**PRACTICAL 1**

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

**THEORY:**

The first phase in a compiler reads the input source and converts strings in the source to tokens. Using regular expressions, we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically, an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.

The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

letter(letter|digit)\*

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

• repetition, expressed by the “\*” operator

• alternation, expressed by the “|” operator

• concatenation

We need one text file also and write example C code so that our program will identify the different types of tokens.

#include<stdio.h>

void main()

{

int ab;

printf("hii");

}

**CODE:**

%%

"#" {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"|"float"|"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)

{

printf("Unable To Open File");

}

else

{

yyin=fp;

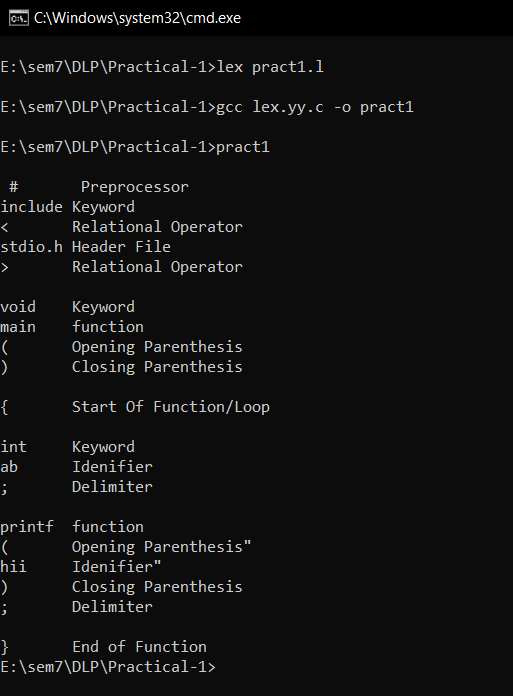
}

yylex();

return 0;

}

**OUTPUT:**

****

**CONCLUSION:**

In this practical we have implemented lexical analyzer which will identified the tokens present in the C program.

**PRACTICAL 2**

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

**THEORY:**

Regular expressions in lex are composed of meta characters. Within a character class, normal operators lose their meaning. Two operators allowed in a character class are the hyphen (“-”) and circumflex (“^”). When used between two characters, the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string, the longest match wins. In case both matches are the same length, then the first pattern listed is used.

**... definitions ...**

**%%**

**... rules ...**

**%%**

**... subroutines ...**

Input to Lex is divided into three sections, with %% dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file: %%

**CODE:**

bin (0|1)\*

oct [0-7]

char [A-Za-z]\*

dec [0-9]\*

digit [0-9]

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

expo {digit}+("."{digit}+)?("E"("+"|"-")?{digit}+)?

hex [0-9a-fA-f]+

%%

{bin} printf("Enter no. is a binary number");

{oct} printf("Enter no. is a Octal number");

{char} printf("Enter input is Char");

{dec} printf("Enter input is decimal number");

{float} printf("Enter input is float number");

{expo} printf("Enter input is expo. number");

{hex} printf("Enter input is hex number");

%%

int yywrap()

{

return 1;

}

int main()

{

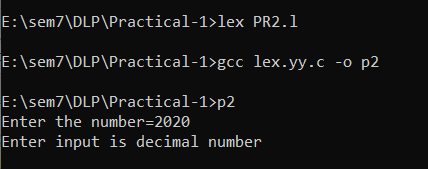
printf("Enter the number=");

yylex();

return 0;

}

**OUTPUT:**

****

**CONCLUSION:**In this practical we have implemented lexical analyzer for identification of numbers.

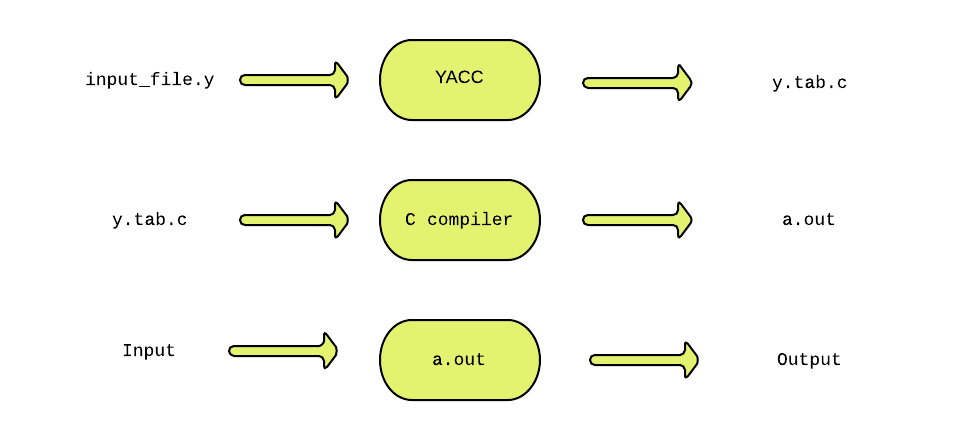
**PRACTICAL 3**

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

**THOERY:**

**Introduction to YACC:**

YACC (Yet Another Compiler Compiler) is a tool used to generate a parser. This document is a tutorial for the use of YACC to generate a parser for ExpL. YACC translates a given Context Free Grammar (CFG) specifications (input in input\_file.y) into a C implementation (y.tab.c) of a corresponding push down automaton (i.e., a finite state machine with a stack). This C program when compiled, yields an executable parser.



The source SIL program is fed as the input to the generated parser ( a.out ). The parser checks whether the program satisfies the syntax specification given in the input\_file.y file.

YACC was developed by Stephen C. Johnson at Bell labs.

**Parser:**

A parser is a program that checks whether its input (viewed as a stream of tokens) meets a given grammar specification. The syntax of SIL can be specified using a Context Free Grammar. As mentioned earlier, YACC takes this specification and generates a parser for SIL.

**Context Free Grammar (CFG):**

A context free grammar is defined by a four tuple (N,T,P,S) - a set N of non-terminals, a set T of terminals (in our project, these are the tokens returned by the lexical analyzer and hence we refer to them as tokens frequently), set P of productions and a start variable S. Each production consists of a non-terminal on the left side (head part) and a sequence of tokens and non-terminals (of zero or more length) on the right side (body part). We will explore productions further in detail later in this documentation. For more about context free grammars refer to this wiki .

Example: This example is an Infix to Postfix converter implemented using YACC. The rules part of the YACC program has been shown below:

**CODE:**

%{

/\*\*\* 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;

}

}

main()

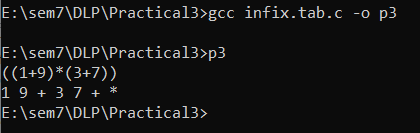
{

yyparse();

return 1;

}

**OUTPUT:**

****

**CONCLUSION:**

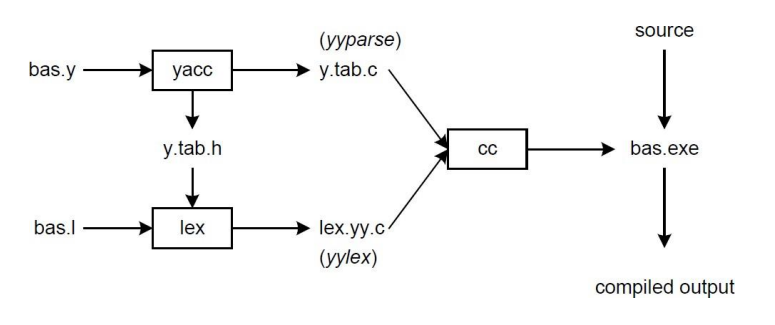
In this practical we implemented infix to postfix conversion using parser and YACC.

**PRACTICAL 4**

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

**THEORY:**

Before 1975 writing a compiler was a very time-consuming process. Then Lesk [1975] and Johnson [1975] published papers on lex and yacc. These utilities greatly simplify compiler writing. Implementation details for lex and yacc may be found in Aho [2006]. Flex and bison, clones for lex and yacc, can be obtained for free from GNU and Cygwin



**CODE:**

DIGIT [0-9]

%option noyywrap

%%

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

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

**C1.y**

%{

#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)

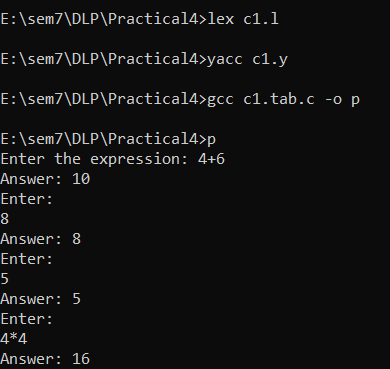
{

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

exit (1);

}

**OUTPUT:**

****

**CONCLUSION:**

Here I have Implemented a Calculator using LEX and YACC.

**PRACTICAL 5**

**AIM:** Implementation of Syntax Tree

**CODE:**

#include<conio.h>

#include<stdio.h>

int main(){

FILE \*fp;

int i=0,j=0,k,l,row,col,s,x;

char a[10][10],ch,main[50],search;

//clrscr();

fp=fopen("syntax.txt","r+");

while((ch=fgetc(fp))!=EOF){

if(ch=='\n'){

row=i;

col=j;

j=0;

i++;

}

else{

a[i][j]=ch;

j++;

}

}

printf("\n");

for(k=0;k<row+1;k++){

for(l=0;l<col;l++){

printf("%c",a[k][l]);

}

printf("\n");

}

i=0;

s=0;

for(k=0;k<row+1;k++){

main[i]=a[k][1];

i++;

if(a[k][3]=='t'){

search=a[k][4];

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

if(main[l]==search){

main[i]=main[l];

i++;

break;

}

}

main[i]=a[k][5];

s=5;

i++;

}

else{

main[i]=a[k][3];

// printf("\n%c",main[i]);

i++;

main[i]=a[k][4];

// printf(",%c\n",main[i]);

s=4;

i++;

}

s++;

if(a[k][s]=='t') {

s++;

search=a[k][s];

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

if(main[l]==search) {

main[i]=main[l];

i++;

break;

}

}

}

else{

main[i]=a[k][s];

i++;

}

}

for(x=i-1;x>=0;x=x-4){

printf("\ntt%c: root->%c ",main[x-3],main[x-1]);

if(main[x-2]>48 &&main[x-2]<59)

printf("lc->t%c ",main[x-2]);

else

printf("lc->%c ",main[x-2]);

if(main[x]>48 &&main[x]<59)

printf("rc->t%c ",main[x]);

else

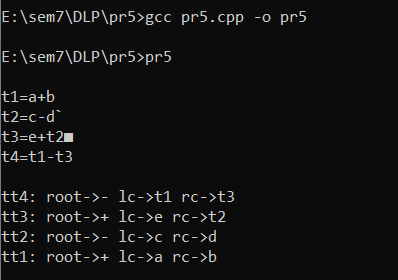
printf("rc->%c ",main[x]);

}

getch();

}

**OUTPUT:**

****

**CONCLUSION:**

Here we have implemented Syntax tree.