO4- Implement Symbol table.

Mapping of CO & PO:

COURSE OUTCOMES	PROGRAM OUTCOMES											
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
CO-1	3	3	3	2	3				1	1	2	2
CO-2	3	3	3	2	2				1	1	2	2
CO-3	3	2	2	2	2					1	1	1
CO-4	3	2	3	2	3				2	1	2	1

Mapping of CO & PSO:

COURSE OUTCOMES	PSO	
CO-1	1	1
CO-2	1	1

INSTRUCTIONAL METHODS:

Direct Instructions:

I. Through Projectors & White Board with Marker

Interactive Instruction:

I. Programs

Indirect Instructions:

I. Problem solving

LEARNING MATERIALS:

Text/Lab Manual

ASSESSMENT OF OUTCOMES:

1. End term Practical exam (Conducted by RTU, KOTA)
2. Daily Lab interaction.

OUTCOMES WILL BE ACHIEVED THROUGH FOLLOWING:

1. Lab Teaching (through Projectors and White Board).
2. Discussion on Programs.

INSTRUCTIONS OF LAB

DO’s

1. Please switch off the Mobile/Cell phone before entering Lab.
2. Enter the Lab with complete source code and data.
3. Check whether all peripheral are available at your desktop before proceeding for program.

4. Intimate the lab In charge whenever you are incompatible in using the system or in case software get corrupted/ infected by virus.
5. Arrange all the peripheral and seats before leaving the lab.
6. Properly shutdown the system before leaving the lab.
7. Keep the bag outside in the racks.
8. Enter the lab on time and leave at proper time.
9. Maintain the decorum of the lab.
10. Utilize lab hours in the corresponding experiment.
11. Get your CD / Pen drive checked by lab In charge before using it in the lab.

DON'TS

1. No one is allowed to bring storage devices like Pan Drive /Floppy etc. in the lab.
2. Don't mishandle the system.
3. Don't leave the system on standing for long
4. Don't bring any external material in the lab.
5. Don't make noise in the lab.
6. Don't bring the mobile in the lab. If extremely necessary, then keep ringers off.
7. Don't enter in the lab without permission of lab in charge.
8. Don't litter in the lab.
9. Don't delete or make any modification in system files.
10. Don't carry any lab equipment's outside the lab.

We need your full support and cooperation for smooth functioning of the

INSTRUCTIONS FOR STUDENT

BEFORE ENTERING IN THE LAB

- All the students are supposed to prepare the theory regarding the next program.
- Students are supposed to bring the practical file and the lab copy.
- Previous programs should be written in the practical file.
- Any student not following these instructions will be denied entry in the lab.

WHILE WORKING IN THE LAB

- Adhere to experimental schedule as instructed by the lab in charge.
- Get the previously executed program signed by the instructor.
- Get the output of the current program checked by the instructor in the lab copy.
- Each student should work on his/her assigned computer at each turn of the lab.
- Take responsibility of valuable accessories.
- Concentrate on the assigned practical and do not play games.
- If anyone caught red handed carrying any equipment of the lab, then he will have to face serious consequences.

Introduction about Lab:

Compiler is a program that reads a program written in one language – the source language – and translates it in to an equivalent program in another language – the target language. ANALYSIS-SYNTHESIS MODEL OF COMPILATION There are two parts to compilation: Analysis and Synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of source program. The synthesis part constructs the desired target program from the intermediate representation. Of the two parts, synthesis requires the most specialize technique. PHASES OF A COMPILER A compiler operates in six phases, each of which transforms the source program from one representation to another. The first three phases are forming the bulk of analysis portion of a compiler. Two other activities, symbol table management and error handling, are also interacting with the six phases of compiler. These six phases are lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization and code generation. LEXICAL ANALYSIS In compiler, lexical analysis is also called linear analysis or scanning. In lexical analysis the stream of characters making up the source program is read from left to right and grouped into tokens that are sequences of characters having a collective meaning. SYNTAX ANALYSIS It is also called as Hierarchical analysis or parsing. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually, a parse tree represents the grammatical phrases of the source program. SEMANTIC ANALYSIS The semantic analysis phase checks the source program for semantic errors MKCE-DEPARTMERMATION TECHNOLOGY
A compiler or interpreter for a programming language is often decomposed into
two parts:
1. Read the source program and discover its structure.
2. Process this structure, e.g. to generate the target program.

Lex and *Yacc* can generate program fragments that solve the first task.
The task of discovering the source structure again is decomposed into subtasks:
1. Split the source file into tokens (*Lex*).
2. Find the hierarchical structure of the program (*Yacc*).

Lex - A Lexical Analyzer Generator

Lex is a program generator designed for lexical processing of character input
streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes
regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

Lex helps write programs whose control flow is directed by instances of regular
expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.
Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is

recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.
The lexical analysis programs written with Lex accept ambiguous specifications
and choose the longest match possible at each input point. If necessary, substantial look ahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.
Lex can generate analyzers in either C or Ratfor, a language which can be
translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

Experiment No.1

Aim: - Introduction: Objective, scope and outcome of the course.

Objective: The objective of this course is to explore the principles of compiler. To introduce the major concept areas of language translation and compiler design.

- To enrich the knowledge in various phases of compiler and its use, code optimization techniques, machine code generation, and use of symbol table.
- To extend the knowledge of parser by parsing LL parser and LR parser.

  Prerequisites: Knowledge of regular expression, pattern, automate theory, compiler theory.

  Scope: Experience working with really big data structure and complex interaction between algorithms.
- These scope rules need a more complicated organization of symbol table than a list of associations between names and attributes. Tables are organized into stack and each table contains the list of names and their associated attributes. ... When the declaration is compiled then the table is searched for a name.
- Scope is a source-code level concept, and a property of name bindings, particularly variable or function name bindings—names in the source code are references to entities in the program—and is part of the behavior of a compiler or interpreter of a language.

Experiment No. 2

Aim: To identify whether given string is keyword or not.

Tools / Software: Turbo C++ /Code Blocks

Description: Keyword is a predefined or reserved word which is available in C++ library with a fixed meaning and used to perform an internal operation. C++ Language supports more than 64 keywords. Every **Keyword** exists in lower case letters like auto, break, case, const, continue, int etc.32 Keywords in C++ Language which is also available in the C language.

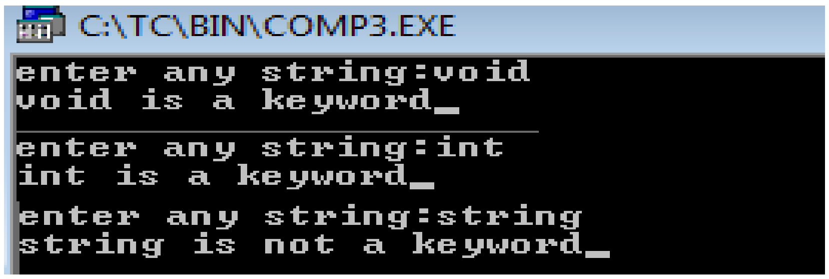
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Source Code:

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

void main()

{

 char keyword[32][10]={"auto","double","int","struct","break","else","long",
      "switch","case","enum","register","typedef","char",
      "extern","return","union","const","float","short",
      "unsigned","continue","for","signed","void","default",
      "goto","sizeof","voltile","do","if","static","while"} ;
char string[10];
 int flag=0,i;
 printf("enter any string:");
 gets(string);
 for(i=0;i<32;i++)
 {
   if(strcmp(string,keyword[i])==0)
   {
     flag=1;
   }
 }
 if(flag==1)
```

```
   printf("%s is a keyword",string);

   else

   printf("%s is not a keyword",string);



getch();

}
```

Output:



```
enter any string:void
void is a keyword_
enter any string:int
int is a keyword_
enter any string:string
string is not a keyword_
```

Viva Questions:

Q1- Define the term “Keyword”.
Q2- Describe the compiler.
Q3- Differentiate between interpreter and compiler.
Q4- Describe the two parts of a Compilation?
Q5- describe the linear analysis.

Experiment No. 3

Aim: Write a C program to count total no. of keywords in a file. [Taking file from user]

Tools / Software: Turbo C++ /Code Blocks

Description: Keywords are reserved words which cannot be used as variable names in program.

- There are 32 keywords in the C programming language.

Compare the string with each keyword if the string is same then string is keyword.

Step 1: Taking a file from user.

Step 2: Check if the given string is a keyword or not.

Step 3: Count the total no. of keyword in given file.

Source Code:

```
 * using Array Structure
 */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define KEYMAX 32

struct keyword
{
    char word[10];
    int occur;
};

int binarysearch(char [], struct keyword[]);

int main()
{
```

```
int i = 0, j = 0, pos;
char string[100], unit[20], c;
struct keyword key[32] = {"auto", 0, "break", 0, "case", 0,
                    "char", 0, "const", 0, "continue", 0,
                    "default", 0, "do", 0, "double", 0,
                    "else", 0, "enum", 0, "extern", 0,
                    "float", 0, "for", 0, "goto", 0,
                    "if", 0, "int", 0, "long", 0,
                    "register", 0, "return", 0, "short", 0,
                    "signed", 0, "sizeof", 0, "static", 0,
                    "struct", 0, "switch", 0, "typedef", 0,
                    "union", 0, "unsigned", 0, "void", 0,
                    "volatile", 0, "while", 0,};

printf("Enter string: ");
do
{
    fflush(stdin);
    c = getchar();
    string[i++] = c;

} while (c != '\n');
string[i - 1] = '\0';
printf("The string entered is: %s\n", string);
for (i = 0; i < strlen(string); i++)
{
    while (i < strlen(string) && string[i] != ' ' && isalpha(string[i]))
    {
        unit[j++] = tolower(string[i++]);
    }
    if (j != 0)
```

```
        {
            unit[j] = '\0';
            pos = binarysearch(unit, key);
            j = 0;
            if (pos != -1)
            {
                key[pos].occur++;
            }
        }
    }
    printf("***********************\n Keyword\tCount\n***********************\n");
    for (i = 0; i < KEYMAX; i++)
    {
        if (key[i].occur)
        {
            printf("    %s\t  %d\n", key[i].word, key[i].occur);
        }
    }

    return 0;
}

int binarysearch(char *word, struct keyword key[])
{
    int low, high, mid;

    low = 0;
    high = KEYMAX - 1;
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (strcmp(word, key[mid].word) < 0)
        {
            high = mid - 1;
        }
        else if (strcmp(word, key[mid].word) > 0)
        {
            low = mid + 1;
        }
        else
        {
            return mid;
        }
    }

    return -1;
}
```

Output :-

```
$ gcc keywordoccur.c
$ ./a.out
Enter string: break, float and double are c keywords. float and double are primitive data types.
The string entered is: break, float and double are c keywords. float and double are primitive data
types.
***********************
  Keyword          Count
***********************
   break            1
   double           2
   float            2
```

Viva Questions:

Q1- Describe the phases of compiler.
Q2- Describe the Token.
Q3- Define phase and pass.
Q4- Describe the single pass and multi pass.
Q5- Define the debugging.

Experiment No.4

Aim: Write a program to count total no of operators in a file. [Taking file from user].

Tools / Software: Turbo C++ /Code Blocks

Description: An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators −

- **Arithmetic Operators -** Arithmetic operators are the symbols that represent arithmetic math operations. Examples include + (addition operator), - (subtraction operator), * (multiplication operator), and / (division operator).
- **Relational Operators-** A relational operator is a programming language construct or operator that tests or defines some kind of relation between two entities. ... In languages such as C, relational operators return the integers 0 or 1, where 0 stands for false and any non-zero value stands for true.
- **Logical Operators-** A logical operator is a symbol or word used to connect two or more expressions such that the value of the compound expression produced depends only on that of the original expressions and on the meaning of the operator. Common logical operators include AND, OR, and NOT.
- **Bitwise Operators -** The Bitwise Operator in C is a type of operator that operates on bit arrays, bit strings, and tweaking binary values with individual bits at the bit level. For handling electronics and IoT-related operations, programmers use bitwise operators. It can operate faster at a bit level
- **Assignment Operators -** Assignment operators are used to assigning value to a variable. The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value. ... "=": This is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left

Source Code:

```
#include<stdio.h>
#include<conio.h>
int main()

{

char  s[200];

int count = 0, i;

printf(“Enter the string:\n”);

scanf(“%[^\n]s”, s)’;

for ( i = 0; s[ i ] != ‘\0’ ; i++ )

{

if   ( s[ i ]  = = ‘+’  | |   s[ i ]  = = ‘-’  | |   s[ i ]  = = ‘*’  | |   s[ i ]  = = ‘/’  | |  )

  count++;

}

printf(“Number of Operators in a given string are:%d\n”, count);

}
```

Viva Questions:

Q1- Describe the function sizeof() in C?.
Q2- Which operator has the highest precedence?
Q3- How can we find size of a variable without using sizeof() operator?
Q4- Describe the use of bit wise operators?
Q5- Define the modulus Operators in C?

Experiment No. 5

Aim: Write a C Program to Count total occurrence of each character in a given file. [Taking file from user]

Tools / Software: Turbo C++ /Code Blocks

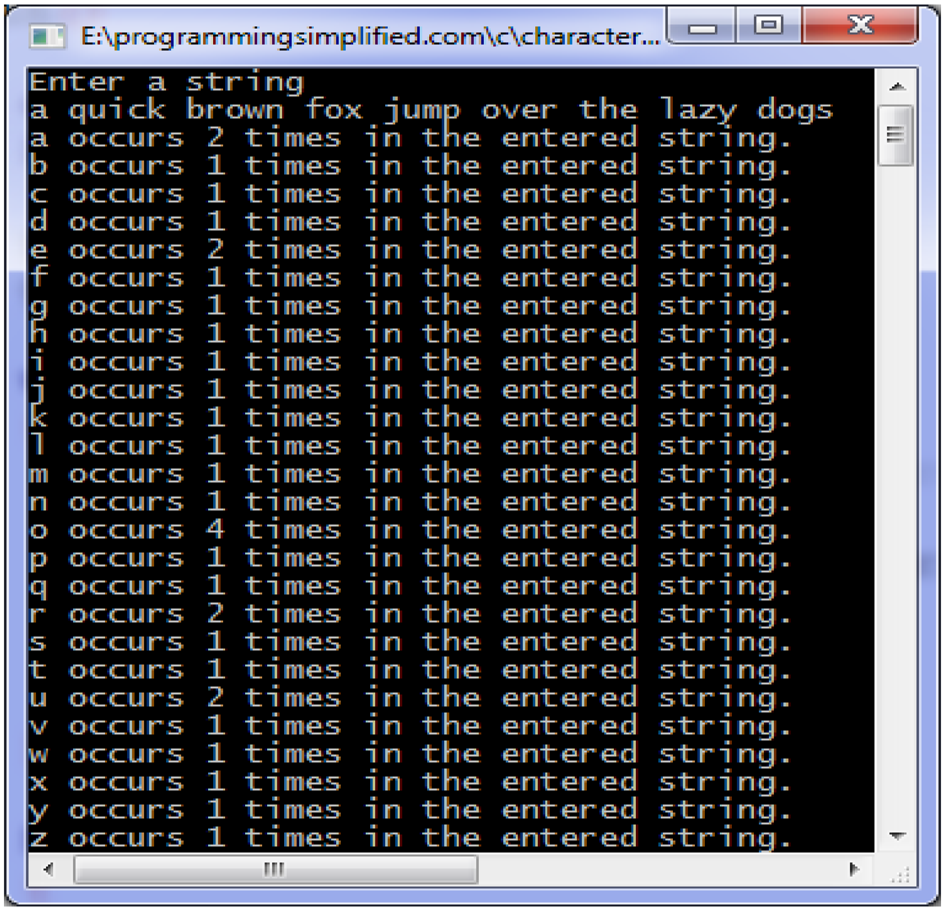
Description: C program to find the frequency of characters in a string: This program counts the frequency of characters in a string, i.e., which character is present how many times in the string. For example, in the string "code" each of the characters 'c,' 'd,' 'e,' and 'o' has occurred one time. Only ***lower case alphabets*** are considered, other characters (uppercase and special characters) are ignored. You can easily modify this program to handle uppercase and special symbols.

Source Code:

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

int main()
{
  char string[100];
  int c = 0, count[26] = {0}, x;

  printf("Enter a string\n");
  gets(string);

  while (string[c] != '\0') {
  /** Considering characters from 'a' to 'z' only and ignoring others. */

    if (string[c] >= 'a' && string[c] <= 'z') {
      x = string[c] - 'a';
      count[x]++;
    }

    c++;
  }

  for (c = 0; c < 26; c++)
      printf("%c occurs %d times in the string.\n", c + 'a', count[c]);

  return 0;
}
```

Explanation of "count[string[c]-'a']++", suppose input string begins with 'a' so c is 0 initially and string[0] = 'a' and string[0] - 'a' = 0 and we increment count[0] i.e. 'a' has occurred one time and repeat this till the complete string is scanned.

Output:



```
Enter a string
a quick brown fox jump over the lazy dogs
a occurs 2 times in the entered string.
b occurs 1 times in the entered string.
c occurs 1 times in the entered string.
d occurs 1 times in the entered string.
e occurs 2 times in the entered string.
f occurs 1 times in the entered string.
g occurs 1 times in the entered string.
h occurs 1 times in the entered string.
i occurs 1 times in the entered string.
j occurs 1 times in the entered string.
k occurs 1 times in the entered string.
l occurs 1 times in the entered string.
m occurs 1 times in the entered string.
n occurs 1 times in the entered string.
o occurs 4 times in the entered string.
p occurs 1 times in the entered string.
q occurs 1 times in the entered string.
r occurs 2 times in the entered string.
s occurs 1 times in the entered string.
t occurs 1 times in the entered string.
u occurs 2 times in the entered string.
v occurs 1 times in the entered string.
w occurs 1 times in the entered string.
x occurs 1 times in the entered string.
y occurs 1 times in the entered string.
z occurs 1 times in the entered string.
```

`
-

Viva Questions:

Q1- Describe use of fopen() function.
Q2- Describe the purpose of ftell
Q3- Define the purpose of rewind().
Q4- Describe the sequential access file?
Q5- Define the syntax error.

Experiment No. 6

Aim: Write a C program to insert, delete and display the entries in Symbol Table.

Tools / Software: Turbo C++ /Code Blocks

Description: A Symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source

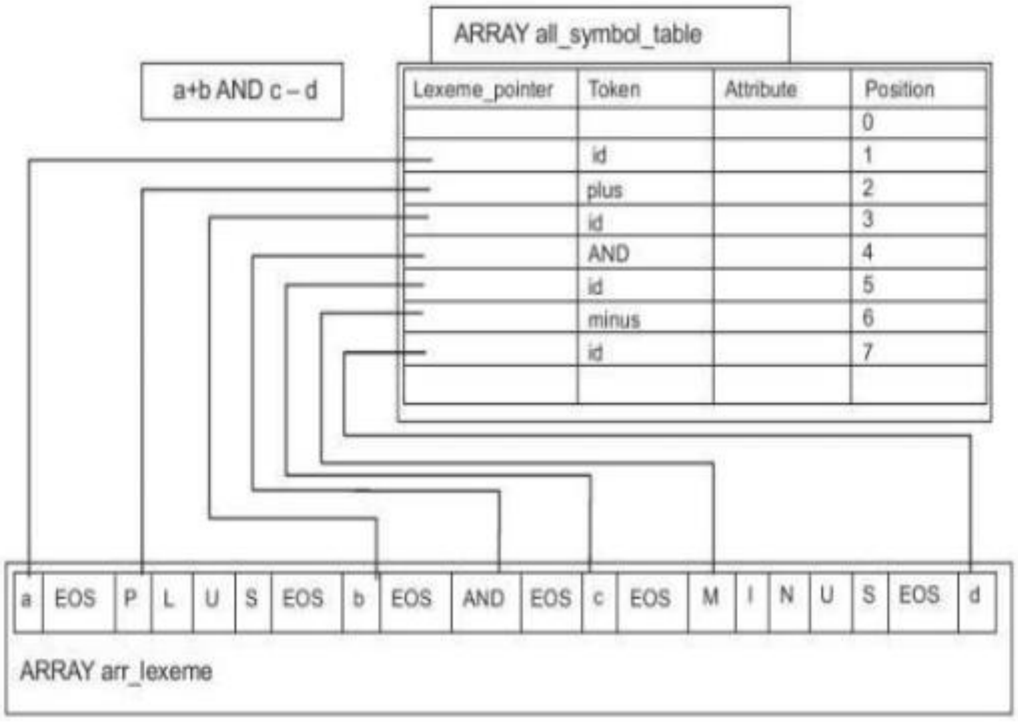
Possible entries in a symbol table:

- Name: a string
- Attribute:

1. Reserved word
2. Variable name
3. Type Name
4. Procedure name
5. Constant name

- Data type
- Scope information: where it can be used.
- Storage allocation

SYMBOL TABLE



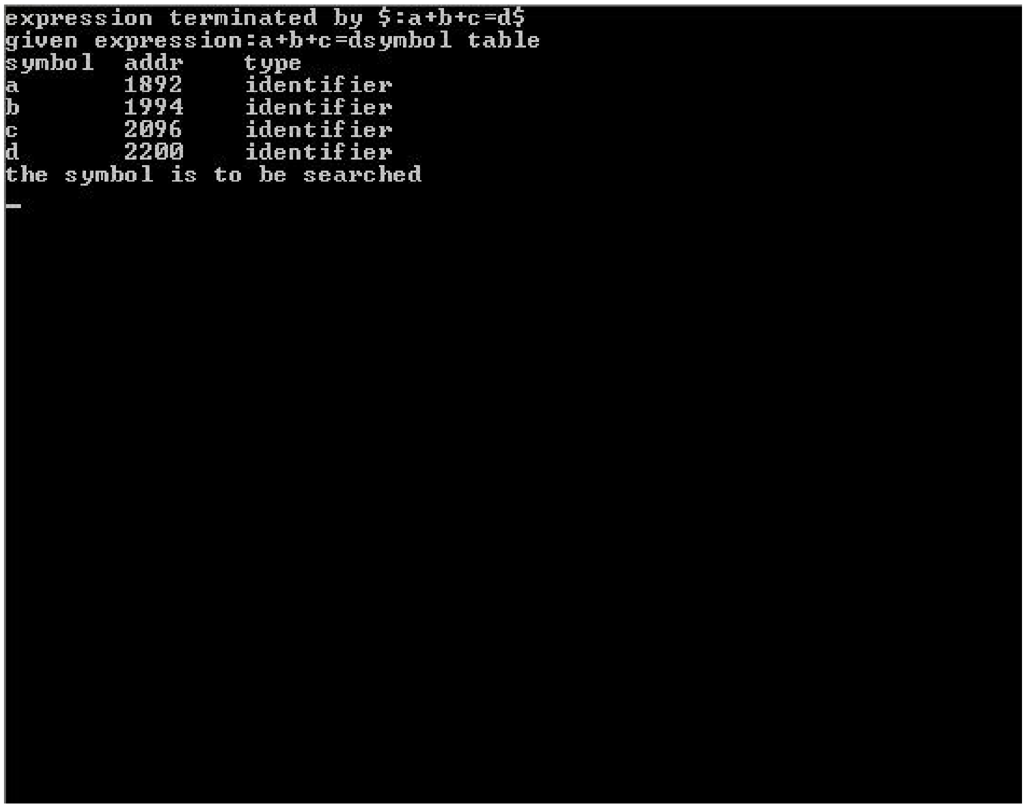
Source code:

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
#include<string.h>
#include<stdlib.h>
#define NULL 0
int size=0;
void Insert();
void Display();
void Delete();
int Search(char lab[]);void Modify();
struct SymbTab
{
char label[10],symbol[10];
int addr;
struct SymbTab *next;};
struct SymbTab *first,*last;
void main()
{
int op,y;
char la[10];
clrscr();
do
{
printf("\n\tSYMBOL TABLE IMPLEMENTATION\n");
printf("\n\t1.INSERT\n\t2.DISPLAY\n\t3.DELETE\n\t4.SEARCH\n\t5.MODIFY\n\t6.END\n");
printf("\n\tEnter your option : ");
scanf("%d",&op);
switch(op)
{
case 1:
Insert();
break;
case 2:
Display();
break;
case 3:
Delete();
break;
case 4:
printf("\n\tEnter the label to be searched : ");
scanf("%s",la);
y=Search(la);
printf("\n\tSearch Result:");
if(y==1)
printf("\n\tThe label is present in the symbol table\n");
else
printf("\n\tThe label is not present in the symbol table\n");
break;
case 5:
Modify();
break;
case 6:
exit(0);
}
}while(op<6);
getch();
}
void Insert()
{
int n;
char l[10];
printf("\n\tEnter the label : ");
scanf("%s",l);
n=Search(l);
if(n==1)
printf("\n\tThe label exists already in the symbol table\n\tDuplicate can't be inserted");
else
{
struct SymbTab *p;
p=malloc(sizeof(struct SymbTab));
strcpy(p->label,l);
printf("\n\tEnter the symbol : ");
scanf("%s",p->symbol);
printf("\n\tEnter the address : ");
scanf("%d",&p->addr);
p->next=NULL;
if(size==0)
{
first=p;
last=p;
}
else
{
last->next=p;
last=p;
}
size++;
}
printf("\n\tLabel inserted\n");
}
void Display()
{
```

```
int i;
struct SymbTab *p;
p=first;
printf("\n\tLABEL\t\tSYMBOL\t\tADDRESS\n");
for(i=0;i<size;i++)
{
printf("\t%s\t\t%s\t\t%d\n",p->label,p->symbol,p->addr);
p=p->next;
}
}
int Search(char lab[])
{
int i,flag=0;
struct SymbTab *p;
p=first;
for(i=0;i<size;i++)
{
if(strcmp(p->label,lab)==0)
flag=1;
p=p->next;
}
return flag;
}
void Modify()
{
char l[10],nl[10];
int add,choice,i,s;
struct SymbTab *p;
p=first;
printf("\n\tWhat do you want to modify?\n");
printf("\n\t1.Only the label\n\t2.Only the address\n\t3.Both the label and address\n");
printf("\tEnter your choice : ");
scanf("%d",&choice);
switch(choice)
{
case 1:
printf("\n\tEnter the old label : ");
scanf("%s",l);
s=Search(l);
if(s==0)
printf("\n\tLabel not found\n");
else
{
printf("\n\tEnter the new label : ");
scanf("%s",nl);
for(i=0;i<size;i++)
{
if(strcmp(p->label,l)==0)
strcpy(p->label,nl);
p=p->next;
}
printf("\n\tAfter Modification:\n");
Display();
}
break;
case 2:
printf("\n\tEnter the label where the address is to be modified : ");
scanf("%s",l);
s=Search(l);
if(s==0)
printf("\n\tLabel not found\n");
else
{
printf("\n\tEnter the new address : ");
scanf("%d",&add);
for(i=0;i<size;i++)
{
if(strcmp(p->label,l)==0)
p->addr=add;
p=p->next;
}
printf("\n\tAfter Modification:\n");
Display();
}
break;
case 3:
printf("\n\tEnter the old label : ");
scanf("%s",l);
s=Search(l);
if(s==0)
printf("\n\tLabel not found\n");
else
{
printf("\n\tEnter the new label : ");
scanf("%s",nl);
printf("\n\tEnter the new address : ");
scanf("%d",&add);
for(i=0;i<size;i++)
{
if(strcmp(p->label,l)==0)
{
strcpy(p->label,nl);
p->addr=add;
}
p=p->next;
}
```

```
printf("\n\tAfter Modification:\n");
Display();
}
break;
}
}
void Delete()
{
int a;
char l[10];
struct SymbTab *p,*q;
p=first;
printf("\n\tEnter the label to be deleted : ");
scanf("%s",l);
a=Search(l);
if(a==0)
printf("\n\tLabel not found\n");
else
{
if(strcmp(first->label,l)==0)
first=first->next;
else if(strcmp(last->label,l)==0)
{
q=p->next;
while(strcmp(q->label,l)!=0)
{
p=p->next;
q=q->next;
}
p->next=NULL;
last=p;
}
else
{
q=p->next;
while(strcmp(q->label,l)!=0)
{
p=p->next;
q=q->next;
}
p->next=q->next;
}
size–;
printf("\n\tAfter Deletion:\n");
Display();
}
}
```

Output:



```
expression terminated by $:a+b+c=d$
given expression:a+b+c=dsymbol table
symbol  addr    type
a       1892    identifier
b       1994    identifier
c       2096    identifier
d       2200    identifier
the symbol is to be searched
_
```

Viva Questions:

Q1- Describe Symbol table.
Q2- List the phases that constitute the front-end of compiler.
Q3- Define the back- end phases of a compiler.
Q4- Describe the various methods of implementing three address statements.
Q5- Suggest a suitable approach for computing hash function.

Experiment No. 7

Aim: Write a LEX program to identify following:

1. Valid mobile number & email-id
2. Valid url
3. Valid identifier
4. Valid date (dd/mm/yyyy)
5. Valid time (hh:mm:ss)

Tools / Software: LEX/ Flex Tool.

Description: FLEX (Fast Lexical Analyzer Generator) is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language. The function yylex() is the main flex function which runs the Rule Section.

- The lex is used in the manner depicted. A specification of the lexical analyzer is preferred by

creating a program lex.1 in the lex language.

- Then lex.1 is run through the lex compiler to produce a 'c' program lex.yy.c.
- The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expression of lex.1 together with a standard routine that uses table of recognize leximes.
- Lex.yy.c is run through the 'C' compiler to produce as object program a.out, which is the lexical

analyzer that transform as input stream into sequence of tokens.

Source Code:

1. Valid mobile number & email-id

Valid mobile number

```
%{
   /* Definition section */
%}
  /* Rule Section */
%%
```

```
  [1-9][0-9]{9} {printf("\nMobile Number Valid\n");}
  .+ {printf("\nMobile Number Invalid\n");}
  %%
  // driver code
int main()
{
    printf("\nEnter Mobile Number : ");
    yylex();
    printf("\n");
    return 0;
}
```

Output:

Input: 7017175023

Output: Mobile Number Valid

Input: 0001112223

Output: Mobile Number Invalid

Valid mobile email-id

```
%
{
  int flag = 0; %
} %
% [a - z.0 - 9 _] + @[a - z] + ".com" | ".in"
flag = 1; %
%
main() {
  yylex();
  if (flag == 1)
```

```
    printf("Accepted");
  else
    printf("Not Accepted");
}
```

Output:

Input :

csetanmayjain@gmail.com

Output :

Valid

Input :

!23@43.com

Output :

Not Valid

2. Valid url

```
%%
((http)|(ftp))s?:\/\/[a-zA-Z0-9]{2, }(\.[a-z]{2, })
       +(\/[a-zA-Z0-9+=?]*)* {printf("\nURL Valid\n");}

.+ {printf("\nURL Invalid\n");}

%%
  // driver program
void main()
 {
    printf("\nEnter URL : ");
    yylex();
    printf("\n");
 }
```

Output:

Input: geeksforgeeks

Output: INVALID URL

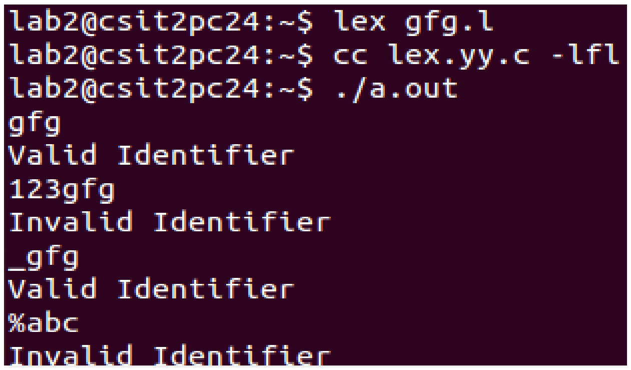
Input: https://www.geeksforgeeks.org

Output: VALID URL

3. Valid identifier

```
% {
#include <stdio.h>
    %
}
      / rule section % %
    // regex for valid identifiers
    ^[a - z A - Z _][a - z A - Z 0 - 9 _] * printf("Valid Identifier");
  // regex for invalid identifiers
^[^a - z A - Z _] printf("Invalid Identifier");
.;
% %
      main()
{
    yylex();
}
```

Output:



```
lab2@csit2pc24:~$ lex gfg.l
lab2@csit2pc24:~$ cc lex.yy.c -lfl
lab2@csit2pc24:~$ ./a.out
gfg
Valid Identifier
123gfg
Invalid Identifier
_gfg
Valid Identifier
%abc
Invalid Identifier
```

4. Valid date (dd/mm/yyyy)

```
%{
   /* Definition section */
   #include<stdio.h>
   int i=0, yr=0, valid=0;
%}
  /* Rule Section */
%%
([0-2][0-9]|[3][0-1])\/((0(1|3|5|7|8))|(10|12))
            \/([1-2][0-9][0-9][-0-9]) {valid=1;}
  ([0-2][0-9]|30)\/((0(4|6|9))|11)
         \/([1-2][0-9][0-9][0-9]) {valid=1;}
  ([0-1][0-9]|2[0-8])\/02
            \/([1-2][0-9][0-9][0-9]) {valid=1;}
  29\/02\/([1-2][0-9][0-9][0-9])
     { while(yytext[i]!='/')i++; i++;
       while(yytext[i]!='/')i++;i++;
       while(i<yyleng)yr=(10*yr)+(yytext[i++]-'0');
       if(yr%4==0||(yr%100==0&&yr%400!=0))valid=1;}
  %%
  // driver code
main()
{
  yyin=fopen("vpn.txt", "r");
  yylex();
  if(valid==1) printf("It is a valid date\n");
  else printf("It is not a valid date\n");
}
 int yywrap()
{
 return 1;
}
```

Output:

Input: 02/05/2019

Output: It is a valid date

Input: 05/20/2019

Output: It is not a valid date

5. Valid time (hh:mm:ss)

Viva Questions:

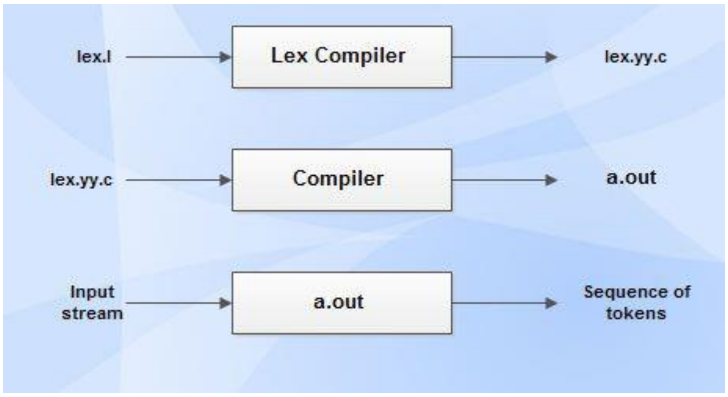
Q1- Describe the is lexeme.
Q2- Differentiate between token and lexeme
Q3- Define lexical analyzer.
Q4- Describe the various error recovery strategies for lexical analysis.
Q5- Define the lex source program.

Experiment No.8

Aim: Write a lex program to count blank spaces, words, lines in a given file.

Tools / Software: LEX/ Flex Tool.

Description: Lex is a computer program that generates lexical analyzers and was written by Mike Lesk and Eric Schmidt. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language. Let's the how to count the blank space, wors, lines, using Lex.

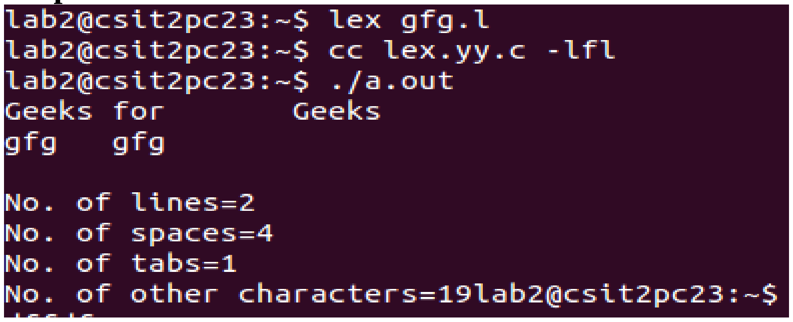


Source Code:

```
%{
#include<stdio.h>
int lc=0, sc=0, tc=0, ch=0; /*Global variables*/
%}
 /*Rule Section*/
%%
\n lc++; //line counter
[ ] sc++; //space counter
\t tc++; //tab counter
. ch++;    //characters counter
%%
 main()
{
```

```
// The function that starts the analysis
yylex();
printf("\nNo. of lines=%d, lc);
printf("\nNo. of spaces=%d, sc);
printf("\nNo. of tabs=%d, tc);
printf("\nNo. of other characters=%d, ch);
  }
```

Output:



```
lab2@csit2pc23:~$ lex gfg.l
lab2@csit2pc23:~$ cc lex.yy.c -lfl
lab2@csit2pc23:~$ ./a.out
Geeks for       Geeks
gfg   gfg

No. of lines=2
No. of spaces=4
No. of tabs=1
No. of other characters=19lab2@csit2pc23:~$
```

Viva Questions:

Q1- Describe the structure of lex program.
Q2- Define an internal command.
Q3- Define the yyleng.
Q4- Describe the Code Motion.
Q5- Define the **Contents** of Activation Record.

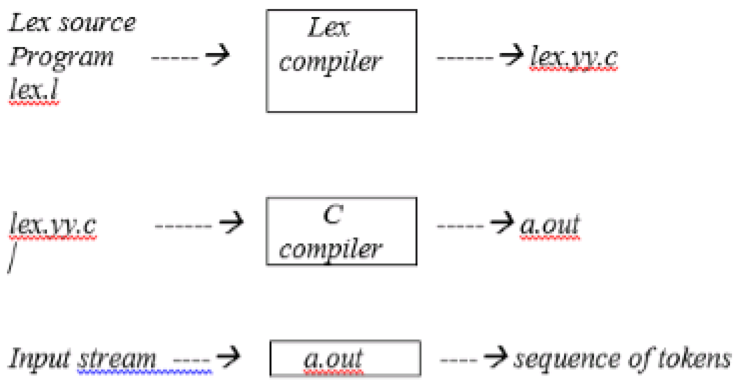
Experiment No. 9

Aim: Write a lex program to count the no. of vowels and consonants in a C file.

Tools / Software: LEX/ Flex Tool.

Description: LEX

- Lex is officially known as a "Lexical Analyser".
- Its main job is to break up an input stream into more usable elements. Or in, other words, to identify the "interesting bits" in a text file.
- For example, if you are writing a compiler for the C programming language, the symbols { } ( ); all have significance on their own.
- The letter a usually appears as part of a keyword or variable name, and is not interesting on its own.
- Instead, we are interested in the whole word. Spaces and newlines are completely uninteresting, and we want to ignore them completely, unless they appear within quotes "like this"
- All of these things are handled by the Lexical Analyser.
- A tool widely used to specify lexical analyzers for a variety of languages
- We refer to the tool as Lex compiler, and to its input specification as the Lex language.



Creating a lexical analyzer with Lex

Source Code:-

```
%{
int vow_count=0;
int const_count =0;
%}
%%
[aeiouAEIOU] {vow_count++;}
```

```
[a-zA-Z] {const_count++;}
%%
main()
{
printf("Enter the string of vowels and consonents:");
yylex();
printf("The number of vowels are:  %d\n",vow);
printf("The number of consonants are:  %d\n",cons);
return 0;
}
```

Output:

```
$$ lex kk.l
$$ gcc kk.c -ll
$$ ./a.out
Enter the string of vowels and consonents
My name is Khan
The number of vowels are: 4
The number of consonants are: 8
```

Viva Questions:

Q1- Differentiate the commands cp and mv.
Q2- Define shell scripts.
Q3- Define the $ls-l.
Q4- Describe the an absolute path name with help of example.
Q5- List The Various Error Recovery Strategies For A Lexical Analysis**.**

Experiment No. 10

Aim: Write a YACC program to recognize strings aaab, abbb using a^nb^n, where b>=0.

Tools / Software: YACC Tool.

Description: Yacc (for "yet another compiler compiler.") is the standard parser generator for the Unix operating system. An open source program, yacc generates code for the parser in the C programming language. The acronym is usually rendered in lowercase but is occasionally seen as YACC or Yacc.

Algo:

- Start the program.
- Write the code for parser. l in the declaration port.
- Write the code for the 'y' parser.
- Also write the code for different arithmetical operations.
- Write additional code to print the result of computation.
- Execute and verify it.
- Stop the program.

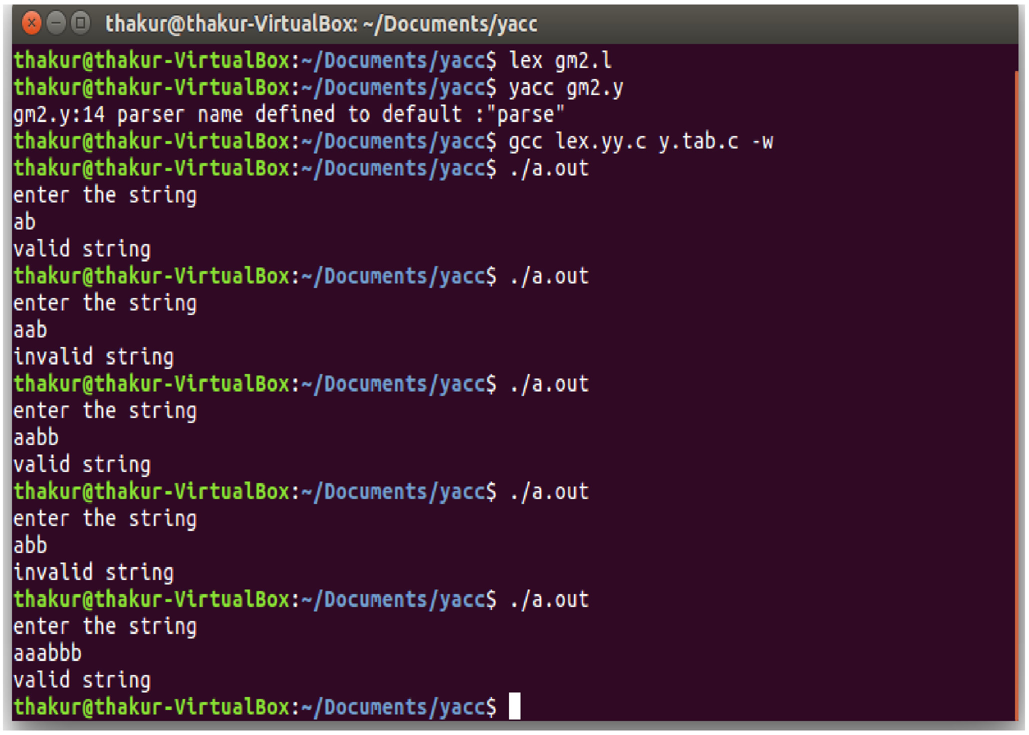
Source Code:

```
%{
  /* Definition section */
 #include "y.tab.h"
 %}
 /* Rule Section */
 %%
[aA] {return A;}
[bB] {return B;}
\n {return NL;}
.  {return yytext[0];}
%%
 int yywrap()
 {
 return 1;
 }
```

Parser Source Code :

```
%{
   /* Definition section */
   #include<stdio.h>
   #include<stdlib.h>
%}
  %token A B NL
  /* Rule Section */
%%
stmt: S NL  { printf("valid string\n");
              exit(0); }
;
S: A S B |
;
%%
  int yyerror(char *msg)
 {
  printf("invalid string\n");
  exit(0);
 }
  //driver code
main()
 {
  printf("enter the string\n");
  yyparse();
 }
```

Output:



```
thakur@thakur-VirtualBox: ~/Documents/yacc
thakur@thakur-VirtualBox:~/Documents/yacc$ lex gm2.l
thakur@thakur-VirtualBox:~/Documents/yacc$ yacc gm2.y
gm2.y:14 parser name defined to default :"parse"
thakur@thakur-VirtualBox:~/Documents/yacc$ gcc lex.yy.c y.tab.c -w
thakur@thakur-VirtualBox:~/Documents/yacc$ ./a.out
enter the string
ab
valid string
thakur@thakur-VirtualBox:~/Documents/yacc$ ./a.out
enter the string
aab
invalid string
thakur@thakur-VirtualBox:~/Documents/yacc$ ./a.out
enter the string
aabb
valid string
thakur@thakur-VirtualBox:~/Documents/yacc$ ./a.out
enter the string
abb
invalid string
thakur@thakur-VirtualBox:~/Documents/yacc$ ./a.out
enter the string
aaabbb
valid string
thakur@thakur-VirtualBox:~/Documents/yacc$
```

Viva Questions:

Q1- Describe the YAAC tool.
Q2- Define the LR(0) items.
Q3- Define the viable prefix.
Q4- Describe the yytext.
Q5- List The Various Error Recovery Strategies For A Lexical Analysis**.**

Experiment No. 11

Aim: Write a YACC program to evaluate an arithmetic expression involving operators +,-,*and /

Tools / Software: YACC Tool.

Description: Here arithmetic expression may have operations like Addition(+), Subtraction(-), Multiplication(*), Division(/) or Modulus(%). Expression may contain balanced round brackets.

YACC program also consists of three sections, "Definitions", "Context Free Grammar and action for each production", "Subroutines/Functions".

In first section, we can mention C language code which may consist of header files inclusion, global variables/ Constants definition/declaration. C language code can be mentioned in between the symbols %{ and %}. Also we can define tokens in the first section. In above program, NUMBER is the token. We can define the associativity of the operations (i.e. left associativity or right associativity). In above yacc program, we have specified left associativity for all operators. Priorities among the operators can also be specified. It is in the increasing order "from top to bottom". For e.g. in our above yacc program, round brackets '(',')' has the higher priority than '*', '/', '%' which has higher priority than '+', '-'. Operators in the same statement have the same priority. For e.g. in our above program all of the '*', '/', '%' have the same priority.

In second section, we mention the grammar productions and the action for each production. $$ refer to the top of the stack position while $1 for the first value, $2 for the second value in the stack.

Third section consists of the subroutines. We have to call yyparse() to initiate the parsing process. yyerror() function is called when all productions in the grammar in second section do not match to the input statement.

Source Code:

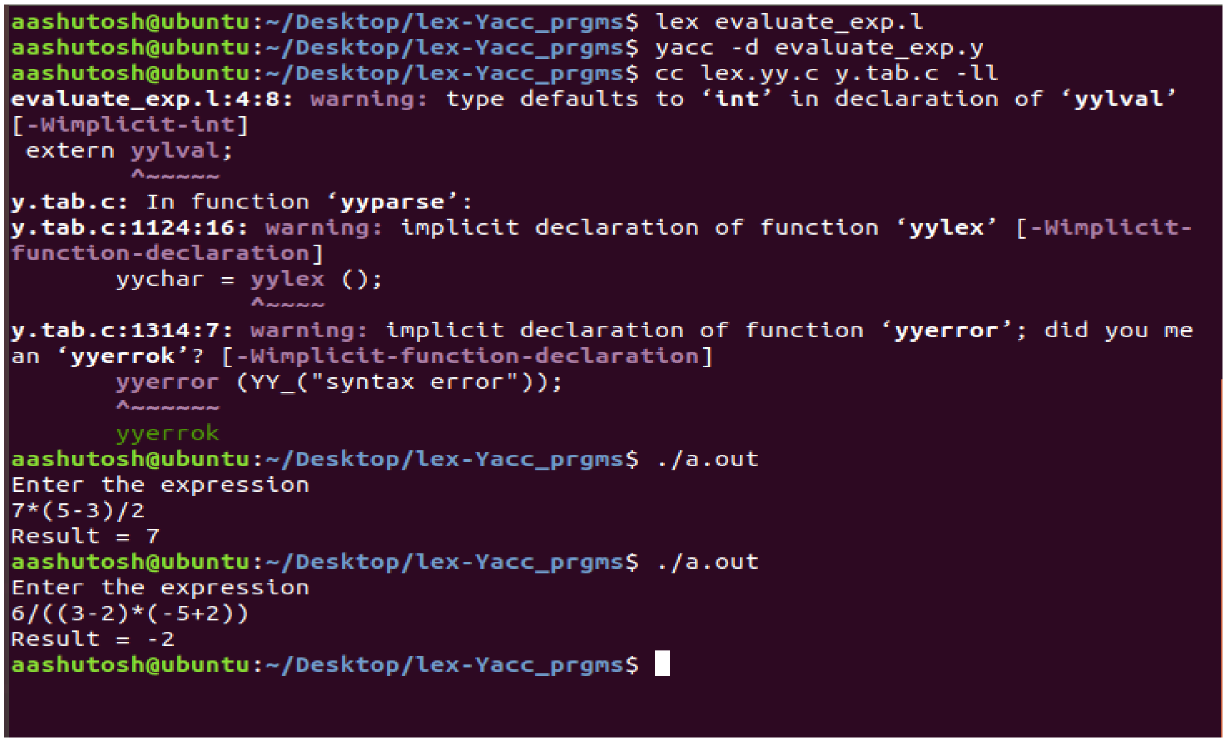
```
%
 {
   /* Definition section*/
   #include "y.tab.h"
   extern yylval;
   %
 }
  % %
      [0 - 9]
   +
 {
   yylval = atoi(yytext);
   return NUMBER;
 }
  [a - zA - Z] + { return ID; }
[\t] + ;
  \n { return 0; }
. { return yytext[0]; }
  % %
```

Parser Source Code:

```
% {
  /* Definition section */
#include
    %
   }
% token NUMBER ID
// setting the precedence
// and associativity of operators
```

```
% left '+' '-'
% left '*' '/'
  /* Rule Section */
%
% E : T
  {
    printf("Result = %d\n", $$);
  return 0;
    }
T : T '+' T { $$ = $1 + $3; }
| T '-' T { $$ = $1 - $3; }
| T '*' T { $$ = $1 * $3; }
| T '/' T { $$ = $1 / $3; }
| '-' NUMBER { $$ = -$2; }
| '-' ID { $$ = -$2; }
| '(' T ')' { $$ = $2; }
| NUMBER { $$ = $1; }
| ID { $$ = $1; };
% %
      int main()
 {
    printf("Enter the expression\n");
    yyparse();
 }
  /* For printing error messages */
int yyerror(char* s)
 {
    printf("\nExpression is invalid\n");
 }
```

Output:



```
aashutosh@ubuntu:~/Desktop/lex-Yacc_prgms$ lex evaluate_exp.l
aashutosh@ubuntu:~/Desktop/lex-Yacc_prgms$ yacc -d evaluate_exp.y
aashutosh@ubuntu:~/Desktop/lex-Yacc_prgms$ cc lex.yy.c y.tab.c -ll
evaluate_exp.l:4:8: warning: type defaults to ‘int’ in declaration of ‘yylval’
[-Wimplicit-int]
 extern yylval;
        ^~~~~~
y.tab.c: In function ‘yyparse’:
y.tab.c:1124:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-
function-declaration]
       yychar = yylex ();
                ^~~~~
y.tab.c:1314:7: warning: implicit declaration of function ‘yyerror’; did you me
an ‘yyerrok’? [-Wimplicit-function-declaration]
       yyerror (YY_("syntax error"));
       ^~~~~~~
       yyerrok
aashutosh@ubuntu:~/Desktop/lex-Yacc_prgms$ ./a.out
Enter the expression
7*(5-3)/2
Result = 7
aashutosh@ubuntu:~/Desktop/lex-Yacc_prgms$ ./a.out
Enter the expression
6/((3-2)*(-5+2))
Result = -2
aashutosh@ubuntu:~/Desktop/lex-Yacc_prgms$ 
```

Viva Questions:

Q1- Describe a Operator Precedence Parser.
Q2- Define the various types of Intermediate Code Representation.
Q3- Define the BNF.
Q4- Describe the list The operations on languages.
Q5- Describe the problems with top down parsing.

Experiment No. 12

Aim: Write a YACC program to recognize string with grammar a^n b^n, n>=0

Tools / Software: YACC Tool.

Description: Yacc (for "yet another compiler compiler.") is the standard parser generator for the Unix operating system. An open source program, yacc generates code for the parser in the C programming language. The acronym is usually rendered in lowercase but is occasionally seen as YACC or Yacc.

Grammars are used to describe the syntax of a programming language. It specifies the structure of expression and statements.

stmt -> if (expr) then stmt

Source Code:

YACC PART:

CODE: (gram.y)

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token A B NL
%%
stmt: S NL {printf("valid string\n");
            exit(0);}
;
S: A S B |
;
%%
int yyerror(char *msg)
```

```
{
printf("invalid string\n");
exit(0);
}
main()
{
printf("enter the string\n");
yyparse();
}
```

LEX PART:

CODE: (gram.l)

```
%{
#include "y.tab.h"
%}
%%
[aA] {return A;}
[bB] {return B;}
\n {return NL;}
. {return yytext[0];}
%%
```

OUTPUT :

```
yacc -d gram.y
lex gram.l
cc y.tab.c lex.yy.c -ly -ll
./a.out
enter the string
ab
valid string
```

```
./a.out
enter the string
aaabb
invalid string

./a.out
enter the string
aabb
valid string

./a.out
enter the string
a
invalid string
```

Viva Questions:

Q1- Describe regular expression**.**
Q2- Describe the context free grammar.
Q3- Define the concept of derivation.
Q4- Describe the functionality of lex and yacc tool.
Q5- The YACC takes C code as input and outputs**__________**

Experiment No. 13

Aim: Write a C program to find first of any grammar**.**

Tools / Software: Turbo C++ / Code Blocks.

Description & Algorithm:

The functions follow and followfirst 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-teminal 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. Grammer 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.

Note : For the second case it is very easy to fall prey to an INFINITE LOOP even if the code looks perfect. So it is important to keep track of all the function calls at all times and never call the same function again.

Below is the implementation :

<u>Source Code:-</u>

```
// C program to calculate the First and

// Follow sets of a given grammar

#include<stdio.h>

#include<ctype.h>

#include<string.h>

// Functions to calculate Follow

void followfirst(char, int, int);

void follow(char c);

// Function to calculate First

void findfirst(char, int, int);
```

```
int count, n = 0;

// Stores the final result

// of the First Sets

char calc_first[10][100];

// Stores the final result

// of the Follow Sets

char calc_follow[10][100];

int m = 0;

// Stores the production rules

char production[10][10];

char f[10], first[10];

int k;

char ck;

int e;

  int main(int argc, char **argv)

{

    int jm = 0;

    int km = 0;

    int i, choice;

    char c, ch;
```

```
count = 8;

// The Input grammar

strcpy(production[0], "E=TR");

strcpy(production[1], "R=+TR");

strcpy(production[2], "R=#");

strcpy(production[3], "T=FY");

strcpy(production[4], "Y=*FY");

strcpy(production[5], "Y=#");

strcpy(production[6], "F=(E)");

strcpy(production[7], "F=i");

int kay;

char done[count];

int ptr = -1;

// Initializing the calc_first array

for(k = 0; k < count; k++) {

    for(kay = 0; kay < 100; kay++) {

        calc_first[k][kay] = '!';

    }

}

int point1 = 0, point2, xxx;

for(k = 0; k < count; k++)
```

```
{
    c = production[k][0];
    point2 = 0;
    xxx = 0;
      // Checking if First of c has
    // already been calculated
    for(kay = 0; kay <= ptr; kay++)
        if(c == done[kay])
            xxx = 1;
       if (xxx == 1)
        continue;
      // Function call
    findfirst(c, 0, 0);
    ptr += 1;
      // Adding c to the calculated list
    done[ptr] = c;
    printf("\n First(%c) = { ", c);
    calc_first[point1][point2++] = c;
      // Printing the First Sets of the grammar
    for(i = 0 + jm; i < n; i++) {
        int lark = 0, chk = 0;
```

```
            for(lark = 0; lark < point2; lark++) {
              if (first[i] == calc_first[point1][lark])
              {
                  chk = 1;
                  break;
              }
          }
          if(chk == 0)
          {
              printf("%c, ", first[i]);
              calc_first[point1][point2++] = first[i];
          }
      }
      printf("}\n");
      jm = n;
      point1++;
  }
  printf("\n");
  printf("-----------------------------------------------\n\n");
  char donee[count];
  ptr = -1;
```

```
// Initializing the calc_follow array
for(k = 0; k < count; k++) {
    for(kay = 0; kay < 100; kay++) {
        calc_follow[k][kay] = '!';
    }
}
point1 = 0;
int land = 0;
for(e = 0; e < count; e++)
{
    ck = production[e][0];
    point2 = 0;
    xxx = 0;
      // Checking if Follow of ck
    // has alredy been calculated
    for(kay = 0; kay <= ptr; kay++)
        if(ck == donee[kay])
            xxx = 1;
      if (xxx == 1)
        continue;
```

```
land += 1;

   // Function call

follow(ck);

ptr += 1;

   // Adding ck to the calculated list

donee[ptr] = ck;

printf(" Follow(%c) = { ", ck);

calc_follow[point1][point2++] = ck;

 // Printing the Follow Sets of the grammar

for(i = 0 + km; i < m; i++) {

    int lark = 0, chk = 0;

    for(lark = 0; lark < point2; lark++)

    {

        if (f[i] == calc_follow[point1][lark])

        {

            chk = 1;

            break;

        }

    }

    if(chk == 0)

    {
```

```
                printf("%c, ", f[i]);

                calc_follow[point1][point2++] = f[i];

            }

        }

        printf(" }\n\n");

        km = m;

        point1++;

    }

}

 void follow(char c)

{

    int i, j;

      // Adding "$" to the follow

    // set of the start symbol

    if(production[0][0] == c) {

        f[m++] = '$';

    }

    for(i = 0; i < 10; i++)

    {

        for(j = 2;j < 10; j++)

        {
            if(production[i][j] == c)
            {
                if(production[i][j+1] != '\0')
                {
                    // Calculate the first of the next
                    // Non-Terminal in the production
                    followfirst(production[i][j+1], i, (j+2));
                }
                  if(production[i][j+1]=='\0' && c!=production[i][0])
                {
                    // Calculate the follow of the Non-Terminal
                    // in the L.H.S. of the production
                    follow(production[i][0]);
                }
            }
        }
    }
}
void findfirst(char c, int q1, int q2)
{
    int j;
```

```
 // The case where we
// encounter a Terminal
if(!(isupper(c))) {
    first[n++] = c;
}
for(j = 0; j < count; j++)
{
    if(production[j][0] == c)
    {
        if(production[j][2] == '#')
        {
            if(production[q1][q2] == '\0')
                first[n++] = '#';
            else if(production[q1][q2] != '\0'
                     && (q1 != 0 || q2 != 0))
            {
                // Recursion to calculate First of New
                // Non-Terminal we encounter after epsilon
                findfirst(production[q1][q2], q1, (q2+1));
            }
            else
```

```
                    first[n++] = '#';
                }
                else if(!isupper(production[j][2]))
                {
                    first[n++] = production[j][2];
                }
                else
                {
                    // Recursion to calculate First of
                    // New Non-Terminal we encounter
                    // at the beginning
                    findfirst(production[j][2], j, 3);
                }
            }
        }
    }
     void followfirst(char c, int c1, int c2)
    {
        int k;
          // The case where we encounter
        // a Terminal
```

```
if(!(isupper(c)))
    f[m++] = c;
else
{
    int i = 0, j = 1;
    for(i = 0; i < count; i++)
    {
        if(calc_first[i][0] == c)
            break;
    }
      //Including the First set of the
    // Non-Terminal in the Follow of
    // the original query
    while(calc_first[i][j] != '!')
    {
        if(calc_first[i][j] != '#')
        {
            f[m++] = calc_first[i][j];
        }
        else
        {
                if(production[c1][c2] == '\0')
                {
                    // Case where we reach the
                    // end of a production
                    follow(production[c1][0]);
                }
                else
                {
                    // Recursion to the next symbol
                    // in case we encounter a "#"
                    followfirst(production[c1][c2], c1, c2+1);
                }
            }
            j++;
        }
    }
}
```

Output:

```
First(E)= { (, i, }

First(R)= { +, #, }

First(T)= { (, i, }

First(Y)= { *, #, }
First(F)= { (, i, }


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


Follow(E) = { $, ),  }
Follow(R) = { $, ),  }
Follow(T) = { +, $, ),  }
Follow(Y) = { +, $, ),  }
Follow(F) = { *, +, $, ),  }
```

Viva Questions:

Q1- Describe Predictive parser.
Q2- Define the Recursive Decent Parser.
Q3- Define the types of analysis can we do using Parser.
Q4- Describe the algorithm for First and follow.
Q5- Describe the Rule for calculating the First.

Reference

V.V Das, Compiler Design using FLEX and YACC, PHI

BOOKS:

Text books:

1. Aho, Ullman and Sethi: Compilers, Addison Wesley.
2. Holub, Compiler Design in C, PHI.

Reference Books:

1. V.V Das, Compiler Design using FLEX and YACC, PHI