

RAJ KUMAR GOEL INSTITUTE OF TECHNOLOGY

5thKM Stone Delhi, Meerut Road, Near Raj Nagar Extension Road, Ghaziabad, UP-201003

Approved by AICTE, N. Delhi & Affiliated to Dr. A.P.J. Abdul Kalam Technical University, Lucknow

NBA Accredited Program (B. Tech- ECE, IT) & B. Pharma



LABORATORY MANUAL

Faculty Name : Mr.Sakil Ahmad
Course Name : Compiler Design Lab
Year/Sem : 3rd/6th
Email ID : sakilfcs@rkgit.edu.in

Department : CSE
Course Code : BCS-652
NBA Code : C309
Academic Year : 2024-25

Department of Computer Science and Engineering

VISION OF THE INSTITUTE

To continually develop excellent professionals capable of providing sustainable solutions to challenging problems in their fields and prove responsible global citizens.

MISSION OF THE INSTITUTE

We wish to serve the nation by becoming a reputed deemed university for providing value based professional education.

VISION OF THE DEPARTMENT

To be recognized globally for delivering high quality education in the ever changing field of computer science & engineering, both of value & relevance to the communities we serve.

MISSION OF THE DEPARTMENT

1. To provide quality education in both the theoretical and applied foundations of Computer Science and train students to effectively apply this education to solve real world problems.
2. To amplify their potential for lifelong high quality careers and give them a competitive advantage in the challenging global work environment.

PROGRAM EDUCATIONAL OUTCOMES (PEOs)

PEO 1: Learning: Our graduates to be competent with sound knowledge in field of Computer Science & Engineering.

PEO 2: Employable: To develop the ability among students to synthesize data and technical concepts for application to software product design for successful careers that meet the needs of Indian and multinational companies.

PEO 3: Innovative: To develop research oriented analytical ability among students to prepare them for making technical contribution to the society.

PEO 4: Entrepreneur / Contribution: To develop excellent leadership quality among students which they can use at different levels according to their experience and contribute for progress and development in the society.

PROGRAM OUT COMES (POs)

Engineering Graduates will be able to:

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

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

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

PO4: 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.

PO5: 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.

PO6: 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.

PO7: 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.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multi-disciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to

Comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

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

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

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1: The ability to use standard practices and suitable programming environment to develop software solutions.

PSO2: The ability to employ latest computer languages and platforms in creating innovative career opportunities.

COURSE OUTCOMES (COs)

C309.1	Formulate the knowledge of lex tool & yacc tool to develop a scanner & parser.
C309.2	Design Lexical analyzer for given language using C and Lex/Yacc tools.
C309.3	Design and analyze top down and bottom up parsers.
C309.4	Generate the intermediate code.
C309.5	Generate machine code from the intermediate code forms.

CO-PO MAPPING

CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
C309.1	2	1	2	1					2				1	
C309.2	2	1	1	2	3				2				1	1
C309.3	1	1	2	1					2					2
C309.4	2	1		2					2					2
C309.5	2		2	2					2				1	
CO	1.8	.8	1.4	1.6	0.6				2				0.6	1

Raj Kumar Goel Institute of Technology, Ghaziabad

Department of Computer Science & Engineering

Sr. No.	Title of Experiment	Corresponding CO
1	Design and implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and new lines.	C309.1
2	Implementation of Lexical Analyzer using Lex Tool	C309.2
3	Generate YACC specification for a few syntactic categories. a) Program to recognize a valid arithmetic expression that uses operator +, -, * and /. b) Program to recognize a valid variable which starts with a letter followed by any number of letters or digits. c) Implementation of Calculator using LEX and YACC d) Convert the BNF rules into YACC form and write code to generate abstract syntax tree	C309.2
4	Write program to find ϵ – closure of all states of any given NFA with ϵ Transition.	C309.1
5	Write program to convert NFA with ϵ transition to NFA without ϵ transition.	C309.1
6	Write program to convert NFA to DFA	C309.1
7	Write program to minimize any given DFA.	C309.1
8	Construct a Shift Reduce Parser for a given language.	C309.3
9	Write program to find Simulate First and Follow of any given grammar.	C309.3
10	Construct a recursive descent parser for an expression.	C309.3
11	Develop an operator precedence parser for a given language.	C309.3
12	Write a program to perform loop unrolling.	C309.5
13	Write a program to perform constant propagation.	C309.5
14	Implement Intermediate code generation for simple expressions.	C309.4
15	Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly instructions can be simple move, add, sub, jump etc.	C309.4
CONTENT BEYOND SYLLABUS		
16	Write a C program for constructing of LL (1) parsing.	C309.3
17	Write a program to Design LALR Bottom up Parser.	C309.3

INTRODUCTION

The language Processors comprises assemblers, compilers and interpreters. It deals with the recognition the translation, and the execution of formal languages It is closely related to compiler construction. Compiler is system software that converts high level language into low level language. We human beings can't program in machine language (low level lang.) understood by computers so we program in high level language and compiler is the software which bridges the gap between user and computer.

Students will gain the knowledge of

- Lexical Analysis for decomposing a character stream into lexical units
- Syntax analysis for recovering context-free structure from an input stream, error correction
- Semantic analysis for enforcing non-context-free requirements, attribute grammars.
- Semantic definition, for describing the meaning of a phrase(we rely on interpretive definition)
- Implementation of programming concepts, control structures
- Data representation, implementation of data structures
- Partial evaluation, for removing interpretation overhead
- Code generation: instruction selection, register allocation
- Further semantic analysis: document validation, type checking.

PREFACE

This lab is as a part of B.Tech. VI semester for CSE students. Compiler design principles provide an in-depth view of translation and optimization process. This lab enables the students to practice basic translation mechanism by designing complete translator for a mini language and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end with recommended systems/software requirements following the university prescribed textbooks. The expected outcomes from the students are:

1. By this laboratory, students will understand the practical approach of how a compiler works.
2. This will enable him to work in the development phase of new computer languages in Industry.

We extend our sincere thanks to Department of Computer Science and Engineering, R.K.G.I.T., Ghaziabad, for giving us this opportunity to prepare a Compiler Design Lab Manual.

Mr. Sakil Ahmad
Assistant Professor
Department of Computer Science & Engineering

DO'S AND DONT'S

DO's

1. Conform to the academic discipline of the department.
2. Enter your credentials in the laboratory attendance register.
3. Read and understand how to carry out an activity thoroughly before coming to the laboratory.
4. Ensure the uniqueness with respect to the methodology adopted for carrying out the experiments.
5. Shut down the machine once you are done using it.

DONT'S

1. Eatables are not allowed in the laboratory.
2. Usage of mobile phones is strictly prohibited.
3. Do not open the system unit casing.
4. Do not remove anything from the computer laboratory without permission.
5. Do not touch, connect or disconnect any plug or cable without your faculty/laboratory technician's permission.

GENERAL SAFETY INSTRUCTIONS

1. Know the location of the fire extinguisher and the first aid box and how to use the min case of an emergency.
2. Report fire or accidents to your faculty /laboratory technician immediately.
3. Report any broken plugs or exposed electrical wires to your faculty/laboratory technician immediately.
4. Do not plug in external devices without scanning them for computer viruses.

GUIDELINES FOR LABORTORY RECORDPREPARATION

While preparing the lab records, the student is required to adhere to the following guidelines:

Contents to be included in Lab Records:

1. Cover page
2. Vision
3. Mission
4. PEOs
5. POs
6. PSOs
7. COs
8. CO-PO-PSO mapping
9. Index
10. Experiments
 - Aim
 - Source code
 - Input-Output

A separate copy needs to be maintained for pre-lab written work.

The student is required to make the Lab File as per the format given on the next two pages.

RAJ KUMAR GOEL INSTITUTE OF TECHNOLOGY

5thKM Stone Delhi, Meerut Road, Near Raj Nagar Extension Road, Ghaziabad, UP-201003

Approved by AICTE, N. Delhi & Affiliated to Dr. A.P.J. Abdul Kalam Technical University, Lucknow

NBA Accredited Program (B. Tech- ECE, IT) & B. Pharma



COMPILER DESIGN LAB FILE (BCS 652)

Name	
Roll No.	
Section- Batch	

Raj Kumar Goel Institute of Technology, Ghaziabad
Department of Computer Science & Engineering

INDEX

Experiment No.	Experiment Name	Date of Conduction	Date of Submission	Faculty Signature

Department of Computer Science & Engineering

GUIDELINES FOR ASSESSMENT

Students are provided with the details of the experiment (Aim, pre-experimental questions, procedure etc.) to be conducted in next lab and are expected to come prepared for each lab class.

Faculty ensures that students have completed the required pre-experiment questions and they complete the in-lab programming assignment(s) before the end of class. Given that the lab programs are meant to be formative in nature, students can ask faculty for help before and during the lab class.

Students' performance will be assessed in each lab based on the following Lab Assessment Components:

Assessment Criteria-1: Performance (Max. marks = 5)

Assessment Criteria-2: VIVA (Max. marks = 5)

Assessment Criteria-3: Record (Max. marks = 5)

In each lab class, students will be awarded marks out of 5 under each component head, making it total out of 15 marks.

Department of Computer Science & Engineering

EXPERIMENT#1

Aim: Design and implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and new lines.

Description:

Lexical analysis or scanning is the process where the stream of characters making up the source program is read from left-to-right and grouped into tokens. Tokens are sequences of characters with a collective meaning. There are usually only a small number of tokens for a programming language: constants (integer, double, char, string, etc.), operators (arithmetic, relational, logical), punctuation, and reserved words.

PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.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 key word", 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("\nEnter the program");
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))
{
```

Department of Computer Science & Engineering

```
tokenvalue=c-'0';
c=getc(f1);
while(isdigit(c)) {
    tokenvalue*=10+c-'0';
    c=getc(f1);
}
num[i++]=tokenvalue;
ungetc(c,f1);
}
elseif(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);
}
elseif(c==' ' || c=='\t')
    printf(" ");
else
    if(c=='\n')
        lineno++;
    else
        putc(c,f3);
}
fclose(f2);
fclose(f3);
fclose(f1);
printf("\nThe 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;
    }
}
```


Department of Computer Science & Engineering

```
fclose(f2);
f3=fopen("specialchar", "r");
printf("\nSpecial characters are");
while((c=getc(f3))!=EOF)
printf("%c",c);
printf("\n");
fclose(f3);
printf("Total no. of lines are:%d",lineno);
}
```

INPUT:

Enter Program \$ for termination:

```
{
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(-t1+t2*t3); /* this is a comment on 2 lines */
} endif
}
```

OUTPUT:

Variables :a[3]t1 t2 t3
Operator : - + * / >
Constants : 2 1 3 6 5 99 -25
Keywords : int if then else endif
Special Symbols : , ; () { }
Comments : this is a comment on 2 lines

QUESTIONS:

1. EXPLAIN COMPILER PHASES.
2. DIFFERENCE BETWEEN COMPILER AND INTERPRETER.

EXPERIMENT #2

Aim: To write a c program for implementing a lexical analyzer using LEX tool

ALGORITHM:

Step1: Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%. The format is as follows: definitions %% rules %% user subroutines.

Step2: In definition section, the variables make up the left column, and their definitions make up the right column. Any C statements should be enclosed in %{..}%. Identifier is defined such that the first Letter of an identifier is alphabet and remaining letters are alphanumeric.

Step3: In rules section, the left column contains the pattern to be recognized in an input file to yylex(). The right column contains the C program fragment executed when that pattern is recognized. The various patterns are keywords, operators, new line character, number, string, identifier, beginning and end of block, comment statements, preprocessor directive statements etc.

Step4: Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.

Step5: When yylex() matches a string in the input stream, it copies the matched text to an external character array, yytext, before it executes any actions in the rules section.

Step6: In user subroutine section, main routine calls yylex(). yywrap() is used to get more input.

Step7: The lex command uses the rules and actions contained in file to generate a program, lex.yy.c, which can be compiled with the cc command. That program can then receive input, break the input into the logical pieces defined by the rules in file, and run program fragments contained in the actions in file.

PROGRAM CODE:

```
//Implementation of Lexical Analyzer using Lex tool
%{
int COMMENT=0;

% }

identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf("\n%s is a preprocessor directive",yytext);
}
int |
```

Department of Computer Science & Engineering

```
float |
char |
double |
while |
for |
struct |
typedef |
do |
if |
break |
continue |
void |
switch |
return |
else |
goto {printf("\n\t%s is a keyword",yytext);}
"/*" {COMMENT=1;} {printf("\n\t %s is a COMMENT",yytext);}
{identifier}\( {if(!COMMENT)printf("\nFUNCTION \n\t%s",yytext);}
\{ {if(!COMMENT)printf("\n BLOCK BEGINS");}
\} {if(!COMMENT)printf("BLOCK ENDS ");}
{identifier}\([0-9]*\)? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\'.*\' {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}
\\(:)? {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}
\(( ECHO;
= {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
%%

int main(int argc, char **argv)
{
FILE *file;
file=fopen("var.c","r");
if(!file)
{
printf("could not open the file");
exit(0);
}
yyin=file;
yylex();
printf("\n");
return(0);
}

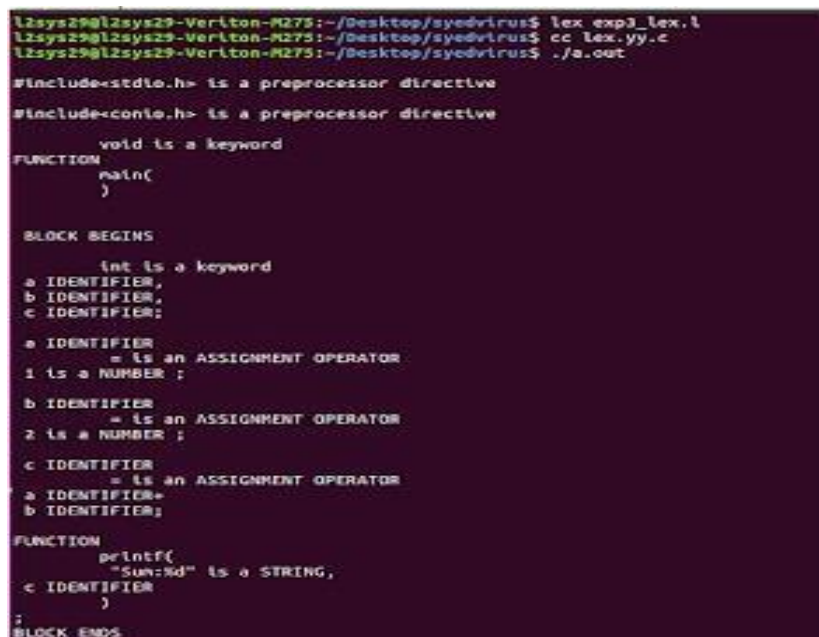
int yywrap()
{
return(1);
}
```

Department of Computer Science & Engineering

INPUT:

```
//var.c
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,c;
    a=1;
    b=2;
    c=a+b;
    printf("Sum:%d",c);
}
```

OUTPUT:



```
l2sys29@l2sys29-Verlton-M275:~/Desktop/syedvirus$ lex exp3_lex.l
l2sys29@l2sys29-Verlton-M275:~/Desktop/syedvirus$ cc lex.yy.c
l2sys29@l2sys29-Verlton-M275:~/Desktop/syedvirus$ ./a.out

#include<stdio.h> is a preprocessor directive
#include<conio.h> is a preprocessor directive
void is a keyword
FUNCTION
main(
)

BLOCK BEGINS
    int is a keyword
    a IDENTIFIER,
    b IDENTIFIER,
    c IDENTIFIER;
    a IDENTIFIER
    = is an ASSIGNMENT OPERATOR
    1 is a NUMBER ;
    b IDENTIFIER
    = is an ASSIGNMENT OPERATOR
    2 is a NUMBER ;
    c IDENTIFIER
    = is an ASSIGNMENT OPERATOR
    a IDENTIFIER+
    b IDENTIFIER;
FUNCTION
printf(
    "Sum:%d" is a STRING,
    c IDENTIFIER
)
;
BLOCK ENDS
```

QUESTIONS:

1. ANALYZE THE PARSE TREE AND SYNTAX TREE.
2. WRITE SHORT NOTES ON LEX TOOL.

EXPERIMENT #3

Aim: Generate YACC specification for a few syntactic categories:

- a) Program to recognize a valid arithmetic expression that uses operator +, -, * and/.
- b) Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.
- c) Implementation of Calculator using LEX and YACC.
- d) Convert the BNF rules into YACC form and write code to generate abstract syntax tree

Description:

A parser generators program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.

YACC (yet another compiler-compiler) is an LALR(1)(Look Ahead, Left-to-right, Rightmost derivation producer with 1 look ahead token) parser generator. YACC was originally designed for being complemented by Lex.

For Compiling YACC Program:

Write lex program in a file file.l and yacc in a file file.y

Open Terminal and Navigate to the Directory where you have saved the files.

type lex file.l

type yacc file.y

type cc lex.yy.c y.tab.h -ll

type ./a.out

a) Program to recognize a valid arithmetic expression that uses operator +, -, * and/.

Program name:arith_id.l

```
% {
/* This LEX program returns the tokens for the expression */
#include "y.tab.h"
% }

%%
"=" {printf("\n Operator is EQUAL");
}
"+",
{
printf("\n Operator is PLUS");
}
"_"
{
printf("\n Operator is MINUS");
}
"/"
{printf("\n Operator isDIVISION");
}
"*"
{
```

Department of Computer Science & Engineering

```
printf("\n Operator is MULTIPLICATION");  
}
```

```
[a-zA-Z]*[0-9]*  
{  
printf("\n Identifier is %s",yytext);  
return ID;  
}  
return yytext[0];  
\n return 0;  
%%
```

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

Program name: variable_test.y

```
% {  
#include  
/* This YACC program is for recognising the Expression*/  
% }  
%token ID INT FLOAT DOUBLE  
%%  
D;TL  
;  
L:L,ID  
|ID;  
  
%%  
extern FILE *yyin;  
main()  
{  
do  
{  
yyparse();  
}while(!feof(yyin));  
}  
  
yyerror(char*s)  
{  
}
```

Department of Computer Science & Engineering

```
}
```

Program Name : arith_id.

```
% {
#include
/* This YYAC program is for recognizing the Expression */
% }
%%
statement: A '=' E
| E {
printf("\n Valid arithmetic expression");
$$ = $1;
};
```

```
E: E '+' ID
| E '-' ID
| E '*' ID
| E '/' ID
| ID
;
%%
extern FILE *yyin;
main()
{
do
{
yyparse();
}while(!feof(yyin));
}
```

```
yyerror(char*s)
{
}
```

Output:

```
[root@localhost]# lex arith_id.l
[root@localhost]# yacc -d arith_id.y
[root@localhost]# gcc lex.yy.c y.tab.c
[root@localhost]# ./a.out
x=a+b;
```

```
Identifier is x
Operator is EQUAL
Identifier is a
Operator is PLUS
Identifier is b
```

b) Program to recognise a valid variable which starts with a letter followed by any number of letters or digits.

Program name: variable_test.l

```
% {
/* This LEX program returns the tokens for the Expression */
```

Department of Computer Science & Engineering

```
#include "y.tab.h"
% }
%%
"int " {return INT;}
"float" {return FLOAT;}
"double" {return DOUBLE;}
[a-zA-Z]*[0-9]*{
printf("\nIdentifier is %s",yytext);
return ID;
}
return yytext[0];
\n return 0;
int yywrap()
{
return 1;
}
```

Program name: variable_test.y

```
% {
#include
/* This YACC program is for recognising the Expression*/
% }
%token ID INT FLOAT DOUBLE
%%
D;TL
;
L:L,ID
|ID
;
T:INT
|FLOAT
|DOUBLE
;
%%
extern FILE *yyin;
main()
{
do
{
yyparse();
}while(!feof(yyin));
}
yyerror(char*s)
{
}
```

Output:

```
[root@localhost]# lex variable_test.I
[root@localhost]# yacc -d variable_test.y
[root@localhost]# gcc lex.yy.c y.tab.c
[root@localhost]# ./a.out
```

Department of Computer Science & Engineering

int a,b;

Identifier is a

Identifier is b[root@localhost]#

c) Implementation of Calculator using LEX and YACC

Program name: calci.l

```
% {
#include "y.tab.h" /*defines the tokens*/
#include ,math.h.
% }
%%
/*To recognise a valid number*/
([0-9] + |([0-9]*.[0-9]+)([eE][+-]?[0-9]+)?) { yylval.dval = atof(yytext);
return NUMBER;}
/*For log no | Log no (log base 10)*/
log | LOG {return LOG;}

/*For ln no (Natural Log)*/
ln {return nLOG;}

/*For sin angle*/
sin | SIN {return SINE;}

/*For cos angle*/
cos | COS {return COS;}

/*For tan angle*/
tan | TAN {return TAN;}
/*For memory*/
mem {return MEM;}

[t] ; /*Ignore white spaces*/

/*End of input*/
\$ {return 0;}

/*Catch the remaining and return a single character token to
the parser*/
\n| return yytext[0];
%%
```

Program Name : calci.y

```
% {
double memvar;
% }

/*To define possible symbol types*/
%token NUMBER
%token MEM
%token LOG SINE nLOG COS TAN
```

Department of Computer Science & Engineering

```
/*Defining the precedences and associativity*/
%left '-' '+' /*Lowest precedence*/
%left '*' '/'
%right '^'
%left LOG SINE nLOG COS TAN /*Highest precedence*/

/*No associativity*/
%nonassoc UMINUS /*Unary Minus*/

/*Sets the type for non-terminal*/
%type expression
%%
/*Start state*/
start: statement '\n'
| start statement '\n'
;

/*For storing the answer(memory)*/
statement: MEM '=' expression {memvar=$3;}
| expression {printf("Answer = %g\n", $1);}
; /*For printing the Answer*/
/*For binary arithmetic operations*/
expression: expression '+' expression {$$ = $1 + $3;}
| expression '-' expression {$$ = $1 - $3;}
| expression '*' expression {$$ = $1 * $3;}
| expression '/' expression
{ /*Tohandle divide by zero case*/
If($3 == 0)
yyerror("divide by zero");
else
$$ = $1 / $3;
}
| expression '^' expression {$$ = pow($1, $3);}
;
/*For unary operators*/
expression: '-' expression %prec UMINUS {$$ = -$2;}
/*%prec UMINUS signifies that unary minus should have the highest precedence*/
| '(' expression ')' {$$ = $2}
| LOG expression {$$ = log($2)/log(10);}
| nLOG expression {$$ = log($2);}
/*Trigonometric functions*/
| SINE expression {$$ = sin($2 * 3.141592654 / 180);}
| COS expression {$$ = cos($2 * 3.141592654 / 180);}
| TAN expression {$$ = tan($2 * 3.141592654 / 180);}
| NUMBER {$$ = $1;}
| MEM {$$ = $1;}
; /*Retrieving the memory contents*/
%%
main()
{
printf("Enter the expression:");
yyvsparse();
}
int yyerror(char *error)
{
```

Department of Computer Science & Engineering

```
fprintf(stderr,"%s\n",error);  
}
```

Output:

The output of the program can be obtained by following commands

```
[root@localhost]# lex calci.l  
[root@localhost]# yacc -d calci.y  
[root@localhost]# cc y.tab.c lex.yy.c -ll -ly -lm  
[root@localhost]# ./a.out
```

Enter the expression: 2+@

Answer =4

2 * 2 + 5 /4

Answer = 5.25

mem = cos 45

sin 45/mem

Answer = 1

ln 10

Answer = 2.

QUESTIONS:

1. DIFFERENCE BETWEEN LEX TOOL AND YACC TOOL.
2. WRITE SHORT NOTES ON YACC TOOL.

EXPERIMENT #4

Aim: Write program to find ϵ – closure of all states of any given NFA with ϵ transition.

Description:

A nondeterministic finite automaton (NFA), or nondeterministic finite state machine, does not need to obey these restrictions. In particular, every DFA is also an NFA.

Using the subset construction algorithm, each NFA can be translated to an equivalent DFA, i.e. a DFA recognizing the same formal language. Like DFAs, NFAs only recognize regular languages. Sometimes the term NFA is used in a narrower sense, meaning an automaton that properly violates an above restriction i.e. that is *not* a DFA.

ALGORITHM:

Step1: Start the Program.

Step2: Enter the regular expression R over alphabet E.

Step3: Decompose the regular expression R into its primitive components Step4: For each component construct finite automata.

Step5: To construct components for the basic regular expression way that corresponding to that way compound regular expression.

Step6: Stop the Program

PROGRAM :

```
// Program to find epsilon closure of a given NFA

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

char result[20][20], copy[3], states[20][20];
void add_state(char a[3], int i) {
    strcpy(result[i], a);
}
void display(int n) {
    int k=0;
    printf("nnn Epsilon closure of %s = { ", copy);
    while(k < n) {
        printf(" %s", result[k]);
    }
}
```

Department of Computer Science & Engineering

```
k++;
}
printf(" } nnn");
}
int main(){
FILE *INPUT;
INPUT=fopen("input.dat","r");
char state[3];
int end,i=0,n,k=0;
char state1[3],input[3],state2[3];
printf("\n Enter the no of states: ");
scanf("%d",&n);
printf("\n Enter the states n");
for(k=0;k<3;k++){
scanf("%s",states[k]);
}

for( k=0;k<n;k++){
i=0;
strcpy(state,states[k]);
strcpy(copy,state);
add_state(state,i++);
while(1){
end = fscanf(INPUT,"%s%s%s",state1,input,state2);
if (end == EOF ){
break;
}

if( strcmp(state,state1) == 0 ){
if( strcmp(input,"e") == 0 ) {
add_state(state2,i++);

strcpy(state, state2);
}
}

display(i);
rewind(INPUT);
}

return 0;
}
```

INPUT& OUTPUT:

```
q0 0 q0
q0 1 q1
q0 e q1
q1 1 q2
```

q1 e q2

Enter the no of states: 3

Enter the states

q0

q1

q2

Epsilon closure of q0= { q0 q1 q2 }

Epsilon closure of q1= { q1 q2 }

Epsilon closure of q2= { q2 }

QUESTIONS:

1. DEFINE AUTOMATA.
2. DIFFERENCE BETWEEN NFA AND DFA.

EXPERIMENT #5

Aim: Write program to convert NFA with ϵ transition to NFA without ϵ transition.

PROGRAM CODE:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int st;
struct node *link;
};
void findclosure(int,int);
void insert_trantbl(int ,char, int);
int findalpha(char);
void findfinalstate(void);
void unionclosure(int);
void print_e_closure(int);
static int set[20],nostate,noalpha,s,notransition,nofinal,start,finalstate[20],c,r,buffer[20];
char alphabet[20];
static int e_closure[20][20]={0};
struct node * transition[20][20]={NULL};
void main()
{
int i,j,k,m,t,n;
struct node *temp;
printf("enter the number of alphabets?\n");
scanf("%d",&noalpha);
getchar();
printf("NOTE:- [ use letter e as epsilon]\n");
printf("NOTE:- [e must be last character ,if it is present]\n");
printf("\nEnter alphabets?\n");
for(i=0;i<noalpha;i++)
{
alphabet[i]=getchar();
getchar();
}
printf("Enter the number of states?\n");

scanf("%d",&nostate);
printf("Enter the start state?\n");
scanf("%d",&start);
printf("Enter the number of final states?\n");
scanf("%d",&nofinal);
printf("Enter the final states?\n");
for(i=0;i<nofinal;i++)
```

```
scanf("%d",&finalstate[i]);
printf("Enter no of transition?\n");
scanf("%d",&notransition);
printf("NOTE:- [Transition is in the form--> qno alphabet qno]\n",notransition);
printf("NOTE:- [States number must be greater than zero]\n");
printf("\nEnter transition?\n");
for(i=0;i<notransition;i++)
{
scanf("%d %c%d",&r,&c,&s);
insert_trantbl(r,c,s);
}
printf("\n");
for(i=1;i<=nostate;i++)
{
c=0;
for(j=0;j<20;j++)
{
buffer[j]=0;
e_closure[i][j]=0;
}
findclosure(i,i);
}
printf("Equivalent NFA without epsilon\n");
printf(".....\n");
printf("start state:");
print_e_closure(start);
printf("\nAlphabets:");
for(i=0;i<noalpha;i++)
printf("%c ",alphabet[i]);
printf("\n States :");
for(i=1;i<=nostate;i++)
print_e_closure(i);
printf("\nTransitions are...:\n");
for(i=1;i<=nostate;i++)
{
for(j=0;j<noalpha-1;j++)
{
for(m=1;m<=nostate;m++)
set[m]=0;
for(k=0;e_closure[i][k]!=0;k++)
{
t=e_closure[i][k];
temp=transition[t][j];

while(temp!=NULL)
unionclosure(temp->st);
temp=temp->link;
}
}
}
```



```
printf("\n");
print_e_closure(i);
printf("%c\t",alphabet[j] );
printf(" ");
for(n=1;n<=nostate;n++)
{
if(set[n]!=0)
printf("q%d,",n);
}
printf(" ");
}
printf("\n Final states:");
findfinalstate();
}
void findclosure(int x,int sta)
{
struct node *temp;
int i;
if(buffer[x])
return;
e_closure[sta][c++]=x;
buffer[x]=1;
if(alphabet[noalpha-1]=='e' && transition[x][noalpha-1]!=NULL)
{
temp=transition[x][noalpha-1];
while(temp!=NULL)
{
findclosure(temp->st,sta);
temp=temp->link;
}
}
}
void insert_trantbl(int r,char c,int s)
{
int j;
struct node *temp;
j=findalpha(c);
if(j==999)
{
printf("error\n");
exit(0);
}
temp=(struct node *) malloc(sizeof(struct node));
temp->st=s;
temp->link=transition[r][j];
transition[r][j]=temp;
}
int findalpha(char c)
{

```

Department of Computer Science & Engineering

```
int i;
for(i=0;i<noalpha;i++)
{
if(alphabet[i]==c)
return i;
}
return(999);
}
void unionclosure(int i)
{
int j=0,k;
while(e_closure[i][j]!=0)
{
k=e_closure[i][j];
set[k]=1;
j++;
}
}
void findfinalstate()
{
int i,j,k,t;
for(i=0;i<nofinal;i++)
{
for(j=1;j<=nostate;j++)
{
for(k=0;e_closure[j][k]!=0;k++)
{
if(e_closure[j][k]==finalstate[i])
{
print_e_closure(j);
}
}
}
}
}
void print_e_closure(int i)
{
int j;
printf("{ ");
for(j=0;e_closure[i][j]!=0;j++)
printf("q%d,",e_closure[i][j]);
printf("}\t");
}
```

INPUT & OUTPUT:

```
enter the number of alphabets?
4
NOTE:- [ use letter e as epsilon]
NOTE:- [e must be last character ,if it is present]
Enter alphabets?
a
b
c
e
Enter the number of states?
3
Enter the start state?
1
Enter the number of final states?
1
Enter the final states?
3
Enter no of transition?
5
NOTE:- [Transition is in the form--> qno  alphabet  qno]
NOTE:- [States number must be greater than zero]
Enter transition?
1      a      1
1      e      2
2      b      2
2      e      3
3      c      3

-----
Equivalent NFA without epsilon
start state:<q1,q2,q3,>
Alphabets:a b c e
States :<q1,q2,q3,>      <q2,q3,>      <q3,>
Intransitions are....:

<q1,q2,q3,>      a      <q1,q2,q3,>
<q1,q2,q3,>      b      <q2,q3,>
<q1,q2,q3,>      c      <q3,>
<q2,q3,>         a      <>
<q2,q3,>         b      <q2,q3,>
<q2,q3,>         c      <q3,>
<q3,>            a      <>
<q3,>            b      <>
<q3,>            c      <q3,>
Final states:<q1,q2,q3,>      <q2,q3,>      <q3,>      _
```

QUESTIONS:

- 1 MINIMIZE THE DFA WITH EXAMPLE.
- 2 CONVERT NFA WITH ϵ TRANSITION TO NFA WITHOUT ϵ TRANSITION.

EXPERIMENT #6

Aim: Write program to convert NFA to DFA

Description:

Step1: $q_0=2^0=1$, $q_1=2^1=2$, $q_2=2^2=4$

Step2: Similarly union of states will be represented as:

$q_0q_1=2^0+2^1=3$, $q_1q_2=2^1+2^2=6$, $q_0q_1q_2=2^0+2^1+2^2=7$

Step3: Do not give any condition for "phi".....

That case is not handled.....(Coz I M LAZY:P)

Step4: Follow zero based indexing everywhere

Step5: Program assumes that if "Number of states are=n" then they are numbered as $q_0, q_1, q_2, \dots, q_{(n-1)}$

Step6: If you find any bug, message me and forgive me for the error.

Program :

```
#include<string.h>
#include<stdio.h>
#include<math.h>
int ninputs;
int dfa[100][2][100] = {0};
int state[10000] = {0};
char ch[10], str[1000];
int go[10000][2] = {0};
int arr[10000] = {0};
int main()
{
    int st, fin, in;
    int f[];
    int i,j=3,s=0,final=0,flag=0,curr1,curr2,k,l;
    int c;
    printf("\nFollow the one based indexing\n");
    printf("\nEnter the number of states::");
    scanf("%d",&st);
    printf("\nGive state numbers from 0 to %d",st-1);
    for(i=0;i<st;i++)
        state[(int)(pow(2,i))] = 1;
    printf("\nEnter number of final states\t");
    scanf("%d",&fin);
    printf("\nEnter final states::");
    for(i=0;i<fin;i++)
    {
```

Department of Computer Science & Engineering

```
scanf("%d",&f[i]);
}
int p,q,r,rel;
printf("\nEnter the number of rules according to NFA::");
scanf("%d",&rel);
printf("\n\nDefine transition rule as \"initial state input symbol final state\\n");
for(i=0; i<rel; i++)
{
scanf("%d%d%d",&p,&q,&r);
if (q==0)
dfa[p][0][r] = 1;
else
dfa[p][1][r] = 1;
}
printf("\nEnter initial state::");
scanf("%d",&in);
in = pow(2,in);
i=0;
printf("\nSolving according to DFA");
int x=0;
for(i=0;i<st;i++)
{
for(j=0;j<2;j++)
{
int stf=0;
for(k=0;k<st;k++)
{
if(dfa[i][j][k]==1)
stf = stf + pow(2,k);
}
go[(int)(pow(2,i))][j] = stf;
printf("%d-%d-->%d\n",(int)(pow(2,i)),j,stf);
if(state[stf]==0)
arr[x++] = stf;
state[stf] = 1;
}
}
//for new states
for(i=0;i<x;i++)
{
printf("for %d-----",arr[x]);
for(j=0;j<2;j++)
{
int new=0;
for(k=0;k<st;k++)
{
if(arr[i] & (1<<k))
{
int h = pow(2,k);
```

Department of Computer Science & Engineering

```
if(new==0)
new = go[h][j];
new = new | (go[h][j]);
}
}
if(state[new]==0)
{
arr[x++] = new;
state[new] = 1;
}
}
}
printf("\nThe total number of distinct states are::\n");
printf("STATE    01\n");
for(i=0;i<10000;i++)
{
    if(state[i]==1)
    {
        //printf("%d**",i);
        int y=0;
        if(i==0)
        printf("q0 ");
        else
        for(j=0;j<st;j++)
        {
            int x = 1<<j;
            if(x&i)
            {
                printf("q%d ",j);
                y = y+pow(2,j);
                //printf("y=%d",y);
            }
        }
        //printf("%d",y);
        printf("    %d%d",go[y][0],go[y][1]);
        printf("\n");
    }
}
j=3;
while(j--)
{
    printf("\nEnter string");
    scanf("%s",str);
    l = strlen(str);
    curr1 = in;
    flag = 0;
    printf("\nString takes the following path-->\n");
    printf("%d-",curr1);
```

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

Department of Computer Science & Engineering

```
{
curr1 = go[curr1][str[i]-'0'];
printf("%d-",curr1);
}

printf("\n Final state - %d\n",curr1);

for(i=0;i<fin; i++)
{
if(curr1 & (1<<f[i]))
{
flag = 1;
break;
}
}
if(flag)
printf("\n String Accepted");
else
printf("\n String Rejected");
}
return 0;
}
```

Input & Output:

Follow the one based indexing

Enter the number of states::3

Give state numbers from 0 to 2

Enter number of final states 1

Enter final states::4

Enter the number of rules according to NFA::4

Define transition rule as "initial state input symbol final state"

1 0 1

1 1 1

1 0 2

2 0 4

Enter initial state::1

Solving according to DFA1-0-->0

1-1-->0

2-0-->6

2-1-->2

4-0-->0

4-1-->0

for 0 ---- for 0 ----

The total number of distinct states are::

STATE 0 1

q0 0 0

q0 0 0

q1 6 2

q2 0

Department of Computer Science & Engineering

QUESTIONS:

1. DESCRIBE AUTOMATA.
2. CONVERT NFA TO DFA.

EXPERIMENT #7

Aim: Write program to minimize any given DFA.

ALGORITHM:

Suppose there is a DFA $D < Q, \Sigma, q_0, \delta, F >$ which recognizes a language L . Then the minimized DFA $D < Q', \Sigma, q_0, \delta', F' >$ can be constructed for language L as:

Step 1: We will divide Q (set of states) into two sets. One set will contain all final states and other set will contain non-final states. This partition is called P_0 .

Step 2: Initialize $k = 1$

Step 3: Find P_k by partitioning the different sets of P_{k-1} . In each set of P_{k-1} , we will take all possible pair of states. If two states of a set are distinguishable, we will split the sets into different sets in P_k .

Step 4: Stop when $P_k = P_{k-1}$ (No change in partition)

Step 5: All states of one set are merged into one. No. of states in minimized DFA will be equal to no. of sets in P_k .

PROGRAM :

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int **transition Map; // 2D array which is used to store state transitions. Transition Map[i][j]
is the state
reached when state i is given symbol j
int **partition Transition Map; // same as transition Map, except row indices represent
partition numbers,
not state numbers
int start State; // The starting state. This is used as the root for DFS to eliminate unreachable
states
long int reachable; // A bitset to represent states that are reachable
long int allStates; // A bitset to represent all states in the FSM
long int finalStates; // A bitset to represent final states in the FSM
long int nonFinalStates; // A bitset to represent non-final states in the FSM
long int *P; // array of partitions. Each partition is a bitset of states
void dfs(int v)
{
    reachable |= (1 << v);
    // Try exploring all paths..
    for(int i=0; i<26; i++)
```

Department of Computer Science & Engineering

```
if((transitionMap[v][i] != -1) && ((reachable & (1 << transitionMap[v][i])) == 0))
{
    dfs(transitionMap[v][i]);
}
}
int main(){
// We start off with no states
finalStates = 0;
allStates = 0;
// Initialize our transition maps. We set transition[i][j] to be -1 in order to indicate that
state/partition i
does not transition when given symbol j
transitionMap = (int**)malloc(64*sizeof(int*));
for (int i = 0; i < 64; i++){
    transitionMap[i] = (int*) malloc(26*sizeof(int));
    for (int j = 0; j < 26; j++){
        transitionMap[i][j] = -1;
    }
}
partitionTransitionMap = (int**)malloc(64*sizeof(int*));
for (int i = 0; i < 64; i++){
    partitionTransitionMap[i] = (int*) malloc(26*sizeof(int));
    for (int j = 0; j < 26; j++){
        partitionTransitionMap[i][j] = -1;
    }
}
// read start state
char buff[125];
fgets(buff, sizeof(buff), stdin);
char *p = strtok(buff, " ");
startState = atoi(p);
// read final states
fgets(buff, sizeof(buff), stdin);
p = strtok(buff, " ");
while (p != NULL)
{
    int state = atoi(p);
    finalStates |= 1 << (state);
    p = strtok(NULL, " ");
}
// read transitions
int from;
char symbol;
int to;
while (fscanf(stdin, "%d %c %d", &from, &symbol, &to) != EOF) {
    transitionMap[from][symbol-'a'] = to; // add transition
    // add from and to states to the allStates bitset
    llStates |= (1 << from);
    allStates |= (1 << to);
}
```

```
// initialize reachable bitset to 0 and run dfs to determine reachable states
reachable = 0;
dfs(startState);
// filter unreachable states
allStates &= reachable;
finalStates &= reachable;
// initialize array of partitions to include empty bitsets
P = (long int*) malloc(64*sizeof(long int));
for (int i = 0; i < 64 ; i++){
P[i] = 0; // no partition exists
}
// P should include two partitions to start: final states and non-final states
nonFinalStates = allStates & ~finalStates;
P[0] = finalStates;
P[1] = nonFinalStates;
int nextPartitionIndex = 2; // Store how many partitions have been added already
// There will be at most 64 partitions. At each iteration, we operate on a partition and add at
most 1
more partition
for (int i = 0; i < 64; i++){
// A bitset for a new partition. This partition will include all states that are distinct from the state
corresponding to the leftmost bit in P[i]
long int newPartition = 0;
// Done partitioning
if (P[i] == 0){
break;
}
/* Try to find leftmost bit in the bitset. This loop will only run to its entirety once when that bit
is found*/
for (int j = 63; j >= 0; j--) {
// Potential leftmost bit. If found, this bit will remain in the bit set.
long int staticState = (long int) 1 << j;
// Check if this state is in the current bitset
if ((P[i] & (staticState)) != 0){
// The leftmost bit state will be associated with this partition. Therefore, we must copy over the
transitions for this state to the transitions for
// the corresponding partition
partitionTransitionMap[i] = transitionMap[j];
// Check for states that should be removed from this partition. All states will be bits right of the
staticState bit
for (int k = j - 1; k >= 0; k -- ){
// Potential state to remove
long int otherState = (long int) 1 << k;
// Check if this state is in the current bitset
if ((P[i] & (otherState)) != 0){
/* Iterate across the entire alphabet and check if staticState and otherState can transition to
Different*/
partitions.
for (int l = 0; l < 26; l++){
```

```
int staticNext = -1; // next partition for static
int otherNext = -1; // next partition for other
for (int m = 0; m < nextPartitionIndex; m++){
    if ((P[m] & (1 << transitionMap[j][l])) != 0){
        staticNext=m;          //found staticnext
    }
    if ((P[m] & (1 << transitionMap[k][l])) != 0){
        otherNext = m; // found other next
    }
}

// If partitions differ, remove the other state and add it to the new partition. Then break, since we
are
done with this partition
if (transitionMap[j][l] != transitionMap[k][l] && (staticNext != otherNext)){
    P[i] &= ~(1 << k);
    newPartition |= (1 << k);
    break;
}
}
}
}
break;
}
}
// New partition exists. Add it to P and increment nextPartitionIndex
if (newPartition != 0){
    P[nextPartitionIndex] = newPartition;
    nextPartitionIndex++;
}
}
// find and print start partition
int startPartition = 0;
for (int i = 0; i < nextPartitionIndex; i++){
    if ((P[i] & (1 << startState)) != 0 ){
        startPartition = i;
        break;
    }
}
printf("%d \n", startPartition);
// find and print final partitions
for (int i = 0; i < nextPartitionIndex; i++){
    if ((P[i] & finalStates) != 0){
        printf("%d ", i);
    }
}
printf("\n");
// find and print all transitions
for (int i = 0; i < nextPartitionIndex; i++){
```

```
for (int j = 0; j < 26; j++) {  
if (partitionTransitionMap[i][j] != -1){  
for (int k = 0; k < nextPartitionIndex; k++){  
if ((P[k] & (1 << partitionTransitionMap[i][j])) != 0){  
printf("%d %c %d\n", i, j + 'a', k);  
}  
}  
}  
}  
}  
return 0;  
}
```

INPUT:

```
0  
2 5  
0 a1  
1 b 2  
0 b 3  
3 a4  
4 b 5
```

OUTPUT:

```
3  
0  
1 b 0  
2 a1  
3 a1  
3 b 2
```

QUESTIONS:

1. MINIMIZE ANY GIVEN DFA.
2. EXPLAIN INTERMEDIATE CODE GENERATION.

EXPERIMENT #8

Aim: Implementation of shift reduce parsing algorithm.

Description:

Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root. In other words, it is a process of “reducing” (opposite of deriving a symbol using a production rule) a string w to the start symbol of a grammar.

ALGORITHM:

STEP1: Initial State: the stack consists of the single state, s_0 ; ip points to the first character in w .

STEP 2: For top-of-stack symbol, s , and next input symbol, a a case action of $T[s,a]$

STEP 3: **shift x :** (x is a STATE number) push a , then x on the top of the stack and advance ip to point to the next input symbol.

STEP 4: **reduce y :** (y is a PRODUCTION number) Assume that the production is of the form $A \Rightarrow \beta$ pop $2 * |\beta|$ symbols of the stack.

STEP 5: At this point the top of the stack should be a state number, say s' . push A , then goto of $T[s',A]$ (a state number) on the top of the stack.

PROGRAM CODE:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
void main()
{
    clrscr();
    puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
    puts("enter input string ");
    gets(a);
    c=strlen(a);
    strcpy(act,"SHIFT->");
    puts("stack \t input \t action");
    for(k=0,i=0; j<c; k++,i++,j++)
```

```
{
if(a[j]=='i' && a[j+1]=='d')
{
stk[i]=a[j];
stk[i+1]=a[j+1];
stk[i+2]='\0';
a[j]=' ';
a[j+1]=' ';
printf("\n%s\t%s\t%s\t%s",stk,a,act);
check();
}
else
{
stk[i]=a[j];
stk[i+1]='\0';
a[j]=' ';
printf("\n%s\t%s\t%s\t%symbols",stk,a,act);
check();
}
}
getch();
}
voidcheck()
{
strcpy(ac,"REDUCE TO E");
for(z=0; z<c; z++)
if(stk[z]=='i' && stk[z+1]=='d')
{
stk[z]='E';
stk[z+1]='\0';
printf("\n%s\t%s\t%s\t%s",stk,a,ac);
j++;
}
for(z=0; z<c; z++)
if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')
{
stk[z]='E';
stk[z+1]='\0';
stk[z+2]='\0';
printf("\n%s\t%s\t%s\t%s",stk,a,ac);
i=i-2;
}
for(z=0; z<c; z++)
if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E')
{
stk[z]='E';
stk[z+1]='\0';
stk[z+2]='\0';
printf("\n%s\t%s\t%s\t%s",stk,a,ac);
i=i-2;
}
for(z=0; z<c; z++)
if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')
{
stk[z]='E';
```


Department of Computer Science & Engineering

```
stk[z+1]='\0';
stk[z+1]='\0';
printf("\n%s\t%s\t%s",stk,a,ac);
i=i-2;
}
}
```

INPUT & OUTPUT:

GRAMMAR is E=E+E E=E*E E=(E) E=id input string is (id*id)+id		
stack	input	action
\$ (id*id)+id\$	SHIFT ->symbols
\$(id	*id)+id\$	SHIFT ->id
\$(E	*id)+id\$	REDUCE TO E
\$(E*	id)+id\$	SHIFT ->symbols
\$(E*id) +id\$	SHIFT ->id
\$(E*E) +id\$	REDUCE TO E
\$(E) +id\$	REDUCE TO E
\$(E)	+id\$	SHIFT ->symbols
\$E	+id\$	REDUCE TO E
\$E+	id\$	SHIFT ->symbols
\$E+id	\$	SHIFT ->id
\$E+E	\$	REDUCE TO E
\$E	\$	REDUCE TO E_

QUESTIONS:

1. DESCRIBE SHIFT REDUCE PARSING TECHNIQUES.
2. EXPLAIN 3 ADDRESS CODE GENERATOR.

EXPERIMENT#9

Aim: Write program to find Simulate First and Follow of any given grammar.

Description:

First and Follow sets are needed so that the parser can properly apply the needed production rule at the correct position.

PROGRAM CODE:

```
// 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 Inputgrammar
```

Department of Computer Science & Engineering

```
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)
    {
```

Department of Computer Science & Engineering

```
printf("%c, ", first[i]);
calc_first[point1][point2++] = first[i];
}
}
printf("}\n");
jm = n;
point1++;
}
printf("\n");
printf(".....\n\n");
chardonee[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 already 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;
}
}
```

Department of Computer Science & Engineering

```
}
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)) {
```

Department of Computer Science & Engineering

```
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;
}
}
```

Department of Computer Science & Engineering

```
//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++;
}
}
```

INPUT:

```
E -> TR
R -> +T R | #
T -> F Y
Y -> *F Y | #
    F -> (E) | i
```

OUTPUT:

```
First (E) = {(, i,}
First (R) = {+, #,}
First (T) = {(, i,}
First (Y) = {*, #,}
First (F) = {(, i,}
```

Department of Computer Science & Engineering

Follow (E) = { \$,), }

Follow (R) = { \$,), }

Follow (T) = { +, \$,), }

Follow (Y) = { +, \$,), }

Follow (F) = { *, +, \$,), }

QUESTIONS:

1. HOW TO CALCULATE FIRST AND FOLLOW OF ANY GIVEN GRAMMAR.
2. WRITE POST FIX NOTATION WITH EXAMPLE

EXPERIMENT #10

Aim: Construction of recursive descent parsing for the following grammar

```
E->TE'  
E'-> +TE' | -TE' | null  
T-> FT'  
T'-> *FT' | /FT' | null  
F-> id/ (E)/ num
```

Description:

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.

ALGORITHM:

Step1:start.

Step2:Declare the prototype functions E(),EP(),T(),TP(),F() Step3:

Read the string to be parsed.

Step 4: Check the productions

Step 5: Compare the terminals and Non-terminals Step

6: Read the parse string

Step 7: stop the production

PROGRAM CODE:

```
#include "stdio.h"  
#include "conio.h"  
char input[100];  
char prod[100][100];  
int pos=-1,l,st=-1;  
char id,num;  
void E();  
void T();  
void F();  
void advance();
```

Department of Computer Science & Engineering

```
void Td();
void Ed();
void advance()
{
    pos++;
    if(pos<1)
    {
        if(input[pos]>='0'&& input[pos]<='9')
        {
            num=input[pos];
            id='\0';
        }
        if((input[pos]>='a' || input[pos]>='A')&&(input[pos]<='z' || input[pos]<='Z'))
        { id=input[pos];
          num='\0';
        }
    }
}
void E()
{
    strcpy(prod[++st],"E->TE");
    T();
    Ed();
}
void Ed()
{
    int p=1;
    if(input[pos]=='+')
    {
        p=0;
        strcpy(prod[++st],"E'->+TE");
        advance();
        T();
        Ed();
    }
    if(input[pos]=='-')
    { p=0;
      strcpy(prod[++st],"E'->-TE");
      advance();
      T();
      Ed();
    }
}
// Recursive Descent Parser
if(p==1)
{
    strcpy(prod[++st],"E'->null");
}
}
void T()
{
```

Department of Computer Science & Engineering

```
strcpy(prod[++st],"T->FT");
F();
Td();
}
void Td()
{
int p=1;
if(input[pos]=='*')
{
p=0;
strcpy(prod[++st],"T'->*FT");
advance();
F();
Td();
}
if(input[pos]=='/')
{ p=0;
strcpy(prod[++st],"T'->/FT");
advance();
F();
Td();
}
if(p==1)
strcpy(prod[++st],"T'->null");
}
void F()
{
if(input[pos]==id) {
strcpy(prod[++st],"F->id");
advance();      }
if(input[pos]=='(')
{
strcpy(prod[++st],"F->(E)");
advance();
E();
if(input[pos]==')') {
//strcpy(prod[++st],"F->(E)");
advance();      }
}
if(input[pos]==num)
{
strcpy(prod[++st],"F->num");
advance();
}
}
int main()
{
int i;
printf("Enter Input String ");
```

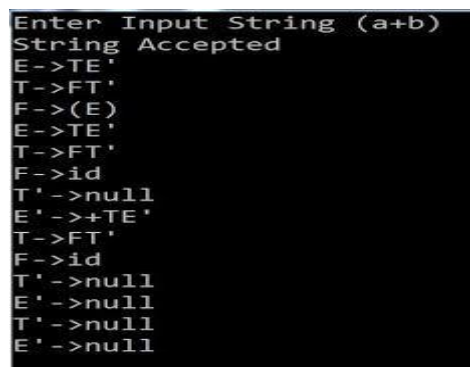
Department of Computer Science & Engineering

```
scanf("%s",input);
l=strlen(input);
input[l]='$';
advance();
E();
if(pos==l)
{
printf("String Accepted\n");
for(i=0;i<=st;i++)
{
printf("%s\n",prod[i]);
}
}
else
{
printf("String rejected\n");
}
getch();
return 0;
}
```

INPUT:

E->TE'
E'-> +TE' | -TE' | null
T-> FT'
T'-> *FT' | /FT' | null
F-> id/ (E)/ num

OUTPUT:



```
Enter Input String (a+b)
String Accepted
E->TE'
T->FT'
F->(E)
E->TE'
T->FT'
F->id
T'->>null
E'->+TE'
T->FT'
F->id
T'->>null
E'->>null
T'->>null
E'->>null
```

QUESTIONS:

1. DESCRIBE RECURSIVE DESCENT PARSER TECHNIQUES.
2. EXPLAIN TOP DOWN PARSER WITH EXAMPLE.

EXPERIMENT #11

Aim: - Write a C program to implement operator precedence parsing

Description:

An operator precedence grammar is a context-free grammar that has the property (among others) that no production has either an empty right-hand side or two adjacent nonterminals in its right-hand side. These properties allow precedence relations to be defined between the terminals of the grammar. A parser that exploits these relations is considerably simpler than more general-purpose parsers such as LALR parsers. Operator-precedence parsers can be constructed for a large class of context-free grammars.

ALGORITHM:

Step 1: Push # onto stack

Step 2: Read first input symbol and push it onto stack Step 3: Do

Obtain OP relation between the top terminal symbol on the stack and the next input symbol If the OP relation is < or =

- i. Pop top of the stack into handle, include non-terminal symbol if appropriate.
- ii. Obtain the relation between the top terminal symbol on the stack and the left most terminal symbol in the handle.
- iii. While the OP relation between terminal symbols is = o Do
Pop top terminal symbol and associated non-terminal symbol on stack into handle
- iv. Match the handle against the RHS of all productions v. Push N onto the stack Step 4: Until end-of-file and only # and N are on the stack.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main(){
/*OPERATOR PRECEDENCE PARSER*/
char stack[20],ip[20],opt[10][10][1],ter[10];
int i,j,k,n,top=0,col,row;

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

Department of Computer Science & Engineering

```
{
stack[i]=NULL;
ip[i]=NULL;
for(j=0;j<10;j++)
{
opt[i][j][1]=NULL;
}
}

printf("Enter the no.of terminals :\n");
scanf("%d",&n);
printf("\nEnter the terminals :\n");
scanf("%s",&ter);
printf("\nEnter the table values :\n");
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
printf("Enter the value for %c %c:",ter[i],ter[j]);
scanf("%s",opt[i][j]);
}
}

printf("\n**** OPERATOR PRECEDENCE TABLE ****\n");

for(i=0;i<n;i++)
{
printf("\t%c",ter[i]);
}
printf("\n");
for(i=0;i<n;i++){printf("\n%c",ter[i]);
for(j=0;j<n;j++){printf("\t%c",opt[i][j][0]);}}
stack[top]='$';
printf("\nEnter the input string:");
scanf("%s",ip);
```

Department of Computer Science & Engineering

```
i=0;

printf("\nSTACK\t\tINPUT STRING\t\tACTION\n");
printf("\n%s\t\t%s\t\t",stack,ip);
while(i<=strlen(ip))
{
for(k=0;k<n;k++)
{
if(stack[top]==ter[k])
col=k;
if(ip[i]==ter[k])
row=k;
}
if((stack[top]=='$')&&(ip[i]=='$')){
printf("String is accepted\n");
break;}
else if((opt[col][row][0]=='<') ||(opt[col][row][0]=='='))
{ stack[++top]=opt[col][row][0];
stack[++top]=ip[i];
printf("Shift %c",ip[i]);
i++;
}
else{
if(opt[col][row][0]=='>')
{
while(stack[top]!='<'){--top;}
top=top-1;
printf("Reduce");
}
else
{
printf("\nString is not accepted");
break;
}
}
```

```
}
printf("\n");
for(k=0;k<=top;k++)
{
printf("%c",stack[k]);
}
printf("\t\t\t");
for(k=i;k<strlen(ip);k++){
printf("%c",ip[k]);
}
printf("\t\t\t");
}
getch();
}
/*
```

INPUT & OUTPUT:

Enter the value for * *:>

Enter the value for * \$:>

Enter the value for \$ i:<

Enter the value for \$ +:<

Enter the value for \$ *:<

Enter the value for \$\$:accept

**** OPERATOR PRECEDENCE TABLE ****

	i	+	*	\$
i	e	>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	a
*/				

Enter the input string:

Department of Computer Science & Engineering

i*i

STACK	INPUT STRING	ACTION
\$	i*i	Shift i
\$<i	*i	Reduce
\$	*i	Shift *
\$<*	i	Shift i
\$<*<i		String is not accepted

QUESTIONS:

1. DESCRIBE OPERATER PRECEDENCE TECHNIQUE.
2. EXPLAIN AMBIGUOUS GRAMMAR

Department of Computer Science & Engineering

EXPERIMENT #12

Aim: Write a program to perform loop unrolling.

Description:

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

PREEXPERIMENT

Q1. Define loop unrolling.

Q2. Give example of loop unrolling.

ALGORITHM:

Step1: Start Step2: Declare n

Step3: Enter n value

Step4: Loop rolling display countbit1 or move to next step 5 Step5: Loop unrolling display countbit2

Step6: End

PROGRAM:

Program 1:

```
// This program does not uses loop unrolling.  
#include<stdio.h>
```

```
int main(void)  
{  
    for (int i=0; i<5; i++)  
        printf("Hello\n"); //print hello 5 times  
    return 0;  
}
```

Program 2:

Department of Computer Science & Engineering

```
// This program uses loop unrolling.
#include<stdio.h>
int main(void)
{
    // unrolled the for loop in program 1
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");
    return 0;
}
```

Output:

```
Hello
Hello
Hello
Hello
Hello
```

Illustration:

Program 2 is more efficient than program 1 because in program 1 there is a need to check the value of *i* and increment the value of *i* every time round the loop. So small loops like this or loops where there is fixed number of iterations are involved can be unrolled completely to reduce the loop overhead.

QUESTIONS:

1. DEFINE LOOP UNROLLING.
2. EXPLAIN RECURSION

Department of Computer Science & Engineering

EXPERIMENT #13

Aim: Write a Program to implement Code Optimization Techniques (eg. Constant propagation etc)

Description:

The algorithm we shall present basically tries to find for every statement in the program a mapping between variables, and values of $N \cup T \cup \perp$. If a variable is mapped to a constant number, that number is the variables value in that statement on every execution. If a variable is mapped to T (top), its value in the statement is not known to be constant, and in the variable is mapped to \perp (bottom), its value is not initialized on every execution, or the statement is unreachable. The algorithm for assigning the mappings to the statements is an iterative algorithm that traverses the control flow graph of the algorithm, and updates each mapping according to the mapping of the previous statement, and the functionality of the statement. The traversal is iterative, because non-trivial programs have circles in their control flow graphs, and it ends when a “fixed-point” is reached – i.e., further iterations don’t change the mappings.

ALGORITHM:

Step1: Start
Step2: Create an input file which contains three address code
Step3: Open the file in read mode
Step4: If the file pointer returns NULL, exit the program else goto 5
Step5: Scan the input symbol from left to right.
Common sub-expression elimination.
Step6: Store the first expression in a string.
Step7: Compare the string with the other expressions in the file.
Step8: If there is a match, remove the expression from the input file.
Step9: Perform the step 5 to 8 for all the input symbols in the file.
Dead Code Elimination
Step10: Scan the input symbol from the file from left to right.
Step11: Get the operand before the operator from the three address code.
Step12: Check whether the operand is used in any other expression in the three address code.
Step13: If the operand is not used, then eliminate the complete expression from the three address code else goto 14.
Step14: Perform step 11-13 for all the operands in the three address code till end of file is reached.
Step15: Stop.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct op{
char l;
```

Department of Computer Science & Engineering

```
char r[20];
}

op[10], pr[10];
voidmain()
{
int a,i,k,j,n,z=0,m,q;
char *p, *l;
char temp,t;
char *temp;
clrscr();
printf("Enter the no. of values");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\tleft\t");
op[i].l=getche();
printf("\tright:\t");
scanf("%s",op[i].r);
}
printf("intermediate code\n");
for(i=0;i<n;i++)
{
printf("%c=",op[i].l);
printf("%s\n",op[i].r);
}
for(i=0;i<n-1;i++)
{
temp=op[i].l;
for(j=0;j<n;j++)
{
p=(strchr(op[j].r,temp);
if (p)
{
pr[z].l=op[i].l;
strcpy(pr[z].r,op[i].r);
z++;
printf("\n after dead code elimination\n");
for(k=0;k<z;k++)
{
printf("%c\t=",pr[k].l);
printf("%s\n",pr[k].r);
}
// sub expression elimination
for(m=0;m<z; m++)
{
tem=pr[m].r;
for(j=m+1;j<z;j++)
{
```

Department of Computer Science & Engineering

p=strstr(tem,pr[j].r)

```
if (p)
{
t=pr[j].l;
pr[j].l=pr[m].l;
for(i=0;i<z;i++)
{
l=strchr(pr[i].r,t);
if (l){
a=l-pr[i].r;
//printf("pos:%d",a);
pr[i].r[a]=pr[m].l;}
}
}
}
printf("eliminate common sub expression\n");
for(i=0;i<z;i++)
printf("%c\t=",pr[i].l);
printf("%s\n",pr[i].r);
//duplicate production elimination
for(i=0;i<z;i++)
{
for(j=i+1;j<z;j++)
{
q=strcmp(pr[i].r,pr[j].r);
if((pr[i].l==pr[j].l)&&!q)
{
pr[i].l='\0';
strcpy(pr[i].r,'\0');
}
}
}
printf("optimized code");
for(i=0;i<z;i++)
{
if(pr[i].l!='\0'){
printf("%c=",pr[i].l);
printf("%s\n",pr[i].r);
}
}
getch();
}
```

OUTPUT:

```
lefta      right 9
leftb      rightc+d
leftc      rightc+d
leftd      rightb+e
leftf      right f
intermediate code
a=9
b=c+d
e=c+d
f=b+e
r=f
```

after dead code elimination

```
b =c+d
e =c+d
f =b+b
r =f
optimizedcode=c+d
f=b+b
r=f
```

QUESTIONS:

1. DESCRIBE CODE OPTIMIZATION TECHNIQUES.
2. WHICH PHASE IS OPTIONAL PHASE?

EXPERIMENT #14

Aim: Implement Intermediate code generation for simple expressions.

Description:

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
inti=1,j=0,no=0,tmpch=90;
char str[100],left[15],right[15];
void findopr();
void explore();
void fleft(int);
void fright(int);
structexp{
    int pos;
    charop;
}
k[15];
void main()
{
    clrscr();
    printf("\t\tINTERMEDIATE CODE GENERATION\n\n");
    printf("Enter the Expression :");
    scanf("%s",str);
    printf("The intermediate code: \t\tExpression\n");
    findopr();
    explore();
    getch();
}
void findopr()
{
    for(i=0;str[i]!='\0';i++)
        if(str[i]==':')
        {
            k[j].pos=i;
            k[j++].op=':';
        }
    for(i=0;str[i]!='\0';i++)
        if(str[i]=='/')
        {
            k[j].pos=i;
```

Department of Computer Science & Engineering

```
k[j++].op='/';
}

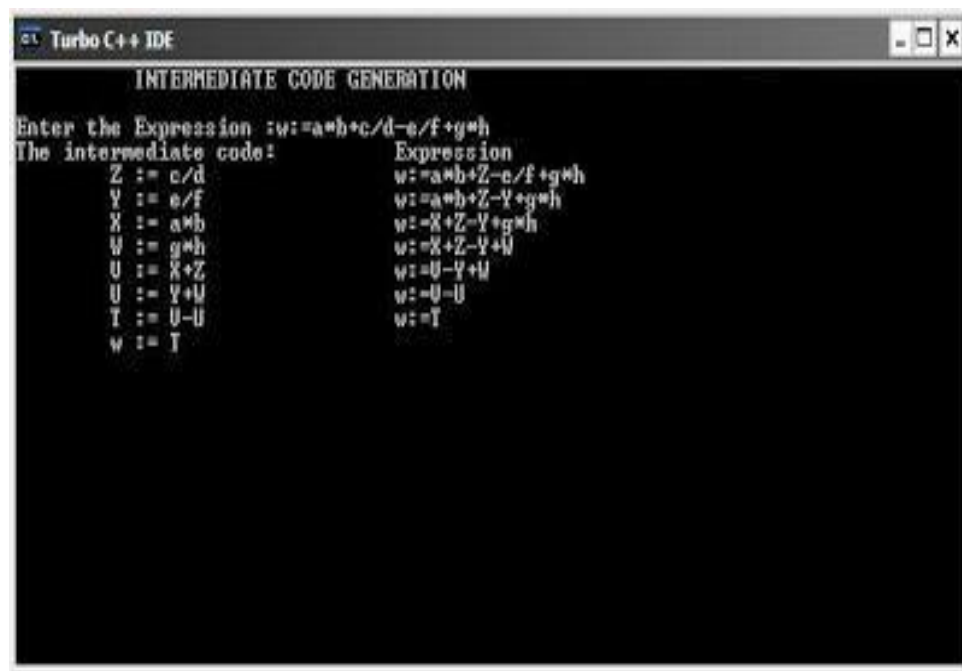
for(i=0;str[i]!='\0';i++)
    if(str[i]=='*')
    {
        k[j].pos=i;
        k[j++].op='*';
    }
    for(i=0;str[i]!='\0';i++)
        if(str[i]=='+')
        {
            k[j].pos=i;
            k[j++].op='+';
        }
        for(i=0;str[i]!='\0';i++)
            if(str[i]=='-')
            {
                k[j].pos=i;
                k[j++].op='-';
            }
    }
void explore()
{
    i=1;
    while(k[i].op!='\0')
    {
        fleft(k[i].pos);
        fright(k[i].pos);
        str[k[i].pos]=tmpch--;
        printf("\t%c := %s%c%s\t\t",str[k[i].pos],left,k[i].op,right);
    }
    for(j=0;j < strlen(str);j++)
        if(str[j]!='$')
            printf("%c",str[j]);
        printf("\n");
        i++;
    }
    fright(-1);
    if(no==0)
    {
        fleft(strlen(str));
        printf("\t%s := %s",right,left);
        getch();
        exit(0);
    }
    printf("\t%s := %c",right,str[k[--i].pos]);
    getch();
}

void fleft(int x)
{
    int w=0,flag=0;
    x--;
    while(x!= -1&&str[x]!='+'&&str[x]!='*'&&str[x]!='='&&str[x]!='\0'&&str[x]!='-'
    '&&str[x]!='/'&&str[x]!=':')
    {
        if(str[x]!='$'&&flag==0)
        {
            left[w++]=str[x];
            left[w]='\0';
            str[x]='$';
            flag=1;
        }
    }
}
```

```
    }
    x--;
  }
}

void fright(int x)
{
  int w=0,flag=0;
  x++;
  while(x!= -1&& str[x]!='+'&&str[x]!='*&&str[x]!='\o'&&str[x]!='='&&str[x]!=':'&&str[x]!='-'
'&&str[x]!='/')
  {
    if(str[x]!='$'&& flag==0)
    {
      right[w++]=str[x];
      right[w]='\o';
      str[x]='$';
      flag=1;
    }
    x++;
  }
}
```

Output



QUESTIONS:

1. EXPLAIN INTERMEDIATE CODE GENERATION.
2. EXPLAIN SEMANTIC ANALYZER.

EXPERIMENT # 15

Aim: Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly instructions can be simple move, add, sub, jumpetc.

ALGORITHM:

- i. Start the program.
- ii. Get the three variables from statements and stored in the text filek.txt
- iii. Compile the program and give the path of text file.
- iv. Execute the program
- v. Target code for the given statement was produced.
- vi. Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
char icode[10][30], str[20],opr[10];
int i=10;
clrscr();
printf("\n Enter the set of intermediate code(terminated by exit):\n");
do
{
scanf("%s", icode[i]);
}while(strcmp(icode[i++], "exit")!=0);
printf("\n target code generation");
printf("\n*****");
i=0;
do{
strcpy(str,icode[i]);
switch(str[3]){
case '+':
strcpy(opr,"ADD");
break;
case '-':
strcpy(opr,"SUB");
break;
case '*':
strcpy(opr, "MUL");
break;
```

Department of Computer Science & Engineering

```
case '/':
    strcpy(opr, "DIV");
    break;

}
printf("\n\tMov %c,R%d", str[2], i);
printf("\n\t%s%c,R%d", opr, str[4], i);
printf("\n\tMov R%d,%c", i, str[0]);
}while(strcmp(icode[++i], "exit")!=0);
getch();
}
```

OUTPUT:

Enter the set of intermediate code terminated by exit):

```
a=a*b
c=f*h
g=a*h
f=q+w
t=q-j
exit
target code generation
*****
Mov a,R0
Mov b,R0
Mov R0,a
Mov f,R1
Mul h,R1
Mov R1,c
Mov a,R2
Mul h,R2
Mov R2,g
Mov q,R3
ADD w,R3
Mov R3,f
Mov q,R4
SUB j,R4
```

QUESTIONS:

1. EXPLAIN 3 ADDRESS CODE.
2. DESCRIBE INDIRECT TRIPLE

EXPERIMENT #16

Aim: Write a C program for constructing of LL (1) parsing.

Description:

An LL parser is a top-down parser for a subset of context-free languages. It parses the input from Left to right, performing Left most derivation of the sentence. An LL parser is called an $LL(k)$ parser if it uses k tokens of look-ahead when parsing a sentence. If such a parser exists for a certain grammar and it can parse sentences of this grammar without backtracking then it is called an $LL(k)$ grammar. $LL(k)$ grammars can generate more languages the higher the number k of look ahead tokens.

ALGORITHM:

Step1: If $A \rightarrow \alpha$ is a production choice, and there is a derivation $\alpha \Rightarrow^* a\beta$, where a is a token, then we add $A \rightarrow \alpha$ to the table entry $M[A, a]$.

Step 2: If $A \rightarrow \alpha$ is a production choice, and there are derivations $\alpha \Rightarrow^* \varepsilon$ and $S \Rightarrow^* \beta A a \gamma$, where S is the start symbol and a is a token (or \$), the new add $A \rightarrow \alpha$ to the table entry $M[A, a]$

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char s[20],stack[20];
void main()
{
char m[5][6][3]={ "tb"," ","","tb"," "," "," ","+tb"," "," ","n","n","fc"," "," ","fc"," "," ","",
"n","*fc","a","n","n","i"," "," ","(e)"," "," "};
int size[5][6]={2,0,0,2,0,0,3,0,0,1,1,2,0,0,2,0,0,1,3,0,1,1,1,0,0,3,0,0};
int i,j,k,n,str1,str2;
clrscr();
printf("\n Enter the input string: ");
scanf("%s",s);
strcat(s,"$");
n=strlen(s);
stack[0]='$';
stack[1]='e';
i=1;
j=0;

printf("\nStack   Input\n");
```

Department of Computer Science & Engineering

```
printf("_____\n");
while((stack[i]!='$')&&(s[j]!='$'))
{
    if(stack[i]==s[j])
    {
        i--;
        j++;
    }
    switch(stack[i])
    {
        case 'e': str1=0;
        break;
        case 'b': str1=1;
        break;
        case 't': str1=2;
        break;
        case 'c': str1=3;
        break;
        case 'f':str1=4;
        break;
    }
    switch(s[j])
    {
        case 'i': str2=0;
        break;
        case '+': str2=1;
        break;
        case '*': str2=2;
        break;
        case '(':str2=3;
        break;
        case ')':str2=4;
        break;
        case '$': str2=5;
        break;
    }
    if(m[str1][str2][0]=='\0')
    {
        printf("\nERROR");
        exit(0);
    }
    else if(m[str1][str2][0]=='n')
        i--;
    else if(m[str1][str2][0]=='i')
        stack[i]='i';
    else
    {
        for(k=size[str1][str2]-1;k>=0;k--)
        {
            stack[i]=m[str1][str2][k];
        }
    }
}
```

Department of Computer Science & Engineering

```
i++;  
}  
i--;  
}  
for(k=0;k<=i;k++)  
printf(" %c",stack[k]);  
printf(" ");  
for(k=j;k<=n;k++)  
printf("%c",s[k]);  
printf(" \n");  
}  
printf("\n SUCCESS");  
getch();  
}
```

INPUT & OUTPUT:

Enter the input string:i*i+i

Stack	INPUT
\$bt	i*i+i\$
\$bcf	i*i+i\$
\$bci	i*i+i\$
\$bc	*i+i\$
\$bcf*	*i+i\$
\$bcf	i+i\$
\$bci	i+i\$
\$bc	+i\$
\$b	+i\$
\$bt+	+i\$
\$bt	i\$
\$bcf	i\$
\$bci	i\$
\$bc	\$
\$b	\$
\$	\$
sucess	

QUESTIONS:

1. TO CHECK THE GIVEN GRAMMAR IS LL (1) OR NOT.
2. EXPLAIN LR PARSING TECHNIQUES.

EXPERIMENT #17


Aim: - Write a program to Design LALR Bottom up Parser.

Description:

An LALR parser or Look-Ahead LR parser is a simplified version of a canonical LR parser, to parse (separate and analyze) a text according to a set of production rules specified by a formal grammar for a computer language. ("LR" means left-to-right, right most derivation.)

- Works on intermediate size of grammar
- Number of states are same as in SLR(1)

ALGORITHM:

STEP1: Represent by its CLOSURE, those items that are either the initial item [S'  do not have the . at the left end of the RHS

STEP2: Compute shift, reduce, and goto actions for the state derived from I directly from CLOSURE(I)

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
void push(char *,int *,char);
char stacktop(char *);
void isproduct(char,char);
int ister(char);
int isnter(char);
int isstate(char);
void error();
void isreduce(char,char);
char pop(char *,int *);
void printt(char *,int *,char [],int);
void rep(char [],int);
struct action
{
char row[6][5]; }
```

Department of Computer Science & Engineering

```
{ "sf", "emp", "emp", "se", "emp", "emp" },
{ "emp", "sg", "emp", "emp", "emp", "acc" },
{ "emp", "rc", "sh", "emp", "rc", "rc" },
{ "emp", "re", "re", "emp", "re", "re" },
{ "sf", "emp", "emp", "se", "emp", "emp" },
{ "emp", "rg", "rg", "emp", "rg", "rg" },
{ "sf", "emp", "emp", "se", "emp", "emp" },
{ "sf", "emp", "emp", "se", "emp", "emp" },
{ "emp", "sg", "emp", "emp", "sl", "emp" },
{ "emp", "rb", "sh", "emp", "rb", "rb" },
{ "emp", "rb", "rd", "emp", "rd", "rd" },
{ "emp", "rf", "rf", "emp", "rf", "rf" }
};

struct gotol
{
char r[3][4];
};

const struct gotol G[12]={
{ "b", "c", "d" },
{ "emp", "emp", "emp" },
{ "emp", "emp", "emp" },
{ "emp", "emp", "emp" },
{ "i", "c", "d" },
{ "emp", "emp", "emp" },
{ "emp", "j", "d" },
{ "emp", "emp", "k" },
{ "emp", "emp", "emp" },
{ "emp", "emp", "emp" },
};

char ter[6]={ 'i', '+', '*', ')', '(', '$' };
char nter[3]={ 'E', 'T', 'F' };
char states[12]={ 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'm', 'j', 'k', 'l' };
char stack[100];
int top=-1;
char temp[10];
struct grammar
{
char left; char right[5]; };
};
```

Department of Computer Science & Engineering

Construct the grammar:

```
{ 'E', "e+T" },  
{ 'E', "T" },  
{ 'T', "T*F" },  
{ 'T', "F" },  
{ 'F', "(E)" },
```

```
{ 'F', "i" },  
};  
void main()  
{  
char inp[80],x,p,dl[80],y,bl='a';  
int i=0,j,k,l,n,m,c,len;  
clrscr();  
printf(" Enter the input :");  
scanf("%s",inp);  
len=strlen(inp);  
inp[len]='$';  
inp[len+1]='\0';  
push(stack,&top,bl);  
printf("\n stack \t\t\t input");  
printt(stack,&top,inp,i);  
do  
{  
x=inp[i];  
p=stacktop(stack);  
isproduct(x,p);  
if(strcmp(temp,"emp")==0)  
error();  
if(strcmp(temp,"acc")==0)  
break;  
else  
{  
if(temp[0]=='s')  
{  
push(stack,&top,inp[i]);  
push(stack,&top,temp[1]);  
i++;  
}  
else  
{  
if(temp[0]=='r')  
{  
j=isstate(temp[1]);  
strcpy(temp,rl[j-2].right);  
dl[0]=rl[j-2].left;  
dl[1]='\0';  
n=strlen(temp);
```

```
pop(stack,&top);
    for(m=0;dl[m]!='\0';m++)
push(stack,&top,dl[m]);
l=top;
y=stack[l-1];
isreduce(y,dl[0]);
for(m=0;temp[m]!='\0';m++)
push(stack,&top,temp[m]);

}
}
}
printt(stack,&top,inp,i);
}while(inp[i]!='\0');
if(strcmp(temp,"acc")==0)
printf(" \n accept the input ");
else
printf(" \n do not accept the input ");
getch();
}
void push(char *s,int *sp,char item)
{
if(*sp==100)
printf(" stack is full ");
else
{
*sp=*sp+1;
[*sp]=item;
}
}
char stacktop(char *s)
{
char i;
i=s[top];
return i;
}
void isproduct(char x,char p)
{
int k,l;
k=ister(x);
l=isstate(p);

strcpy(temp,A[l-1].row[k-1]);
}
int ister(char x)
{
int i;
for(i=0;i<6;i++)
if(x==ter[i])
```

```
return i+1;
return 0;

}
int isnter(char x)
{

    int i;
    for(i=0;i<3;i++)
        if(x==nter[i])
            return i+1;
    return 0;

}
int isstate(char p)
{
    int i;
    for(i=0;i<12;i++)
        if(p==states[i])
            return i+1;
    return 0;
}
void error()
{
    printf(" error in the input ");
    exit(0);
}
void isreduce(char x,char p)
{
    int k,l;
    k=isstate(x);
    l=isnter(p);
    strcpy(temp,G[k-1].r[l-1]);
}
char pop(char *s,int *sp)
{
    char item;
    if(*sp==-1)
        printf(" stack is empty ");
    else
    {
        item=s[*sp];
        *sp=*sp-1;
    }
    return item;
}
void printt(char *t,int *p,char inp[],int i)
{
    int r;
```

```
case 'a': printf("0");
break;
case 'b': printf("1");
break;
case 'c': printf("2");
break;
case 'd': printf("3");
break;
case 'e': printf("4");
break;
case 'f': printf("5");
break;
case 'g': printf("6");
break;
case 'h': printf("7");
break;
case 'm': printf("8");
break;
case 'j': printf("9");
break;
case 'k': printf("10");
break;
case 'l': printf("11");
break;
default :printf("%c",t[r]);
break;
}
}
```

INPUT & OUTPUT:



QUESTIONS:

1. DESCRIBE CLR PARSING TECHNIQUES.
2. EXPLAIN LALR PARSING TECHNIQUES.

REFERENCES:

- (1) Aho,Ullman,Sethi-----CompilerConstruction.
- (2) AllenI.Holub-----CompilerConstruction.

APPENDIX-I:

SYLLABUS (AS PER AKTU)

1. Design and implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and newlines.
2. To write a c program for implementing a lexical analyzer using LEX tool
3. Generate YACC specification for a few syntactic categories:
 - a) Program to recognize a valid arithmetic expression that uses operator +, −, * and/.
 - b) Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.
 - c) Implementation of Calculator using LEX and YACC
 - d) Convert the BNF rules into YACC form and write code to generate abstract syntax tree
4. Write program to find ϵ – closure of all states of any given NFA with ϵ transition.
5. Write program to convert NFA with ϵ transition to NFA without ϵ transition.
6. Write program to convert NFA to DFA
7. Write program to minimize any given DFA.
8. Implementation of shift reduce parsing algorithm.
9. Write program to find Simulate First and Follow of any given grammar.
10. Construction of recursive descent parsing for the following grammar
11. Write a C program to implement operator precedence parsing
12. Write a program to perform loop unrolling.
13. Write a Program to implement Code Optimization Techniques (eg. Constant propagation etc.)
14. Implement Intermediate code generation for simple expressions.
15. Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly instructions can be simple move, add, sub, jump etc.
16. Write a C program for constructing of LL (1) Parsing.
17. Write a program to Design LALR Bottom up Parse.

