Course Objectives: To enlighten the student with knowledge base in compiler design and its applications

Course Outcomes: The end of the course student will be able to

- x Design simple lexical analyzers
- x Determine predictive parsing table for a CFG
- x Apply Lex and Yacc tools
- x Examine LR parser and generating SLR Parsing table
- x Relate Intermediate code generation for subset C language

List of Experiments:

1. Write a C program to identify different types of Tokens in a given Program.

2. Write a Lex Program to implement a Lexical Analyzer using Lex tool.

3. Write a C program to Simulate Lexical Analyzer to validating a given input String.

4. Write a C program to implement the Brute force technique of Top down Parsing.

5. Write a C program to implement a Recursive Descent Parser.

6. Write C program to compute the First and Follow Sets for the given Grammar.

7. Write a C program for eliminating the left recursion and left factoring of a given grammar

8. Write a C program to check the validity of input string using Predictive Parser.

9. Write a C program for implementation of LR parsing algorithm to accept a given input string.

10. Write a C program for implementation of a Shift Reduce Parser using Stack Data Structure to accept a given input string of a given grammar.

11. Simulate the calculator using LEX and YACC tool.

12. Generate YACC specification for a few syntactic categories.

13. Write a C program for generating the three address code of a given expression/statement.

14. Write a C program for implementation of a Code Generation Algorithm of a given expression/statement.

CONTENTS

1.Write a c program to identify different types of tokens in a given program

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

// Returns 'true' if the character is a DELIMITER.
boolisDelimiter(char ch)
{
	if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
		ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
		ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
		ch == '[' || ch == ']' || ch == '{' || ch == '}')
		return (true);
	return (false);
}

// Returns 'true' if the character is an OPERATOR.
boolisOperator(char ch)
{
	if (ch == '+' || ch == '-' || ch == '*' ||
		ch == '/' || ch == '>' || ch == '<' ||
		ch == '=')
		return (true);
	return (false);
}

// Returns 'true' if the string is a VALID IDENTIFIER.
boolvalidIdentifier(char* str)
{
	if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
		str[0] == '3' || str[0] == '4' || str[0] == '5' ||
		str[0] == '6' || str[0] == '7' || str[0] == '8' ||
		str[0] == '9' || isDelimiter(str[0]) == true)
		return (false);
	return (true);
}

// Returns 'true' if the string is a KEYWORD.
boolisKeyword(char* str)
{
	if (!strcmp(str, "if") || !strcmp(str, "else") ||
		!strcmp(str, "while") || !strcmp(str, "do") ||
		!strcmp(str, "break") ||
		!strcmp(str, "continue") || !strcmp(str, "int")
		|| !strcmp(str, "double") || !strcmp(str, "float")
		|| !strcmp(str, "return") || !strcmp(str, "char")
		|| !strcmp(str, "case") || !strcmp(str, "char")
		|| !strcmp(str, "sizeof") || !strcmp(str, "long")
		|| !strcmp(str, "short") || !strcmp(str, "typedef")
```

```
                || !strcmp(str, "switch") || !strcmp(str, "unsigned")
                || !strcmp(str, "void") || !strcmp(str, "static")
                || !strcmp(str, "struct") || !strcmp(str, "goto"))
                return (true);
        return (false);
}

// Returns 'true' if the string is an INTEGER.
boolisInteger(char* str)
{
        int i, len = strlen(str);

        if (len == 0)
                return (false);
        for (i = 0; i <len; i++) {
                if (str[i] != '0' &&str[i] != '1' &&str[i] != '2'
                        &&str[i] != '3' &&str[i] != '4' &&str[i] != '5'
                        &&str[i] != '6' &&str[i] != '7' &&str[i] != '8'
                        &&str[i] != '9' || (str[i] == '-' && i > 0))
                        return (false);
        }
        return (true);
}

// Returns 'true' if the string is a REAL NUMBER.
boolisRealNumber(char* str)
{
        int i, len = strlen(str);
        boolhasDecimal = false;

        if (len == 0)
                return (false);
        for (i = 0; i <len; i++) {
                if (str[i] != '0' &&str[i] != '1' &&str[i] != '2'
                        &&str[i] != '3' &&str[i] != '4' &&str[i] != '5'
                        &&str[i] != '6' &&str[i] != '7' &&str[i] != '8'
                        &&str[i] != '9' &&str[i] != '.' ||
                        (str[i] == '-' && i > 0))
                        return (false);
                if (str[i] == '.')
                        hasDecimal = true;
        }
        return (hasDecimal);
}

// Extracts the SUBSTRING.
char* subString(char* str, int left, int right)
{
        int i;
        char* subStr = (char*)malloc(
                                sizeof(char) * (right - left + 2));
```

```
        for (i = left; i <= right; i++)
                subStr[i - left] = str[i];
        subStr[right - left + 1] = '\0';
        return (subStr);
}

// Parsing the input STRING.
void parse(char* str)
{
        int left = 0, right = 0;
        intlen = strlen(str);

        while (right <= len&& left <= right) {
                if (isDelimiter(str[right]) == false)
                        right++;

                if (isDelimiter(str[right]) == true && left == right) {
                        if (isOperator(str[right]) == true)
                                printf("'%c' IS AN OPERATOR\n", str[right]);

                        right++;
                        left = right;
                } else if (isDelimiter(str[right]) == true && left != right
                                || (right == len&&left != right)) {
                        char* subStr = subString(str, left, right - 1);

                        if (isKeyword(subStr) == true)
                                printf("'%s' IS A KEYWORD\n", subStr);

                        else if (isInteger(subStr) == true)
                                printf("'%s' IS AN INTEGER\n", subStr);

                        else if (isRealNumber(subStr) == true)
                                printf("'%s' IS A REAL NUMBER\n", subStr);

                        else if (validIdentifier(subStr) == true
                                        &&isDelimiter(str[right - 1]) == false)
                                printf("'%s' IS A VALID IDENTIFIER\n", subStr);

                        else if (validIdentifier(subStr) == false
                                        &&isDelimiter(str[right - 1]) == false)
                                printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
                        left = right;
                }
        }
        return;
}

// DRIVER FUNCTION
int main()
```

```
{
        // maximum length of string is 100 here
        charstr[100] = "int a = b + 1c; ";

        parse(str); // calling the parse function

        return (0);
}
```

Output

```
'int' IS A KEYWORD
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'b' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'1c' IS NOT A VALID IDENTIFIER
```

2.Write a lex program to implement a lexical analyser using lex tool

Aim: (Tokenizing) Use Lex and yacc to extract tokens from a given source code.

Description:

- A language for specifying lexical analyzer.
- There is a wide range of tools for construction of lexical analyzer. The majority of these tools are based on regular expressions.
- The one of the traditional tools of that kind is lex.

Lex:-

- The lex is used in the manner depicted. A specification of the lexical analyzer is preferred by creating a program lex.1 in the lex language.
- Then lex.1 is run through the lex compiler to produce a 'c' program lex.yy.c.
- The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expression of lex.1 together with a standard routine that uses table of recognize leximes.
- Lex.yy.c is run through the 'C' compiler to produce as object program a.out, which is the lexical analyzer that transform as input stream into sequence of tokens.

Algorithm:

1. First, a specification of a lexical analyzer is prepared by creating a program lexp.l in the LEX language.
2. The Lexp.l program is run through the LEX compiler to produce an equivalent code in C language named Lex.yy.c
3. The program lex.yy.c consists of a table constructed from the Regular Expressions of Lexp.l, together with standard routines that uses the table to recognize lexemes.
4. Finally, lex.yy.c program is run through the C Compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

Program: lexp.l

```
%{
int COMMENT=0;
 %}
identifier [a-zA-Z][a-zA-Z0-9]*
 %%
#.* {printf ("\n %s is a Preprocessor Directive",yytext);}
int |
float |
main |
if |
else |
printf |
scanf |
for |
char |
getch |
while {printf("\n %s is a Keyword",yytext);}
"/*" {COMMENT=1;}
"*/" {COMMENT=0;}
{identifier}\( {if(!COMMENT) printf("\n Function:\t %s",yytext);}
\{ {if(!COMMENT) printf("\n Block Begins");
\} {if(!COMMENT) printf("\n Block Ends");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s is an Identifier",yytext);}
\".*\" {if(!COMMENT) printf("\n %s is a String",yytext);}
[0-9]+ {if(!COMMENT) printf("\n %s is a Number",yytext);}
\)(\;)? {if(!COMMENT) printf("\t");ECHO;printf("\n");}
\( ECHO;
 = {if(!COMMENT) printf("\n%s is an Assmt oprtr",yytext);}
\<= |
\>= |
\< |
== {if(!COMMENT) printf("\n %s is a Rel. Operator",yytext);}
.|\n
%%
int main(int argc, char **argv)
{
if(argc>1)


{
FILE *file;
file=fopen(argv[1],"r");
if(!file)
 {
printf("\n Could not open the file: %s",argv[1]);
exit(0);
}
yyin=file;
}
yylex(); printf("\n\n");
return 0;
```

```
}
int yywrap()
{
return 0;
}
```

Output:

test.c

```
#include<stdio.h>
main()
{
int fact=1,n;
for(int i=1;i<=n;i++)
{
fact=fact*i;
} printf("Factorial Value of N is", fact);
getch();
}
```

```
$ lex lexp.l
$ cc lex.yy.c
$ ./a.out test.c
#include<stdio.h>is a Preprocessor Directive
Function: main( )
Block Begins
int is a Keyword
fact is an Identifier
= is an Assignment Operator
1 is a Number
n is an Identifier
Function: for(
int is a Keyword
i is an Identifier
= is an Assignment Operator
1 is a Number
i is an Identifier
<= is a Relational Operator
n is an Identifier
i is an Identifier
);
Block Begins
fact is an Identifier
= is an Assignment Operator
fact is an Identifier
i is an Identifier
Block Ends
Function: printf(
"Factorial Value of N is" is a String
fact is an Identifier );
Function: getch( );
Block Ends
```

3.write a c program to simulate lexical analyzer to validate a given input string

```
#include<stdio.h>
void main()
{
char s[5];
//clrscr();
printf("\n Enter any operator:");
gets(s);
switch(s[0])
{
case'>': if(s[1]=='=')
printf("\n Greater than or equal");
else
printf("\n Greater than");
break;
case'<': if(s[1]=='=')
printf("\n Less than or equal");
else
printf("\nLess than");
break;
case'=': if(s[1]=='=')
printf("\nEqual to");
else
printf("\nAssignment");
break;
case'!': if(s[1]=='=')
printf("\nNot Equal");
else
printf("\n Bit Not");
break;
case'&': if(s[1]=='&')
printf("\nLogical AND");
else
printf("\n Bitwise AND");
break;
case'|': if(s[1]=='|')
printf("\nLogical OR");
else
printf("\nBitwise OR");
break;
```

```
case'+': printf("\n Addition");
break;
case'-': printf("\nSubstraction");
break;
case'*': printf("\nMultiplication");
break;
case'/': printf("\nDivision");
break;
case'%': printf("Modulus");
break;
default: printf("\n Not a operator");
}
//getch();
}
```

Output

```
Enter any operator:*
Multiplication
```

4.write a c program to implement the brute force technique of top down parsing

```
#include<stdio.h>
//#include<conio.h>
#include<iostream.h>
void main()
{
int a[30];
//clrscr();
int min=10000,temp=0,i,lev,n,noofc,z;
printf("please enter how many number");
cin>>n;
for(i=0;i<n;i++)
a[i]=0;
cout<<"enter value of root";
cin>>a[0];
for(i=1;i<=n/2;i++)
{
cout<<"please enter no of child of parent with value"<<a[i-1]<<":";
cin>>noofc;
for(int j=1;j<=noofc;j++)
{z=(i)*2+j-2;
cout<<"please enter value of child";
cin>>a[z];
}
}
```

Output

```
please enter how many number5
enter value of root4
please enter no of child of parent with value4:2
please enter value of child1
please enter value of child3
please enter no of child of parent with value1:1
please enter value of child1
temp min is5
temp min is6
temp min is7
min is5
```

5.write a c program to implement a recursvie descent parser

```
#include<stdio.h>
#include<string.h>
int E(),Edash(),T(),Tdash(),F();
char *ip;
char string[50];
int main()
{
printf("Enter the string\n");
scanf("%s",string);
ip=string;
printf("\n\nInput\tAction\n--------------------------------\n");

if(E() &&ip=="\0"){
printf("\n--------------------------------\n");
printf("\n String is successfully parsed\n");
}
else{
printf("\n--------------------------------\n");
printf("Error in parsing String\n");
}
}
int E()
{
printf("%s\tE->TE' \n",ip);
if(T())
{
if(Edash())
{
return 1;
}
else
return 0;
}
else
return 0;
}
intEdash()
{
if(*ip=='+')
{
printf("%s\tE'->+TE' \n",ip);
ip++;
```

```
if(T())
{
if(Edash())
{
return 1;
}
else
return 0;
}
else
return 0;
}
else
{
printf("%s\tE'->^ \n",ip);
return 1;
}
}
int T()
{
printf("%s\tT->FT' \n",ip);
if(F())
{

if(Tdash())
{
return 1;
}
else
return 0;
}
else
return 0;
}
intTdash()
{
if(*ip=='*')
{
printf("%s\tT'->*FT' \n",ip);
ip++;
if(F())
{
if(Tdash())
{
return 1;
}
else
return 0;
}
else
return 0;
```

```
}
else
{
printf("%s\tT'->^ \n",ip);
return 1;
}
}
int F()
{
if(*ip=='(')
{
printf("%s\tF->(E) \n",ip);
ip++;
if(E())
{
if(*ip==')')
{
ip++;
return 0;
}
else
return 0;
}
else
return 0;
}

else if(*ip=='i')
{
ip++;
printf("%s\tF->id \n",ip);
return 1;
}
else
return 0;
}
```

Output

```
Enter the string
compilerdesign
Input     Action
--------------------------------
compilerdesign  E->TE'
compilerdesign  T->FT'

--------------------------------
Error in parsing String
```

6.write a c program to compute the first and follow sets for the given grammar

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

// Functions to calculate Follow
voidfollowfirst(char, int, int);
void follow(char c);

// Function to calculate First
voidfindfirst(char, int, int);

int count, n = 0;

// Stores the final result
// of the First Sets
charcalc_first[10][100];

// Stores the final result
// of the Follow Sets
charcalc_follow[10][100];
int m = 0;

// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
charck;
int e;

int main(intargc, char **argv)
{
        intjm = 0;
        int km = 0;
        int i, choice;
        char c, ch;
        count = 8;

        // The Input grammar
        strcpy(production[0], "E=TR");
        strcpy(production[1], "R=+TR");
        strcpy(production[2], "R=#");
```

```
strcpy(production[3], "T=FY");
strcpy(production[4], "Y=*FY");
strcpy(production[5], "Y=#");
strcpy(production[6], "F=(E)");
strcpy(production[7], "F=i");

intkay;
char done[count];
intptr = -1;

// Initializing the calc_first array
for(k = 0; k < count; k++) {
        for(kay = 0; kay< 100; kay++) {
                calc_first[k][kay] = '!';
        }
}
int point1 = 0, point2, xxx;

for(k = 0; k < count; k++)
{
        c = production[k][0];
        point2 = 0;
        xxx = 0;

        // Checking if First of c has
        // already been calculated
        for(kay = 0; kay<= ptr; kay++)
                if(c == done[kay])
                        xxx = 1;

        if (xxx == 1)
                continue;

        // Function call
        findfirst(c, 0, 0);
        ptr += 1;

        // Adding c to the calculated list
        done[ptr] = c;
        printf("\n First(%c) = { ", c);
        calc_first[point1][point2++] = c;

        // Printing the First Sets of the grammar
        for(i = 0 + jm; i < n; i++) {
                int lark = 0, chk = 0;

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

                        if (first[i] == calc_first[point1][lark])
                        {
                                chk = 1;
```

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

        // Initializing the calc_follow array
        for(k = 0; k < count; k++) {
                for(kay = 0; kay< 100; kay++) {
                        calc_follow[k][kay] = '!';
                }
        }
        point1 = 0;
        int land = 0;
        for(e = 0; e < count; e++)
        {
                ck = production[e][0];
                point2 = 0;
                xxx = 0;

                // Checking if Follow of ck
                // has already been calculated
                for(kay = 0; kay<= ptr; kay++)
                        if(ck == donee[kay])
                                xxx = 1;

                if (xxx == 1)
                        continue;
                land += 1;

                // Function call
                follow(ck);
                ptr += 1;

                // Addingck to the calculated list
                donee[ptr] = ck;
                printf(" Follow(%c) = { ", ck);
                calc_follow[point1][point2++] = ck;
```

```
                // Printing the Follow Sets of the grammar
                for(i = 0 + km; i < m; i++) {
                        int lark = 0, chk = 0;
                        for(lark = 0; lark < point2; lark++)
                        {
                                if (f[i] == calc_follow[point1][lark])
                                {
                                        chk = 1;
                                        break;
                                }
                        }
                        if(chk == 0)
                        {
                                printf("%c, ", f[i]);
                                calc_follow[point1][point2++] = f[i];
                        }
                }
                printf(" }\n\n");
                km = m;
                point1++;
        }
}

void follow(char c)
{
        int i, j;

        // Adding "$" to the follow
        // set of the start symbol
        if(production[0][0] == c) {
                f[m++] = '$';
        }
        for(i = 0; i < 10; i++)
        {
                for(j = 2;j < 10; j++)
                {
                        if(production[i][j] == c)
                        {
                                if(production[i][j+1] != '\0')
                                {
                                        // Calculate the first of the next
                                        // Non-Terminal in the production
                                        followfirst(production[i][j+1], i, (j+2));
                                }

                                if(production[i][j+1]=='\0' && c!=production[i][0])
                                {
                                        // Calculate the follow of the Non-Terminal
                                        // in the L.H.S. of the production
                                        follow(production[i][0]);
                                }
```

```
                        }
                }
        }
}

voidfindfirst(char c, int q1, int q2)
{
        int j;

        // The case where we
        // encounter a Terminal
        if(!(isupper(c))) {
                first[n++] = c;
        }
        for(j = 0; j < count; j++)
        {
                if(production[j][0] == c)
                {
                        if(production[j][2] == '#')
                        {
                                if(production[q1][q2] == '\0')
                                        first[n++] = '#';
                                else if(production[q1][q2] != '\0'
                                                && (q1 != 0 || q2 != 0))
                                {
                                        // Recursion to calculate First of New
                                        // Non-Terminal we encounter after epsilon
                                        findfirst(production[q1][q2], q1, (q2+1));
                                }
                                else
                                        first[n++] = '#';
                        }
                        else if(!isupper(production[j][2]))
                        {
                                first[n++] = production[j][2];
                        }
                        else
                        {
                                // Recursion to calculate First of
                                // New Non-Terminal we encounter
                                // at the beginning
                                findfirst(production[j][2], j, 3);
                        }
                }
        }
}

voidfollowfirst(char c, int c1, int c2)
{
        int k;
```

```
		// The case where we encounter
		// a Terminal
		if(!(isupper(c)))
			f[m++] = c;
		else
		{
			int i = 0, j = 1;
			for(i = 0; i < count; i++)
			{
				if(calc_first[i][0] == c)
					break;
			}

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

Output

First(E) = { (, i, }

First(R) = { +, #, }

First(T) = { (, i, }

First(Y) = { *, #, }

First(F) = { (, i, }

Follow(E) = { $,), }

Follow(R) = { $,), }

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

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

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

7.write a c program for eliminating the left recursion and left factoring of a given grammar

Remove left factoring of grammar

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

//Structure Declaration

struct production
{
char lf;
charrt[10];
intprod_rear;
intfl;
};
struct production prodn[20],prodn_new[20];     //Creation of object

//Variables Declaration

int b=-1,d,f,q,n,m=0,c=0;
char terminal[20],nonterm[20],alpha[10],extra[10];
char epsilon='^';

//Beginning of Main Program

void main()
{
clrscr();

  //Input of Special characters
cout<<"\nEnter the number of Special characters(except non-terminals): ";
cin>>q;
cout<<"Enter the special characters for your production: ";
for(intcnt=0;cnt<q;cnt++)
  {
cin>>alpha[cnt];
  }
```

```
  //Input of Productions

cout<<"\nEnter the number of productions: ";
cin>>n;
for(cnt=0;cnt<=n-1;cnt++)
  {
cout<<"Enter the "<< cnt+1<<" production: ";
cin>>prodn[cnt].lf;
cout<<"->";
cin>>prodn[cnt].rt;
prodn[cnt].prod_rear=strlen(prodn[cnt].rt);
prodn[cnt].fl=0;
  }

  //Condition for left factoring

for(int cnt1=0;cnt1<n;cnt1++)
  {
for(int cnt2=cnt1+1;cnt2<n;cnt2++)
   {
if(prodn[cnt1].lf==prodn[cnt2].lf)
        {
cnt=0;
int p=-1;
          while((prodn[cnt1].rt[cnt]!='\0')&&(prodn[cnt2].rt[cnt]!='\0'))
          {
if(prodn[cnt1].rt[cnt]==prodn[cnt2].rt[cnt])
           {
extra[++p]=prodn[cnt1].rt[cnt];
prodn[cnt1].fl=1;
prodn[cnt2].fl=1;
           }
else
           {
if(p==-1)
break;
else
            {
int h=0,u=0;
prodn_new[++b].lf=prodn[cnt1].lf;
strcpy(prodn_new[b].rt,extra);
prodn_new[b].rt[p+1]=alpha[c];
prodn_new[++b].lf=alpha[c];
for(int g=cnt;g<prodn[cnt2].prod_rear;g++)
prodn_new[b].rt[h++]=prodn[cnt2].rt[g];
prodn_new[++b].lf=alpha[c];
for(g=cnt;g<=prodn[cnt1].prod_rear;g++)
prodn_new[b].rt[u++]=prodn[cnt1].rt[g];
                    m=1;
break;
             }
```

```
            }
cnt++;
          }
if((prodn[cnt1].rt[cnt]==0)&&(m==0))
          {
int h=0;
prodn_new[++b].lf=prodn[cnt1].lf;
strcpy(prodn_new[b].rt,extra);
prodn_new[b].rt[p+1]=alpha[c];
prodn_new[++b].lf=alpha[c];
prodn_new[b].rt[0]=epsilon;
prodn_new[++b].lf=alpha[c];
for(int g=cnt;g<prodn[cnt2].prod_rear;g++)
prodn_new[b].rt[h++]=prodn[cnt2].rt[g];
          }
if((prodn[cnt2].rt[cnt]==0)&&(m==0))
          {
int h=0;
prodn_new[++b].lf=prodn[cnt1].lf;
strcpy(prodn_new[b].rt,extra);
prodn_new[b].rt[p+1]=alpha[c];
prodn_new[++b].lf=alpha[c];
prodn_new[b].rt[0]=epsilon;
prodn_new[++b].lf=alpha[c];
for(int g=cnt;g<prodn[cnt1].prod_rear;g++)
prodn_new[b].rt[h++]=prodn[cnt1].rt[g];
          }
c++;
           m=0;
         }
     }
   }

  //Display of Output

cout<<"\n\n*********************************";
cout<<"\n     AFTER LEFT FACTORING     ";
cout<<"\n*********************************";
cout<<endl;
for(int cnt3=0;cnt3<=b;cnt3++)
         {
cout<<"Production "<<cnt3+1<<" is: ";
cout<<prodn_new[cnt3].lf;
cout<<"->";
cout<<prodn_new[cnt3].rt;
cout<<endl<<endl;
         }

for(int cnt4=0;cnt4<n;cnt4++)
   {
if(prodn[cnt4].fl==0)
```

```
   {
cout<<"Production  "<<cnt3++<<" is: ";
cout<<prodn[cnt4].lf;
cout<<"->";
cout<<prodn[cnt4].rt;
cout<<endl<<endl;
   }
 }
getche();
}   //end of main program
```

Output

```
Enter the number of productions: 4
Enter the 1 production: S
->icts
Enter the 2 production: S
->ictSeS
Enter the 3 production: S
->a
Enter the 4 production: C
->b


**************************************
       AFTER LEFT FACTORING
**************************************
Production 1 is: S->ictR

Production 2 is: R->SeS

Production 3 is: R->s

Production  3 is: S->a

Production  4 is: C->b
```

Left recursion

```
#include<stdio.h>
#include<string.h>
void main()
{
   char input[100],l[50],r[50],temp[10],tempprod[20],productions[25][50];
int i=0,j=0,flag=0,consumed=0;
printf("Enter the productions: ");
scanf("%1s->%s",l,r);
printf("%s",r);
while(sscanf(r+consumed,"%[^|]s",temp) == 1 && consumed <= strlen(r))
 {
if(temp[0] == l[0])
 {
flag = 1;
```

```
sprintf(productions[i++],"%s->%s%s'\0",l,temp+1,l);
      }
else
sprintf(productions[i++],"%s'->%s%s'\0",l,temp,l);
consumed += strlen(temp)+1;
   }
if(flag == 1)
{
sprintf(productions[i++],"%s->ε\0",l);
printf("The productions after eliminating Left Recursion are:\n");
for(j=0;j<i;j++)
printf("%s\n",productions[j]);
   }
else
printf("The Given Grammar has no Left Recursion");
}
```

Output

```
Enter the productions: E->E+E|T
E+E|TThe productions after eliminating Left Recursion are:
E->+EE'
E'->TE'
E->ε
```

8.write a C Program for Implementation of Predictive Parser

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

```
   len=strlen(s);
   if(lastentry+1>=MAX)
      Error_Message("Symbpl table is full");
   if(lastchar+len+1>=MAX)
      Error_Message("Lexemes array is full");
   lastentry=lastentry+1;
   symtable[lastentry].token=tok;
   symtable[lastentry].lexptr=&lexemes[lastchar+1];
   lastchar=lastchar+len+1;
   strcpy(symtable[lastentry].lexptr,s);
   return lastentry;
}
/*void Initialize()
{
               struct entry *ptr;
               for(ptr=keywords;ptr->token;ptr+1)
                              insert(ptr->lexptr,ptr->token);
}*/
int lexer()
{
   int t;
   int val,i=0;
   while(1)
   {
      t=getchar();
      if(t==' '||t=='\t');
      else if(t=='\n')
         lineno=lineno+1;
      else if(isdigit(t))
      {
         ungetc(t,stdin);
         scanf("%d",&tokenval);
         return NUM;
      }
      else if(isalpha(t))
      {
         while(isalnum(t))
         {
            buffer[i]=t;
            t=getchar();
            i=i+1;
            if(i>=SIZE)
               Error_Message("Compiler error");
         }
         buffer[i]=EOS;
         if(t!=EOF)
            ungetc(t,stdin);
         val=look_up(buffer);
         if(val==0)
            val=insert(buffer,ID);
         tokenval=val;
         return symtable[val].token;
      }
      else if(t==EOF)
         return DONE;
      else
      {
```

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

```
            display(t,NONE);
            continue;
        case '/' :
            t=lookahead;
            Match(lookahead);
            display(t,NONE);
            continue;
        default :
            return;
        }
    }
}
void E()
{
    int t;
    T();
    while(1)
    {
        switch(lookahead)
        {
        case '+' :
            t=lookahead;
            Match(lookahead);
            T();
            display(t,NONE);
            continue;
        case '-' :
            t=lookahead;
            Match(lookahead);
            T();
            display(t,NONE);
            continue;
        default :
            return;
        }
    }
}
void parser()
{
    lookahead=lexer();
    while(lookahead!=DONE)
    {
        E();
        Match(';');
    }
}
int main()
{
    char ans[10];
    printf("\n Program for recursive descent parsing ");
    printf("\n Enter the expression ");
    printf("And place ; at the end\n");
    printf("Press Ctrl-Z to terminate\n");
    parser();
return 0;
}
```

Output

```
Program for recursive descent parsing
 Enter the expression And place ; at the end
Press Ctrl-Z to terminate
a*b+c;

 Identifier: a
 Identifier: b
Arithmetic Operator: *
 Identifier: c
Arithmetic Operator: +
5*7;

 Number: 5
 Number: 7
Arithmetic Operator: *
*2;
line 5, Syntax error
```

9. write a C program for implementation of LR parsing algorithm to accept a given input string.

ALGORITHM:

1. Get the input expression and store it in the input buffer.
2. Read the data from the input buffer one at the time and convert in to corresponding Non Terminal using production rules available.
3. Perform push & pop operation for LR parsing table construction.
4. Display the result with conversion of corresponding input symbols to production and production reduction to start symbol. No operation performed on the operator.

PROGRAM:

```
#include

#include

char stack[30];

int top=-1;

void push(char c)

{

top++;

stack[top]=c;

}

char pop()

{

char c;

if(top!=-1)
```

```
{
c=stack[top];
top--;
return c;
}
return'x';
}
void printstat()
{
int i;
printf("\n\t\t\t $");
for(i=0;i<=top;i++)
printf("%c",stack[i]);
}
void main()
{
int i,j,k,l;
char s1[20],s2[20],ch1,ch2,ch3;
clrscr();
printf("\n\n\t\t LR PARSING");
printf("\n\t\t ENTER THE EXPRESSION");
scanf("%s",s1);
l=strlen(s1);
j=0;
printf("\n\t\t $");
```

```
for(i=0;i<l;i++)< span="" style="box-sizing: border-box;">
{
if(s1[i]=='i' && s1[i+1]=='d')
{
s1[i]=' ';
s1[i+1]='E';
printstat(); printf("id");
push('E');
printstat();
}
else if(s1[i]=='+'||s1[i]=='-'||s1[i]=='*' ||s1[i]=='/' ||s1[i]=='d')
{
push(s1[i]);
printstat();
}
   }
printstat();
l=strlen(s2);
while(l)
{
ch1=pop();
if(ch1=='x')
{
printf("\n\t\t\t $");
```

```
break;
}
if(ch1=='+'||ch1=='/'||ch1=='*'||ch1=='-')
{
ch3=pop();
if(ch3!='E')
{
printf("errror");
exit();
}
else
{
push('E');
printstat();
}
}
ch2=ch1;
}
getch();
} </l;i++)<>
```

OUTPUT:

```
LR PARSING
ENTER THE EXPRESSION
```

```
id+id*id-id
$
$id
$E
$E+
$E+id
$E+E
$E+E*
$E+E*id
$E+E*E
$E+E*E-
$E+E*E-id
$E+E*E-E
$E+E*E-E
$E+E*E
$E
$
```

10.Write a C program for implementation of a Shift Reduce Parser using Stack Data Structure to accept a given input string of a given grammar.

ALGORITHM:

1. Get the input expression and store it in the input buffer.
2. Read the data from the input buffer one at the time.
3. Using stack and push & pop operation shift and reduce symbols with respect to production rules available.
4. Continue the process till symbol shift and production rule reduce reaches the start symbol.
5. Display the Stack Implementation table with corresponding Stack actions with input symbols.

PROGRAM:

```
#include

#include

#include

#include

char ip_sym[15],stack[15];

int ip_ptr=0,st_ptr=0,len,i;

char temp[2],temp2[2];

char act[15];

void check();

void main()
```

```
{
clrscr();
printf("\n\t\t SHIFT REDUCE PARSER\n");
printf("\n GRAMMER\n");
printf("\n E->E+E\n E->E/E");
printf("\n E->E*E\n E->a/b");
printf("\n enter the input symbol:\t");
gets(ip_sym);
printf("\n\t stack implementation table");
printf("\n stack \t\t input symbol\t\t action");
printf("\n_________\t\t_____________\t\t_____________\n");
printf("\n $\t\t%s$\t\t\t--",ip_sym);
strcpy(act,"shift");
temp[0]=ip_sym[ip_ptr];
temp[1]='\0';
strcat(act,temp);
len=strlen(ip_sym);
for(i=0;i<=len-1;i++)
{
stack[st_ptr]=ip_sym[ip_ptr];
stack[st_ptr+1]='\0';
ip_sym[ip_ptr]=' ';
ip_ptr++;
printf("\n $%s\t\t%s$\t\t\t%s",stack,ip_sym,act);
```

```
strcpy(act,"shift");
temp[0]=ip_sym[ip_ptr];
temp[1]='\0';
strcat(act,temp);
check();
st_ptr++;
}
st_ptr++;
check();
}
void check()
{
int flag=0;
temp2[0]=stack[st_ptr];
temp2[1]='\0';
if((!strcmpi(temp2,"a"))||(!strcmpi(temp2,"b")))
{
stack[st_ptr]='E';
if(!strcmpi(temp2,"a"))
printf("\n $%s\t\t%s$\t\t\tE->a",stack,ip_sym);
else
printf("\n $%s\t\t%s$\t\t\tE->b",stack,ip_sym);
flag=1;
}
if((!strcmpi(temp2,"+"))||(strcmpi(temp2,"*"))||(!strcmpi(temp2,"/")))
```

```
{
flag=1;
}
if((!strcmpi(stack,"E+E"))||(!strcmpi(stack,"E\E"))||(!strcmpi(stack,"E*E")))
{
strcpy(stack,"E");
st_ptr=0;
if(!strcmpi(stack,"E+E"))
printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip_sym);
else
if(!strcmpi(stack,"E\E"))
printf("\n $%s\t\t%s$\t\t\tE->E\E",stack,ip_sym);
else
if(!strcmpi(stack,"E*E"))
printf("\n $%s\t\t%s$\t\t\tE->E*E",stack,ip_sym);
else
printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip_sym);
flag=1;
}
if(!strcmpi(stack,"E")&&ip_ptr==len)
{
printf("\n $%s\t\t%s$\t\t\tACCEPT",stack,ip_sym);
getch();
exit(0);
```

```
}
if(flag==0)
{
printf("\n%s\t\t\t%s\t\t reject",stack,ip_sym);
exit(0);
}
return;
}
```

OUTPUT:

```
                    SHIFT REDUCE PARSER
        GRAMMER
        E->E+E
        E->E/E
        E->E*E
        E->a/b
        Enter the input symbol: a+b
```

11. Simulate the calculator using LEX and YAAC tool.

Aim: Study the LEX and YACC tool and Evaluate an arithmetic expression with parentheses, unary and binary operators using Flex and Yacc. [Need to write yylex() function and to be used with Lex and yacc.].

Description: LEX-A Lexical analyzer generator: Lex is a computer program that generates lexical analyzers ("scanners" or "lexers").Lex is commonly used with the yacc parser generator. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language 1. A lexer or scanner is used to perform lexical analysis, or the breaking up of an input stream into meaningful units, or tokens. 2. For example, consider breaking a text file up into individual words. 3. Lex: a tool for automatically generating a lexer or scanner given a lex specification (.l file).

Structure

The structure of a Lex file is intentionally similar to that of a yacc file; files are divided up into three sections, separated by lines that contain only two percent signs, as follows:

Definition section:
%%
Rules section: %%
C code section:

- The definition section is the place to define macros and to import header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
- The rules section is the most important section; it associates patterns with C statements. Patterns are simply regular expressions. When the lexer sees some text in the input

matching a given pattern, it executes the associated C code. This is the basis of how Lex operates. 14 Prepared by-Ramesh Chotiya GCEK, BHAWANIPATNA CSE Department

➢ The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file and link it in at compile time.

Algorithm:

1) Get the input from the user and Parse it token by token.

2) First identify the valid inputs that can be given for a program.

3) The Inputs include numbers, functions like LOG, COS, SIN, TAN, etc. and operators.

4) Define the precedence and the associativity of various operators like +,-,/,* etc.

5) Write codes for saving the answer into memory and displaying the result on the screen.

6) Write codes for performing various arithmetic operations.

7) Display the possible Error message that can be associated with this calculation.

8) Display the output on the screen else display the error message on the screen.

Program:

CALC.L

```
%{
#include<stdio.h>
#include<stdlib.h>
void yyerror(char *);
#include "y.tab.h"
int yylval;
%}
%%
[a-z] {yylval=*yytext='&'; return VARIABLE;}
[0-9]+ {yylval=atoi(yytext); return INTEGER;}
```

CALC.Y

```
[\t] ;
%%
int yywrap(void)
{
return 1;
}
```

```
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
%{
int yylex(void);
void yyerror(char *);
int sym[26];
%}
%%
PROG:
PROG STMT '\n'
;
STMT: EXPR {printf("\n %d",$1);}
| VARIABLE '=' EXPR {sym[$1] = $3;}
;
EXPR: INTEGER
| VARIABLE {$$ = sym[$1];}
| EXPR '+' EXPR {$$ = $1 + $3;}
| '(' EXPR ')' {$$ = $2;}
%%
void yyerror(char *s)
{
printf("\n %s",s);
return;
}
int main(void)
{
printf("\n Enter the Expression:");
yyparse();
return 0;
}
```

Output:

```
$ lex calc.l
$ yacc -d calc.y
```

```
$ cc y.tab.c lex.yy.c -ll -ly -lm
$ . / a . out
Enter the Expression: ( 5 + 4 ) * 3
Answer: 27
```

12.Generate YACC specification for a few syntactic categories.

```
%{
/* Definition section */
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}

/* Rule Section */
%%
[0-9]+ {
        yylval=atoi(yytext);
        return NUMBER;

    }
[\t] ;

[\n] return 0;

. return yytext[0];
```

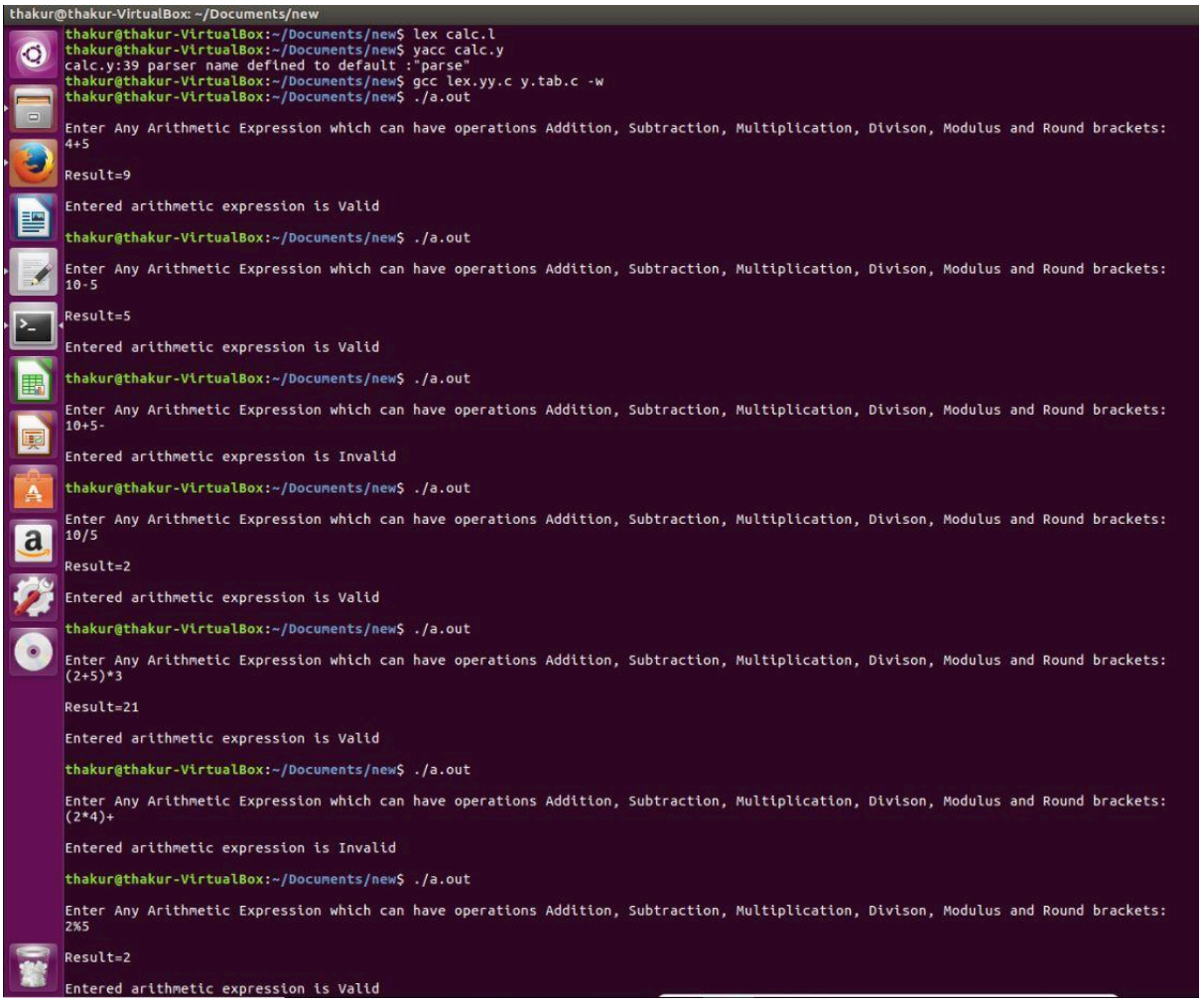
```
%%

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

Parser Source Code :

```
%{
   /* Definition section */
  #include<stdio.h>
  int flag=0;
%}

%token NUMBER

%left '+' '-'

%left '*' '/' '%'

%left '(' ')'

/* Rule Section */
%%

ArithmeticExpression: E{

         printf("\nResult=%d\n", $$);

         return 0;

        };
 E:E'+'E {$$=$1+$3;}

 |E'-'E {$$=$1-$3;}

 |E'*'E {$$=$1*$3;}

 |E'/'E {$$=$1/$3;}

 |E'%'E {$$=$1%$3;}
```

```
 |'('E')' {$$=$2;}

 | NUMBER {$$=$1;}

 ;

%%

//driver code
void main()
{
   printf("\nEnter Any Arithmetic Expression which
                  can have operations Addition,
                  Subtraction, Multiplication, Division,
                         Modulus and Round brackets:\n");

   yyparse();
   if(flag==0)
   printf("\nEntered arithmetic expression is Valid\n\n");
}

void yyerror()
{
   printf("\nEntered arithmetic expression is Invalid\n\n");
   flag=1;
}
```

Output:



13. Write a C program for generating the three address code of a given expression/statement.

In source code I use the Three function pm(),plus(),div(), I use the keyword strlen and I include the string.h header package.

Example Source Code Programming | Generation Three address code Project

```
#include<stdio.h>
#include<string.h>
void pm();
```

```
void plus();
void div();
int i,ch,j,l,addr=100;
char ex[10], exp[10] ,exp1[10],exp2[10],id1[5],op[5],id2[5];
void main()
{
clrscr();
while(1)
{
printf("\n1.assignment\n2.arithmetic\n3.relational\n4.Exit\nEnter the choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\nEnter the expression with assignment operator:");
scanf("%s",exp);
l=strlen(exp);
exp2[0]='\0';
i=0;
while(exp[i]!='=')
{
i++;
}
strncat(exp2,exp,i);
strrev(exp);
exp1[0]='\0';
strncat(exp1,exp,l-(i+1));
strrev(exp1);
printf("Three address code:\ntemp=%s\n%s=temp\n",exp1,exp2);
break;

case 2:
printf("\nEnter the expression with arithmetic operator:");
scanf("%s",ex);
strcpy(exp,ex);
l=strlen(exp);
exp1[0]='\0';

for(i=0;i<l;i++)
{
if(exp[i]=='+'||exp[i]=='-')
{
if(exp[i+2]=='/'||exp[i+2]=='*')
{
pm();
break;
}
else
{
plus();
```

```
break;
}
}
else if(exp[i]=='/'||exp[i]=='*')
{
div();
break;
}
}
break;

case 3:
printf("Enter the expression with relational operator");
scanf("%s%s%s",&id1,&op,&id2);
if(((strcmp(op,"<")==0)||(strcmp(op,">")==0)||(strcmp(op,"<=")==0)||(strcmp(op,">=")==0)||(
strcmp(op,"==")==0)||(strcmp(op,"!=")==0))==0)
printf("Expression is error");
else
{
printf("\n%d\tif %s%s%s goto %d",addr,id1,op,id2,addr+3);
addr++;
printf("\n%d\t T:=0",addr);
addr++;
printf("\n%d\t goto %d",addr,addr+2);
addr++;
printf("\n%d\t T:=1",addr);
}
break;
case 4:
exit(0);
}
}
}
void pm()
{
strrev(exp);
j=l-i-1;
strncat(exp1,exp,j);
strrev(exp1);
printf("Three address code:\ntemp=%s\ntemp1=%c%ctemp\n",exp1,exp[j+1],exp[j]);
}
void div()
{
strncat(exp1,exp,i+2);
printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
}
void plus()
{
strncat(exp1,exp,i+2);
printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
```

}

Example Generation of Three Address Project Output Result

1. assignment
2. arithmetic
3. relational
4. Exit
Enter the choice:1
Enter the expression with assignment operator:
a=b
Three address code:
temp=b
a=temp

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:
a+b-c
Three address code:
temp=a+b
temp1=temp-c

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:
a-b/c
Three address code:
temp=b/c
temp1=a-temp

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:
a*b-c
Three address code:
temp=a*b
temp1=temp-c

1.assignment
2.arithmetic

```
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:a/b*c
Three address code:
temp=a/b
temp1=temp*c
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:3
Enter the expression with relational operator
a
<=
b

100 if a<=b goto 103
101 T:=0
102 goto 104
103 T:=1

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:4
```

14. Write a C program for implementation of a Code Generation Algorithm of a given expression/statement.

AIM : C program to implement intermediate code generation for simple expression.

Program

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int i=1,j=0,no=0,tmpch=90;
char str[100],left[15],right[15];
void findopr();
void explore();
void fleft(int);
void fright(int);
struct exp
{
int pos;
char op;
}k[15];
void main()
{
printf("\t\tINTERMEDIATE CODE GENERATION\n\n");
printf("Enter the Expression :");
scanf("%s",str);
printf("The intermediate code:\n");
findopr();
explore();
}
void findopr()
{
```

```
for(i=0;str[i]!='\0';i++)
if(str[i]==':')
{
k[j].pos=i;
k[j++].op=':';
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='/')
{
k[j].pos=i;
k[j++].op='/';
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='*')
{
k[j].pos=i;
k[j++].op='*';
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='+')
{
k[j].pos=i;
k[j++].op='+';
}
for(i=0;str[i]!='\0';i++)
```

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

```
}

printf("\t%s := %c",right,str[k[--i].pos]);

getch();

}

void fleft(int x)

{

int w=0,flag=0;

x--;

while(x!=
-1 &&str[x]!= '+' &&str[x]!='*'&&str[x]!='='&&str[x]!='\0'&&st
r[x]!='-'&&str[x]!='/'&&str[x]!=':')

{

if(str[x]!='$'&& flag==0)

{

left[w++]=str[x];

left[w]='\0';

str[x]='$';

flag=1;

}

x--;

}

}

void fright(int x)

{

int w=0,flag=0;
```

```
x++;

while(x!=
-1 && str[x]!= '+'&&str[x]!='*'&&str[x]!='\0'&&str[x]!='='&&st
r[x]!=':'&&str[x]!='-'&&str[x]!='/')

{

if(str[x]!='$'&& flag==0)

{

right[w++]=str[x];

right[w]='\0';

str[x]='$';

flag=1;

}

x++;

}

}
```

```
                INTERMEDIATE CODE GENERATION

Enter the Expression :w:=a*b+c/d-e/f+g*h
The intermediate code:
        Z := c/d
        Y := e/f
        X := a*b
        W := g*h
        V := X+Z
        U := Y+W
        T := V-U
        w := T
Process returned 0 (0x0)   execution time : 43.188 s
Press any key to continue.
```