COMPILER DESIGN

LAB MANUAL

Subject Code:	**20C07301**
Regulation:	**R20**
Class:	**III Year II Semester (CSE)**

Department of Computer Science & Engineering

QIS COLLEGE OF ENGINEERING AND TECHNOLOGY

Vegamukkapalem- **523272**, Ongole, Andhra Pradesh

(AUTONOMOUS)

QIS COLLEGE OF ENGINEERING AND TECHNOLOGY
Vegamukkapalem- **523272**, Ongole, Andhra Pradesh
(AUTONOMOUS)

COMPUTER SCIENCE AND ENGINEERING

Vision:

To facilitate transformation of students into highly skilled, knowledgeable and competent software professional focusing on ethical values and societal commitment.

Mission:

- To impart quality education to meet the needs of the industry and research in the field of computer science and engineering.
- To encourage an environment conductive to innovation, creativity, team spirit and entrepreneurial leadership in Computer Science and Engineering.
- To foster networking with Alumni, Software Industries, Institutes and other stakeholders for effective interaction.
- To practice and promote high standards of ethical values through societal commitment.

PEO's (PROGRAMME EDUCATIONAL OBJECTIVES):

1. PEO1. Graduates will have solid foundation in fundamentals of computer science and engineering required to solve computing problems and create innovative software products and solutions for the real life problems.
2. PEO2. Graduates will have technical competence and skills to use modern and cost-effective tools and technologies and have extensive and effective practical skills in computer science and engineering to pursue a career as a computer engineer.
3. PEO3. Graduates will have attributes like professionals with world class academic excellence, ethics, best practices, values, social concerns, lifelong learning and openness to other international cultures to meet the global needs.
4. PEO4. Graduates will have managerial and entrepreneur skills with cross-cultural etiquettes, leading to a sustainable competitive edge in R&D and meeting societal needs.

Program Outcomes	
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 engineeringproblems 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 methodsincluding 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 insocietal 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 multidisciplinary settings.
PO10	**Communication**: Communicate effectively on complex engineering activities with the engineeringcommunity 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	
PSO1	**Domain Knowledge:** An ability to understand, analyze and develop computer programs in the areas related to Algorithms, System Software, Multimedia, Web Design, Big Data and Analytics and Networking for efficient design of computer based systems of varying complexity to meet the need of the software industry.
PSO2	**Process Management:** An Ability to organize and apply standard practices and strategies in software product development by managing and monitoring the resources and safeguarding the data.

COMPILER DESIGN LAB SYLLABUS

Sl. No.	List of Experiments
1.	Write a C Program for dividing the input program into lexemes.
2.	Write a C Program to Simulate FIRST and FOLLOW of grammar.
3.	Write a C program for implementing the operator precedence.
4.	Design a lexical analyzer for the given language. The lexical analyzer should ignore redundant spaces, tabs and new lines, comments etc.
5.	Write a C Program to Implement a Recursive Descent Parser.
6.	Write a C program to construct LL (1) parser.
7.	Write a C Program to Implement a Predictive Parser.
8.	Write a C Program to construct shift- reduce parser.
9.	Write a C program to generate three address code.
10.	Design a LALR bottom up parser for the given language.
11.	Convert the BNF rules into YACC form and write code to generate abstract syntax tree.
12.	A program to generate machine code from the abstract syntax tree generated by the parser.
	Additional Experiments
1.	a) Program to convert the infix to prefix expression. b) Program to convert the infix to postfix expression.
2.	Program to implement code optimization.

ATTAINMENT OF PROGRAM OUTCOMES& PROGRAM SPECIFIC OUTCOMES

Sl. No	NAME OF THE EXPERIMENT	Program Outcomes Attained	Program Specific Outcomes Attained
1	Write a C Program for dividing the input program into lexemes.	PO1, PO2, PO3	PSO1
2	Write a C Program to Simulate FIRST and FOLLOW of grammar.	PO1, PO2, PO3,PO4	PSO1, PSO2
3	Write a C program for implementing the operator precedence.	PO1, PO2	PSO1, PSO2
4	Design a lexical analyzer for the given language. The lexical analyzer should ignore redundant spaces, tabs and new lines, comments etc.	PO1, PO2	PSO1
5	Write a C Program to Implement a Recursive Descent Parser.	PO1, PO2, PO3,PO4	PSO1, PSO2
6	Write a C program to construct LL (1) parser.	PO1, PO2, PO3,PO4	PSO1, PSO2
7	Write a C Program to Implement a Predictive Parser	PO1, PO2, PO3,PO4	PSO1, PSO2
8	Write a C Program to construct shift- reduce parser.	PO1, PO2	PSO1, PSO2
9	Write a C program to generate three address code.	PO1, PO2, PO4	PSO1
10	Design a LALR bottom up parser for the given language	PO1, PO2, PO4	PSO1
11	Convert the BNF rules into YACC form and write code to generate abstract syntax tree.	PO1, PO2	PSO1
12	A program to generate machine code from the abstract syntax tree generated by the parser.	PO1, PO2, PO3,PO4, PO5	PSO1
	Additional Experiments		
1	a) Program to convert the infix to prefix expression. b) Program to convert the infix to postfix expression.	PO1, PO2	PSO1
2	Program to implement code optimization.	PO1, PO2	PSO1

COMPILER DESIGN LAB

OBJECTIVE:

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

OUTCOMES:

Upon the completion of Compiler Design practical course, the student will be able to:

1. Acquire knowledge in different phases and passes of Compiler, and specifying different types of tokens by lexical analyzer, and also able to use the Compiler tools like LEX, YACC, etc.
2. Parser and its types i.e. Top-down and Bottom-up parsers.
3. Construction of LL, SLR, CLR and LALR parse table.
4. Syntax directed translation, synthesized and inherited attributes.
5. Analyze the Code generation Techniques and Symbol Table.
6. Techniques for code optimization.

Experiment No: 1

Name of the Experiment: Write a C Program for dividing the input program into lexemes.

Aim: To write a program for dividing the giving input program into lexems.

Description:

Lexical Analyser is the first phase of a compiler. Its main task is to read the input characters and to produce output as a sequence of tokens that parse uses for syntax analysis.

The lexical analyser may also perform certain secondary tasks that of user interface. Once such task is stripping out from the source comments and white spaces in the form of the blank tabs and new line and characters. It correlates the error messages from the compiler with the source program.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
FILE *fp;
Static char ch;
char * keyword[20]={“void”,”main”,”int”,”float”};
char brace[20]={‘(‘,’)’,’{‘,’}’,’[‘,’]’};
char operator[20]={‘+’,’-‘,’*’,’/’,’%’,’<’,’>’,’=’};
char symbol[20]={‘;’,’/’,’”’,’:’,’,’};
char identifier[32];
char number[3];
int i;
main()
{
  fp=fopen(“prg.c”,”w”);
  clrscr();
  printf(“Write the input program:\n”);
  while(((ch=getchar()!=’@’))
  {
     putc(ch,fp);
  }
 fclose(fp);
 fp=fopen(“prg.c”,”r”);
 ch=getc(fp);
 printf(“\n -------------\n”);
 printf(“tokens                    lexemes \n”);
 printf(“----------------\n”);
 while(!feof(fp))
 {
   scantoken();
 }
 fclose(fp);
 getch();
}
int scantoken()
{
   int j;
   for(i=0;i<5;i++)
   if(strcmp(ch,brace[i])==0)
   {
     printf("paranthesis ------------->%c \n",ch);
     ch=getc(fp);
   }
   for(i=0;i<8;i++)
   if(strcmp(ch,operator[i])==0)
   {
     printf("operator--------------->%c \n",ch);
     ch=getc(fp);
   }
   for(i=0;i<5;i++)
   if(strcmp(ch,symbol[i])==0)
   {
     printf("symbol-------------- >%c \n",ch);
     ch=getc(fp);
   }
   if(isalpha(ch))
   {
     identifier[0]=ch;
     identifier[1]='\0';
     j=1;
     ch=getc(fp);
     while(isalnum(ch))
     {
     identifier[j++]=ch;
     ch=getc(fp);
     }
     identifier[j]='\0';
     if(chkkey()==1)
     printf("keyword -------------- >%s \n",identifier);
     else
     printf("identifier---------------->%s \n",identifier);
  }
  else if(isdigit(ch))
   {
     number[0]=ch;
     j=1;
     ch=getc(fp);
   }
   identifier[j]='\0';
   printf("number--------------- >%s \n",number);
  }
  else if(isspace(ch))
         ch=getc(fp);
}
```

```
int chkkey()
{
   int flag=0;
   for(i=0;i<3;i++)
   if(strcmp(keyword[i],identifier)==0)
   {
     flag=1;
     break;
   }
  return flag;
}
```

Input:

Write the input program

```
void main()
{
  int a,b;
  a=a+b;
}
@
```

Output:

```
--------------------------------
tokens                          lexems

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

keyword--------------------------->void

keyword------------------------->main

paranthesis------------->(

paranthesis------------->)

paranthesis------------->{

keyword------------------------->int

identifier--------------->a

symbol----------------->,

identifier---------------> b

symbol----------------->;

identifier--------------->a

operator------------------------>=
identifier--------------->a

operator------------------------>+

identifier--------------->b

symbol----------------->;

paranthesis------------->}
```

Result: The program is successfully executed.

Lab Viva Questions:

1. What is token?
2. What is lexeme?
3. What is the difference between token and lexeme?
4. Define phase and pass?
5. What is the difference between phase and pass?
6. What is the difference between compiler and interpreter?

Experiment No: 2

Name of the Experiment: Write a C Program to Simulate FIRST and FOLLOW of grammar.

Aim: To calculate the first and follow of the given grammar.

Description: The first and follow functions allow us to fill the entries of a predictive parsing table.

Program:

```
 #include<stdio.h>
#include<conio.h>
#include<string.h>
int n,m=0,p,i=0,j=0;
char a[10][10],f[10];
void follow(char c);
void first(char c);
int main()
{
   int i,z;
char c,ch;
clrscr();
printf("Enter the no.of productions:");
scanf("%d",&n);
printf("Rnter productions:\n");
for(i=0;i<n;i++)
sccanf("%S%c",a[i],&ch);
do
{
  m=0;
printf("Enter the elements whose first & follow is to be found:");
scanf("%c",&c);
first(c );
printf("first(%c)={"c);
for(i=0;i<m;i++)
{
   if(i<m-1)
printf("%c",f[i]);
else
printf("%c",f[i]);
}
printf("}\n");
strcpy(f," ");
flushall();
m=0;follow();
printf("follow(%c)={",c);
for(i=0;i<m;i++)
{
   if(i<m-1)
printf(“%c”,f[i]);
else
printf(“%c”,f[i]);
}
printf(“}\n”);
printf(“continue (0/1)”);
scanf(“%d%c”,&z,&ch);
}
while(z==1)
return(0)
}
void first(char c)
{
  int k;
if(!isupper(c))
f[m++]=c;
for(k=0;k<n;k++)
{
  if(a[k][0]==c)
 {
    if(a[k][2]=’$’)
  follow(a[k][0]);
 else if(islower(a[k][2]))
   f[m+1]=a[k][2];
else first(a[k][2]);
  }
 }
}
void  follow( char c)
{
 if(a[0][0]==c)
f[m+1]=’$’;
for(i=0;i<n;i++)
{
  for(j=0;j<strlen(a[i]);j++)
{
  if(a[i][j]==c)
{
  if(a[i][j+1]!=’\0’)
    first(a[i][j+1]);
if(a[i][j+1]==’\0’ &&c!=a[i][0])
   follow(a[i][0]);
       }
    }
   }
}
```

Output:

```
 Enter the no.of productions:
S->aSb
S->Aa
A->bc
Enter the elements whose first and follow is to be found:S
First(S): {a, b}
Follow(S): {$, b}
Continue (0/1):0
```

Result: The program executed successfully.

Lab Viva Questions:

1. What is top-down parsing?
2. What are the disadvantages of brute force method?
3. What is context free grammar?
4. What is parse tree?
5. What is ambiguous grammar?
6. What are the derivation methods to generate a string for the given grammar?
7. What is the output of parse tree?

Experiment No: 3

Name of the Experiment: Write a C program for implementing the operator precedence

Aim: To write a Program for implementing Operator Precedence.

Description: The precedence is used to determine how an expression involving more than one operator is evaluated. An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operators.

Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
  char stack[20],ip[20],opt[10][10][1] ,ter[10];
  int i,j,k,n,top=0;
  clrscr();
for(i=0;i<10;i++)
{
  stack[i]=NULL;
  ip[i]=NULL;
  for(j=0;j<10;j++)
{
    opt[i][j][1]=NULL;
}
printf("Enter no.of terminals:");
scanf("%d",&n);
printf("\nEnter the terminals");
scanf("%s",ter);
printf("\nEnter the table values");
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("\t%c",ter[i]);
  for(j=0;j<n;j++)
  printf("\t%c",opt[i][j][0]);
}
getch();
}
```

Output:

```
Enter no.of terminals: 3
Enter the terminals +*/
Enter the table values:
Enter the value for + +:>
Enter the value for + *:<
Enter the value for + /:<
Enter the value for * + :>
Enter the value for * *:>
Enter the value for */:>
Enter the value for / +:>
Enter the value for / *:>
Enter the value for / /: >
Operator precedence table:
     +    *    /
+    >    <    <

*    >    >    >

/    >    >    >
```

Result: The program executed successfully.

Lab Viva Questions:

1. What is top-down parsing?
2. What are the disadvantages of brute force method?
3. What is context free grammar?
4. What is parse tree?
5. What is ambiguous grammar?
6. What are the derivation methods to generate a string for the given grammar?
7. What is the output of parse tree?

Experiment No: 4

Name of the Experiment: Design a lexical analyzer for the given language. The lexical analyzer should ignore redundant spaces, tabs and new lines, comments etc.

Aim: To implement lexical analyzer for the given language.

Description: Lexical analyser will scan the i/p statement and identifies the lexems and tokens.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
{
        char exp[20],id[10],dig[10],ch;
        int i,j;
        clrscr();
        printf("enter expression:");
        scanf("%s",&exp);
void main()

        for(i=0;i<strlen(exp);)
        {
                ch=exp[i];
                j=0;
                if(ch>='a'&&ch<='z')
                {
                        id[j++]=ch;
                        i++;
                        while((exp[i]>='a'&&exp[i]<='z')||(exp[i]>='0'&&exp[i]<='9'))
                        {
                                id[j++]=exp[i++];
                        }
                                id[j]='\0';

                        printf("\nidentifier:%s",id);
                }
                else if(ch=='+'||ch=='-'||ch=='*'||ch=='/'||ch=='%'||ch=='=')
                {
                        printf("\noperator:%c",ch);
                        i++;
                }
                else if(ch>='0'&&ch<='9')
                {
                        dig[j++]=ch;
                        i++;
                        while(exp[i]>='0'&&exp[i]<='9')
                        dig[j++]=exp[i++];
                        dig[j]='\0';
                        printf("\nconstant:%s",dig);
                }
        }//for
        getch();
}
```

Output:

```
[qiscse@localhost ~]$  gcc  lexical.c
[qiscse@localhost ~]$ ./a.out
Enter the expression: a=b+c
identifier : a
operator: =
identifier : b
operator:  +
identifier : c
```

Result: The program executed successfully.

Lab Viva Questions:

1. List the different sections available in LEX compiler?
2. What is an auxiliary definition?
3. How can we define the translation rules?
4. What is regular expression?
5. What is finite automaton?

Experiment No: 5

Name of the Experiment: Write a C Program to Implement a Recursive Descent Parser.

Aim: To write a program to implement recursive descent parsing for the grammar
S->aAd
A->ab/aA/null

Description:

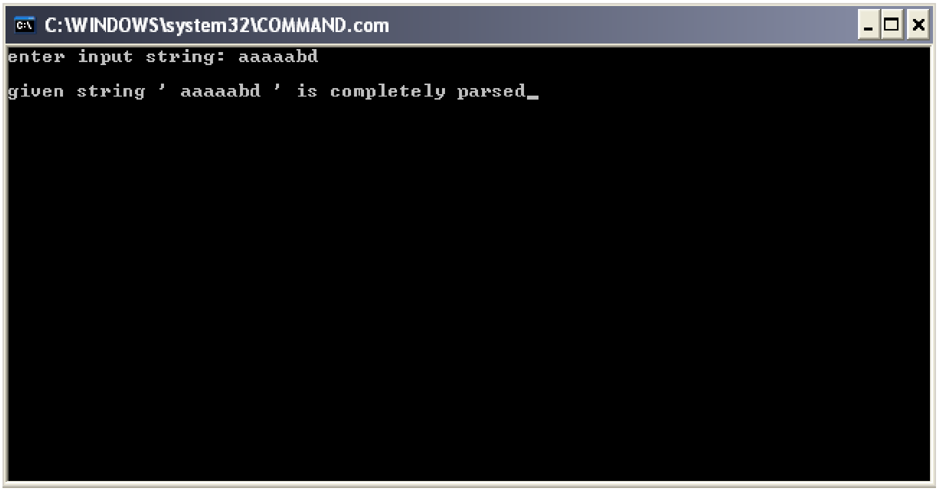
Recursive descent parsing is one of the top down parsing methods. This allows backtracking technique. First we continue with first production if that the parsed production doesn't satisfy the string. Then we again go back and store with some other new production. This technique is called "**backtracking**".

Program:

```
/*recursive descent parsing for the grammar
  S->aAd
  A->ab/aA/null*/

 #include<stdio.h>
 #include<conio.h>
 char str[20];
 int flag=1;
 int i=0;
 void main()
 {
  int S();
  clrscr();
  printf("enter input string: ");
  gets(str);
  S();
  if(flag==1&&str[i]=='\0')
   printf("\ngiven string ' %s ' is completely parsed",str);
  else
   printf("sorry! not parsed");
  getch();
 }
 int S()
 {
  int A();
  if(str[i]=='a')
  {
   i++;
   A();
   if(str[i]=='d')
   {
    i++;
    return 0;
   }
   else
   {
    flag=0;
    return 0;
   }
  }
 }
int A()
{
 if(str[i]=='a')
 {
  i++;
  if(str[i]=='b')
  {
   i++;
   return 0;
  }
  else
  {
   A();
   return 0;
  }
 }
 else
 {
 // flag=1;
  return 0  }}
```

<u>**Output:**</u>



C:\WINDOWS\system32\COMMAND.com

```
enter input string: aaaaabd

given string ' aaaaabd ' is completely parsed_
```

Result: Recursive Descent parsing for the given grammar is implemented.

Lab Viva Questions:

1. What are the problems in top down parsing?
2. Define recursive-descent parser?
3. Define predictive parsers?
4. What are the problems in top down parsing?

Experiment No: 6

Name of the Experiment: Write a C program to construct LL (1) parser.

Aim: To construct LL(1) parser for a given expression.

Description: It is a top down parser. It is simple to implement. For LL(1) the first L means the input scanned from left to right. The second L means it uses left most derivation for the input string. The number 1 indicates that one look ahead symbol to predict the parsing.

Program:

```
#include <conio.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void main()
{
int i=0,j=0,l1,k=0,m=0,len,n=0,o=0,o1=0,var=0,f=0,c=0,f1=0;
char str[30],str1[40]="E",temp[20],temp1[20],temp2[20],tt[20],t3[20];
char t[10];
char array[6][5][10]= {
"NT", "<id>","+","*","$",
"E", "Te","Error","Error","Error",
"e", "Error","+Te","Error","\0",
"T", "Vt","Error","Error","Error",
"t", "Error","\0","*Vt","\0",
"V", "<id>","Error","Error","Error"
};
clrscr();
strcpy(temp1,'\0');
strcpy(temp2,'\0');
printf( "\n\tLL(1)  PARSER  TABLE \n");
for(i=0;i<6;i++)
{
for(j=0;j<5;j++)
{

printf("%s\t",array[i][j]);
}
printf("\n");
}
printf("\n");
rep:
printf("\n\tENTER THE STRING :");
gets(str);
if(str[strlen(str)-1] != '$')
{
printf("END OF STRING MARKER SHOULD BE '$'");
getch();
```

```
goto rep;
}
printf("\n\tCHECKING VALIDATION OF THE STRING ");
printf("\n\t%s", str1);
i=0;
while(i<strlen(str))
{
again:
if(str[i] == ' ' && i<strlen(str))
{
printf("\n\tSPACES IS NOT ALLOWED IN SOURSE STRING ");
getch();
exit(1);
}
temp[k]=str[i];
temp[k+1]='\0';
f1=0;
again1:
if(i>=strlen(str))
{
getch();
exit(1);
}
for(l1=1;l1<=4;l1++)
{
if(strcmp(temp,array[0][l1])==0)
{
f1=1;
m=0,o=0,var=0,o1=0;
strcpy(temp1,'\0');
strcpy(temp2,'\0');
//int len=strlen(str1);
while(m<strlen(str1) && m<strlen(str))
{
if(str1[m]==str[m])
{
var=m+1;
temp2[o1]=str1[m];
m++;
o1++;
}
else
{
if((m+1)<strlen(str1))
{
m++;
temp1[o]=str1[m];
o++;
}
else
m++;
}

}
temp2[o1] = '\0';
temp1[o] = '\0';
t[0] = str1[var];
t[1] = '\0';
for(n=1;n<=5;n++)
{
if(strcmp(array[n][0],t)==0)
break;
}
strcpy(str1,temp2);
strcat(str1,array[n][l1]);
strcat(str1,temp1);
printf("\n\t%s",str1);
getch();

if(strcmp(array[n][l1],'\0')==0)
{
if(i==(strlen(str)-1))
{ len=strlen(str1);
str1[len-1]='\0';
printf("\n\t%s",str1);
printf("\n\n\tENTERED STRING IS VALID");
getch();
exit(1);
}
strcpy(temp1,'\0');
strcpy(temp2,'\0');
strcpy(t,'\0');
goto again1;
}
if(strcmp(array[n][l1],"Error")==0)
{
printf("\n\tERROR IN YOUR SOURCE STRING");
getch();
exit(1);
}
strcpy(tt,'\0');
strcpy(tt,array[n][l1]);
strcpy(t3,'\0');
f=0;
for(c=0;c<strlen(tt);c++)
{
t3[c]=tt[c];
t3[c+1]='\0';
```

```
if(strcmp(t3,temp)==0)

{
f=0;
break;
}
else
f=1;
}

if(f==0)
{
strcpy(temp,'\0');
strcpy(temp1,'\0');
strcpy(temp2,'\0');
strcpy(t,'\0');
i++;
k=0;
goto again;
}
else
{
strcpy(temp1,'\0');
strcpy(temp2,'\0');
strcpy(t,'\0');
goto again1;
}
}
}
i++;
k++;
}
if(f1==0)
printf("\nENTERED STRING IS INVALID");
else
printf("\n\n\tENTERED STRING IS VALID");
getch();
}
```

Output:

```
NT      <id>    +       *       $
E       Te      Error  Error   Error
e       Error  +Te     Error
T       Vt      Error  Error   Error
t       Error          *Vt
V       <id>    Error   Error   Error
```

```
 ENTER THE STRING :<id>+<id>*<id>$


CHECKING VALIDATION OF THE STRING
        E
        Te
        Vte
        <id>te
        <id>e
        <id>+Te
        <id>+Vte
        <id>+<id>te
        <id>+<id>*Vte
        <id>+<id>*<id>te
        <id>+<id>*<id>e
        <id>+<id>*<id>

        ENTERED STRING IS VALID
```

Result: The program executed successfully.

Lab Viva Questions:

1. Define parse trees?
2. Define context-free grammar?
3. What is LL (1) grammar?
4. What is meant by left recursion?
5. Define context-free language?

Experiment No: 7

Name of the Experiment: Write a C Program to Implement a Predictive Parser

Aim: To write a program for implementing the predictive parsing

Description:

This is one of the methods of top down parsing. We can call this as recursive descent parsing with no back tracking for this predictive parsing algorithm, the grammar that we are giving as input should be a non-left recursive and left factored. In this algorithm, we write procedure from each nonterminal and then start matching process which is nothing but parsing.

We compare each and every symbol of the input starting with the given production of the grammar. Whenever a match is found, we increment look ahead pointer, a pointer which points to input symbol is being scanned.

Program:

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
char table[10][10][10],nt[10],t[10],inp[20],stack[20];
int nut,nun,i=0,top=0;
int getnt(char);
int gett(char);
void replace(char,char);
void  main()
{
   int i,j;
   clrscr();
   printf("Enter no.of terminals:\n");
   scanf("%d",&nut);
  printf("Enter no.of non terminals:\n");
  scanf("%d",&nun);
  printf("Enter all nonterminals:\n");
  scanf("%s",&nt);
  printf("Enter all terminals:\n");
  scanf("%s",&t);
  printf("\n");
  printf("\n");
  for(i=0;i<nun;i++)
  for(j=0;j<nut;j++)
  {
    printf("Enter for %c and %c:",nt[i],t[j]);
    scanf("%s",table[i][j]);
  }
   printf("table is \n \n");
   for(j=0;j<nut; j++)
   printf("\t %c", t[j]);
   printf("\n");

for(i=0; i<nun;i++)
```

```
 {
   printf("%c \t", nt[i]);
   for(j=0;j<nut;j++)
   {
      printf("%s \t",table[i][j]);
   }
   printf("\n");
   getch();
  }
  printf("Enter string to be parsed:\n");
  scanf("%s",inp);
  stack[top++]='$';
  stack[top++]=nt[0];
  i=0;
  while(1)
  {
    if((stack[top-1]=='$')&&(inp[i]=='$'))
    {
      printf("str accepted \n");
      return;
    }
    else if(!isupper(stack[top-1]))
            {
               if(stack[top-1]==inp[i])
               {
                 i++;
                 top--;
               }
               else
               {
                 printf("\t ERROR !!\n Not accepted \n");
                 break;
               }
           }
           else
           {
             replace(stack[top-1],inp[i]);
           }
    }
   getch();
}
int getnt(char x)
{
   int a;
   for(a=0;a<nun;a++)
   if(x==nt[a])
   return a;
   return 100;
}
int gett(char x)
```

```
{
   int a;
   for(a=0;a<nut;a++)
   if(x==t[a])
   return a;
   return 100;
}
void replace(char NT,char T)
{
   int in1,it1,len;
   char str[10];
   in1=getnt(NT);
   it1=gett(T);
   if((in1!=100)&&(it1!=100))
   {
     strcpy(str,table[in1][it1]);
     if(strcmp(str,”#”)==0)
     {
       printf(“\t Error \n”);
       return;
    }
    else if(strcmp(str,”@”)==0)
            top--;
    else
     {
       len=strlen(str);
        len--;
        top--;
        do
        {
          stack[top++]=str[len--];
        }
        while(len>=0);
     // printf(“\n stack=%s”,stack);
     }
   }
else
   printf(“\t Not valid \n”);
}
```

Output:

```
Enter no. of terminals:3
Enter no. of nonterminals:3
Enter all nonterminals:SAB
Enter all terminals:ab$
Enter for S and a:AaAb
Enter for S and b:BbBa
Enter for S and $:#
Enter for A and a:@
Enter for A and b:@
Enter for A and $:#
Enter for B and a:@
Enter for B and b:@
Enter for B and $:#
table is
              a        b        $
S           AaAb     BbBa       #
A             @        @        #
B             @        @        #
Enter string to be parsed
ab$
Accepted
```

Result: The Program executed successfully.

Lab Viva Questions:

1. Define parse trees?
2. Define ambiguity?
3. What are the various types of errors in program?
4. What re the various error-recovery strategies?
5. Write a grammar to define simple arithmetic expression?

Experiment No: 8

Name of the Experiment: Write a C Program to construct shift- reduce parser.

Aim: To write a ‘c’ program for implementing the shift reduce.

Description: Shift-Reduce parsing is a form of bottom – up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. As we shall see, the handle always appeared the top of the stack just before it is identified as the handle.

We use $ to mark the bottom of the stack and also the right end of the input. Conventionally, when discussing bottom-up parsing, we show the top of the stack on the right, rather than on the left as we did for top – down parsing. Initially, the stack is empty, and the string ‘w’ is on the input, as follows.

Stack	Input
$	W$

Program:

```
  #include<stdio.h>
 #include<conio.h>
 #include<string.h>
   int top=0,l=0,j,k;
    char st[20],ip[20],str[20],op[20];
    void push(char);
    void  pop();
    void disp();
    void main()
 {
 int i,n=0;
 clrscr();
 printf(“Enter the input string:”);
 scanf(“%s”,&ip);
 j=0;
 strcat(ip,”$”);
 l=strlen(ip);
 push(‘$’);
 printf(“\n\t\t stack \t\t input \t\t action”);
 disp();
 for(i=0;i<=l;i++)
 {
if(ip[j]==’i’)
{
     printf(“\t\t shift”);
     push(ip[j]);
     j++;
    disp();
  }

else
if(ip[j]=='+' || ip[j]=='-' || ip[j]=='*' || ip[j]=='/' )
{
n++;
printf("\t\t shift");
push(ip[j]);
j++;
disp();
}
if(st[k]=='i')
{
st[k]='E';
printf("\t\t Reduce");
disp();
}
}
while(n>0)
{
pop();
pop();
printf("\t\t Reduce ");
disp();
n--;
}
if((strcmp(str,"$E")==0)&&(ip[j]=='$'))
printf("\t\t Accepted");
else
printf("\n\n\t\t Error");
getch();
}
void push(char c)
{
k=top;
st[top++]=c;
}
void pop()
{
top--;
 }
void disp()
 {
int i;
printf("\n\t\t");
for(i=0;i<top;i++)
{
printf("%c",st[i]);
str[i]=st[i];
}
//printf("\t\t string =%s\n",str);
str[i++]='\0';
```

```
printf("\t\t");
for(i=j;i<l;i++)
printf("%c",ip[i]);
}
```

Output:

```
Enter the input string: i+i*i

            stack          input          action

           $              i+i*i$          shift
           $i              +i*i$          Reduce
           $E              +i*i$          shift
           $E+              i*i$          shift
           $E+i             *i$           Reduce
           $E+E             *i$           shift
           $E+E*             i$           shift
           $E+E*i             $           Reduce
           $E+E*E            $            Reduce
           $E+E              $             Reduce
           $E                 $           Accepted
```

Result: The program executed successfully.

Lab Viva Questions:

1. What is shift reduce parsing?
2. Define Handles?
3. What are the four possible action of a shift reduce parser?
4. What is an operator grammar?

Experiment No: 9

Name of the Experiment: Write a C program to generate three address code.

Aim: To generate three address codes.

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<stdlib.h>
#include<string.h>
struct three
{
char data[10],temp[7];
}s[30];
void main()
{
char d1[7],d2[7]="t";
int i=0,j=1,len=0;
FILE *f1,*f2;
clrscr();
f1=fopen("sum.txt","r");
f2=fopen("out.txt","w");
while(fscanf(f1,"%s",s[len].data)!=EOF)
len++;
itoa(j,d1,7);
strcat(d2,d1);
strcpy(s[j].temp,d2);
strcpy(d1,"");
strcpy(d2,"t");
if(!strcmp(s[3].data,"+"))
{
fprintf(f2,"%s=%s+%s",s[j].temp,s[i+2].data,s[i+4].data);
j++;
}
else if(!strcmp(s[3].data,"-"))
{
fprintf(f2,"%s=%s-%s",s[j].temp,s[i+2].data,s[i+4].data);
j++;
}
for(i=4;i<len-2;i+=2)
{
```

```
itoa(j,d1,7);
strcat(d2,d1);
strcpy(s[j].temp,d2);
if(!strcmp(s[i+1].data,"+"))
fprintf(f2,"\n%s=%s+%s",s[j].temp,s[j-1].temp,s[i+2].data);
else if(!strcmp(s[i+1].data,"-"))
fprintf(f2,"\n%s=%s-%s",s[j].temp,s[j-1].temp,s[i+2].data);
strcpy(d1,"");
strcpy(d2,"t");
j++;
}
fprintf(f2,"\n%s=%s",s[0].data,s[j-1].temp);
fclose(f1);
fclose(f2);
getch();
}
```

Input: sum.txt

```
out = in1 + in2 + in3 - in4
```

Output: out.txt

```
t1=in1+in2
t2=t1+in3
t3=t2-in4
out=t3
```

Result: The program executed successfully.

Lab Viva Questions:

1. Define intermediate code?
2. Define parse trees and syntax trees
3. What is a three-address code?
4. Name any types of three-address statements?

Experiment No: 10

Name of the Experiment: Design a LALR bottom up parser for the given language.

Aim: To write a program for implementing the LALR parser.

Description:
The bottom of the syntax analysis is also known as shift reducing parsing. the general method of shift reduced parsing also called as LR parsing. Shift reducing parser attempts to construct a parse tree for an Input streambegining at the levels and waiting up toward the roots.
At the each reduction step a particular sub string matching the right side of the production is replaced by the symbol on the left of the production and if the substring is chosen correctly at each step. a right most derivation is traced out in reverse.

The possible opertions or actions are:

1) Shift
2) Reduce
3) Accept
4) Error

Program:

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
#include<stdlib.h>
char action[20][10][3],nt[10],t[10],inp[20],stack[20][2],a[3];
char pro[20][10],str[10],str2[10],state[20][2],gto[20][10][2];
int t1,s1,x;
int nut,nun,i=0,top=0,np,numst,nent,k;
int getnt(char);
int gett(char);
int getst(char*);
void main()
{
int i,j,len,n1,n2,i1,j1;
char ch,ans='y',temp[5];
clrscr();
printf("enter no of ter\n");
scanf("%d",&nut);
printf("enter no of nonter\n");
scanf("%d",&nun);
printf("enter no of productions\n");
scanf("%d",&np);
for(i=1;i<=np;i++)
{
printf("enter productions\n");
scanf("%s",pro[i]);
}
for(i=1;i<=np;i++)
```

```
{

printf("production%d=%s\n",i,pro[i]);
}
printf("enter all non ter\n");
scanf("%s",nt);
printf("enter all ter\n");
scanf("%s",t);
/*for(i=0;i<nut;i++)
printf("%c\t",t[i]);
printf("\n");
for(j=0;j<nun;j++)
printf("%c\t",nt[j]);
printf("\n"); */
printf("enter no of states\n");
scanf("%d",&numst);
for(i=0;i<numst;i++)
{
printf("enter state name\n");
scanf("%s",state[i]);
}
printf("enter action as s2 for shift 2\n r2 reduce2\n ac for accept\n e for error\n");
for(i=0;i<numst;i++)
for(j=0;j<nut;j++)
{
strcpy(action[i][j]," ");
printf("enter for %d and %c",i,t[j]);
scanf("%s",action[i][j]);

}

for(i=0;i<numst;i++)
for(j=0;j<nun;j++)
{
strcpy(gto[i][j]," ");
}
printf("enter no of entries\n");
scanf("%d",&nent);
for(k=0;k<nent;k++)
{
printf("enter state no");
scanf("%d",&i1);
flushall();
printf("enter non ter\n");
scanf("%c",&ch);
j1=getnt(ch);
flushall();
printf("enter gto[%d][%d]=",i1,j1);
scanf("%c",&gto[i1][j1]);
}
```

```
printf("\t\tACTION TABLE\n States");

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

printf("\t %c",t[j]);
printf("\n********************************\n");


for(i=0;i<numst;i++)
{
printf("|%s\t|",state[i]);
for(j=0;j<nut;j++)
{
printf("|%s\t|",action[i][j]);
}
printf("\n------------------------------------------\n");
}

printf("\t\tGOTO TABLE \nStates");
for(j=0;j<nun;j++)
printf("|\t %c|",nt[j]);

printf("\n**************************\n");


for(i=0;i<numst;i++)
{
printf("|%s\t|",state[i]);
for(j=0;j<nun;j++)
{
printf("|%s\t|",gto[i][j]);
}
printf("\n-------------------------------\n");
}
do
{
strcpy(inp," ");
printf("enter str to parse\n");
scanf("%s",inp);
top=0;
strcpy(stack[top++],"$");
strcpy(stack[top++],"0");
i=0;
while(1)
{
t1=gett(inp[i]);
s1=getst(stack[top-1]);
strcpy(str,action[s1][t1]);
if(str[0]=='a')
{
```

```
printf("str accepted");
break;

}

else if(str[0]=='s')
{
temp[0]=inp[i++];
temp[1]='\0';
strcpy(stack[top++],temp);
for(i1=0;i1<strlen(str);i1++)
if((i1!=0)&&isdigit(str[i1]))
temp[i1-1]=str[i1];
strcpy(stack[top++],temp);

}
else if(str[0]=='r')
{
a[0]=str[1];
a[1]='\0';

x=atoi(a);
strcpy(str2,pro[x]);
len=strlen(str2);
len=len-3;
top=top-2*len;
temp[0]=str2[0];
temp[1]='\0';
strcpy(stack[top++],temp);
n1=getst(stack[top-2]);
n2=getnt(str2[0]);
strcpy(stack[top++],gto[n1][n2]);

}
else
{
printf("error not valid string\n");
break;
}
}
flushall();
ch=' ';
printf("do u want to continue\n");
scanf("%c",&ch);
}while(ans==ch);
}
int getnt(char x)
{
int a;
for(a=0;a<nun;a++)
```

```
if(x==nt[a])
return a;
return 100;
}
int gett(char x)
{
int a;
for(a=0;a<nut;a++)
if(x==t[a])
return a;
return 100;
}

int getst(char st[])
{
int l;
for(l=0;l<numst;l++)
if(strcmp(st,state[l])==0)
return l;
return 100;
}
```

OUT PUT:

```
enter no of ter
3
enter no of nonter
2
enter no of productions
3
enter productions
S->CC
enter productions
C->cC
enter productions
C->d
production1=S->CC
production2=C->cC
production3=C->d
enter all non ter
SC
enter all ter
cd$
enter no of states
7
enter state name
0
enter state name
1
enter state name
2
```

```
enter state name
3
enter state name
4

enter state name
5
enter state name
6
enter action as s2 for shift 2
 r2 reduce2
 ac for accept
 e for error3

enter for 0 and cs3
enter for 0 and ds4
enter for 0 and $e
enter for 1 and ce
enter for 1 and de
enter for 1 and $ac
enter for 2 and cs3
enter for 2 and ds4
enter for 2 and $e
enter for 3 and cs3
enter for 3 and ds4
enter for 3 and $e
enter for 4 and cr3
enter for 4 and dr3
enter for 4 and $r3
enter for 5 and ce
enter for 5 and de
enter for 5 and $r1
enter for 6 and cr2
enter for 6 and dr2
enter for 6 and $r2
enter no of entries
4
enter state no0
enter non ter
S
enter gto[0][0]=1
enter state no0
enter non ter
C
enter gto[0][1]=2
enter state no2
enter non ter
C
enter gto[2][1]=5
enter state no3
enter non ter
C
enter gto[3][1]=6



ACTION TABLE

 States   c       d       $
*********************************
|0       ||s3    ||s4    ||e     |
--------------------------------
|1       ||e     ||e     ||ac    |
--------------------------------
|2       ||s3    ||s4    ||e     |
--------------------------------
|3       ||s3    ||s4    ||e     |
--------------------------------
|4       ||r3    ||r3    ||r3    |
--------------------------------
|5       ||e     ||e     ||r1    |
--------------------------------
|6       ||r2    ||r2    ||r2    |
--------------------------------
              GOTO TABLE
States   |  S||     C|
***************************
|0        ||1     ||2     |
------------------------
|1         ||     ||      |
------------------------
|2         ||     ||5     |
------------------------
|3       ||     ||6     |
------------------------
|4       ||     ||      |
------------------------
|5       ||     ||      |
------------------------
|6       ||     ||      |
------------------------
enter str to parse
- - - - - - - - - - - - - - - -
enter str to parse
cd$
error not valid string
do u want to continue
y
enter str to parse
dd$
```

```
str accepted
do u want to continue
n
```

Result: The program executed successfully.

Lab Viva Questions:

1. Why bottom-up parsing is also called as shift reduce parsing?
2. What are the different types of bottom up parsers?
3. What is mean by LR (0) items?
4. Write the general form of LR(1) item?
5. What is YACC?

Experiment No: 11

Name of the Experiment: Convert the BNF rules into YACC form and write code to generate abstract syntax tree.
Aim: To implement a program for converting The BNF rules into YACC form and write code to generate abstract syntax tree.

Description:

The grammar is used to construct a parse tree from an expression.

So if you already have a parse tree, you don't need the grammar.

Depending on how much work your parser does, the resulting tree that is formed from parsing an expression could already be an abstract syntax tree. Or it could be a simple parse tree which requires a second pass to construct the ast.

To construct the parse tree from a grammar and an expression, you would first have to convert your grammar into working code.

Typically, you would split the work into a tokenizer which splits the input stream representing the expression into a list of tokens, and a parser which takes the list of tokens and constructs a parse tree\ast from it.

So the expression "1 + 2*(3+4)" might be split into a list of tokens like this:

1	int
+	add_operator
2	int
*	mul_operator
(	lparen
3	int
+	add_operator
4	int
)	rparen

The first column is the actual text value. The second represents the token type.

These tokens are fed into the parser, which is built from your grammar and recognizes the tokens and builds the parse tree.

Program:

/*Program name:<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];
%%
```

output:

```
lex lex.l
```

/*Program name:<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 ';'
| CONDST
| 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;
int b;
int c;
int x;
a=5;
b=2;
c=4;
X=a*b+c;
}
```

Output:

```
yacc -d parser.y
gcc lex.yy.c y.tab.c -ll -lm
[cse@putty ~]$ ./a.out test.c
           -------------------
            Pos Operator Arg1 Arg2 Result
           -------------
            0      =      5          a
            1      =      2          b
            2      =      4          c
            3      *      a     b    t0
            4      +      t0    c    t1
            5      =      t1         X
            ---------------
```

Result: The program executed successfully.

Lab Viva Questions:

1. What is Abstract Syntax tree?
2. What are BNF Rules?
3. What is DAG representation?
4. How LALR (1) states are generates?
5. In which condition the user has to supply more information to YACC?

Experiment No: 12

Name of the Experiment: A program to generate machine code from the abstract syntax tree generated by the parser.

Aim: To write a program in order to generate machine code from abstract syntax tree generated by the parser

Description:
Code generation is the process by which compilers convert some intermediate representation of source code into form (machine code) that can be readily executed by a machine.
Sophisticated compilers typically perform multiple passes over various intermediate forms. This multistage process is used because many algorithms for code optimization are easier to apply one at a time or because the input to one optimization relies on the processing performing by another optimization. This organization also facilitates the creation of a single compiler that can target multiple architectures as only the last of the code generation stages needs to change from target to target.

Program:
```
#include<stdio.h>
#include<conio.h>
void oneaddr();
void twoaddr();
void threeaddr();
void store();
char alp[10],op[10];
int j=0,i=0,h=0,m=0,k=0;
void main()
{
  char ip[10];
  int choice;
  clrscr();
  printf("\n *****MENU****\n 1.Single Address Instruction");
  printf("\n 2.Two Address Instruction \n3.Three Address
               Instruction");
  printf("\n Enter the postfix expression:");
  scanf("%s",&ip);
  printf("\n select the TARGET CODE instruction format:");
  scanf("%d",&choice);
  printf("OPCODE ADDRESS \n---------- \n");
  while(ip[h]!='\0')
  {
   if(isalpha(ip[h]))
   {
     alp[i++]=ip[h++];
   }
   else
   {
     op[j]=ip[h++];
switch(choice)
        {
           case 1:oneaddr();
                        break;
           case 2:twoaddr();
                        break;
           case 3:threeaddr();
                        break;
           default:printf("Invalid choice");
        }
      }
    }
    getch();
}
void  twoaddr()
{
   if(m<=0)
   {
     printf("LOAD \t %c R1 \n",alp[--i]);
     printf("LOAD \t %c R2 \n",alp[--i]);
     m++;
   }
   else
   printf("LOAD \t %c,R2 \n",alp[--i]);
   switch(op[j])
   {
     case '+':printf("ADD \t R2,R1\n");
                    break;
     case '*':printf("MUL \t R2,R1\n");
                    break;
     case '-':printf("SUB \t R2,R1\n");
                   break;
     case '/':printf("DIV \tR2,R1\n");
                   break;
     default:printf("INVALID CASE");
   }
}
void  threeaddr()
{
    if(m<=0)
   {
     printf("LOAD \t %c,R1 \n",alp[--i]);
     printf("LOAD \t %c,R2 \n",alp[--i]);
     m++;
   }
   else
   printf("LOAD \t %c,R2 \n",alp[--i]);
   if(k==2)
   k=0;
   switch(op[j])
```

```
{
     case ‘+’:if(k==0)
                    printf(“ADD \t R1,R2,R3\n”);
                    else
                    printf(“ADD \t R3,R2,R1\n”);
                    k++;
                    break;
     case ‘*’:if(k==0)
                    printf(“MUL \t R1,R2,R3\n”);
                    else
                    printf(“MUL \t R3,R2,R1\n”);
                    k++;
                    break;
     case ‘-’:if(k==0)
                   printf(“SUB \t R1,R2,R3\n”);
                   else
                   printf(“SUB \t R3,R2,R1\n”);
                   k++;
                   break;
     case ‘/’: if(k==0)
                   printf(“DIV \t R1,R2,R3\n”);
                   else
                   printf(“DIV \t R3,R2,R1\n”);
                   k++;
                   break;
     default:printf(“INVALID CASE”);
    }
}
void  oneaddr()
{
   printf(“LOAD \t %c ,R1 \n”,alp[--i]);
   switch(op[j])
   {
     case ‘+’:printf(“ADD \t %c,R1\n”,alp[--i]);
                    store();
                    break;
    case ‘*’:printf(“MUL \t %c,R1\n”,alp[--i]);
                   store();
                   break;
     case ‘-’:printf(“SUB \t %c,R1\n”,alp[--i]);
                   store();
                   break;
     case ‘/’:printf(“DIV \t %c,R1\n”,alp[--i]);
                   store();
                   break;
    default:printf(“Invalid Case”);
   }
}
void store()
{
```

```
printf("STORE  \t R1,t\n");
   alp[i++]='t';
}
```

Output:

```
*****MENU*****
1. Single Address Instruction
2. Two Address Instruction
3. Three Address Instruction
Enter the post expression:ab+c-
select the TARGET CODE INTRUCTION format:1
OPCODE          ADDRESS
-----------------------
LOAD             b,R1
ADD              a,R1
STORE            R1,t
LOAD             c,R1
SUB              t,R1
STORE            R1,t
*****MENU******
1. Single Address Instruction
2. Two Address Instruction
3. Three Address Instruction
Enter the post expression:ab+c-
select the TARGET CODE INTRUCTION format:2
OPCODE          ADDRESS
-----------------------
LOAD             b,R1
LOAD             a,R2
ADD              R2,R1
LOAD             c,R2
SUB              R2,R1
*****MENU******
1. Single Address Instruction
2. Two Address Instruction
3. Three Address Instruction
Enter the post expression:ab+c-
select the TARGET CODE INTRUCTION format:3
OPCODE          ADDRESS
-----------------------
LOAD             b,R1
LOAD             a,R2
ADD              R1,R2,R3
LOAD             c,R2
SUB              R3,R2,R1
```

Result:

The program was successfully executed.

Lab Viva Questions:

1 What are the different forms of object code?
2 What is mean by relocatable object code?
3 What is the cost of register to register operation?
4 What is address descriptor?
5 What is register descriptor?

ADDITIONAL EXPERIMENTS

Experiment No: 1.a) Infix to Prefix

Aim: To write a program to convert an infix expression into prefix expression.

Description: The prefix notation in which the operators are written before the operators, it is also called "polish notation". In the however of the
Polish mathematician JANKKASIEWIC2, develop when written in the prefix notation as "+AB". As the operator '+' is written before the operands A and B.so this operation is called prefix notation.

Program:

```
        #include<stdio.h>
         #include<conio.h>
         #include<ctype.h>
         char pop();
         void push(char);
          void check();
         char  s[20],a[20],b[20];
         int   top=-1,t,i=0,j=0;
         void main()
          {
             char X;
             clrscr();
             printf("Enter the infix expression:");
             gets(a);
              strrev(a);
              push('#');
              while(a[i]!='\0')
    {
      t=a[i];
      if(isalpha(t) || isdigit(t))
      b[j++]=t;
       else
      if( t=='+' || t=='-' || t=="*" || t=='/' || t==')'  ||  t=='('  )
      {
       switch(t)
       {
       Case  ')' : push(t);
                  break;
       case  '+':
       case  '-':
       case  '*':
       case  '/':  check();
                  push(t);
                  break;
       case '(':   do
                  {
                  X=pop();
```

```
                           b[j++]=x;
                           }
                            While(x!=’)’);
                            j=j-1;
                            break;
         }
       }
   i=i+1;
}
while(s[top]!=’#’)
b[j++]=pop();
b[j]=’\0’;
strrev(b);
printf(“prefix expression is %s”,b);
getch();
}
void check()
{
while(priority(t)<=priority(s[top]))
b[j++]=pop();
}
int  priority(char ch)
{
if(ch==’*’  || char ==’/’)
return(2);
if(ch==’+’  || ch==’-‘)
return(1);
else
return(0);
}
 void  push(char c)
{
s[++top]=c;
}
char pop()
{
return(s[top--]);
}
```

Output:

```
 Enter the infix expression a+b*c
    The prefix expression is *+abc
```

Result: The program is successfully executed.

Experiment No: 1.b) Infix to Postfix

AIM: To write a c program to convert an infix expression to postfix expression
.
DESCRIPTION: The expression in which the operator lies b/w the operands is called the infix expression where as the postfix expression in which the operator is placed after the operands. for example: A+B is the infix expression for the infix one is AB+. For this conversation we use the stack for the storage of the operands and paranthesis .To stack we always give the priority of the stack's top.

PROGRAM:
```
  #include<stdio.h>
   #include<conio.h>
   #include<ctype.h>
   void push(char);
   void check();
    char pop();
    char s[20],a[20],b[20];
    int top=-1,t,i=0,j=0;
    void main()
    {
    char x;
    clrscr();
    printf("Enter the infix expression ");
    gets(a);
    push('#');
    while(a[i] ! ='\0')
    {
    t=a[i];
    if(isalpha(t))
    b[j++]=t;
    else
       if(t=='+'  ||  t=='-'  ||  t=='*'  ||  t=='/' || t=='('   ||  t==')' )
       {
        switch(t)
        {
         case '(' : push(t);
                    break;
         case '+':
         case '-' :
         case '/' :
         case '*': check();
                    push(t);
                    break;
         case  ')': do
                    {
                    x=pop();
                    b[j++]=x;
                    }
                  while( x!= ’(‘ );
                    j=j-1;
                    break;
         }
        }
      i=i+1;
    }
    while(s[top] ! = ‘#’ )
    b[j++]=pop();
    b[j] =’\0’;
    printf(“postfix expression is %s”,b);
    getch();
   }
    void check()
    {
    while(priority(t)<=priority(s[top]))
      b[j++]=pop();
      }
      int  priority(char ch)
       {
       if( ch== ’*’ || ch== ’/’ )
        return(2);
       if( ch==’+’ || ch==’-‘ )
    return(1);
    else
    return(0);
  }
void push(char c)
  {
  s[++top]=c;
  }
char pop()
{
return(s[top--]);
}
```

Output: Enter the infix expression a+((b/c)*d)

```
              The postfix expression is abc/d*+
```

Result: The program is successfully executed.

Experiment No: 2) Code Optimization

Aim: To write a c program for implementing code optimization for the given expression.
Description:

The machine independent code optimization phase attempts to improve the intermediate code so that better target code will result .Usually better means faster, but other objectives may be desired, such as shorter code or target code that consumes less power.

For example, a straight forward algorithm generates the intermediate code, using an instruction for each operation in the tree representation that comes from the semantic analyzer.

Program:

```
#include<stdio.h>
#include<conio.h>
void gen();
void codegen();
char s[15],v[4];
char a[10]={‘A’,’B’,’C’,’D’,’E’,’F’,’G’,’H’,’I’,’J’};
char p[4]={‘/’,’*’,’+’,’-‘};
int i,j,k,q=0,l,m;
void main()
{
  clrscr();
  printf(“\n \n pls enter the expression \t”);
  gets(s);
  printf(“\n \t \t Code Optimization”);
  printf(“\n \t \t----------------“);
  for(i=0;i<strlen(p);i++)
  {
    for(j=0;j<strlen(s);j++)
    {
      if(p[i]==s[j])
      {
        k=j;
        v[0]=s[j-1];
        v[1]=s[j+1];
        v[2]=s[j];
         gen();
      }
    }
  }
  getch();
}
void gen()
{
   for(m=0;m<2;m++)
   {
     for(k=j;s[k]!=’\0’;k++)
     s[k]=s[k+1];
   }

   s[j-1]=a[2];
```

```
 v[3]=s[j-1];
   codegen();
   q++;
   j=-1;
}
 void codegen()
{
   switch(v[2])
  {
    case ‘*’:printf(“\n\n\t\t MUL %c %c %c”,v[3],v[0],v[1]);
                 break;
    case ‘/’:printf(“\n\n\t\t DIV %c %c %c”,v[3],v[0],v[1]);
                 break;
    case ‘+’:printf(“\n\n\t\t ADD %c %c %c”,v[3],v[0],v[1]);
                 break;
    case ‘-’:printf(“\n\n\t\t SUB %c %c %c”,v[3],v[0],v[1]);
                 break;
  }
}
```

Output:

```
Enter the expression: a+b*c
Code Optimization
MUL A  b  c
ADD B  a   A
```

Result: The Program is successfully executed.