**Experiment-VII: Implementation of CLR parser.**

**1.Problem Statement :** program to implement CLR for the following Grammar

S->CC

C->aC/d

**Description :**  token can look like anything that is useful for processing an input text stream or text

**Source Code:**

#include<stdio.h>

#include<string.h>

int i,j,z;

void printtable();

main()

{

char b[5]={'a','d','$','s','c'};

printf("\n------------------clr----------------------\n");

printf("\nthe given grammer\n");

printf("\nS->CC\nC->aC\nC->d\n");

printf("\n the parsing table of the grammer\n");

for(z=0;z<5;z++)

{

printf("\t%c",b[z]);

}

printtable();

}

void printtable()

{

#include<stdio.h>

#include<string.h>

int i,j,z;

void printtable();

main()

{

char b[5]={'a','d','$','s','c'};

printf("\n------------------clr----------------------\n");

printf("\nthe given grammer\n");

printf("\nS->CC\nC->aC\nC->d\n");

printf("\n the parsing table of the grammer\n");

for(z=0;z<5;z++)

{

printf("\t%c",b[z]);

}

printtable();

}

void printtable()

{

for(i=0;i<=9;i++)

{

printf("\n---------------------------------------------------\n");

printf("%d",i);

for(j=0;j<=4;j++)

{

if(t[i][j]==-5)

printf("\t");

else if(t[i][j]>=1&&t[i][j]<10)

printf("\t%d",t[i][j]);

else if(t[i][j]==99)

printf("\taccepted\t");

//else if(t[i][j]%2==0)

else if(t[i][j]%2==0)

printf("\tr%d",t[i][j]/10);

else if(t[i][j]%2==1)

{

printf("\ts%d",t[i][j]/10);

continue;

}

/\*else

printf("\t"); \*/

}

}

}

**Input / Output:**

$ gcc clr.c

$ ./a.out

------------------clr----------------------

the given grammer

S->CC

C->aC

C->d

the parsing table of the grammer

a d $ s c

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

0 s3 s4 1 2

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

1 accepted

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

2 s6 s7 5

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

3 s3 s4 8

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

4 r3 r3

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

5 s6 r1

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

6 9

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

7 r2 s7 r3

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

8 r2

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

9 r2

$

**Experiment-VIII:** Implementation of LALR Parser using ANTLR

**1.Problem Statement :** Implementation of LALR Parser using ANTLR

**Description :**  ANTLR(Another Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.

**Steps to parse a grammer using antlr tool:**

1. Create a folder in c drive called antlr.
2. Copy grammar file (for ex: Calculator.g4 file) , antlr-4.5-complete.jar file and two batch files (antlr4.bat and grun.bat) files in to c:\antlr folder.
3. go to command prompt.
4. set classpath=.;C:\antlr\antlr-4.5-complete.jar.
5. compile a grammer file for ex take Calculator grammer file
6. antlr4 Calculator.g4 // on success java files will be generated.
7. javac \*.java //compile java files.then
8. grun Calculator calculator –gui.
9. ex: 3\*(4+5) enter and press ctrl+z.
10. Finally a parse tree inspector generates for the given grammer.
11. Click on save png then a png image file will be created for the given grammer.

**Source Code:**

**Calculator.g4:**

grammar Calculator;

calculator : expression;

expression

: expression operator=('\*'|'/') expression # MultiplyDivide

| expression operator=('+'|'-') expression # AddSubtract

| '-' expression # Negate

| Number # Number

| '(' expression ')' # Parenthesis

;

Number : DIGIT+ '.' DIGIT\*

| '.' DIGIT+

| DIGIT+

;

DIGIT: ('0'..'9');

WS: [ \t\r\n]+ -> skip;

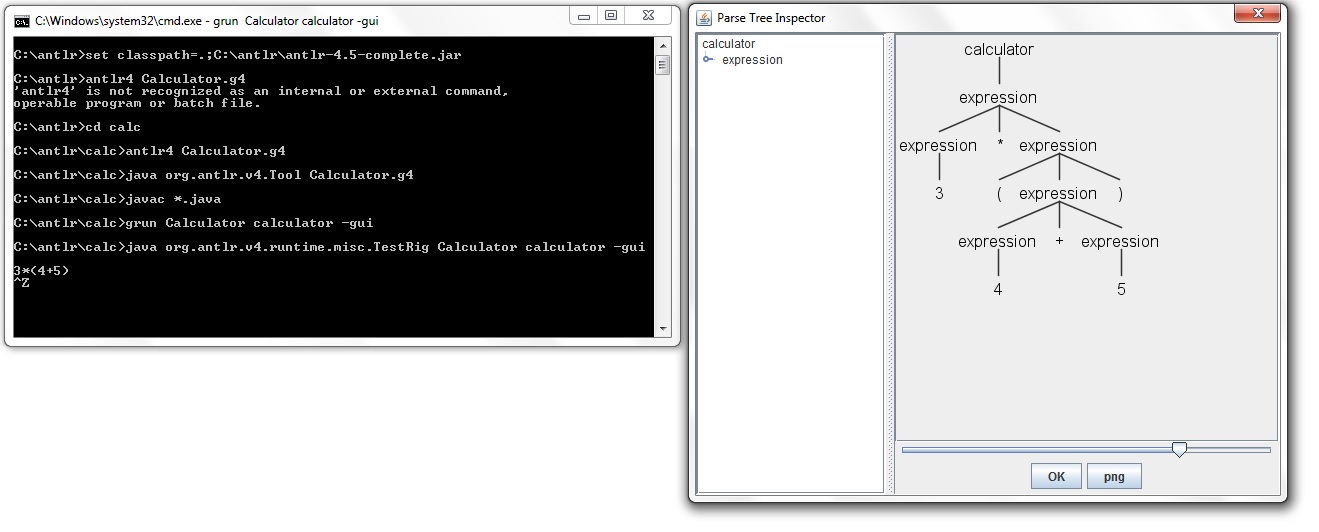
**antlr4.bat:**

java org.antlr.v4.Tool %\*

**grun.bat:**

java org.antlr.v4.runtime.misc.TestRig %\*.

**Input / Output::**



**Experiment-IX: Construct dependency graph for the given SDD**

**1.Problem Statement :** YACC program toConstruct dependency graph for the given SDD

**Description :**  Dependency graphs tool for determining an evaluation order for the attributes instances in a given parse tree.

Annotated parse tree shows the values of attributes, a dependency graph helps to determine how those values can be computed.

Dependency graphs

1. Edges express constraints implied by the semantic rules.
2. Each attribute is associated to a node
3. If a semantic rule associated with a production p defines the value of synthesized attribute A.b in terms of the value of X.c, then graph has an edge from X.c to A.b

If a semantic rule associated with a production p defines the value of inherited attribute B.c in terms of value of X.a, then graph has an edge from X.a to B.c

**Source Code:**

dep.l

%{

#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;}

sin|SIN {return SIN;}

cos|COS {return COS;}

[\t];

\$; {return 0;}

\n|. {return yytext[0];}

%%

dep.y

%{

#include<math.h>

#include<stdio.h>

int sno=1;

%}

%union{

double dval;

}

%type<dval>Expr

%type<dval> T

%type<dval> F

%token <dval>NUMBER

%token LOG,SIN,COS

%left '+','-'

%left '\*','/'

%left SIN COS

%%

S :Expr'\n' {printf("%g \n",$1); printf("%d . E.val = %g\n",sno,$1); sno++;}

;

Expr:Expr '+' T {$$=$1+$3; printf("%d . E.val = %g\n",sno,$$); sno++;}

|T {printf("%d . E.val = %g\n",sno,$$); sno++;}

;

T: T '\*' F {$$=$1\*$3; printf("%d . T.val = %g\n",sno,$$); sno++;}

|F {printf("%d . T.val = %g\n",sno,$$); sno++;}

;

F : NUMBER {$$=$1; printf("%d . F.val = %g\n",sno,$$); sno++; }

;

%%

int main()

{

yyparse();

return 0;

}

int yywrap()

{

return 1;

}

int yyerror(char \*err)

{

printf("%s",err);

}

**Input / Output:**

$ lex dep.l

$ yacc -d dep.y

$ gcc lex.yy.c y.tab.c -ll -lm

$ ./a.out

6+8\*7

1 . F.val = 6

2 . T.val = 6

3 . E.val = 6

4 . F.val = 8

5 . T.val = 8

6 . F.val = 7

7 . T.val = 56

8 . E.val = 62

**Experiment-X: Intermediate Code generation using YACC**

**1.Problem Statement :** C program for Three address code generation

**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:**

**three.l**

%{

#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];}

%%

**Three.y**

%{

#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++;

}}

main()

{

printf("enter expression");

yyparse();

threeaddresscode();

}

yywrap()

{

return 1;

**}**

**Input / Output:**

$ lex three.l

$ yacc -d three.y

$ gcc lex.yy.c y.tab.c -ll -lm

$ ./a.out

enter expression (a\*b)+(c\*d)

three addrtess codes

B:= a \* b

C:= c \* d

D:= B + B

**2.Problem Statement :**YACC program for generate Quadruple.

**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;

**}**

**Input / 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

**Experiment-III: Construct the DAG for given three address code.**

**1.Problem Statement :**C program for Directed Acyclic Graph

**Description :**  A directed acyclic graph (DAG) is a directed graph that contains no cycles. a DAG gives a picture of how the value computed by a statement in a basic block is used in subsequent statements of the block.

Constructing a DAG from three-address statements is a good way of determining common sub-expressions (expressions computed more than once) within a block, determining which names are used inside the block but evaluated outside the block, and determining which statements of the block could have their computed value used outside the block.

1. A DAG for a basic block has following labels on the nodes

Leaves are labeled by unique identifiers, either variable names or constants.

Interior nodes are labeled by an operator symbol.

Nodes are also optionally given a sequence of identifiers for labels.

**Source Code:**

**dag.c**

#include<stdio.h>

#include<ctype.h>

#define size 20

typedef struct node{

char data;

struct node \*left;

struct node \*right;

}btree;

btree \*stack[size];

int top;

main() {

btree \*root;

char exp[80];

btree \*create(char exp[80]);

void dag(btree \*root);

printf("\nEnter the postfix expression:\n");

scanf("%s",exp);

top=-1;

root=create(exp);

printf("\nThe tree is created.....\n");

printf("\nInorder DAG is : \n\n");

dag(root);

return 0; }

btree \*create(char exp[]) {

btree \*temp;

int pos;

char ch;

void push(btree\*);

btree \*pop();

pos=0;

ch=exp[pos];

printf("%c\t",ch);

while(ch!='\0') {

temp=((btree\*)malloc(sizeof(btree)));

temp->left=temp->right=NULL;

temp->data=ch;

printf("%c",temp->data);

if(isalpha(ch))

push(temp);

else if(ch=='+' ||ch=='-' || ch=='\*' || ch=='/') {

temp->right=pop();

temp->left=pop();

push(temp); }

else

printf("\n Invalid char Expression\n");

pos++;

ch=exp[pos]; }

temp=pop();

return(temp); }

void push(btree \*Node)

{

if(top+1 >=size)

printf("Error:Stack is full\n");

top++;

stack[top]=Node;

}

btree\* pop()

{

btree \*Node;

if(top==-1)

printf("\nerror: stack is empty..\n");

Node=stack[top];

top--;

return(Node);

}

void dag(btree \*root)

{

btree \*temp;

temp=root;

if(temp!=NULL)

{

dag(temp->left);

printf("%c",temp->data);

dag(temp->right);

}}

**Input / Output:**

$ gcc dag.c

$ ./a.out

Enter the postfix expression:

abcd+\*-

a abcd+\*-

The tree is created.....

Inorder DAG is :

a-b\*c+d

**2.Problem Statement :** Write a C Program to implement a code optimization method “common sub expression elimination”.

**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>

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];

main(){

int a,d;

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;

**}**

**INPUT/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