

COMPILER DESIGN

LABORATORY MANUAL

**B.TECH
(III YEAR – I SEM)
(2018-19)**

**DEPARTMENT OF COMPUTER SCIENCE AND
ENGINEERING**



**MALLA REDDY COLLEGE OF ENGINEERING &
TECHNOLOGY**

(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12 (B) of UGC ACT 1956

Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, India

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION

- To improve the quality of technical education that provides efficient software engineers with an attitude to adapt challenging IT needs of local, national and international arena, through teaching and interaction with alumni and industry.

MISSION

- Department intends to meet the contemporary challenges in the field of IT and is playing a vital role in shaping the education of the 21st century by providing unique educational and research opportunities.

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

PEO1 – ANALYTICAL SKILLS

To facilitate the graduates with the ability to visualize, gather information, articulate, analyze, solve complex problems, and make decisions. These are essential to address the challenges of complex and computation intensive problems increasing their productivity.

PEO2 – TECHNICAL SKILLS

To facilitate the graduates with the technical skills that prepare them for immediate employment and pursue certification providing a deeper understanding of the technology in advanced areas of computer science and related fields, thus encouraging to pursue higher education and research based on their interest.

PEO3 – SOFT SKILLS

To facilitate the graduates with the soft skills that include fulfilling the mission, setting goals, showing self-confidence by communicating effectively, having a positive attitude, get involved in team-work, being a leader, managing their career and their life.

PEO4 – PROFESSIONAL ETHICS

To facilitate the graduates with the knowledge of professional and ethical responsibilities by paying attention to grooming, being conservative with style, following dress codes, safety codes, and adapting themselves to technological advancements.

PROGRAM SPECIFIC OUTCOMES (PSOs)

After the completion of the course, B. Tech Information Technology, the graduates will have the following Program Specific Outcomes:

1. **Fundamentals and critical knowledge of the Computer System:-** Able to Understand the working principles of the computer System and its components , Apply the knowledge to build, asses, and analyze the software and hardware aspects of it .
2. **The comprehensive and Applicative knowledge of Software Development:** Comprehensive skills of Programming Languages, Software process models, methodologies, and able to plan, develop, test, analyze, and manage the software and hardware intensive systems in heterogeneous platforms individually or working in teams.
3. **Applications of Computing Domain & Research:** Able to use the professional, managerial, interdisciplinary skill set, and domain specific tools in development processes, identify the research gaps, and provide innovative solutions to them.

PROGRAM OUTCOMES (POs)

Engineering Graduates should possess the following:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
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.
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.
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.
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.
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.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
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 projects and in multi disciplinary 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 change.



MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(UGC-Autonomous Institution , Govt. of India)

(Permanently Affiliated to JNTUH, Approved by AICTE-Accredited by NBA & NAAC- A-Grade; ISO 9001:2008 Certified)

Maisammaguda, Dhulapally Post, Via Hakimpet, Secunderabad – 500100

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Compiler Design Lab Manual (R15A0586)

TABLE OF CONTENTS

Sl. No	Description of the Item / Title	Page No.	Remarks
1.	Importance/Rationale behind the CD Lab	i	
2.	General Laboratory Instructions	ii	
3.	Objectives & Outcomes	iii	
4.	Software / Hardware Requirements		
5.	Case Study: Description of the Syntax of the source Language(mini language) for which the compiler components are designed	1	
6.	Write a C Program to Scan and Count the number of characters, words, and lines in a file.	4	
7.	Write a C Program to implement NFAs that recognize identifiers, constants, and operators of the mini language.	5	
8.	Write a C Program to implement DFAs that recognize identifiers, constants, and operators of the mini language.	6	
9.	Design a lexical analyzer for the given language. The lexical analyzer should ignore redundant spaces, tabs and new lines, comments etc.	7	
10	Implement the lexical analyzer using JLex, flex or other lexical analyzer generating tools.	11	
11	Design Predictive Parser for the given language	13	
12	Design a LALR bottom up parser for the given language	18	
13	Convert the BNF rules into Yacc form and write code to generate abstract syntax tree.	20	
14	A program to generate machine code from the abstract syntax tree generated by the parser.	26	
Additional/Extra Programs			
15	Lex Program to convert abc to ABC	30	
16	Write a lex program to find out total number of vowels, and consonants from the given input sting.	31	
17	Implementation of Predictive Parser	32	
18	Implementation of Recursive Descent Parser	37	
19	Implementation of SLR Parser	38	

Importance of Compiler Design Lab

Every software has to be translated to its equivalent machine instruction form so that it will be executed by the underlying machine. There are many different types of system softwares that help during this translation process.

Compiler is one such an important **System Software** that converts High level language programs to its equivalent machine (low level) language. It is impossible to learn and use machine language in software development for the users. Therefore we use high level computer languages to write programs and then convert such programs into machine understandable form with the help of mediator softwares such as compilers, interpreters, assemblers etc. Thus compiler bridges the gap between the user and the machine, i.e., computer.

It's a very complicated piece of software which took 18 man years to build first compiler. To build this software we must understand the principles, tools, and techniques used in its working.

The compiler goes through the following sequence of steps called phases of a compiler.

- 1) Lexical Analysis
- 2) Syntax Analysis
- 3) Semantic Analysis
- 4) Intermediate Code Generation
- 5) Code Optimization
- 6) Code Generation.

In performing its role, compiler also uses the services of two essential components namely, Symbol Table and Error Handler

The objective of the Compiler Design Laboratory is to understand and implement the principles, techniques, and also available tools used in compiler construction process. This will enable the students to work in the development phase of new computer languages in industry.



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to starting time), those who come after 5 minutes will not be allowed into the lab.
2. Plan your task properly much before to the commencement, come prepared to the lab with the synopsis / program / experiment details.
3. Student should enter into the laboratory with:
 - a. Laboratory observation notes with all the details (Problem statement, Aim, Algorithm, Procedure, Program, Expected Output, etc.,) filled in for the lab session.
 - b. Laboratory Record updated up to the last session experiments and other utensils (if any) needed in the lab.
 - c. Proper Dress code and Identity card.
4. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.
5. Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.
6. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.
7. Computer labs are established with sophisticated and high end branded systems, which should be utilized properly.
8. Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions. Misuse of the equipment, misbehaviors with the staff and systems etc., will attract severe punishment.
9. Students must take the permission of the faculty in case of any urgency to go out ; if anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.
10. Students should LOG OFF/ SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.

Head of the Department

Principal



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OBJECTIVES AND OUTCOMES

OBJECTIVES:

- To provide an Understanding of the language translation peculiarities by designing complete translator for mini language.

OUTCOMES : By the end of the semester, students will be able

- To understand the practical approach of how a compiler works.
- To Apply the techniques used in Compiler Construction
- To generate few phases of the compiler for the mini language using tools

RECOMMENDED SYSTEM / SOFTWARE REQUIREMENTS:

1. Intel based desktop PC with minimum of 166MHz or faster processor with at least 64 MB RAM and 100 MB free disk space.
2. C ++ Compiler and JDK kit, Lex or Flex and YACC tools (Unix/Linux utilities)

USEFUL TEXT BOOKS / REFERECES / WEBSITES :

1. Modern compiler implementation in C, Andrew w.Appel, Revised Edn, Cambridge University Press
2. Principles of Compiler Design. – A.V Aho, J.D Ullman ; Pearson Education.
3. **lex&yacc** , -John R Levine, Tony Mason, Doug Brown; O'reilly.
4. **Compiler Construction**, - LOUDEN, Thomson.
5. Engineering a compiler – Cooper& Linda, Elsevier
6. Modern Compiler Design – Dick Grune, Henry E. Bal, Cariel TH Jacobs, Wiley Drearetech

SOURCE LANGUAGE (A Case Study)

Consider the following mini language, a simple procedural High Level Language, operating on integer data with a syntax looking vaguely like a simple C crossed with Pascal. The syntax of the language is defined by the following BNF grammar:

```
<program> ::= <block>
<block> ::= { <variable definition> <slist> }
           | { <slist> }
<variable definition> ::= int <vardeflist> ;
<vardeflist> ::= <vardec> | <vardec>, <vardeflist>
<vardec> ::= <identifier> | <identifier> [<constant>]
<slist> ::= <statement> | <statement> ; <slist>
<statement> ::= <assignment> | <ifstatement> | <whilestatement> | <block>
               | <printstatement> | <empty>
<assignment> ::= <identifier> = <expression>
               | <identifier> [<expression>] = [<expression>]
<ifstatement> ::= if <bexpression> then <slist> else <slist> endif
               | if <bexpression> then <slist> endif
<whilestatement> ::= while <bexpression> do <slist> enddo
<printstatement> ::= print{ <expression> }
<expression> ::= <expression> <addingop> <term> | <term> | <addingop> <term>
<bexpression> ::= <expression> <relop> <expression>
<relop> ::= < | <= | = | >= | > | !=
<addingop> ::= + | -
<term> ::= <term> <multop> <factor> | <factor>
<multop> ::= * | /
<factor> ::= <constant> | <identifier> | <identifier> [<expression>]
           | (<expression>)
<constant> ::= <digit> | <digit> <constant>
<identifier> ::= <identifier> <letterordigit> | <letter>
<letterordigit> ::= a|b|c|...|y|z
<digit> ::= 0|1|2|3|...|8|9
<empty> ::= has the obvious meaning
```

Comments : zero or more characters enclosed between the standard C/Java style comment brackets /*...*/. The language has the rudimentary support for 1-Dimensional arrays. Ex: int a[3] declares a as an array of 3 elements, referenced as a[0],a[1],a[2].

Sample Program written in this language is :

```
{
  int a[3],t1,t2;
  t1=2;
  a[0]=1; a[1]=2; a[t1]=3;
  t2= -(a[2]+t1*6) / a[2]-t1);
  if t2>5 then
    print(t2);
  else
    {
      int t3;
      t3=99;
      t2=25;
      print(-11+t2*t3); /* this is not a comment on two lines */
    }
```

```
}  
Endif  
}
```

1. Write a C Program to Scan and Count the number of characters, words, and lines in a file.
2. Write a C Program to implement NFAs that recognize identifiers, constants, and operators of the mini language.
3. Write a C Program to implement DFAs that recognize identifiers, constants, and operators of the mini language.
4. Design a Lexical analyzer for the above language. The lexical analyzer should ignore redundant spaces, tabs and newlines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value.
5. Implement the lexical analyzer using JLex, flex, flex or lex or other lexical analyzer generating tools.
6. Design Predictive parser for the given language
7. Design LALR bottom up parser for the above language.
8. Convert the BNF rules into Yacc form and write code to generate abstract syntax tree.
9. Write program to generate machine code from the abstract syntax tree generated by the parser.
The following instruction set may be considered as target code.

The following is a simple register-based machine, supporting a total of 17 instructions. It has three distinct internal storage areas. The first is the set of 8 registers, used by the individual instructions as detailed below, the second is an area used for the storage of variables and the third is an area used for the storage of program. The instructions can be preceded by a label. This consists of an integer in the range 1 to 9999 and the label is followed by a colon to separate it from the rest of the instruction. The numerical label can be used as the argument to a jump instruction, as detailed below.

In the description of the individual instructions below, instruction argument types are specified as follows:

R specifies a register in the form R0, R1, R2, R3, R4, R5, R6 or R7 (or r0, r1, etc).

L specifies a numerical label (in the range 1 to 9999).

V specifies a "variable location" (a variable number, or a variable location pointed to by a register - see below).

A specifies a constant value, a variable location, a register or a variable location pointed to by a register (an indirect address). Constant values are specified as an integer value, optionally preceded by a minus sign, preceded by a # symbol. An indirect address is specified by an @ followed by a register.

So, for example an A-type argument could have the form 4 (variable number 4), #4 (the constant value 4), r4 (register 4) or @r4 (the contents of register 4 identifies the variable location to be accessed).

The instruction set is defined as follows:

LOAD A, R

loads the integer value specified by A into register R.

STORE R, V

stores the value in register R to variable V.

OUT R

outputs the value in register R.

NEG R

negates the value in register R.

ADD A, R

adds the value specified by A to register R, leaving the result in register R.

SUB A, R

subtracts the value specified by A from register R, leaving the result in register R.

MUL A, R

multiplies the value specified by A by register R, leaving the result in register R.

DIV A, R

divides register R by the value specified by A, leaving the result in register R.

JMP L

causes an unconditional jump to the instruction with the label L.

JEQ R, L

jumps to the instruction with the label L if the value in register R is zero.

JNE R, L

jumps to the instruction with the label L if the value in register R is not zero.

JGE R, L

jumps to the instruction with the label L if the value in register R is greater than or equal to zero.

JGT R, L

jumps to the instruction with the label L if the value in register R is greater than zero.

JLE R, L

jumps to the instruction with the label L if the value in register R is less than or equal to zero.

JLT R, L

jumps to the instruction with the label L if the value in register R is less than zero.

NOP

is an instruction with no effect. It can be tagged by a label.

STOP

stops execution of the machine.

All programs should terminate by executing a STOP instruction.

1. **Problem Statement:** Write a C Program to Scan and Count the number of characters, words, and lines in a file.

AIM : To Write a C Program to Scan and Count the number of characters, words, and lines in a file.

ALGORITHM / PROCEDURE/PROGRAM:

1. Start
2. Read the input file/text
3. Initialize the counters for characters, words, lines to zero
4. Scan the characters, words, lines and
5. increment the respective counters
6. Display the counts
7. End

Input: Enter the Identifier input string/file :

These are few sentences in mini
Language

Output:

No of characters: 35
No of words : 7
No of lines : 2

[Viva Questions]

1. What is Compiler?
2. List various language Translators.
3. Is it necessary to translate a HLL program? Explain.
4. List out the phases of a compiler?
5. Which phase of the compiler is called an optional phase?why

2. Problem Statement: Write a C Program to implement NFAs that recognize identifiers, constants, and operators of the mini language.

AIM: To Write a C Program to implement NFAs that recognize identifiers, constants, and operators of the mini language.

ALGORITHM / PROCEDURE / PROGRAM:

1. Start
2. Design the NFA (N) to recognize Identifiers, Constants, and Operators
3. Read the input string w give it as input to the NFA
4. NFA processes the input and outputs “Yes” if $w \in L(N)$, “No” otherwise
5. Display the output
6. End

Input: Enter the Identifier input: sum

Output: Yes

Input: Enter a Constant input: 4567

Output: Yes

Input: Enter an operator input: +

Output: Yes

[Viva Questions]

1. What is a Preprocessor and what is its role in compilation?
2. Which language is both compiled and interpreted?
3. List out the languages that are interpreted?
4. Explain the working of a NFA?
5. When do you prefer to design an NFA to DFA?

3. **Problem Statement:** Write a C Program to implement DFAs that recognize identifiers, constants, and operators of the mini language.

AIM: To Write a C Program to implement DFAs that recognize identifiers, constants, and operators of the mini language.

ALGORITHM / PROCEDURE:

- 1 Start
- 2 Design the DFAs (M) to recognize Identifiers, Constants, and Operators
- 3 Read the input string w give it as input to the DFA M
- 4 DFA processes the input and outputs “Yes” if $w \in L(M)$, “No” otherwise
- 5 Display the output
- 6 End

Input: Enter the Identifier input: sum

Output: Yes

Input: Enter a Constant input: 4567

Output: Yes

Input: Enter an operator input: +

Output: Yes

[Viva Questions]

1. What is an Interpreter?
2. What are the other language processors you know?
3. What is the difference between DFA / minimum DFA?
4. Write the difference between the interpreter and Compiler

4. Problem Statement: Design a Lexical analyzer. The lexical analyzer should ignore redundant spaces and new lines. It should also ignore comments. Although the syntax specification says those identifiers can be arbitrarily long, you may restrict the length to some reasonable Value.

AIM: Write a C/C++ program to implement the design of a Lexical analyzer to recognize the tokens defined by the given grammar.

ALGORITHM / PROCEDURE:

We make use of the following two functions in the process.

look up() – it takes string as argument and checks its presence in the symbol table. If the string is found then returns the address else it returns NULL.

insert() – it takes string as its argument and the same is inserted into the symbol table and the corresponding address is returned.

1. Start
2. Declare an array of characters, an input file to store the input;
3. Read the character from the input file and put it into character type of variable, say 'c'.
4. If 'c' is blank then do nothing.
5. If 'c' is new line character line=line+1.
6. If 'c' is digit, set token Val, the value assigned for a digit and return the 'NUMBER'.
7. If 'c' is proper token then assign the token value.
8. Print the complete table with
Token entered by the user,
Associated token value.
9. Stop

PROGRAM / SOURCE CODE :

```
#include<string.h>
#include<ctype.h>
#include<stdio.h>
void keyword(char str[10])
{
    if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||strcmp("int",str)
    )==0||strcmp("float",str)==0||strcmp("char",str)==0||strcmp("double",str)==0||strcmp("static",
    str)==0||strcmp("switch",str)==0||strcmp("case",str)==0)
        printf("\n%s is a keyword",str);
    else
        printf("\n%s is an identifier",str);
}
main()
{
```

```

FILE *f1,*f2,*f3;
char c, str[10], st1[10];
int num[100], lineno=0, tokenvalue=0,i=0,j=0,k=0;
printf("\n Enter the c program : ");/*gets(st1);*/
f1=fopen("input","w");
while((c=getchar())!=EOF)
putc(c,f1);
fclose(f1);
f1=fopen("input","r");
f2=fopen("identifier","w");
f3=fopen("specialchar","w");
while((c=getc(f1))!=EOF)
{
    if(isdigit(c))
    {
        tokenvalue=c-'0';
        c=getc(f1);
        while(isdigit(c))
        {
            tokenvalue*=10+c-'0';
            c=getc(f1);
        }
        num[i++]=tokenvalue;
        ungetc(c,f1);
    }
    else
    if(isalpha(c))
    {
        putc(c,f2);
        c=getc(f1);
        while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
        {
            putc(c,f2);
            c=getc(f1);
        }
    }
}

```

```

                putc(' ',f2);
                ungetc(c,f1);
            }
            else
            if(c==' '||c=='\t')
                printf(" ");
            else
            if(c=='\n')
                lineno++;
            else
                putc(c,f3);
        }
        fclose(f2);
        fclose(f3);
        fclose(f1);
        printf("\n The no's in the program are :");
        for(j=0; j<i; j++)
            printf("%d", num[j]);
        printf("\n");
        f2=fopen("identifier", "r");
        k=0;
        printf("The keywords and identifiers are:");
        while((c=getc(f2))!=EOF)
        {
            if(c!=' ')
                str[k++]=c;
            else
            {
                str[k]='\0';
                keyword(str);
                k=0;
            }
        }
        fclose(f2);
        f3=fopen("specialchar","r");

```

```
printf("\n Special characters are : ");
while((c=getc(f3))!=EOF)
    printf("%c",c);
printf("\n");
fclose(f3);
printf("Total no. of lines are:%d", lineno);
}
```

Output :

Enter the C program: a+b*c

Ctrl-D

The no's in the program are:

The keywords and identifiers are:

a is an identifier and terminal

b is an identifier and terminal

c is an identifier and terminal

Special characters are:

+ *

Total no. of lines are: 1

[Viva Questions]

5. What is lexical analyzer?
6. Which compiler is used for lexical analysis?
7. What is the output of Lexical analyzer?
8. Which Finite state machines are used in lexical analyzer design?
9. What is the role of regular expressions, grammars in Lexical Analyzer?

5. Problem Statement: Implement the lexical analyzer using JLex, flex or other lexical Analyzer generating tools.

AIM: To Implement the lexical analyzer using JLex, flex or lex other lexical analyzer generating Tools.

ALGORITHM / PROCEDURE:

Input : LEX specification files for the token

Output : Produces the source code for the Lexical Analyzer with the name lex.yy.c and displays the tokens from an input file.

1. Start
2. Open a file in text editor
3. Create a Lex specifications file to accept keywords, identifiers, constants, operators and relational operators in the following format.
 - a) %
Definition of constant /header files
%}
 - b) Regular Expressions
%%
Transition rules
%%
 - c) Auxiliary Procedure (main() function)
4. Save file with .l extension e.g. **mylex.l**
5. Call lex tool on the terminal e.g. [root@localhost]# lex mylex.l. This lex tool will convert “.l” file into “.c” language code file i.e., **lex.yy.c**
6. Compile the file lex.yy.c using C / C++ compiler. e.g. **gcc lex.yy.c**. After compilation the file lex.yy.c, the output file is in **a.out**
7. Run the file a.out giving an input(text/file) e.g. **./a.out**.
8. Upon processing, the sequence of tokens will be displayed as output.
9. Stop

LEX SPECIFICATION PROGRAM / SOURCE CODE (lexprog.l) :

```
// ***** LEX Tool program to identify C Tokens *****//
DIGIT      [0-9]
LETTER     [A-Z a-z]
DELIM      [ \t\n]
WS         {DELIM}+
ID         {(LETTER)[LETTER/DIGIT]}+
INTEGER    {DIGIT}+
%%
{WS}       { printf("\n special characters\n"); }
{ID}       { printf("\n Identifiers\n"); }
{DIGIT}    { printf("\n Intgers\n"); }
If         { printf("\n Keywords\n"); }
```

```

else                { printf("\n keywords\n"); }
">"               { printf("\n Logical Operators\n"); }
"<"               { printf("\n Logical Operators\n"); }
"<="             { printf("\n Logical Operators\n"); }
">="             { printf("\n Logical Operators\n"); }
"="               { printf("\n Logical Operators\n"); }
"!="             { printf("\n Logical Operators\n"); }
"&&"            { printf("\n Relational \n"); }
"||"             { printf("\n Relational Operatos\n"); }
"!"             { printf("\n Relational Operators\n"); }
"+"             { printf("\n Arithmetic Operator\n"); }
"_"             { printf("\n Arithmetic Operator\n"); }
"*"             { printf("\n Arithmetic Operator\n"); }
"/"             { printf("\n Arithmetic Operator\n"); }
"%"             { printf("\n Arithmetic Operator\n"); }
%%
main()
{
    yylex();
}

```

OUTPUT :

```

[root@localhost]# lex lexprog.l
[root@localhost]# cc lex.yy.c
[root@localhost]# ./a.out lexprog

```

TEST CASES:

INPUT	OUTPUT
If	Keyword
%	Arithmetic Operator
>=	Relational Operator
&&	Logical Operator

[Viva Questions]

1. What is are the functions of a Scanner?
2. What is Token?
3. What is lexeme, Pattern?
4. What is purpose of Lex?
5. What are the other tools used in Lexical Analysis?

6. Problem Statement : Design a Predictive Parser for the following grammar

G: { $E \rightarrow TE'$, $E' \rightarrow +TE' \mid 0$, $T \rightarrow FT'$, $T' \rightarrow *FT' \mid 0$, $F \rightarrow (E) \mid id$ }

AIM: To write a 'C' Program to implement for the Predictive Parser (Non Recursive Descent parser) for the

given grammar ,

Given the parse Table:

	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow 0$	$E' \rightarrow 0$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow 0$	$T' \rightarrow *FT'$		$T' \rightarrow 0$	$T' \rightarrow 0$
F	$F \rightarrow id$			$F \rightarrow (E)$		

ALGORITHM / PROCEDURE :

Input : string w\$, Predictive Parsing table M

Output : A Left Most Derivation of the input string if it is valid , error otherwise.

- Step1: Start
- Step2: Declare a character array w[10] and Z as an array
- Step3: Enter the string with \$ at the end
- Step4: if (A(w[z])) then increment z and check for (B(w[z])) and if satisfies increment z and check for 'd' if d is present then increment and check for (D(w[z]))
- Step5: if step 4 is satisfied then the string is accepted
Else string is not
- Step 6: Exit

SOURCE CODE :

```
// ***IMPLEMENTATION OF PREDICTIVE / NON-RECURSIVE DESCENT PARSING *****/
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
char ch;
#define id 0
#define CONST 1
#define mulop 2
#define addop 3
#define op 4
#define cp 5
#define err 6
#define col 7
#define size 50
int token;
```

```

char lexbuff[size];
int lookahead=0;
int main()
{
    clrscr();
    printf(" Enter the string :");
    gets(lexbuff);
    parser();
    return 0;
}
parser()
{
    if(E())
        printf("valid string");
    else
        printf("invalid string");
    getch();
    return 0;
}
E()
{
    if(T())
    {
        if(EPRIME())
            return 1;
        else
            return 0;
    }
    else
        return 0;
}
T()
{
    if(F())
    {
        if(TPRIME())
            return 1;
        else
            return 0;
    }
    else
        return 0;
}
EPRIME()
{
    token=lexer();
    if(token==addop)
    {
        lookahead++;
        if(T())
        {
            if(EPRIME())

```

```

        return 1;
    else
        return 0;
    }
    else
        return 0;
    }
else
return 1;
}
TPRIME()
{
    token=lexer();
    if(token==mulop)
    {
        lookahead++;
        if(F())
        {
            if(TPRIME())
                return 1;
            else
                return 0;
        }
        else return 0;
    }
    else return 1;
}
F()
{
    token=lexer();
    if(token==id)
        return 1;
    else
    {
        if(token==4)
        {
            if(E())
            {
                if(token==5)
                    return 1;
                else return 0;
            }
            else return 0;
        }
    }
    else return 0;
}
}
lexer()
{
    if(lexbuff[lookahead]!='\n')
    {
        while(lexbuff[lookahead]=='\t')

```

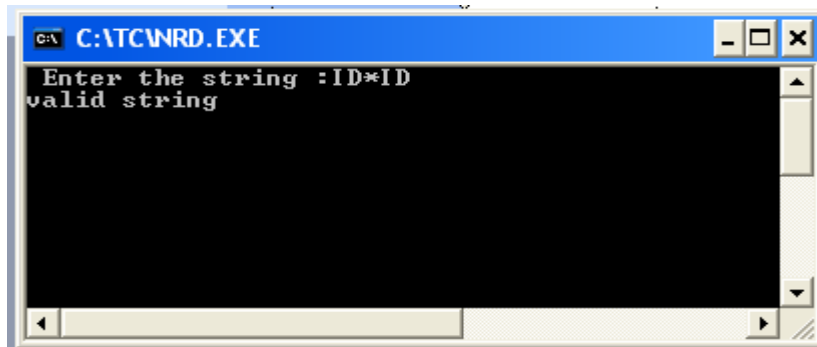
```

lookahead++;
if(isalpha(lexbuff[lookahead]))
{
    while(isalnum(lexbuff[lookahead]))
        lookahead++;
    return(id);
}
else
{
    if(isdigit(lexbuff[lookahead]))
    {
        while(isdigit(lexbuff[lookahead]))
            lookahead++;
        return CONST;
    }
    else
    {
        if(lexbuff[lookahead]=='+')
        {
            return(addop);
        }
        else
        {
            if(lexbuff[lookahead]=='*')
            {
                return(mulop);
            }
            else
            {
                if(lexbuff[lookahead]=='(')
                {
                    lookahead++;
                    return(op);
                }
                else
                {
                    if(lexbuff[lookahead]==')')
                    {
                        return(op);
                    }
                    else
                    {
                        return(err);
                    }
                }
            }
        }
    }
}
else
return (col);

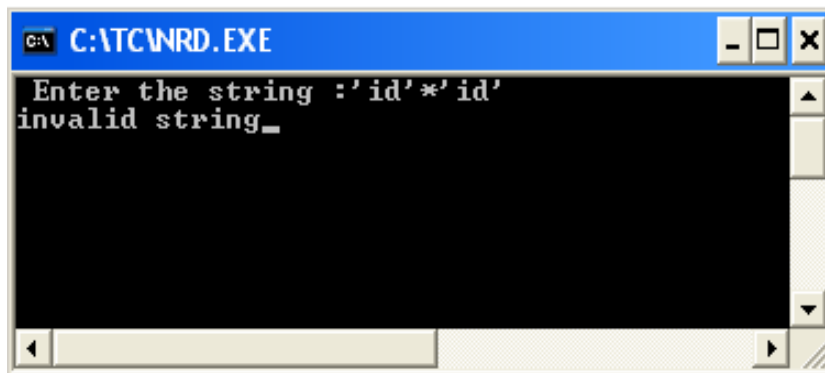
```

}

OUTPUT :



```
C:\TC\NRD.EXE
Enter the string :ID*ID
valid string
```



```
C:\TC\NRD.EXE
Enter the string :\'id\'*\'id\'
invalid string_
```

Viva Questions:

1. What is a parser and state the Role of it?
2. Types of parsers? Examples to each
3. What are the Tools available for implementation?
4. How do you calculate FIRST(),FOLLOW() sets used in Parsing Table construction?

7. Problem Statement: Design a LALR Bottom Up Parser for the given grammar

AIM: To Design and implement an LALR bottom up Parser for checking the syntax of the Statements in the given language.

ALGORITHM/PROCEDURE/CODE:

LALR Bottom Up Parser

```
<parser.l>
%{
#include<stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+ {yylval.dval=atof(yytext);
return DIGIT;
}
\n|. return yytext[0];
%%
<parser.y>
%{
/*This YACC specification file generates the LALR parser for the program
considered in experiment 4.*/
#include<stdio.h>
%}
%union
{
double dval;
}
%token <dval> DIGIT
%type <dval> expr
%type <dval> term
%type <dval> factor
%%
line: expr '\n' {
;
printf("%g\n", $1);
}
expr: expr '+' term {$$=$1 + $3 ;}
| term
;
term: term '*' factor {$$=$1 * $3 ;}
| factor
;
factor: '(' expr ')' {$$=$2 ;}
| DIGIT
;
%%
int main()
{
yyparse();
```

```
}  
yyerror(char *s)  
{  
printf("%s",s);  
}
```

Output:

```
$lex parser.l  
$yacc -d parser.y  
$cc lex.yy.c y.tab.c -ll -lm  
$./a.out  
2+3  
5.0000
```

Viva Questions?

1. What is yacc? Are there any other tools available for parser generation?
2. How do you use it?
3. Structure of parser specification program
4. How many ways we can generate the Parser
5. How a

8. Problem statement: Convert The BNF rules into YACC form and write code to generate abstract syntax tree.

AIM: To Implement the process of conversion from BNF rules to Yacc form and generate Abstract Syntax Tree.

ALGORITHM/PROCEDURE

```
<int.l>
%{
#include"y.tab.h"
#include<stdio.h>
#include<string.h>
int LineNo=1;
% }
identifier [a-zA-Z][_a-zA-Z0-9]*
number [0-9]+|([0-9]*\.[0-9]+)
%%
main\(\) return MAIN;
if return IF;
else return ELSE;
while return WHILE;
int |
char |
float return TYPE;
{identifier} {strcpy(yylval.var,yytext);
return VAR;}
{number} {strcpy(yylval.var,yytext);
return NUM;}
\< |
\> |
\>= |
\<= |
== {strcpy(yylval.var,yytext);
return RELOP;}
[ \t] ;
\n LineNo++;
. return yytext[0];
%%
<int.y>
%{
#include<string.h>
#include<stdio.h>
struct quad
{
char op[5];
char arg1[10];
char arg2[10];
char result[10];
}QUAD[30];
struct stack
```

```

{
    int items[100];
    int top;
}stk;
int Index=0,tIndex=0,StNo,Ind,tInd;
extern int LineNo;
%}
%union
{
    char var[10];
}
%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
%left '-' '+'
%left '*' '/'
%%
PROGRAM : MAIN BLOCK
;
BLOCK: '{' CODE '}'
;
CODE: BLOCK
| STATEMENT CODE
| STATEMENT
;
STATEMENT: DESCT ';'
| ASSIGNMENT ';'
| CONST
| WHILEST
;
DESCT: TYPE VARLIST
;
VARLIST: VAR ';' VARLIST
| VAR
;
ASSIGNMENT: VAR '=' EXPR{
strcpy(QUAD[Index].op,"=");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,$1);
strcpy($$,QUAD[Index++].result);
}
;
EXPR :   EXPR '+' EXPR { AddQuadruple("+",$1,$3,$$);}
|       EXPR '-' EXPR { AddQuadruple("-", $1,$3,$$);}
|       EXPR '*' EXPR { AddQuadruple("*", $1,$3,$$);}
|       EXPR '/' EXPR { AddQuadruple("/", $1,$3,$$);}
|       '-' EXPR      { AddQuadruple("UMIN", $2, "", $$);}
|       '(' EXPR ')'   { strcpy($$, $2); }
|       VAR
|       NUM
;

```

```

CONDST: IFST{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
| IFST ELSEST
;
IFST: IF '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
};
ELSEST: ELSE{
tInd=pop();
Ind=pop();
push(tInd);
sprintf(QUAD[Ind].result,"%d",Index);
}
BLOCK{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
};
CONDITION: VAR RELOP VAR {AddQuadruple($2,$1,$3,$$);
StNo=Index-1;
}
| VAR
| NUM
;
WHILEST: WHILELOOP{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",StNo);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
;
WHILELOOP: WHILE '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");

```

```

push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
;
%%
extern FILE *yyin;
int main(int argc,char *argv[])
{
FILE *fp;
int i;
if(argc>1)
{
fp=fopen(argv[1],"r");
if(!fp)
{
printf("\n File not found");
exit(0);
}
yyin=fp;
}
yyparse();
printf("\n\n\t\t -----""\n\t\t Pos Operator Arg1 Arg2 Result" "\n\t\t
-----");
for(i=0;i<Index;i++)
{
printf("\n\t\t %d\t %s\t %s\t %s\t
%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i].arg2,QUAD[i].result);
}
printf("\n\t\t -----");
printf("\n\n");
return 0;
}
void push(int data)
{
stk.top++;
if(stk.top==100)
{
printf("\n Stack overflow\n");
exit(0);
}
stk.items[stk.top]=data;
}
int pop()
{

```

```

int data;
if(stk.top== -1)
{
printf("\n Stack underflow\n");
exit(0);
}
data=stk.items[stk.top--];
return data;
}
void AddQuadruple(char op[5],char arg1[10],char arg2[10],char result[10])
{
strcpy(QUAD[Index].op,op);
strcpy(QUAD[Index].arg1,arg1);
strcpy(QUAD[Index].arg2,arg2);
sprintf(QUAD[Index].result,"t%d",tIndex++);
strcpy(result,QUAD[Index++].result);
}
yyerror()
{
printf("\n Error on line no:%d",LineNo);
}
Input:
$vi test.c
main()
{
int a,b,c;
if(a<b)
{
a=a+b;
}
while(a<b)
{
a=a+b;
}
if(a<=b)
{
c=a-b;
}
else
{
c=a+b;
}
}

```

Output:

```

$ lex int.l
$ yacc -d int.y
$ gcc lex.yy.c y.tab.c -ll -lm
$ ./a.out test.c

```

OUTPUT:

Pos	Operator	Arg1	Arg2	Result
0	<	a	b	to
1	==	to	FALSE	5
2	+	a	b	t1
3	=	t1		a
4	GOTO			5
5	<	a	b	t2
6	==	t2	FALSE	10
7	+	a	b	t3
8	=	t3		a
9	GOTO			5
10	<=	a	b	t4
11	==	t4	FALSE	15
12	-	a	b	t5
13	=	t5		c
14	GOTO		17	
15	+	a	b	t3
16	=	t6		c

Viva Questions:

1. What is abstract syntax tree?
2. What is quadruple?
3. What is the difference between Triples and Indirect Triple?
4. State different forms of Three address statements.
5. What are different intermediate code forms?

9. Problem Statement: A program to generate machine code from the abstract syntax tree generated by the parser.

Aim: To write a C Program to Generate Machine Code from the Abstract Syntax Tree using the specified machine instruction formats.

ALGORITHM / PROCEDURE / SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int label[20];
int no=0;
int main()
{
FILE *fp1,*fp2;
char fname[10],op[10],ch;
char operand1[8],operand2[8],result[8];
int i=0,j=0;
printf("\n Enter filename of the intermediate code");
scanf("%s",&fname);
fp1=fopen(fname,"r");
fp2=fopen("target.txt","w");
if(fp1==NULL || fp2==NULL)
{
printf("\n Error opening the file");
exit(0);
}
while(!feof(fp1))
{
fprintf(fp2,"\n");
fscanf(fp1,"%s",op);
i++;
if(check_label(i))
fprintf(fp2,"\nlabel#%d",i);
if(strcmp(op,"print")==0)
{
fscanf(fp1,"%s",result);
fprintf(fp2,"\n\t OUT %s",result);
}
if(strcmp(op,"goto")==0)
{
fscanf(fp1,"%s %s",operand1,operand2);
fprintf(fp2,"\n\t JMP %s,label#%s",operand1,operand2);
label[no++]=atoi(operand2);
}
if(strcmp(op,"[]")==0)
{
fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t STORE %s[%s],%s",operand1,operand2,result);
}
}
```

```

if(strcmp(op,"uminus")==0)
{
fscanf(fp1,"%s %s",operand1,result);
fprintf(fp2,"\n\t LOAD -%s,R1",operand1);
fprintf(fp2,"\n\t STORE R1,%s",result);
}
switch(op[0])
{
case '*': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t MUL R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '+': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t ADD R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '-': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t SUB R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '/': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t DIV R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '%': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t DIV R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '=': fscanf(fp1,"%s %s",operand1,result);
fprintf(fp2,"\n\t STORE %s %s",operand1,result);
break;
case '>': j++;
fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t JGT %s,label#%s",operand2,result);
label[no++] = atoi(result);
break;
case '<': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1); fprintf(fp2,"\n\t
JLT %s,label#%d",operand2,result);
label[no++] = atoi(result);
break;

```

```

}
}
fclose(fp2); fclose(fp1);
fp2=fopen("target.txt","r");
if(fp2==NULL)
{
printf("Error opening the file\n");
exit(0);
}
do
{
ch=fgetc(fp2);
printf("%c",ch);
}while(ch!=EOF);
fclose(fp1);
return 0;
}
int check_label(int k)
{
int i;
for(i=0;i<no;i++)
{
if(k==label[i])
return 1;
}
return 0;
}

```

Input :

```

$ vi int.txt
= t1 2
[] = a 0 1
[] = a 1 2
[] = a 2 3
*t1 6 t2
+a[2] t2 t3
-a[2] t1 t2
/t3 t2 t2
uminus t2 t2
print t2
goto t2 t3
=t3 99
uminus 25 t2
*t2 t3 t3
uminus t1 t1
+t1 t3 t4
print t4

```

Output :

Enter filename of the intermediate code: int.txt

```
STORE t1, 2
STORE a[0], 1
STORE a[1], 2
STORE a[2], 3
LOAD t1, R0
LOAD 6, R1
ADD R1, R0
STORE R0, t3
LOAD a[2], R0
LOAD t2, R1
ADD R1, R0
STORE R0, t3
LOAD a[t2], R0
LOAD t1, R1
SUB R1, R0
STORE R0, t2
LOAD t3, R0
LOAD t2, R1
DIV R1, R0
STORE R0, t2
LOAD t2, R1
STORE R1, t2
LOAD t2, R0
JGT 5, label#11
Label#11: OUT t2
JMP t2, label#13
Label#13: STORE t3, 99
LOAD 25, R1
STORE R1, t2
LOAD t2, R0
LOAD t3, R1
MUL R1, R0
STORE R0, t3
LOAD t1, R1
STORE R1, t1
LOAD t1, R0
LOAD t3, R1
ADD R1, R0
STORE R0, t4
OUT t4
```

CD –ADDITIONAL / EXTRA PROGRAMS

- 1). **Problem Statement:** Write a LEX Program to convert the substring “abc” to “ABC” from the given input string.

Lex Program

```
%{
/*The Program replaces the substring abc by ABC from
the given input string*/
#include<stdio.h>
#include<string.h>
int i;
%}
%%
[a-zA-Z]* {
for(i=0;i<=yyleng;i++)
{
If((yytext[i]=='a')&&( yytext[i+1]=='b')&&( yytext[i+2]=='c'))
{
yytext[i]='A';
yytext[i+1]='B';
yytext[i+2]='C';
}
}
Printf("%s", yytext);
}
[\t]*return;
.*{ECHO;}
\n {printf("%s", yytext);}
%%
main()
{
yytext();
}
int yywrap()
{
return 1;
}
```

2). **Problem Statement:** Write a lex program to find out total number of vowels and consonants from the given input sting.

Lex Program

```
%{
/*Definition section*/
int vowel_cnt=0,consonant_cnt=0;
%}
vowel [aeiou]+
consonant [^aeiou]
eol \n
%%
{eol} return 0;
[t]+ ;
{vowel} {vowel_cnt++;}
%%
int main()
{
printf("\n Enter some input string\n");
yylex();
printf("vowels=%d and consonant=%d\n",vowel_cnt,consonant_cnt);
return 0;
}
Int yywrap()
{
return 1;
}

%{
/*Definition section*/
int vowel_cnt=0,consonant_cnt=0;
%}
vowel [aeiou]+
consonant [^aeiou]
eol \n
%%
{eol} return 0;
[t]+ ;
{vowel} {vowel_cnt++;}
%%
int main()
{
printf("\n Enter some input string\n");
yylex();
printf("vowels=%d and consonant=%d\n",vowel_cnt,consonant_cnt);
return 0;
}
Int yywrap()
{
return 1;
}
```

3) **Problem Statement:** Implementation of Predictive Parser.

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
#include<stdlib.h>
#define SIZE 128
#define NONE -1
#define EOS '\0'
#define NUM 257
#define KEYWORD 258
#define ID 259
#define DONE 260
#define MAX 999
char lexemes[MAX];
char buffer[SIZE];
int lastchar=-1;
int lastentry=0;
int tokenval=DONE;
int lineno=1;
int lookahead;
struct entry
{
char *lexptr;
int token;
}symtable[100]; struct entry
keywords[]={{"if",KEYWORD,"else",KEYWORD,"for",KEYWORD,"int",KEYWORD,
"float",KEYWORD,"double",KEYWORD,"char",KEYWORD,"struct",KEYWORD,"ret
urn",KEYWORD,0,0};
void Error_Message(char *m)
{
fprintf(stderr,"line %d, %s \n",lineno,m);
exit(1);
}
int look_up(char s[ ])
{
int k; for(k=lastentry;k>0;k--)
if(strcmp(symtable[k].lexptr,s)==0)
return k;
return 0;
}
int insert(char s[ ],int tok)
{
int len;
len=strlen(s); if(lastentry+1>=MAX)
Error_Message("Symbpl table is full");
if(lastchar+len+1>=MAX)
Error_Message("Lexemes array is full");
lastentry=lastentry+1;
symtable[lastentry].token=tok;
symtable[lastentry].lexptr=&lexemes[lastchar+1];
```

```

lastchar=lastchar+len+1;
strcpy(symtable[lastentry].lexptr,s);
return lastentry;
}
/*void Initialize()
{
struct entry *ptr;
for(ptr=keywords;ptr->token;ptr+1)
insert(ptr->lexptr,ptr->token);
}*/
int lexer()
{
int t;
int val,i=0;
while(1)
{
t=getchar();
if(t==' '||t=='\t');
else if(t=='\n')
lineno=lineno+1;
else if(isdigit(t))
{
ungetc(t,stdin);
scanf("%d",&tokenval);
return NUM;
}
else if(isalpha(t))
{
while(isalnum(t))
{
buffer[i]=t;
t=getchar();
i=i+1;
if(i>=SIZE)
Error_Message("Compiler error");
}
buffer[i]=EOS;if(t!=EOF)
ungetc(t,stdin);
val=look_up(buffer);
if(val==0)
val=insert(buffer,ID);
tokenval=val;
return symtable[val].token;
}
else if(t==EOF)
return DONE;
else
{
tokenval=NONE;
return t;
}
}
}

```

```

}
void Match(int t)
{
if(lookahead==t)
lookahead=lexer();
else
Error_Message("Syntax error");
}
void display(int t,int tval)
{
if(t=='+'||t=='-'||t=='*'||t=='/')
printf("\nArithmetic Operator: %c",t);
else if(t==NUM)
printf("\n Number: %d",tval);
else if(t==ID)
printf("\n Identifier: %s",symtable[tval].lexptr);
else
printf("\n Token %d tokenval %d",t,tokenval);
}
void F()
{
//void E();
switch(lookahead)
{
case '(' : Match('(');
E();
Match(')');
break;
case NUM : display(NUM,tokenval);
Match(NUM);
break;
case ID : display(ID,tokenval);
Match(ID);
break;
default : Error_Message("Syntax error");
}
}
void T()
{
int t;
F();
while(1)
{
switch(lookahead)
{
case '*' : t=lookahead;
Match(lookahead);
F();
display(t,NONE);
continue;
case '/' : t=lookahead;
Match(lookahead);

```

```

display(t,NONE);
continue;
default : return;
}
}
}
void E()
{
int t;
T();
while(1)
{
switch(lookahead)
{
case '+' : t=lookahead;
Match(lookahead);
T();
display(t,NONE);
continue;
case '-' : t=lookahead;
Match(lookahead);
T();
display(t,NONE);
continue;
default : return;
}
}
}
void parser()
{
lookahead=lexer();
while(lookahead!=DONE)
{
E();
Match(';');
}
}
main()
{
char ans[10];
printf("\n Program for recursive decent parsing ");
printf("\n Enter the expression ");
printf("And place ; at the end\n");
printf("Press Ctrl-Z to terminate\n");
parser();
}

```

Output:

Program for recursive decent parsing
Enter the expression And place ; at the end
Press Ctrl-Z to terminate

a+b*c;
Identifier: a
Identifier: b
Identifier: c
Arithmetic Operator: *
Arithmetic Operator: +
2*3;
Number: 2
Number: 3
Arithmetic Operator: *
+3;
line 5,Syntax error
Ctrl-Z

4. Problem Statement: Implement RECURSIVE DESCENT PARSER

AIM: To Implement the RECURSIVE DESCENT PARSER for the given grammar / language

ALGORITHM/ PROCEDURE:

Input : A string w\$, where w is a string of terminals

Output : w is accepted if it can be derived by the given grammar, else not accepted.

- Step1: Start
- Step2: declare w[10] as char and Z as an array
- Step3: enter the string with \$ at the end
- Step4: if (A(w[z])) then increment z and check for (B(w[z])) and if satisfies increment z and check for 'd' if d is present then increment and check for (D(w[z]))
- Step5: if step 4 is satisfied then the string is accepted
Else string is not
- Step6: give for the grammar $A \rightarrow bc/ab$ in the loop A(int k)
- Step7S: Describe the grammar $b \rightarrow c/d$ in the loop B (int k)
- Step8: Similarly describe the grammar $D \rightarrow d/abcd$
- Step9: if steps7,8,9 are satisfied accordingly string is accepted
Else string is not accepted
- Step10: Stop

5. Problem Statement : Design SLR Parser

AIM: Design SLR bottom up parser for the above language

ALGORITHM

SStep1: Start
Step2: Initially the parser has s0 on the stack where s0 is the initial state and w\$ is in buffer
Step3: Set ip point to the first symbol of w\$
Step4: repeat forever, begin
Step5: Let S be the state on top of the stack and a symbol pointed to by ip
Step6: If action [S, a] =shift S then begin
Push S1 on to the top of the stack
Advance ip to next input symbol
Step7: Else if action [S, a], reduce A->B then begin
Pop 2* |B| symbols of the stack
Let S1 be the state now on the top of the stack
Step8: Output the production A → B
End
Step9: else if action [S, a]=accepted, then return
Else
Error()
End
Step10: Stop

SOURCE CODE

```
// ***** IMPLEMENTATION OF LALR PARSING PROGRAM *****//

#include<stdio.h>
#include<conio.h>
#include<string.h>
#define MAX 50
void push(char item);
char pop(void);
int top=-1;
int call(char c);
char stack[MAX],input[10],str2[15],str1[8]="",c;
void prn(int j)
{
    int i;
    for(i=j;input[i]!='\0';i++)
        printf("%c",input[i]);
}
void prnstack(int top)
{
    int i;
    for(i=0;i<top;i++)
        printf("%c",stack[i]);
}
```

```

void main()
{
    char str1[6],*cmp="",c[8]="";
    int i=0,cn=0, k,j;
    FILE *ptr, *gptr;
    clrscr();
    printf("\n\n enter the expression :\n");
    scanf("%s",input);
    push('0');
    printf("\n\n\t STACK \t\t COMPARISION \t\t OUTPUT \n\n");
    do
    {
        printf("");
        prnstack(top);
        printf("\t\t");
        prn(i);
        if(strcmp(cmp,"1$")==0)
        {
            strcpy(str2,"accepted");
            printf("\n\nthe input is accepted");
            getch();
            exit(0);
        }
        else
        {
            cmp[0]=stack[top];
            cmp[1]=input[i];
            cmp[2]='\0';
            if((ptr=fopen("d:\\ltable.doc","r"))==NULL)
                printf("\n\n FILE CAN NOT BE OPEN");
            else
            {
                while(!feof(ptr))
                {
                    fscanf(ptr, "%s%s",str1,str2);
                    if(strcmp(str1,cmp)==0)
                    {
                        if(str2[0]=='s')
                        {
                            push(input[i]);
                            push(str2[1]);
                            i++;
                            break;
                        }
                        else if(str2[0]=='r')
                        {
                            cn=call(str2[1]);
                            for(k=0;k<(cn*2);k++)
                                pop();
                            c[0]=stack[top];
                            push(str2[0]);
                            c[1]=stack[top];
                        }
                    }
                }
            }
        }
    }
    while(1);
}

```

```

        c[2]='\0';
        if(strcmp(c,"0E")==0)
            push('1');
        else if(strcmp(c,"0T")==0)
            push('2');
        else if(strcmp(c,"0F")==0)
            push('3');
        else if(strcmp(c,"0E")==0)
            push('8');
        else if(strcmp(c,"0T")==0)
            push('2');
        else if(strcmp(c,"0F")==0)
            push('3');
        else if(strcmp(c,"0T")==0)
            push('9');
        else if(strcmp(c,"0F")==0)
            push('3');
        else if(strcmp(c,"0F")==0)
            push('t');
    }
    else if(strcmp(str2,"0")==0)
    {
        printf("\n\n the string is not accepted");
        break;
    }
}
}

fclose(ptr);
}
printf("\t\t%s\t\t\n",cmp,str2);
}
while(input[i]!='\0');
getch();
}
int call(char c)
{
    int count =0;
    switch(c)
    {
        case 1: strcpy(str2,"E->E+T");
                count=3;
                break ;

        case 2: strcpy(str2,"E->T");
                count=1;
                break;

        case 3: strcpy(str2,"T->T*F");
                count=3;
                break;
    }
}

```

```

    case 4: strcpy(str2,"T->F");
        count=1;
        break;

    case 5: strcpy(str2,"F->(E)");
        count=3;
        break;

    case 6: strcpy(str2,"F->id");
        count=1;
        break;
    }
    return count ;
}
void push(char item)
{
    if(top==MAX)
        printf("\n\n stack overflow");
    else
    {
        top=top+1;
        stack[top]=item;
    }
}
char pop(void)
{
    char item;
    if(top==-1)
        printf("\n\n stack underflow");
    else
    {
        item=stack[top];
        top--;
    }
    return item;
}

```

We have to down load from drive “D:\ lrtable.txt”

SLR PARSER TABLE

States	Action						GOTO		
	Id	+	*	(	)	\$	E	T	F
0	S5			S4			1	2	3
1						accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6				S11			
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

OUTPUT:

id*(id+id)\$

Grammer accepted