**PRACTICAL 1**

**AIM:** Implement a lexical analyzer for a subset of C using LEX Implementation should support Error handling.

## IMPLEMENTATION:

* lex <filename with .l extension>
* gcc <newly created .c file> -o <file name for exe file>
* <filename of exe file>

In this case, create an extra text file named abc.txt which will contain some C code to work as input for lexical analysis.

## PROGRAM:

%%

"#" {printf("\n %s \t Preprocessor",yytext);} "main"|"printf"|"scanf" {printf("\n%s\tfunction",yytext);}

"if"|"else"|"int"|"unsigned"|"long"|"char"|"switch"|"case"|"struct"|"do"|"while"|"void"|"for"|"fl oat"|"continue"|"break"|"include" { printf("\n%s\tKeyword",yytext); }

[\_a-zA-Z][\_a-zA-Z0-9]\* {printf("\n%s\tIdenifier",yytext);} "+"|"/"|"\*"|"-" {printf("\n%s\tOperator",yytext);}

"="|"<"|">"|"!="|"=="|"<="|">=" {printf("\n%s\tRelational Operator",yytext);} "%d"|"%s"|"%c"|"%f" {printf("\n%s\tTokenizer",yytext);}

"stdio.h"|"conio.h"|"math.h"|"string.h"|"graphics.h"|"dos.h" {printf("\n%s\tHeader File",yytext);}

";"|"," {printf("\n%s\tDelimiter",yytext);}

"("|")" {if(strcmp(yytext,"(")==0)

{

printf("\n%c\tOpening Parenthesis",yytext[0]);

}

else

{

printf("\n%c\tClosing Parenthesis",yytext[0]);

}

;}

"{" {printf("\n%s\tStart Of Function/Loop",yytext);} "}" {printf("\n%s\tEnd of Function",yytext);}

%%

int yywrap(void)

{

return 1;

}

int main()

{

int i;

FILE \*fp;

fp=fopen("abc.txt","r");

if(fp==NULL)

{

}

else

{

}

printf("Unable To Open File");

yyin=fp;

yylex(); return 0;

}

## OUTPUT:

**CONCLUSION:**

In this practical, we learnt about lex files and implemented a program for lexical analysis.

# PRACTICAL 2

**AIM:** Implement a lexical analyzer for identification of numbers.

## IMPLEMENTATION:

* lex <filename with .l extension>
* gcc <newly created .c file> -o <file name for exe file>
* <filename of exe file>

## PROGRAM:

bin (0|1)+

char [A-Za-z]+ digit [0-9]

oct [0-7]

dec [0-9]\*

float {digit}+("."{digit}+)

exp {digit}+("."{digit}+)?("E"("+"|"-")?{digit}+)? hex [0-9a-fA-F]+

%%

{bin} {printf("\n %s= it is a binary number",yytext);}

{char} {printf("\n %s=it is a char",yytext);}

{oct} {printf("\n %s=it is a octal number",yytext);}

{digit} {printf("\n %s=it is a digit",yytext);}

{dec} {printf("\n %s=it is a decimal",yytext);}

{float} {printf("\n %s=it is a float",yytext);}

{exp} {printf("\n %s=it is a exp",yytext);}

{hex} {printf("\n %s=it is a hex",yytext);}

%%

int yywrap()

{

return 1;

}

int main()

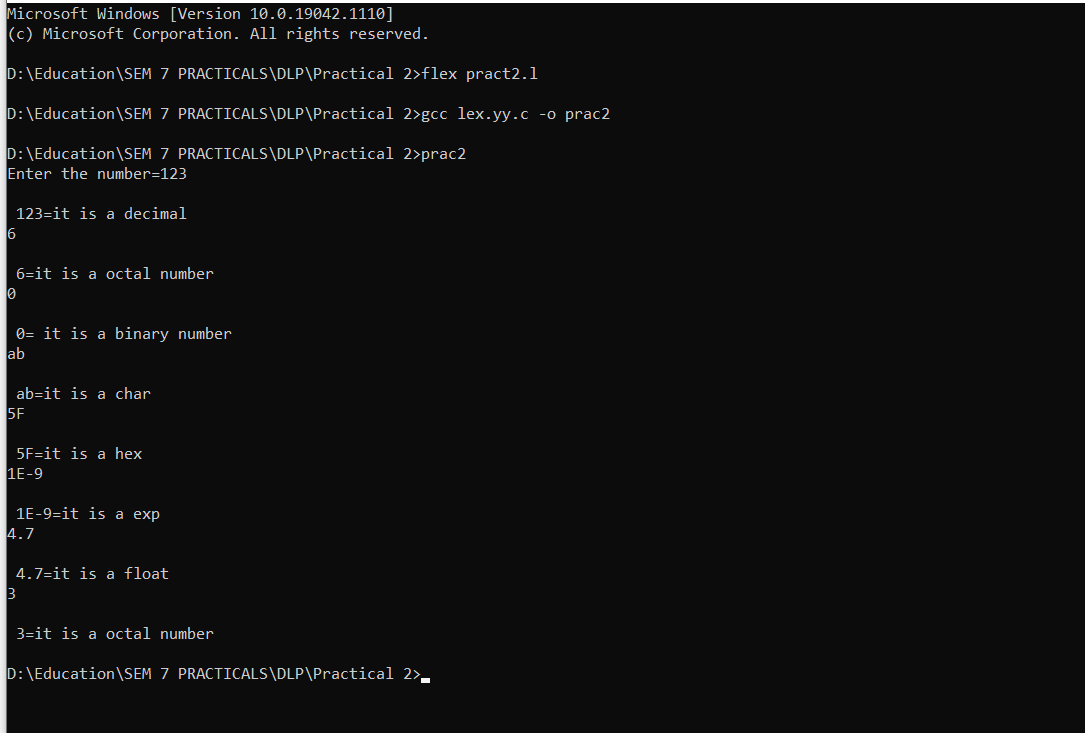
{

printf("Enter the number="); yylex();

return 0;

}

## OUTPUT:



**CONCLUSION:**

In this practical, we learnt about lexical analysis for numbers and characters.

**PRACTICAL 3**

**AIM:** Implement a Calculator using LEX and YACC.

**IMPLEMENTATION:**

• lex <filename with .l extension>

• yacc <filename with .y extension>

• gcc <newly created .c file from yacc> -o <file name for exe file>

• <filename of exe file>

**PROGRAM:**

Lex File:

DIGIT [0-9]

%option noyywrap

%%

{DIGIT} { yylval=atof(yytext); return NUM;}

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

Yacc File:

%{

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

#define YYSTYPE double

%}

%token NUM

%left '+' '-'

%left '\*' '/'

%%

S : S E '\n' { printf("Answer: %g \nEnter:\n", $2); }

| S '\n'

|

| error '\n' { yyerror("Error: Enter once more \n" );yyerrok; }

;

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

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

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

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

| NUM

;

%%

#include "lex.yy.c" int main()

{

printf("Enter the expression: "); yyparse();

}

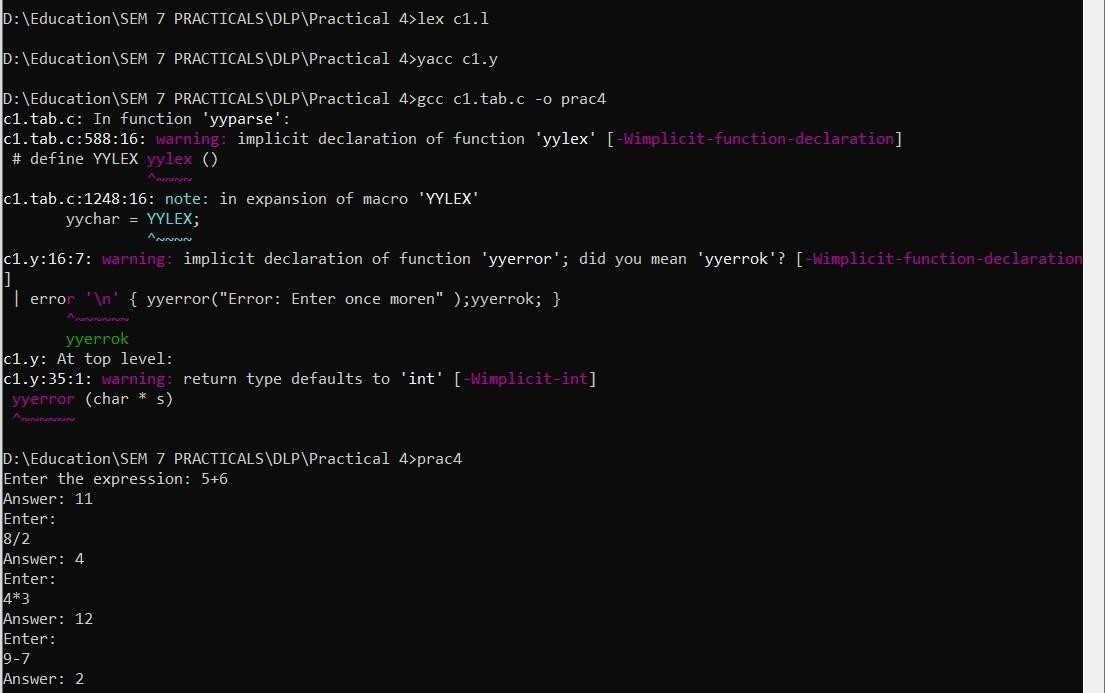
yyerror (char \* s)

{

printf ("% s \n", s); exit (1);

}

**OUTPUT:**



**CONCLUSION:**

In this practical, we learnt implemented a calculator using lex and yacc which takes expression as input and perform basic arithmetic operations.

**PRACTICAL 4**

**AIM:** Implement a program to identify keywords and identifiers using finite automata.

**IMPLEMENTATION:**

• lex <filename with .l extension>

• yacc <filename with .y extension>

• gcc <newly created .c file from yacc> -o <file name for exe file>

• <filename of exe file>

**PROGRAM:**

Lex File:

#include <stdio.h>

#include <string.h>

// Define keywords

char \*keywords[] = {"int", "float", "if", "else", "while", "for", "return"};

// Function to check if a string is a keyword

int isKeyword(char \*str) {

for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++) {

if (strcmp(keywords[i], str) == 0) {

return 1; // It's a keyword

}

}

return 0; // It's not a keyword

}

int main() {

char input[1000];

printf("Enter C code:\n");

fgets(input, sizeof(input), stdin);

char token[100];

int i = 0, j = 0;

while (input[i] != '\0') {

if (input[i] == ' ' || input[i] == '\n' || input[i] == '\t') {

// Whitespace, end current token and check if it's a keyword

token[j] = '\0';

if (j > 0 && isKeyword(token)) {

printf("Keyword: %s\n", token);

} else if (j > 0) {

printf("Identifier: %s\n", token);

}

j = 0; // Reset token

} else {

// Add character to the current token

token[j] = input[i];

j++;

}

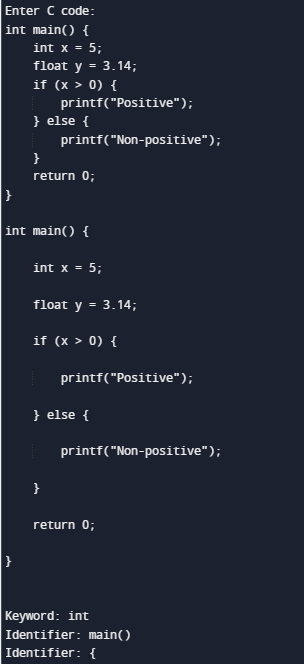
i++;

}

return 0;

}

**OUTPUT:**



**CONCLUSION:**

In this practical, we learnt implemented a calculator using lex and yacc which takes expression as input and perform basic arithmetic operations.

**PRACTICAL 5**

**AIM:** Write an ambiguous CFG to recognize an infix expression and implement a parser that recognizes the infix expression using YACC.

**IMPLEMENTATION:**

• yacc <filename with .y extension>

• gcc <newly created .c file> -o <file name for exe file>

• <filename of exe file>

**PROGRAM:**

%{

/\*\*\* Auxiliary declarations section \*\*\*/

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

/\* Custom function to print an operator\*/ void print\_operator(char op);

/\* Variable to keep track of the position of the number in the input \*/ int pos=0;

char p;

%}

/\*\*\* YACC Declarations section \*\*\*/

%token NUM

%left '+'

%left '\*'

%%

/\*\*\* Rules Section \*\*\*/

start : expr '\n' {exit(1);}

;

expr: expr '+' expr {print\_operator('+');}

| expr '\*' expr {print\_operator('\*');}

| '(' expr ')'

| NUM {printf("%c ",p);}

;

%%

/\*\*\* Auxiliary functions section \*\*\*/

void print\_operator(char c){ switch(c){

case '+' : printf("+ "); break;

case '\*' : printf("\* "); break;

}

return;

}

yyerror(char const \*s)

{

printf("yyerror %s",s);

}

yylex(){ char c;

c = getchar(); p=c; if(isdigit(c)){

pos++; return NUM;

}

else if(c == ' '){

yylex(); /\*This is to ignore whitespaces in the input\*/

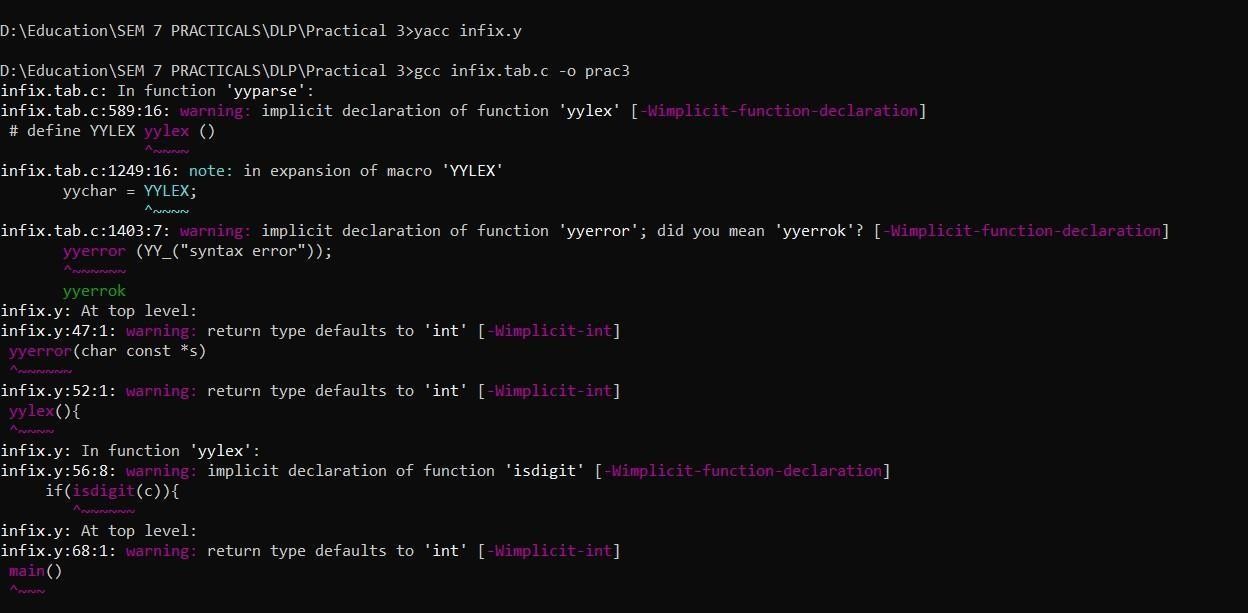
}

else {

return c;

}

}

**OUTPUT:**



**CONCLUSION:**

In this practical, we learnt about yacc and performed infix to postfix conversion.

**PRACTICAL 6**

**AIM:** Implement a C program to find FIRST and FOLLOW set of given

grammar.

**IMPLEMENTATION:**

• yacc <filename with .y extension>

• gcc <newly created .c file> -o <file name for exe file>

• <filename of exe file>

**PROGRAM:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX\_RULES 10

#define MAX\_SYMBOLS 10

#define MAX\_FIRST\_SET 10

#define MAX\_FOLLOW\_SET 10

char grammar[MAX\_RULES][MAX\_SYMBOLS];

char nonTerminals[MAX\_RULES];

int numRules;

int numNonTerminals;

typedef struct {

char symbol;

char firstSet[MAX\_FIRST\_SET];

int numFirst;

char followSet[MAX\_FOLLOW\_SET];

int numFollow;

} NonTerminalInfo;

NonTerminalInfo nonTerminalInfo[MAX\_RULES];

// Function to add a symbol to a set

void addToSet(char set[], int \*num, char symbol) {

for (int i = 0; i < \*num; i++) {

if (set[i] == symbol) {

return;

}

}

set[\*num] = symbol;

(\*num)++;

}

// Function to calculate FIRST set

void calculateFirstSet(char nonTerminal) {

for (int i = 0; i < numRules; i++) {

if (grammar[i][0] == nonTerminal) {

// If the production starts with the non-terminal

if (islower(grammar[i][3])) {

// If the production has a terminal symbol

addToSet(nonTerminalInfo[nonTerminal - 'A'].firstSet,

&nonTerminalInfo[nonTerminal - 'A'].numFirst, grammar[i][3]);

} else if (grammar[i][3] != nonTerminal) {

// If the production has another non-terminal

calculateFirstSet(grammar[i][3]);

int j;

for (j = 0; j < nonTerminalInfo[grammar[i][3] - 'A'].numFirst; j++) {

addToSet(nonTerminalInfo[nonTerminal - 'A'].firstSet,

&nonTerminalInfo[nonTerminal - 'A'].numFirst,

nonTerminalInfo[grammar[i][3] - 'A'].firstSet[j]);

}

if (j == nonTerminalInfo[grammar[i][3] - 'A'].numFirst) {

addToSet(nonTerminalInfo[nonTerminal - 'A'].firstSet,

&nonTerminalInfo[nonTerminal - 'A'].numFirst, '#');

}

}

}

}

}

// Function to calculate FOLLOW set

void calculateFollowSet(char nonTerminal) {

if (nonTerminal == grammar[0][0]) {

addToSet(nonTerminalInfo[nonTerminal - 'A'].followSet,

&nonTerminalInfo[nonTerminal - 'A'].numFollow, '$');

}

for (int i = 0; i < numRules; i++) {

int j;

for (j = 3; grammar[i][j] != '\0'; j++) {

if (grammar[i][j] == nonTerminal) {

// If nonTerminal appears in a production

for (int k = j + 1; grammar[i][k] != '\0'; k++) {

if (isupper(grammar[i][k])) {

// If a non-terminal follows

int l;

for (l = 0; l < nonTerminalInfo[grammar[i][k] - 'A'].numFirst; l++) {

addToSet(nonTerminalInfo[nonTerminal - 'A'].followSet,

&nonTerminalInfo[nonTerminal - 'A'].numFollow,

nonTerminalInfo[grammar[i][k] - 'A'].firstSet[l]);

}

if (l == nonTerminalInfo[grammar[i][k] - 'A'].numFirst) {

addToSet(nonTerminalInfo[nonTerminal - 'A'].followSet,

&nonTerminalInfo[nonTerminal - 'A'].numFollow, '#');

}

} else {

// If a terminal symbol follows

addToSet(nonTerminalInfo[nonTerminal - 'A'].followSet,

&nonTerminalInfo[nonTerminal - 'A'].numFollow, grammar[i][k]);

break;

}

}

if (grammar[i][k] == '\0') {

// If nonTerminal appears at the end of a production

for (int l = 0; nonTerminals[l] != '\0'; l++) {

if (nonTerminals[l] == grammar[i][0]) {

continue;

}

for (int m = 0; m < nonTerminalInfo[nonTerminals[l] - 'A'].numFollow;

m++) {

addToSet(nonTerminalInfo[nonTerminal - 'A'].followSet,

&nonTerminalInfo[nonTerminal - 'A'].numFollow,

nonTerminalInfo[nonTerminals[l] - 'A'].followSet[m]);

}

}

}

}

}

}

}

int main() {

printf("Enter the number of rules: ");

scanf("%d", &numRules);

printf("Enter the grammar rules (e.g., S->AB): \n");

for (int i = 0; i < numRules; i++) {

scanf("%s", grammar[i]);

nonTerminals[i] = grammar[i][0];

nonTerminals[i + 1] = '\0';

}

numNonTerminals = strlen(nonTerminals);

// Initialize FIRST and FOLLOW sets

for (int i = 0; i < numNonTerminals; i++) {

nonTerminalInfo[nonTerminals[i] - 'A'].symbol = nonTerminals[i];

nonTerminalInfo[nonTerminals[i] - 'A'].numFirst = 0;

nonTerminalInfo[nonTerminals[i] - 'A'].numFollow = 0;

}

// Calculate FIRST sets

for (int i = 0; i < numNonTerminals; i++) {

calculateFirstSet(nonTerminals[i]);

}

// Calculate FOLLOW sets

for (int i = 0; i < numNonTerminals; i++) {

calculateFollowSet(nonTerminals[i]);

}

// Print FIRST and FOLLOW sets

printf("\nFIRST and FOLLOW sets:\n");

for (int i = 0; i < numNonTerminals; i++) {

printf("Non-terminal %c\n", nonTerminalInfo[i].symbol);

printf("FIRST: { ");

for (int j = 0; j < nonTerminalInfo[i].numFirst; j++) {

printf("%c ", nonTerminalInfo[i].firstSet[j]);

}

printf("}\n");

printf("FOLLOW: { ");

for (int j = 0; j < nonTerminalInfo[i].numFollow; j++) {

printf("%c ", nonTerminalInfo[i].followSet[j]);

}

printf("}\n\n");

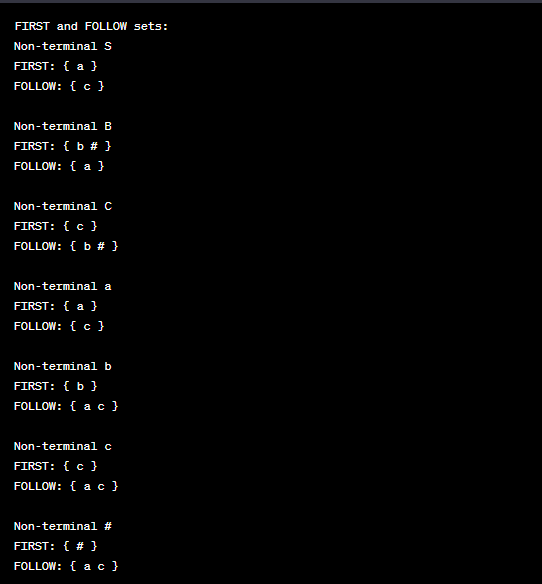
}

return 0;

}

**OUTPUT:**





**CONCLUSION:**

In this practical we implemented a C program to find FIRST and FOLLOW set of given

grammar.

**PRACTICAL 7**

**AIM:** Write a program to remove the Left Recursion from a given grammar.

**IMPLEMENTATION:**

yacc <filename with .y extension>

• gcc <newly created .c file> -o <file name for exe file>

• <filename of exe file>

**PROGRAM:**

In this practical we implemented a C program to find FIRST and FOLLOW set of given

grammar.

#include <stdio.h>

#include <string.h>

#define MAX\_RULES 10

#define MAX\_SYMBOLS 10

char grammar[MAX\_RULES][MAX\_SYMBOLS];

int numRules;

void removeLeftRecursion(char nonTerminal) {

int i, j, k;

char newGrammar[MAX\_RULES][MAX\_SYMBOLS];

for (i = 0; i < numRules; i++) {

if (grammar[i][0] == nonTerminal) {

// A production with left recursion

int m = 0;

for (j = 1; j < strlen(grammar[i]); j++) {

if (grammar[i][j] == '|') {

newGrammar[numRules + m][0] = nonTerminal;

newGrammar[numRules + m][1] = '\'';

newGrammar[numRules + m][2] = '\0';

strcat(newGrammar[numRules + m], &grammar[i][j + 1]);

m++;

j++;

}

}

newGrammar[numRules + m][0] = '#'; // ε (empty production)

newGrammar[numRules + m][1] = '\0';

numRules += m + 1;

for (k = i; k < numRules - m; k++) {

strcpy(grammar[k], newGrammar[k - i]);

}

}

}

}

int main() {

printf("Enter the number of rules: ");

scanf("%d", &numRules);

printf("Enter the grammar rules (e.g., S->a|A): \n");

for (int i = 0; i < numRules; i++) {

scanf("%s", grammar[i]);

}

// Iterate through non-terminals and remove left recursion

for (int i = 0; i < numRules; i++) {

removeLeftRecursion(grammar[i][0]);

}

// Print the modified grammar

printf("\nGrammar after removing left recursion:\n");

for (int i = 0; i < numRules; i++) {

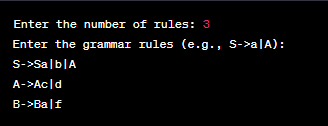
printf("%s\n", grammar[i]);

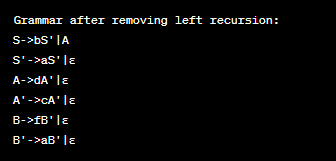
}

return 0;

}

**OUTPUT:**

****

****

**CONCLUSION:**

In this practical we implemented a program to remove the Left Recursion from a given grammar.

**PRACTICAL 8**

**AIM:** Implementation of Context Free Grammar.

**IMPLEMENTATION:**

• gcc <our .c file> -o <file name for exe file>

• <filename of exe file>

In this case, create a syntax.txt file as input for the executable which will contain following statements.

S aBaA S AB

A Bc B c

**PROGRAM:**

//CFG

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

int i,j,k,l,m,n=0,o,p,nv,z=0,t,x=0;

char str[10],temp[20],temp2[20],temp3[20];

struct prod

{

char lhs[10],rhs[10][10]; int n;

}pro[10];

void findter()

{

for(k=0;k<n;k++)

{

if(temp[i]==pro[k].lhs[0])

{

for(t=0;t<pro[k].n;t++)

{

for(l=0;l<20;l++) temp2[l]='\0';

for(l=i+1;l<strlen(temp);l++) temp2[l-i-1]=temp[l];

for(l=i;l<20;l++) temp[l]='\0';

for(l=0;l<strlen(pro[k].rhs[t]);l++)

temp[i+l]=pro[k].rhs[t][l]; strcat(temp,temp2); if(str[i]==temp[i])

return;

else if(str[i]!=temp[i] && temp[i]>=65 && temp[i]<=90) break;

}

break;

}

}

if(temp[i]>=65 && temp[i]<=90) findter();

}

int main()

{

FILE \*f;

// clrscr();

for(i=0;i<10;i++) pro[i].n=0;

f=fopen("input.txt","r"); while(!feof(f))

{

fscanf(f,"%s",pro[n].lhs); if(n>0)

{

if( strcmp(pro[n].lhs,pro[n-1].lhs) == 0 )

{

pro[n].lhs[0]='\0';

fscanf(f,"%s",pro[n-1].rhs[pro[n-1].n]); pro[n-1].n++;

continue;

}

}

fscanf(f,"%s",pro[n].rhs[pro[n].n]); pro[n].n++;

n++;

}

n--;

printf("\n\nTHE GRAMMAR IS AS FOLLOWS\n\n"); for(i=0;i<n;i++)

for(j=0;j<pro[i].n;j++)

printf("%s -> %s\n",pro[i].lhs,pro[i].rhs[j]);

while(1)

{

for(l=0;l<10;l++) str[0]=NULL;

printf("\n\nENTER ANY STRING ( 0 for EXIT ) : "); scanf("%s",str);

if(str[0]=='0')

printf("Exit");

// exit(1); for(j=0;j<pro[0].n;j++)

{

for(l=0;l<20;l++) temp[l]=NULL;

strcpy(temp,pro[0].rhs[j]); m=0;

for(i=0;i<strlen(str);i++)

{

if(str[i]==temp[i]) m++;

else if(str[i]!=temp[i] && temp[i]>=65 && temp[i]<=90)

{

findter(); if(str[i]==temp[i])

m++;

}

else if( str[i]!=temp[i] && (temp[i]<65 || temp[i]>90) ) break;

}

if(m==strlen(str) && strlen(str)==strlen(temp))

{

printf("\n\nTHE STRING can be PARSED !!!"); break;

}

}

if(j==pro[0].n)

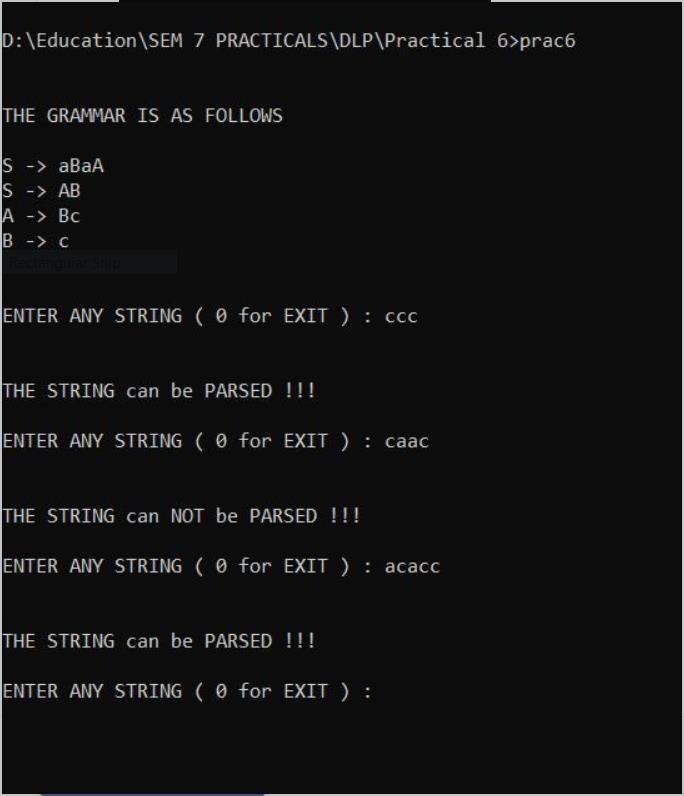
printf("\n\nTHE STRING can NOT be PARSED !!!");

}

getch();

}

**OUTPUT:**



**CONCLUSION:**

In this practical, we learnt about Context Free Grammar and implemented the concept using C.

# PRACTICAL 9

## AIM:

Implementation of code generator.

## IMPLEMENTATION:

* gcc <our .c file> -o <file name for exe file>
* <filename of exe file>

Content of Input1.txt:

a=b+c; d=n+s; p=q;

## PROGRAM:

// Pgm for Code generation by using simple code generation algorithm

#include<stdio.h> #include<string.h> struct table{

char op1[2]; char op2[2]; char opr[2]; char res[2];

}tbl[100];

void add(char \*res,char \*op1, char \*op2,char \*opr)

{

FILE \*ft;

char string[20]; char sym[100];

ft=fopen("result.asm","a+");

if(ft==NULL)

ft=fopen("result.asm","w");

printf("\nUpdating Assembly Code for the Input File : File : Result.asm ; Status [ok]\n");

//sleep(2); strcpy(string,"mov r0,"); strcat(string,op1); if(strcmp(opr,"&")==0)

{

}

else

{

}

//do nothing

strcat(string,"\nmov r1,"); strcat(string,op2);

fputs(string,ft); if(strcmp(opr,"+")==0)

strcpy(string,"\nadd r0,r1\n"); else if(strcmp(opr,"-")==0)

strcpy(string,"\nsub r0,r1\n"); else if(strcmp(opr,"/")==0)

strcpy(string,"\ndiv r0,r1\n"); else if(strcmp(opr,"\*")==0)

strcpy(string,"\nmul r0,r1\n"); else if(strcmp(opr,"&")==0)

strcpy(string,"\n");

else

strcpy(string,"\noperation r0,r1\n");

fputs(string,ft); strcpy(string,"mov ");

}

main()

{

strcat(string,res); strcat(string,", r0\n"); fputs(string,ft); fclose(ft); string[0]='\0';

sym[0]='\0';

int res,op1,op2,i,j,opr; FILE \*fp;

char filename[50]; char s,s1[10];

system("clear"); remove("result.asm"); remove("result.sym"); res=0;op1=0;op2=0;i=0;j=0;opr=0;

printf("\n Enter the Input Filename with no white spaces:"); scanf("%s",filename);

fp=fopen(filename,"r"); if(fp==NULL)

{

}

else

{

printf("\n cannot open the input file !\n"); return(0);

while(!feof(fp))

{

s=fgetc(fp);

if(s=='=')

{

res=1; op1=op2=opr=0; s1[j]='\0';

strcpy(tbl[i].res,s1); j=0;

}

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

{

op1=1; opr=1; s1[j]='\0';

tbl[i].opr[0]=s;

tbl[i].opr[1]='\0';

strcpy(tbl[i].op1,s1); j=0;

}

else if(s==';')

{

if(opr) // for 3 operand format ex: a=b+c;

{

op2=1; s1[j]='\0';

strcpy(tbl[i].op2,s1);

}

else if(!opr) // for 2 operand format ex: d=a;

{

op1=1;

expr

-

op2=0; s1[j]='\0';

strcpy(tbl[i].op1,s1);

strcpy(tbl[i].op2,"&");// simplifying the strcpy(tbl[i].opr,"&"); //-------"--"---------

}

}

system("clear");

}

return 0;

}

}

else

{

}

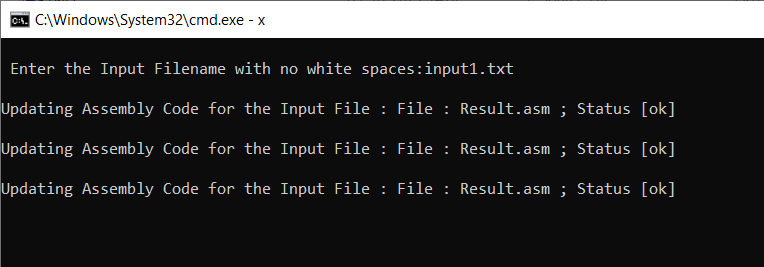
add(tbl[i].res,tbl[i].op1,tbl[i].op2,tbl[i].opr); i++;

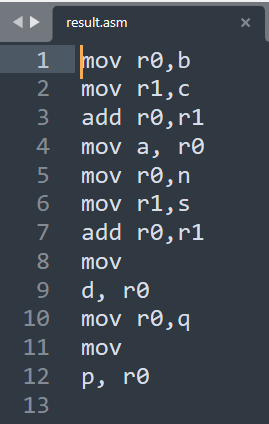
j=0;

opr=op1=op2=res=0;

s1[j]=s; j++;

## OUTPUT:





**CONCLUSION:**

In this practical, we learnt about code generation and implemented the same using C.

**PRACTICAL 10**

**AIM:** Implementation of code optimization for Common sub-expression elimination, Loop invariant code movement.

**IMPLEMENTATION:**

yacc <filename with .y extension>

• gcc <newly created .c file> -o <file name for exe file>

• <filename of exe file>

**PROGRAM:**

#include <stdio.h>

int main() {

int i, sum = 0;

for (i = 1; i <= 10; i++) {

int square = i \* i;

sum += square;

}

printf("Sum of squares: %d\n", sum);

return 0;

}

In this code, the expression int square = i \* i; is computed inside the loop. However, this calculation is loop-invariant because it doesn't depend on the loop variable i. We can optimize the code by moving this calculation outside the loop:

#include <stdio.h>

int main() {

int i, sum = 0;

int square; // Declare outside the loop

for (i = 1; i <= 10; i++) {

square = i \* i; // Calculate once

sum += square;

}

printf("Sum of squares: %d\n", sum);

return 0;

}

By moving the calculation of square outside the loop, we avoid redundant calculations and potentially improve the code's efficiency. This is a simple example of loop-invariant code movement.

**CONCLUSION:**

In this practical we learnt about Loop invariant code movement.