B.E. 3/4-II Sem
Compiler Construction Lab

List of Experiments Prescribed By Osmania University

1. Scanner stand alone program in C
2. Scanner program using Lex
3. First and Follow set of Productions
4. Recursive Decent Parser Generation
5. LL parser generation
6. SLR parser generation
7. LR parser generation
8. Parser program using YACC (Calculator)
9. YACC program for three address code generation
10. YACC program for Quadruple generation
11. Code generation
12. Code optimization

COMPILER CONSTRUCTION LAB

CONTENTS

1. ‘C’ Programs to recognize Comment line, Identifier, Number

AIM: To recognize tokens in the source code.

HARDWARE REQUIREMENTS: Pentium III, 256MBRAM, 80GB HDD,keyboard, mouse, monitor.

SOFTWARE REQUIREMENTS: Windows XP, Turbo C.

DESCRIPTION: C programs using switch statement/ if statement to recognize Comment line which starts with // or enclosing in /* and */ symbols, an Identifier starts with alphabet [A to Z or a to z] and can followed by any number of digits [0 to 9] and/or any number of alphabets and number starts with digit and followed by any number of digits.

SOURCE CODE:

/* program to recognize comment line */

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int state=1;
  int ip=0;
  char input[20];
  clrscr();
  printf("Enter the input string:\n");
  scanf("%s",input);
  while((state>=1)&&(state<=4))
  {
    switch(state)
    {
      case 1:if(input[ip]=='/')
              {
                ip++;
                state=2;
              }
              else
                state=6;
              break;
      case 2:if(input[ip]=='*')
              {
```

```
              ip++;
              state=3;
            }
            else
              state=6;
            break;
     case 3:if(input[ip]=='*')
            {
              ip++;
              state=4;
            }
            else
            {
              ip++;
              state=3;
            }
            break;
     case 4:if(input[ip]=='/')
            {
              ip++;
              state=5;
            }
            else
            {
              if(input[ip]=='*')
              {
                    ip++;
                    state=4;

            }
              else
              {
                    ip++;
                    state=3;
              }
            }
            break;
      }
    }
    if(state==5)
      printf("It is a comment line");
    else
      printf("It is not a comment line");
    getch();
}
```

OUTPUT:

```
Enter the input string:
// am I a comment line ?
  It  is a comment line

Enter the input string
/* program to find xxx
It is not a comment line.
```

/* To recognize key words, identifiers and numbers */

```
#include<stdio.h>
#include<conio.h>
char a[10],b[10];
int sta=0;
int keyword(void);
void identifier(void);
void main()
{
FILE *fp;
char ch;
int i=0,j=0;
clrscr();
fp=fopen("input.c","r");
while((ch=fgetc(fp))!=EOF)
{
	i=0;
	while(ch!='\n')
	{
		a[i]=ch;
		i++;
		a[i]='\0';
		ch=fgetc(fp);
	}
	while(a[i]!='\0')
	{
	j=0;
	while(a[i]!=' ' && a[i]!=';')/*&&(a[i]!=';')&&(a[i]!='\0')*/
	{
		b[j]=a[i];
		j++;
		b[j]='\0';
		i++;
```

```
        }
        i++;
        sta=keyword();
        if(sta==0)
        printf("hddfhh");

        else
        identifier();


        }
        }
fclose(fp);
getch();
}

int keyword()
{
FILE *k;
int i=0,l;
char c;
char key[10][6]={{"char"},{"int"},{"void"},{"if"},{"else"},{"while"},{"do"},{"for"}};
for(i=0;i<10;i++)
{
        l=strcmp(key,b);
        if(l==0)
        {
                printf("%s is an keyword",b);
                return 0;
        }
        else
        {
        }

}
}

void identifier()
{
int i,j=0,k=0;
char c[10],ide[10];
if((b[0]=='_')||(isalpha(b[0])))
{
        ide[j]=b[0];
        j++;
        for(i=1;b[i]!='\0';i++)
        {
                if((isalpha(b[i]))||(isdigit(b[i]))||(b[i]=='_'))
                {
                        ide[j]=b[i];
                        j++;
                        ide[j]='\0';
                }
                else if(b[i]=='"')
                {
                        do
                        {
                                i++;
                                c[k]=b[i];
                                k++;
                                c[k]='\0';
                        }
                        while(b[i]!='"');
                        k--;
                        c[k]='\0';
                }
                else if(b[i]==';')
                {
                        if(j==0)
                                printf("%s is not an identifier",b);
                        else
                                printf("%s is an identifier",b);
                        j=0;
                }
        }
}
if(k==0)
{
}
else
        printf("%s is a constant",c);
}
```

OUTPUT:

Input file input.c

```
 {   If
    Abc
    23
}
```

```
  If is a keyword
 Abc is a identifier
 23 is a constant.
```

2. Implement Lexical analyzer using C

AIM: Implement Lexical analyzer / Scanner using C

HARDWARE REQUIREMENTS: Pentium III, 256MBRAM, 80GB HDD,keyboard, mouse, monitor.

SOFTWARE REQUIREMENTS: Windows XP, Turbo C.

DESCRIPTION: Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a **lexical analyzer**, **lexer** or **scanner.** A **token** is a string of characters, categorized according to the rules as a symbol (e.g. IDENTIFIER, NUMBER, COMMA, etc.). The process of forming tokens from an input stream of characters is called **tokenization** and the lexer categorizes them according to a symbol type. A token can look like anything that is useful for processing an input text stream or text file.

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
#include<stdlib.h>
#define SIZE 128
#define NONE -1
#define EOS '\0'
#define NUM 256
#define KEYWORD 257
#define PAREN 258
#define ID 259
#define ASSIGN 260
#define REL_OP 261
#define DONE 262
#define MAX 999
char lexemes[MAX];
char buffer[SIZE];
int lastchar=-1;
int lastentry =0;
int tokenval=NONE;
int lineno=1;
struct entry
{
char *lexptr;
int token;
}
```

```
symtable[100];
/*stores the list of keywords*/
struct entry keywords[]=
{
"if",KEYWORD,"else",KEYWORD,"FOR",KEYWORD,"INT",KEYWORD,"FLOAT",
KEYWORD,"DOUBLE",KEYWORD,"CHAR",KEYWORD,"STRUCT",KEYWORD,"
RETURN",KEYWORD,
0,0};
/*THIS FUNCTION ISSSUE A COMPILER ERROR*/
void Error_message(char *m)
{
fprintf(stderr,"line %d: %s\n",lineno,m);
exit(1);

}
/*this function is used to search the symbol table for particular entry*/
int look_up(char s[])
{
int k;
for(k=lastentry;k>0;k=k-1)
if(strcmp(symtable[k].lexptr,s)==0)
return k;
return 0;
}
int insert(char s[],int tok)
{
int len;
len=strlen(s);
if((lastentry+1)>=MAX)
Error_message("Symbol table is full");
if((lastchar+len+1)>=MAX)
Error_message("lexemas array is full");
lastentry=(lastentry +1);
symtable[lastentry].token=tok;
symtable[lastentry].lexptr=&lexemes[lastchar+1];
lastchar=lastchar + len +1;
strcpy(symtable[lastentry].lexptr,s);
return lastentry;
}
void Initialize()
{
struct entry *ptr;
for(ptr=keywords;ptr->token;ptr++)
insert(ptr->lexptr,ptr->token);
}
int lexer()
```

```
{
int t;
int val,i=0;
while(1)
{
t=getchar();
if(t==' '||t=='\t')
 ;
else
 if(t=='\n')
lineno=lineno+1;
else if(t=='('||t==')')
return PAREN;
else if(t=='<'||t=='>'||t=='>='||t=='<='||t=='!=')
return REL_OP;
else if(t=='=')
return ASSIGN;
else if(isdigit(t))
{
ungetc(t,stdin);
scanf("%d",&tokenval);
return NUM;
}
else if(isalpha(t))
{
while(isalnum(t))
{
buffer[i]=t;
t=getchar();
i=i+1;
if(i>=SIZE)
Error_message("compiler error");
}
buffer[i]=EOS;
if(t!=EOF)
ungetc(t,stdin);
val=look_up(buffer);
if(val==0)
val= insert(buffer,ID);
tokenval=val;
return symtable[val].token;
}
else if(t==EOF)
return DONE;
else
{
```

```
tokenval=NONE;
return t;
}
}
}
void main()
{
int lookahead;
char ans;
clrscr();
printf("\n\t\t  program for lexical analysis \n");
Initialize();
printf("\n Enter the expresssion  and put ; at the end");
printf("\n press ctrl z to terminate ....\n");
lookahead=lexer();
while(lookahead!=DONE)
{
if(lookahead==NUM)
{
printf("\n NUmber:");
printf("%d",tokenval);
}
if(lookahead=='+'||lookahead=='-'||lookahead=='*'||lookahead=='/')
printf("\n operator");
if(lookahead==PAREN)
printf("\n parenthesis");
if(lookahead==ID)
{
printf("\n Identifier:");
printf("%s",symtable[tokenval].lexptr);
}
if(lookahead==KEYWORD)
printf("\n keyword");
if(lookahead==ASSIGN)
printf("\n Assignment operator");
if(lookahead==REL_OP)
printf("\n relational operator");
lookahead=lexer();
}
}
```

OUTPUT:

```
Program for lexical analysis
Enter the expression and put ; at the end
Press ctrl Z to terminate…
Else;
Keyword
```

3. Simple Lex Programs

AIM:. Lex programs to recognize Keywords, String ending with 00, starts and end with 'k' ,numbers with 1 in its 5th position from right, and To assign line numbers for source code

HARDWARE REQUIREMENTS: Pentium III, 256MBRAM, 80GB HDD,keyboard, mouse, monitor.

SOFTWARE REQUIREMENTS: Windows XP, Turbo C.

DESCRIPTION: Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

SOURCE CODE:

1.//program to recognize keywords

```
%{
#include<stdio.h>
%}
%%
int|float|char { printf(" data type: %s",yytext); }
%%
main()
{
yylex();
return(0);
}
```

OUTPUT:

```
int

datatype: int
```

2.//program to assign line numbers for source code.

```
%{
 #include<stdio.h>
 int lineno=0;
%}
line .*\n
%%
{line}   { printf("%d .%s",lineno++,yytext); }
%%
main()
{
yylex();
return (0);
}
```

OUTPUT:

```
abc
def
ghi

1.abc
2.def
3.ghi
```

3. //program to recognize the strings which are ending with 00

```
%{
 #include<stdio.h>
%}
%%
[a-z A-Z 0-9]++00 { printf("string is acepted",yytext);}
.* { printf("not acepted",yytext);}
%%
main()
{
yylex();
return(0);
}
```

OUTPUT:

```
As100
String is accepted
```

4. Program to recognize the numbers which has 1 in its 5th position from right

```
%{
#include<stdio.h>
%}
%%
[0-9]*1[0-9]{4} { printf("acepted:");}
[0-9]*1[0-9] { printf("NOT acepted:");}
%%
main()
{
yylex();
return(0);
}
```

OUTPUT:

```
3410001
accepted
123456
NOT accepted
```

5. Program to recognize the strings which are starting or ending with 'k'

```
%{
 #include<stdio.h>
%}
begin-with-k k.*
end-with-k .*k
%%
{begin-with-k} { printf(" %s is a word that begin with k",yytext);}
{end-with-k} { printf(" %s is a word that end with k",yytext);}
%%
main()
{
yylex();
return(0);
}
```

OUTPUT:

```
kishore
kishore is a word that begin with k
shaik
shaik is a word that end with k
```

4. Implement lexical analyzer in Lex.

AIM: Implement lexical analyzer in Lex.

HARDWARE REQUIREMENTS: Pentium III, 256MBRAM, 80GB HDD,keyboard, mouse, monitor.

SOFTWARE REQUIREMENTS: Windows XP, Turbo C.

DESCRIPTION : The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial look a head is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it. The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

SOURCE CODE:

```
%{
int COMMENT=0;
%}
identifier [a-z A-Z][a-z A-Z 0-9]*
%%
#.* {printf("\n %s is a PREPROCESSOR DIRECTIVE",yytext);}
int|float|char|double|while|for|do|if|break|continue|void|switch|case|long|struc
t|const|typedef|return|else|goto|main {printf("\n\t %s is a KEYWORD",yytext);}
"%*" {COMMENT=1;}
"*/" {COMMENT=0;}
{identifier}\( {if(!COMMENT) printf("\n\n FUNCTION \n\t%s",yytext); }
\{ {if(!COMMENT) printf("\n BLOCK BEGINS");}
\} {if(!COMMENT) printf("\n 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\t%s is a NUMBER",yytext);}
\)(\;)? {if(!COMMENT) printf("\n\t");
ECHO;
printf("\n");}
```

```
\(ECHO;
= {if(!COMMENT) printf("\n\t is an ASSIGNMENT OPERATOR",yytext);}
\<=|\>=|\<|==|\> {if(!COMMENT) printf("\n\t %s is a RELATIONAL
OPERATOR",yytext)
;}
\b|\t {if(!COMMENT) printf("\n white space",yytext);}
\n { ; }
%%
int main(int argc,char **argv)
{
if(argc>1)
{
FILE *file;
file=fopen(argv[1],"r");
if(!file)
{
printf("could not open %s\n",argv[1]);
exit(0);
}
yyin=file;
}
yylex();
printf("\n\n");
return 0;
}
int yywrap()
{
return 0;
}
```

INPUT: file input.c

```
#include<stdio.h>
Main()
{ int i=10;
}
```

OUTPUT:

```
#include<stdio.h> is a preprocessor directive
Main    KEYWORD
(
)
 BLOCK BEGINS
 int  KEYWORD ; IDENTIFIER
     i IDENTIFIER
   =  is an ASSIGNMENT OPERATOR
   10 is a NUMBER
BLOCK ENDS
```

5. To find FIRST set of productions for given CFG

AIM: C program to find FIRST set of productions for given CFG

HARDWARE REQUIREMENTS: Pentium III, 256MBRAM, 80GB HDD,keyboard, mouse, monitor.

SOFTWARE REQUIREMENTS: Windows XP, Turbo C.

DESCRIPTION: If x is any string of grammar symbols, let FIRST(x) be the set of terminals that begin the strings derived from x. If x=*>e, then e is also in FIRST(x). Define FOLLOW(A), for nonterminals A, to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exists a derivation of the form S=*>xAab for some x and b. Note that there may, at some time during the derivation, have been symbold beteween A and a, but if so, they derived e and disappeared. If A can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A).
To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or e can be added to any FIRST set.
1. If X is terminal, then FIRST(X) is {X}.
2. If X->e is a production, then add e to FIRST(X).
3. If X is nonterminal and X->Y1Y2...Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi) and e is in all of FIRST(Y1),...,FIRST(Yi-1); that is, Y1...Yi-1=*>e. If e is in FIRST(Yj) for all j=1,2,...,k, then add e to FIRST(X). For example, everything in FIRST(Yj) is surely in FIRST(X). If y1 does not derive e, then we add nothing more to FIRST(X), but if Y1=*>e, then we add FIRST(Y2) and so on.

SOURCE CODE:

```
#include"stdio.h"
#include<conio.h>
#define max 10
#define MAX 15
char array[max][MAX],temp[max][MAX];
int c,n,t;void fun(int,int[]);
int fun2(int i,int j,int p[],int key)
{
int k;
if(!key)
{
```

```
for(k=0;k<n;k++)
if(array[i][j]==array[k][0])
break;
p[0]=i;p[1]=j+1;
fun(k,p);
return 0;
}
else
{
for(k=0;k<=c;k++)
{
if(array[i][j]==temp[t][k])
break;
}
if(k>c)return 1;
else return 0;
}
}
void fun(int i,int p[])
{
int j,k,key;
for(j=2;array[i][j]!='\0';j++)
{
if(array[i][j-1]=='/')
{
if(array[i][j]>='A'&&array[i][j]<='Z')
{
key=0;
fun2(i,j,p,key);
}
else
{key=1;
if(fun2(i,j,p,key))
temp[t][++c]=array[i][j];
if(array[i][j]=='[at]'&&p[0]!=-1)
{ //taking ,[at], as null symbol.
if(array[p[0]][p[1]]>='A'&&array[p[0]][p[1]]<='Z')
{
key=0;
fun2(p[0],p[1],p,key);
}
else
if(array[p[0]][p[1]]!='/'&&array[p[0]][p[1]]!='\0')
{
if(fun2(p[0],p[1],p,key))
temp[t][++c]=array[p[0]][p[1]];
}
}
}
}
}
}
char fol[max][MAX],ff[max];int f,l,ff0;
void ffun(int,int);
void follow(int i)
{
int j,k;
for(j=0;j<=ff0;j++)
if(array[i][0]==ff[j])
return 0;
if(j>ff0)ff[++ff0]=array[i][0];
if(i==0)fol[l][++f]='$';
for(j=0;j<n;j++)
for(k=2;array[j][k]!='\0';k++)
if(array[j][k]==array[i][0])
ffun(j,k);
}
void ffun(int j,int k)
{
int ii,null=0,tt,cc;
if(array[j][k+1]=='/'||array[j][k+1]=='\0')
null=1;
for(ii=k+1;array[j][ii]!='/'&&array[j][ii]!='\0';ii++)
{
```

```
if(array[j][ii]<='Z'&&array[j][ii]>='A')
{
for(tt=0;tt<n;tt++)
if(temp[tt][0]==array[j][ii])break;
for(cc=1;temp[tt][cc]!='\0';cc++)
{
if(temp[tt][cc]=='[at]')null=1;
else fol[l][++f]=temp[tt][cc];
}
}
else fol[l][++f]=array[j][ii];
}
if(null)follow(j);
}
void main()
{
int p[2],i,j;
clrscr();
printf("Enter the no. of productions :");
scanf("%d",&n);
printf("Enter the productions :\n");
for(i=0;i<n;i++)
scanf("%s",array[i]);
for(i=0,t=0;i<n;i++,t++)
{
c=0,p[0]=-1,p[1]=-1;
temp[t][0]=array[i][0];
fun(i,p);
temp[t][++c]='\0';
printf("First(%c) : [ ",temp[t][0]);
for(j=1;j<c;j++)
printf("%c,",temp[t][j]);
printf("\b ].\n");
getch();
}
/* Follow Finding */
for(i=0,l=0;i<n;i++,l++)
```

```
{
f=-1;ff0=-1;
fol[l][++f]=array[i][0];
follow(i);
fol[l][++f]='\0';
}
for(i=0;i<n;i++)
{
printf("\nFollow[%c] : [ ",fol[i][0]);
for(j=1;fol[i][j]!='\0';j++)
printf("%c,",fol[i][j]);
printf("\b ]");
getch();
}
}
```

OUTPUT:

```
Enter the no. of productions 4
Enter the productions
A->aBb
C->dBc
B->dCc

First of A is: a
First of C is: d
First of B is: d

Follow[A] : $
Follow[B] : b,c
Follow[C] : c
```

6a. Recursive descent parsing

AIM:. C program for Recursive descent Parsing
HARDWARE REQUIREMENTS: Pentium III, 256MBRAM, 80GBHDD, keyboard, mouse, monitor.

SOFTWARE REQUIREMENTS: Windows XP, Turbo C.

DESCRIPTION:

A recursive descent parser is a static top down parser. For every non terminal Procedure must be implemented to recognize its production body. The first procedure invoked by main function is the procedure for start symbol of the grammar.

SOURCE CODE:

```
/*For grammar E->TE', E'->+TE', T->FT', T'->*FT'*/
/*F->(E)/id*/
#include<stdio.h>
#include<stdlib.h>
int i=0;
char a[10];
void e();
void e1();
void t();
void t1();
void f();
int main()
{
        printf("Enter the string:");
        scanf("%s",a);
        e();
        if(a[i]=='$')
                printf("Succesful Parse");
        else
                printf("Unsuccessful Parse");
        return 0;
}
void e(void)
{
        t();
        e1();
}
void e1(void)
{
        if(a[i]=='+')
        {
                i++;
                t();
                e1();
        }
}
void t(void)
{
        f();
        t1();
}
void t1(void)
{
        if(a[i]=='*')
        {
                i++;
                f();
                t1();
        }
}
void f(void)
{
        if(a[i]=='(')
        {
                i++;
                e();
                if(a[i]==')')
                        i++;
                else
                {
                        printf("')' expected\n");
                        exit(0);
                }
        }
        else if(a[i]=='i')
    i++;
        else
        {
                printf("Invalid Expresion\n");
                exit(0);
        }}
```

OUTPUT: Enter the string i+i$

Successful Parse

6b. LL(1) PARSER

AIM:. C program for LL(1) PARSER

HARDWARE REQUIREMENTS: Pentium III, 256MBRAM, 80GB HDD,keyboard, mouse, monitor.

SOFTWARE REQUIREMENTS: Windows XP, Turbo C.

DESCRIPTION:

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right endmarker to indicate the end of the input string. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $. The parsing table is a two dimensional array M[A,a] where A is a nonterminal, and a is a terminal or the symbol $. The parser is controlled by a program that behaves as follows. The program considers X, the symbol on the top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1 If X= a=$, the parser halts and announces successful completion of parsing.
2 If X=a!=$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3 If X is a nonterminal, the program consults entry M[X,a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example, M[X,a]={X->UVW}, the parser replaces X on top of the stack by WVU(with U on top). As output, we shall assume that the parser just prints the production used; any other code could be executed here. If M[X,a]=error, the parser calls an error recovery routine.

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
char l[20]={'E','G','G','T','U','U','F','F'};
char r[20][20]={"TG","+TG","@","FU","*FU","@","(E)","i"};
char x[5]={'E','G','T','U','F'};
char y[6]={'i','+','*','(',')','$'};
int tab[5][6],np[10],ra=100,z;
static char temp[10];
char ch[20],t1[10];
int i,j,k=0,m,n,p,h,m1,b=0;
void FIRST(char);
void FOLLOW(char);
void main()
{
```

```
clrscr();
printf("\n\n THE GRAMMER IS...:\n\n");
for(i=0;i<8;i++)
{
        printf("%c----->",l[i]);
        for(j=0;j<4;j++)
           printf("%c",r[i][j]);
        printf("\n");
}
for(p=0;p<5;p++)
{
       for(m=0;m<8;m++)
       {   if(x[p]==l[m])
           {
                  n=m;
                  break;
           }
       }
       k=0;ra=100;z=0;
       FIRST(x[p]);
       for(j=0;j<k;j++)
       {
            for(h=0;h<6;h++)
            {
                   if(temp[j]==y[h]&&j<ra)
                   {
                        tab[p][h]=n+1;
                        if(r[n][0]==temp[j])
                               tab[p][h]=n+1;
                        else if(r[n+1][0]==temp[j])
                               tab[p][h]=n+2;
                   }
                   if(temp[j]==y[h]&&j>=ra)
                   {
                        tab[p][h]=n+2;
                        if(r[n][0]==temp[j])
                               tab[p][h]=n+2;
                        else if(r[n+1][0]==temp[j])
                               tab[p][h]=n+3;
                   }
            }
       }
}
printf("\n");
for(i=0;i<6;i++)
       printf("\t %c",y[i]);
```

```
    for(i=0;i<5;i++)
    {
            printf("\n\t %c",x[i]);
            for(j=0;j<6;j++)
                printf("%d \t",tab[i][j]);
    }
    getch();
    return 0;
}
void FOLLOW(char c)
{
    for(i=0;i<8;i++)
    {
       for(j=0;j<8;j++)
       {
              if(c==r[i][j])
              {
                 if(r[i][j+1]>='A' && r[i][j+1]<='Z')
                         FIRST(r[i][j+1]);
                 else if(r[i][j+1]=='\0')
                         FOLLOW(l[i]);
                 else
                 {
                         temp[k]=r[i][j+1];
                         k++;
                 }
              }
       }
    }
    if(c==l[0])
    {
            temp[k]='$';
            k++;
    }
}
void FIRST(char c)
{
    for(i=0;i<20;i++)
    {
            if(c==l[i])
            {
                if(r[i][0]>='A' && r[i][0]<='Z')
                          FIRST(r[i][0]);
                else if(r[i][0]=='@')
                {
                          if(z==0)
```

```
                    {
                        ra=k;
                        z++;
                    }
                    FOLLOW(l[i]);
                }
            }
            else
            {
                    temp[k]=r[i][0];
                    k++;
            }
    }}
```

OUTPUT:

```
The grammar is E→TG
                          G→+TG
                           G→@
                           T→FU
                           U→*FU
                            U→@
                            F→(E)
                             F→i
```

```
         I     +     *     (     )     $
E        1     0     0     1     0     0
G        0     2     0     0     3     3
T        4     0     0     4     0     0
U        0     6     5     0     6     6
F        8     0     0     7     0     0
```

7. SLR parser to parse a given string.

AIM: C program for SLR parser to parse a given string.

HARDWARE REQUIREMENTS: Pentium III, 256MBRAM, 80GB HDD, keyboard, mouse, monitor.

SOFTWARE REQUIREMENTS: Windows XP, Turbo C.

DESCRIPTION:

A Grammar is said to be SLR(1) if and only if, for any state *s* in the SLR(1) automaton, the following conditions are met:
1. For any item *A* -> *a.Xb* in *s* (*X* is a terminal), there is no complete item *B* -> *y.* in *s* with *X* in the follow set of *B*. Violation of this rule is a **Shift-Reduce Conflict**.
2. For any two complete items *A* -> *a.* and *B* -> *b.* in *s*, *Follow(A)* and *Follow(B)* are disjoint (their intersection is the empty set). Violation of this rule is a **Reduce-Reduce Conflict**.

A grammar is said to be SLR(1) if the following Simple LR parser algorithm results in no ambiguity.

1. If state *s* contains any time of the form *A* -> *a.Xb*, where *X* is a terminal, and *X* is the next token in the input string, then the action is to shift the current input token onto the stack, and the new state to be pushed on the stack is the state containing the item *A* -> *aX.b*.
2. If state *s* contains the complete item *A* -> *y.*, and the next token in the input string is in *Follow(A)*, then the action is to reduce by the rule *A* -> *y*. A reduction by the rule *S'* -> *S*, where *S* is the start state, is equivalent to acceptance; this will happen only if the next input token is *$*. In all other cases, the new state in computed as follows. Remove the string *y* and all of its corresponding states from the parsing stack. Correspondingly, back up in the DFA to the state from which the construction of *y* began. By construction, this state must contain an item of the form *B* -> *a.Ab*. Push *A* on to the stack, and push the state containing the item *B* -> *aA.b*.
3. If the next input token is such that neither of the above two cases apply, an error is declared.

SOURCE CODE:

```
int axn[][6][2]= { {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
                   {{-1,-1},{100,6},{-1,-1},{-1,-1},{-1,-1},{102,102}},
                   {{-1,-1},{101,2},{100,7},{-1,-1},{101,2},{101,2}},
                   {{-1,-1},{101,4},{101,4},{-1,-1},{101,4},{101,4}},
                   {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
                   {{-1,-1},{101,6},{101,6},{-1,-1},{101,6},{101,6}},
                   {{100,5},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1}},
                   {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
                   {{-1,-1},{100,6},{-1,-1},{-1,-1},{100,11},{-1,-1}},
                   {{-1,-1},{101,1},{100,7},{-1,-1},{101,1},{101,1}},
                   {{-1,-1},{101,3},{101,3},{-1,-1},{101,3},{101,3}},
                   {{-1,-1},{101,5},{101,5},{-1,-1},{101,5},{101,5}}
                    };//axn Table
int gotot[12][3]={1,2,3,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,2,3,-1,-1,-1,
          -1,9,3,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};  //GoTo table
int a[10];char b[10];int top=-1,btop=-1,i;
void push(int k)
{
 if(top<9)
 a[++top]=k;
}
 void pushb(char k)
{
if(btop<9)
b[++btop]=k;
}
 char TOS()
{
return a[top];
}
void pop()
{
if(top>=0)
top--;
}
void popb()
{
if(btop>=0)
b[btop--]='\0';
}
void display()
{
for(i=0;i<=top;i++)
 printf("%d%c",a[i],b[i]);
}
 void display1(char p[],int m) //Displays The Present Input String
{
int l;
```

```
for(l=m;p[l]!='\0';l++)
{
   printf("%c",p[l]);
}
  printf("\n");
}
void error()
{
 printf("Syntax Error");
}
 void reduce(int p)
{
int len,k,ad;
char src,*dest;
switch(p)
 {
case 1:dest="E+T";
              src='E';
              break;
case 2:dest="T";
       src='E';
       break;
case 3:dest="T*F";
       src='T';
       break;
case 4:dest="F";
       src='T';
       break;
case 5:dest="(E)";
       src='F';
       break;
case 6:dest="i";
       src='F';
       break;
default:dest="\0";
             src='\0';
             break;
 }
for(k=0;k<strlen(dest);k++)
 {
 pop();
 popb();
 }
  pushb(src);
   switch(src)
   {
case 'E':ad=0;
              break;
case 'T':ad=1;
              break;
```

```
case 'F':ad=2;
              break;
default: ad=-1;
              break;
  }
 push(gotot[TOS()][ad]);}
int main()
{
 int j,st,ic;
 char ip[20]="\0",an;
 clrscr();
 printf("Enter any String");
 gets(ip);
 push(0);
 display();
 printf("\t%s\n",ip);
 for(j=0;ip[j]!='\0';)
   {
             st=TOS();
             an=ip[j];
             if(an>='a'&&an<='z')
             ic=0;
             else if(an=='+')
             ic=1;
             else if(an=='*')
             ic=2;
             else if(an=='(')
             ic=3;
             else if(an==')')
             ic=4;
             else if(an=='$')
             ic=5;
             else
             {
               error();
                break;
             }
 if(axn[st][ic][0]==100)
    {
 pushb(an);
 push(axn[st][ic][1]);
 display();
 j++;

 display1(ip,j);
 }
if(axn[st][ic][0]==101)
   {
  reduce(axn[st][ic][1]);
      display();
```

```
            display1(ip,j);
   }


 if(axn[st][ic][1]==102)
   {
  printf("Given String is accepted\n");
     break;

   }

}

            getch();
            return 0;
            }
```

OUTPUT:

```
Enter the string        i+i$

0                     i+i$

0i5                  +i$

0F3                  +i$

0T2                  +i$

0E1                  +i$

0E1+6                 i$

0E1+6i5               $

0E1+6F3               $

0E1+6T9               $

0E1                   $

Given string is accepted.
```

8. Parser generator using YACC(Calculator)

AIM: Parser generator using YACC(Calculator)

HARDWARE REQUIREMENTS: Pentium III, 256MBRAM, 80GB HDD,keyboard, mouse, monitor.

SOFTWARE REQUIREMENTS: Windows XP, Redhat Linux with in built lex and YACC.

DESCRIPTION: YACC generates a parser (the part of a compiler that tries to make syntactic sense of the source code) based on an analytic grammar written in a notation similar to BNF. Yacc generates the code for the parser in the C programming language. The parser generated by yacc requires a lexical analyzer.

SOURCE CODE:

Lex program:

```
%{
#include "y.tab.h"
#include<math.h>
%}
%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {yylval.dval=atof(yytext);
return NUMBER;
}
log|LOG {return LOG;}
ln {return nLOG;}
sin|SIN {return SINE;}
cos|COS {return COS;}
tan|TAN {return TAN;}
mem {return MEM;}
[\t];
\$; {return 0;}
\n|. {return yytext[0];}
%%
```

yacc program

```
%{
#include<stdio.h>
#include<math.h>
double memvar;
%}
%union
```

```
{
double dval;
}
%token<dval>NUMBER
%token<dval>MEM
%token LOG SINE nLOG COS TAN
%left '-''+'
%left '*''/'
%right '^'
%left LOG SINE nLOG COS TAN
%nonassoc UMINUS
%type<dval> expression
%%
start: statement '\n'
|start statement '\n'
;
statement: MEM'='expression { memvar=$3;}
|expression {printf("answer=%g\n",$1);}
;
expression:expression'+'expression {$$=$1+$3;}
|expression'-'expression {$$=$1+$3;}
|expression'*'expression {$$=$1*$3;}
|expression'/'expression
{
if($3==0)
yyerror("divide by zero");
else
$$=$1/$3;}
|expression'^'expression {$$=pow($1,$3);}
;
expression: '-' expression %prec UMINUS {$$=-$2;}
|'('expression')' {$$=$2;}
|LOG expression {$$=log($2)/log(10);}
|nLOG expression {$$=log($2);}
|SINE expression {$$=sin($2*3.14159/180);}
|COS expression {$$=cos($2*3.14159/180);}
|TAN expression {$$=tan($2*3.14159/180);}
|NUMBER { $$ = $1;}
|MEM {$$=memvar;}
;
%%
main()
{
printf("enter expression:");
yyparse();
}
```

```
int yyerror(char *error)
{
fprintf(stderr,"%s\n",error);
}
yywrap()
{  return 1;
}
```

OUTPUT:

```
$ lex lex.l
$ yacc –d yacc.y
$cc lex.yy.c y.tab.c –ll –ly –lm
$./a.out
2+3
Answer=5
SIN 90
Answer=1
```

9. Three address code generation

AIM: Write a C program for code generation

HARDWARE REQUIREMENTS: Pentium III, 256MBRAM, 80GB HDD,keyboard, mouse, monitor.

SOFTWARE REQUIREMENTS: Windows XP, Turbo C.

DESCRIPTION: Three-address code is a sequence of statements of the general form
X:= Op Z
where x, y, and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean-valued data. A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are quadruples, triples, and indirect triples.

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
int tc[10],fb=0,i=0,j=0,k=0,fstar=0,c=-1,c1=0,c2=0,t1,t2;
char m[30],temp[30];
void main(){
int a,d;
clrscr();
for(i=0;i<10;i++)
tc[i]=-1;
printf("\n Code stmt evaluation follow following precedence: ");
printf("\n   1.( ) within the () stmt should be of the form:  x op z");
printf("\n   2.*,/ equal precedence");
printf("\n   3.+,- equal precedence");
printf("\n Enter ur Code Stmt-");
gets(m);
i=0;
while(m[i]!='\0'){
        if(m[i++]=='('){
        fb++;
        break;
        }
        }
i=0;
printf("\nThe Intermediate Code may generated as-");
```

```
if(fb==1){              /* evaluating sub exp */
        while(m[i]!='\0')
                if(m[i]=='('){
                temp[j++]='T';
                tc[k]=k++;
                printf("\nT%d=",k-1);
                i++;
                while(m[i]!=')')
                printf("%c",m[i++]);
                i++;
                }
                else if(m[i]!='(')
                temp[j++]=m[i++];
        if(fb==1){
        temp[j]='\0';
        for(i=0;temp[i]!='\0';i++)
        m[i]=temp[i];
        m[i]='\0';
        }

}             /* end of evluating sub exp */

a=operatormajid('*','/');     /* operator fun call depends on priority */
d=operatormajid('+','-');
if(a==0&&d==0&&m[1]=='=')
        printf("\n%s%d",m,k-1);
getch();
}
/*  ******************************     */
operatormajid(char haj,char haj1){     /* function to evaluate operators */
m1:     for(i=0;m[i]!='\0';i++)
        if(m[i]==haj||m[i]==haj1){
        fstar++;
        break;
        }
        if(fstar==1){
        for(j=0;j<i;j++)
        if(m[j]=='T')c++;
        printf("\nT%d=",k);
        if(m[i-1]=='T'&&m[i+1]=='T'){
        printf("%c%d%c%c%d",m[i-1],tc[c],m[i],m[i+1],tc[c+1]);
        tc[c]=k++;
        for(t2=c+1;t2<9;t2++)
        tc[t2]=tc[t2+1];
        }
        else if(m[i-1]!='T'&&m[i+1]!='T'){
        printf("%c%c%c",m[i-1],m[i],m[i+1]);
        if(c==-1){
        for(t1=9;t1>0;t1--)
        tc[t1]=tc[t1-1];
        tc[0]=k++;
        }
        else if(c>=0){
        for(t1=9;t1>c+1;t1--)
        tc[t1]=tc[t1-1];
        tc[t1]=k++;
        }
        }
        else if(m[i-1]=='T'&&m[i+1]!='T'){
        printf("%c%d%c%c",m[i-1],tc[c],m[i],m[i+1]);
        tc[c]=k++;
        }
        else if(m[i-1]!='T'&&m[i+1]=='T'){
        printf("%c%c%c%d",m[i-1],m[i],m[i+1],tc[c+1]);
        tc[c+1]=k++;
        }
        for(t1=0;t1<i-1;t1++)
        temp[t1]=m[t1];
        temp[t1++]='T';
        for(t2=i+2;m[t2]!='\0';t2++)
        temp[t1++]=m[t2];
        temp[t1++]='\0';

        fstar=0;
        for(i=0;temp[i]!='\0';i++)
        m[i]=temp[i];
        m[i]='\0';
        c=-1;
        goto m1;
        }
        else return 0;
        }
```

OUTPUT:

```
Code stmt evaluation follow following precedence:
 1.( ) within the () stmt should be of the form:  x op z
 2.*,/ equal precedence
 3.+,- equal precedence
 Enter ur Code Stmt

        a+(b*c)-d/(b*c)

  T0=b*C
  T1=b*C
  T2=d/T1
  T3=a+T0
  T4=T3-T2
```

10. YACC program for generate Quadruple

AIM: Write a YACC program for generate Quadruple

HARDWARE REQUIREMENTS: Pentium III, 256MBRAM, 80GB HDD,keyboard, mouse, monitor.

SOFTWARE REQUIREMENTS: Windows XP, Redhat Linux with in built lex and YACC.

DESCRIPTION: A quadruple is a record structure with four fields, which we call op, arg l, arg 2, and result. The op field contains an internal code for the operator. The three-address statement x:= y op z is represented by placing y in arg 1. z in arg 2. and x in result. Statements with unary operators like x: = – y or x: = y do not use arg 2. Operators like param use neither arg2 nor result. Conditional and unconditional jumps put the target label in result

SOURCE CODE:

/* lex program */

```
%{
#include "y.tab.h"
extern char yyval;
%}
number [0-9]+
letter [a-zA-Z]+
%%
{number} {yylval.sym=(char)yytext[0];return number;}
{letter} {yylval.sym=(char)yytext[0]; return letter; }
\n {return 0;}
.  {return yytext[0];}
%%
```

/*yaac program */

```
%{
#include<stdio.h>
#include<string.h>
int nIndex=0;
struct Intercode
{
char operand1;
char operand2;
char opera;
};
```

```
%}
%union
{
char sym;
}
%token <sym> letter number
%type  <sym> expr
%left '-' '+'
%right '*' '/'
%%
Statement: letter '=' expr ';' { addtotable((char)$1,(char)$3,'=' ); }
              | expr ;
              ;
expr: expr '+' expr  { $$=addtotable((char)$1,(char)$3,'+');}
        | expr '-'  expr { $$=addtotable((char)$1,(char)$3,'-');}
        | expr '*'  expr { $$=addtotable((char)$1,(char)$3, '*');}
        | expr '/'  expr { $$=addtotable((char)$1,(char)$3,'/');}
        | '(' expr ')' { $$= (char)$2;}
        |  number  { $$= (char)$1;}
        | letter { $$= (char)$1;}
%%

yyerror(char *s)
{
printf("%s",s);
exit (0);
}
struct Intercode code[20];
char addtotable(char operand1, char operand2,char opera)
{
char temp = 'A';
code[nIndex].operand1 = operand1;
code[nIndex].operand2 = operand2;
code[nIndex].opera = opera;
nIndex++;
temp++;
return temp;
}
threeaddresscode()
{
int nCnt=0;
char temp='A';
printf("\n\n\t three addrtess codes\n\n");
temp++;
while(nCnt<nIndex)
{
```

```
printf("%c:=\t",temp);
if (isalpha(code[nCnt].operand1))
printf("%c\t", code[nCnt].operand1);
else
printf("%c\t",temp);
printf("%c\t", code[nCnt].opera);
if (isalpha(code[nCnt].operand2))
printf("%c\t", code[nCnt].operand2);
else
printf("%c\t",temp);
printf("\n");
nCnt++;
temp++;
}
}
void quadruples()
{
int nCnt=0;
char temp = 'A';
temp++;
printf("\n\n\t Quardruples \n");
printf("\n ID OPERATOR OPERAND1 OPERAND2\n");
while(nCnt<nIndex)
{
printf("\n (%d) \t %c \t",nCnt,code[nCnt].opera);
if(isalpha(code[nCnt].operand1))
printf("%c\t", code[nCnt].operand1);
else
printf("%c\t",temp);
printf("%c\t", code[nCnt].opera);
if(isalpha(code[nCnt].operand2))
printf("%c\t", code[nCnt].operand2);
else
printf("%c\t",temp);
printf("%c\t",temp);
printf("\n");
nCnt++;
temp++;
}
}
main()
{
printf("enter expression");
yyparse();
threeaddresscode();
quadruples();
```

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

OUTPUT:

```
$ lex lexfile.l
    $ yacc –d  yaccfile.y
    $cc lex.yy.c y.tab.c –ll –ly –lm
    $./a.out
    Enter expr a+b*c+d
    Three address code
    B=b*c
    C=a+B
    D=B+d

    ID    OPERATOR         OPERAND1     OPERAND2
    (0)        *             b                  c            B
    (1)        +             a                  B            C
    (2)        +             B                  d            D
```

11.Machine Code Generation

AIM: Write a C Program to machine   a code for given three address code.

HARDWARE REQUIREMENTS: Pentium III, 256MBRAM, 80GB HDD,keyboard, mouse, monitor.

SOFTWARE REQUIREMENTS: Windows XP, Turbo C.

DESCRIPTION:

SOURCE CODE:

```
/* program to generate mechine code*/
 #include<stdio.h>
 #include<conio.h>
 #include<stdlib.h>
 #include<string.h>
 int label[20],no=0;
 int main()
 {   FILE *fp1,*fp2;
     int check_label(int n);
     char fname[100],op[10],ch;
     char op1[8],op2[8],res[8];
     int i=0; int j=0;
     clrscr();
    // printf("testing");
     printf("\n enter filename of intermediate code:");
     scanf("%s",&fname);
    // printf("%s",fname);

     fp1=fopen(fname,"r");
     fp2=fopen("target.txt","w");
     if(fp1==NULL||fp2==NULL)
     {
      printf("\n error in opening files....");
      getch();
      exit(0);
     }
     while(!feof(fp1))
     {
       fprintf(fp2,"\n");
       fscanf(fp1,"%s",op);
       i++;
       if(check_label(i))
         fprintf(fp2,"\n label.# %d:",i);
       if(strcmp(op,"printf")==0)
       {  fscanf(fp1,"%s",res);
```

```
   fprintf(fp2,"\n\t OUT %s",res);
}
if(strcmp(op,"goto")==0)
{  fscanf(fp1,"%s%s",op1,op2);
   fprintf(fp2,"\n\t JMP %s label.# %s",op1,op2);
   label[no++]=atoi(op2);
}
if(strcmp(op,"[]=")==0)
{  fscanf(fp1,"%s%s%s",op1,op2,res);
   fprintf(fp2,"\n\t STORE %s[%s],%s",op1,op2,res);
}
if(strcmp(op,"uminus")==0)
{   fscanf(fp1,"%s%s",op1,res);
    fprintf(fp2,"\n\t LOAD %s, R1",op1);
    fprintf(fp2,"\n\t STORE R1,%s",res);
}
switch(op[0])
{  case '*': fscanf(fp1,"%s%s%s",op1,op2,res);
               fprintf(fp2,"\n\t LOAD %s,R0",op1);
               fprintf(fp2,"\n\t LOAD %s,R1",op2);
               fprintf(fp2,"\n\t MUL R1,R0");
               fprintf(fp2,"\n\t STORE R0,%s",res);
               break;
   case '+': fscanf(fp1,"%s%s%s",op1,op2,res);
               fprintf(fp2,"\n\t LOAD %s,R0",op1);
               fprintf(fp2,"\n\t LOAD %s,R1",op2);
               fprintf(fp2,"\n\t ADD R1,R0");
               fprintf(fp2,"\n\t STORE R0,%s",res);
               break;
   case '-': fscanf(fp1,"%s%s%s",op1,op2,res);
               fprintf(fp2,"\n\t LOAD %s,R0",op1);
               fprintf(fp2,"\n\t LOAD %s,R1",op2);
               fprintf(fp2,"\n\t SUB R1,R0");
               fprintf(fp2,"\n\t STORE R0,%s",res);
               break;
   case '/': fscanf(fp1,"%s%s%s",op1,op2,res);
               fprintf(fp2,"\n\t LOAD %s,R0",op1);
               fprintf(fp2,"\n\t LOAD %s,R!",op2);
               fprintf(fp2,"\n\t DIV R1,R0");
               fprintf(fp2,"\n\t STORE R0,%s",res);
               break;
   case '%': fscanf(fp1,"%s%s%s",op1,op2,res);
               fprintf(fp2,"\n\t LOAD %s,R0",op1);
               fprintf(fp2,"\n\t LOAD %s,R1",op2);
               fprintf(fp2,"\n\t DIV R1,R0");
               fprintf(fp2,"\n\t STORE R0,%s",res);
```

```
                         break;
           case '=': fscanf(fp1,"%s%s",op1,res);
                         fprintf(fp2,"\n\t STORE %s %s",op1,res);
                         break;
           case '>': j++;
                         fscanf(fp1,"%s%s%s",op1,op2,res);
                         fprintf(fp2,"\n\t LOAD %s,R0",op1);
                         fprintf(fp2,"\n\t JGT %s,label.# %s",op2,res);
                         label[no++]=atoi(res);
                         break;
      }
    }
    fclose(fp2);
    fclose(fp1);
    fp2=fopen("target.txt","r");
    if(fp2==NULL)
    {  printf("\n error in opening file target.txt");
     getch();
     exit(0);
    }
    do{  ch=fgetc(fp2);
        printf("%c",ch);
        }while(ch!=EOF);
        fclose(fp2);
    getch();
    return 0;
}
int check_label(int k)
{
 //printf("in check_label");
   int i;
   for(i=0;i<no;i++)
   {  if(k==label[i])
      return 1;
   }
   return 0;
}
```

Input (Target.c)

```
= t1  2
[]=a 01
[]=a 12
[]=a 23
*t1 6 t2
+  a[2] t2  t3
```

```
-   a[t2]  t1  t2
/  t3  t2  t2
uminus t2 t2
>  t2 5 11
goto  t2 13
=  t3 99
+  * t1  t3  t4
print  t4
```

OUTPUT:

```
target. txt
STORE t2, 2
STORE  a[0], 1
STORE a[1], 2
STORE a[2], 3
LOAD t1, R0
LOAD 6, R1
MUL R1, R0
STORE  R0, t2
LOAD a[t2], R0
LOAD t2, R,
ADD R1, R0
STORE R0, T3
LOAD a[t2] , R0
LOAD t1, R1
SUB R1, R0
STORE R0, t2
LOAD t3, R0
LOAD t2, R1
DIV R1, R0
STORE R0, t2
LOAD t2, t1
UMINUS R1
STORE  R1, t2
LOAD t2, R0
JGT 5, lable#, 11
lable #11:
JMP t2 lable # t3
STORE  t3, 99
LOAD t1, R0
LOAD t3, R1
ADD R1, R0
STORE R0, t4
OUT t4
```

12. Code optimization

AIM: Write a C Program to implement a code optimization method “common sub expression elimination”

HARDWARE REQUIREMENTS: Pentium III, 256MBRAM, 80GB HDD,keyboard, mouse, monitor.

SOFTWARE REQUIREMENTS: Windows XP, Turbo C.

DESCRIPTION: An "optimization" must not change the output produced by a program for a given input, or cause an error, such as a division by zero, that was not present in the original version of the source program, speed up programs by a measurable amount, and it must be worth the effort. On of the Optimization technique is eliminate Common sub expressions. Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where its referenced when encountered again – of course providing the variable values in the expression still remain constant.

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
int tc[10],fb=0,i=0,j=0,k=0,p=0,fstar=0,c=-1,c1=0,c2=0,t1,t2,t3,t4,fo=0;
char m[30],temp[30],opt[10][4];
void main(){
int a,d;
clrscr();
for(i=0;i<10;i++)
tc[i]=-1;
printf("\n Code stmt evaluation follow following precedence: ");
printf("\n   1.( ) within the () stmt should be of the form:  x op z");
printf("\n   2.*,/ equal precedence");
printf("\n   3.+,- equal precedence");
printf("\n Enter ur Code Stmt-");
gets(m);
i=0;
while(m[i]!='\0'){
        if(m[i++]=='('){
        fb++;
        break;
        }
        }
i=0;
printf("\nThe Intermediate Code may generated as-");
if(fb==1){                              /* evaluating sub exp */
        while(m[i]!='\0')
                if(m[i]=='('){
                temp[j++]='T';
                i++;
                t3=i;                   /* optimising the code */
           while(m[i]!=')')
                opt[c1][c2++]=m[i++];
                for(t4=c1-1;t4>=0;t4--)
                if(strcmp(opt[c1],opt[t4])==0){
                tc[p++]=t4;
                fo=1;
                }                       /* end of optimising   */
                if(fo==0){
                tc[p++]=k++;
           printf("\nT%d=",k-1);
                while(m[t3]!=')')
                printf("%c",m[t3++]);
                }
                i++;
                c1++;
                c2=fo=0;
                }
                else if(m[i]!='(')
                temp[j++]=m[i++];
        if(fb==1){
      temp[j]='\0';
        for(i=0;temp[i]!='\0';i++)
        m[i]=temp[i];
      m[i]='\0';
        }

}                    /* end of evluating sub exp */
a=operatormajid('*','/');    /* operator fun call depends on priority */
d=operatormajid('+','-');
if(a==0&&d==0&&m[1]=='=')
        printf("\n%s%d",m,k-1);

getch();
}
/*  ****************************     */
operatormajid(char haj,char haj1){    /* function to evaluate operators */
m1:     for(i=0;m[i]!='\0';i++)
        if(m[i]==haj||m[i]==haj1){
        fstar++;
        break;
        }
        if(fstar==1){
        for(j=0;j<i;j++)
        if(m[j]=='T')c++;
        printf("\nT%d=",k);
        if(m[i-1]=='T'&&m[i+1]=='T'){
        printf("%c%d%c%c%d",m[i-1],tc[c],m[i],m[i+1],tc[c+1]);
      tc[c]=k++;
        for(t2=c+1;t2<9;t2++)
        tc[t2]=tc[t2+1];
        }
        else if(m[i-1]!='T'&&m[i+1]!='T'){
        printf("%c%c%c",m[i-1],m[i],m[i+1]);
        if(c==-1){
        for(t1=9;t1>0;t1--)
        tc[t1]=tc[t1-1];
        tc[0]=k++;
        }
        else if(c>=0){
        for(t1=9;t1>c+1;t1--)
        tc[t1]=tc[t1-1];
        tc[t1]=k++;
}
        }
        else if(m[i-1]=='T'&&m[i+1]!='T'){
        printf("%c%d%c%c",m[i-1],tc[c],m[i],m[i+1]);
        tc[c]=k++;
        }
        else if(m[i-1]!='T'&&m[i+1]=='T'){
        printf("%c%c%c%d",m[i-1],m[i],m[i+1],tc[c+1]);
        tc[c+1]=k++;
        }
        for(t1=0;t1<i-1;t1++)
        temp[t1]=m[t1];
        temp[t1++]='T';
        for(t2=i+2;m[t2]!='\0';t2++)
        temp[t1++]=m[t2];
        temp[t1++]='\0';

        fstar=0;
```

```
    for(i=0;temp[i]!='\0';i++)
       m[i]=temp[i];
       m[i]='\0';
       c=-1;
       goto m1;
       }
       else return 0;
       }
```

OUTPUT:

```
Code stmt evaluation follow following precedence:
 1.( ) within the () stmt should be of the form:  x op z
 2.*,/ equal precedence
 3.+,- equal precedence
 Enter ur Code Stmt

        a+(b*c)-d/(b*c)

   T0=b*C
   T1= d/T1
   T2=a+T0
   T3=T2-T1
```