**Aim:- Write a program to generate token using LEX.**

**Topic Name:- LEX**

**Theoretical background:-**

Lex is officially known as a "Lexical Analyser".It's main job is to break up an input stream into more usable elements.Or in, other words, to identify the "interesting bits" in a text file.For example, if you are writing a compiler for the C programming language, the symbols { } ( ) ; all have significance on their own. The letter a usually appears as part of a keyword or variable name, and is not interesting on it's own. Instead, we are interested in the whole word. Spaces and newlines are completely uninteresting, and we want to ignore them completely, unless they appear within quotes "like this".All of these things are handled by the Lexical Analyser.

Yacc is officially known as a "parser".It's job is to analyse the structure of the input stream, and operate of the "big picture".In the course of it's normal work, the parser also verifies that the input is syntactically sound.Consider again the example of a C-compiler. In the C-language, a word can be a function name or a variable, depending on whether it is followed by a ( or a = There should be exactly one } for each { in the program.YACC stands for "Yet Another Compiler Compiler". This is because this kind of analysis of text files is normally associated with writing compilers.However, as we will see, it can be applied to almost any situation where text-based input is being used.

For example, a C program may contain something like:

{

intint;

int = 33;

printf("int: %d\n",int);

}

In this case, the lexical analyser would have broken the input sream into a series of "tokens", like this: {

int

int

;

int

=

33

;

printf

(

"int: %d\n"

,

int

)

;

}

Note that the **lexical analyser** has already determined that where the keyword int appears within quotes, it is really just part of a litteral string. It is up to the **parser** to decide if the token int is being used as a keyword or variable. Or it may choose to reject the use of the name int as a variable name. The parser also ensures that each statement ends with a ; and that the brackets balance.

Lex turns the user's expressions and actions (called source in this memo) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

+-------+

Source ->| Lex | ->yylex

+-------+

+-------+

Input -> |yylex | -> Output

+-------+

An overview of Lex

Figure 1

The general format of Lex source is:

{definitions}

%%

{rules}

%%

{user subroutines}

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

%%

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

**To compile and run lex program:**

(Note:- you have to store lex program using .l extention)

* lex programname.l
* gcc lex.yy.c
* ./a.out

**Code with Comments:-**

%{

#include<stdio.h>

%}

del [,|;]

keyword "int"|"float"|"double"|"char"

digit [0-9]

letter [a-zA-Z]

op [+|\*|/]

op1 "-"

un [\_]

sc [@|$|&|#|!]

%%

({keyword})+ {printf("%s-keyword\n",yytext);}

({digit})+ {printf("%s-digit\n",yytext);}

({op}|{op1})+ {printf("%s-operator\n",yytext);}

({letter}|{un})(({letter}|{digit})\*)+ {printf("%s-identifier \n",yytext);}

({sc}|({digit})\*)+({letter})+ {printf("%s-invalid token \n",yytext);}

({del})+ {printf("%s-deliminator \n",yytext);}

%%

int main(void)

{

yylex();

return 0;

}

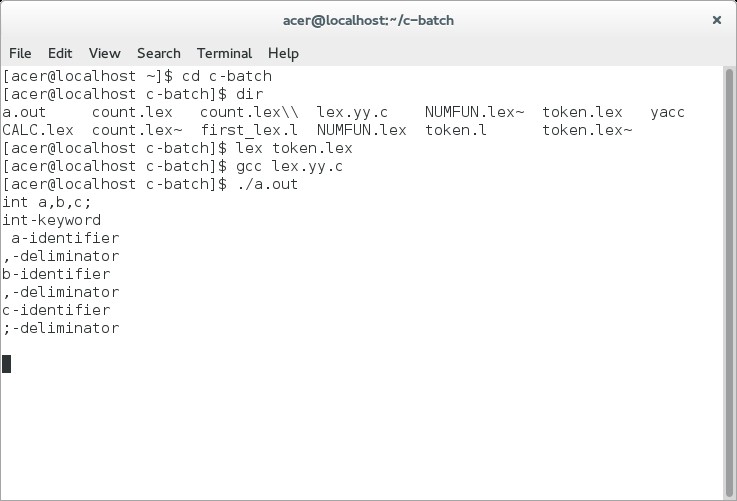
intyywrap()

{

return 1;

}

**Output:**

****

**Aim:-Write a program to count numbers of tokens in given string.**

**Code with Comments:-**

%{

#include<stdio.h>

int k=0,i=0,d=0,del=0,o=0,t=0;

%}

DEL [;]

KEY ["float"|"int"]

OP [+|\*|/|=]

OP1 "-"

DIGIT [0-9]

LETTER [a-zA-Z]

%%

{KEY} {k++,t++;}

({DIGIT})+ {d++,t++;}

{LETTER}({LETTER}|{DIGIT})\* {i++,t++;}

({OP}|{OP1}) {o++,t++;}

{DEL} {del++,t++;}

printf("%d-keyword \n %d-identifier \n %d-operator \n %d-digit\n %d-delimeter\n %d-tokens",k,i,o,d,del,t);

}

%%

void main()

{

yylex();

}

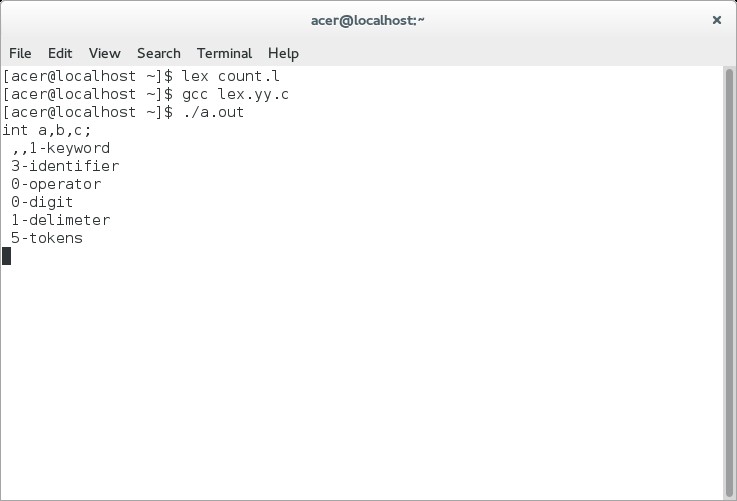
intyywrap()

{

return 1;

}

**Output:**

****

**Aim:-Write a program to convert form words format to number format.**

**Code with Comments:-**

%{

#include<stdio.h>

int a=0,b=0,c=0;

%}

END[;]

%%

ONE {a=a+1;}

TWO {a=a+2;}

THREE {a=a+3;}

FOUR {a=a+4;}

FIVE {a=a+5;}

SIX {a=a+6;}

SEVEN {a=a+7;}

EIGHT {a=a+8;}

NINE {a=a+9;}

TEN {a=a+10;}

ELEVEN {a=a+11;}

TWELVE {a=a+12;}

THIRTEEN {a=a+13;}

FOURTEEN {a=a+14;}

FIFTEEN {a=a+15;}

SIXTEEN {a=a+16;}

SEVENTEEN {a=a+17;}

EIGHTEEN {a=a+18;}

NINETEEN {a=a+19;}

TWENTY {a=a+20;}

THIRTY {a=a+30;}

FOURTY {a=a+40;}

FIFTY {a=a+50;}

SIXTY {a=a+60;}

SEVENTY {a=a+70;}

EIGHTY {a=a+80;}

NINTY {a=a+90;}

HUNDRED {b=a\*100;a=0;}

THOUSAND {c=a\*1000;a=0;}

{END} {printf("%d\n",a+b+c);a=0;b=0;c=0;}

%%

void main ()

{

yylex();

}

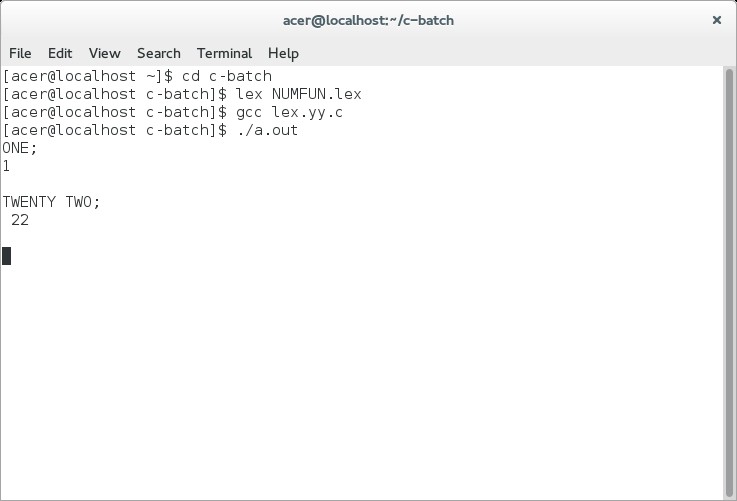
intyywrap ()

{

return 1;

}

**Output:**



**Aim:-**Write a program for Calculator.

**Code with Comments:-**

%{

#include<stdio.h>

inta,b,c,op=0;

%}

DIGIT [0-9][0-9]\*

ADD [+]

SUB "-"

MUL [\*]

DIV [/]

%%

{DIGIT}+ {digit();}

{ADD} {op=1;}

{SUB} {op=2;}

{MUL} {op=3;}

{DIV} {op=4;}

%%

digit()

{

if (op==0)

{

a = atoi(yytext);

}

else

b = atoi(yytext);

switch (op)

{

case 1: c=a+b;

printf("%d + %d = %d\n",a,b,c);

op=0;

break;

case 2: c=a-b;

printf("%d - % d = %d\n",a,b,c);

op=0;

break;

case 3: c=a\*b;

printf("%d \* %d = %d\n",a,b,c);

op=0;

break;

case 4: c=a/b;

printf("%d / %d = %d\n",a,b,c);

op=0;

break;

}

}

void main ()

{

yylex();

}

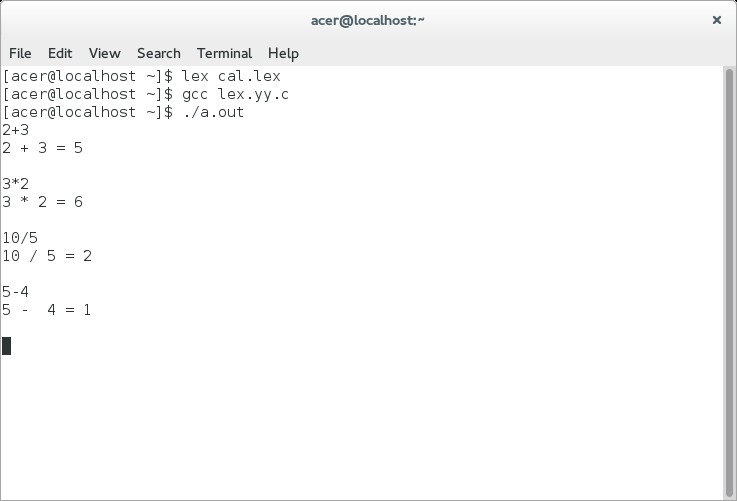
intyywrap ()

{

return 1;

}

**Output:**

****

**Related exercise:**

1. Write a lex program which count number of keywords, identifier and operators.

**Aim:-Write a program for Calculator using YACC.**

**Topic Name:- YACC**

**Theoretical Background:-**

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions. Yacc is written in a portable dialect of C Ritchie Kernighan Language Prentice and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

date :month\_name day ',' year ;

Here, date, month\_name, day, and year represent structures of interest in the input process; presumably, month\_name, day, and year are defined elsewhere. The comma ``,'' is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

July 4, 1776

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal. To avoid confusion, terminal symbols will usually be referred to as tokens.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

month\_name : 'J' 'a' 'n' ;

month\_name : 'F' 'e' 'b' ;

. . .

month\_name : 'D' 'e' 'c' ;

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and month\_name would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a month\_name was seen; in this case, month\_name would be a token.

Literal characters such as ``,'' must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

date : month '/' day '/' year ;

allowing

7 / 4 / 1776

as a synonym for

July 4, 1776

In most cases, this new rule could be ``slipped in'' to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere. Aho Johnson Surveys LR Parsing Aho Johnson Ullman Ambiguous Grammars Aho Ullman Principles Compiler Design Yacc has been extensively used in numerous practical applications, including lint, Johnson Lint the Portable C Compiler, Johnson Portable Compiler Theory and a system for typesetting mathematics. Kernighan Cherry typesetting system CACM

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

**Basic Specifications**

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sec- tions: the declarations, (grammar) rules, and programs. The sections are separated by double percent ``%%'' marks. (The percent ``%'' is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

declarations

%%

rules

%%

programs

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

%%

rules

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /\* . . . \*/, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

A : BODY ;

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ``.'', underscore ``\_'', and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes ``'''. As in C, the backslash ``\'' is an escape character within literals, and all the C escapes are recognized. Thus

'\n' newline

'\r' return

'\'' single quote ``'''

'\\' backslash ``\''

'\t' tab

'\b' backspace

'\f' form feed

'\xxx' ``xxx'' in octal

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar ``|'' can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

A : B C D ;

A : E F ;

A : G ;

can be given to Yacc as

A : B C D

| E F

| G ;

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

empty : ;

Names representing tokens must be declared; this is most simply done by writing

%token name1 name2 . . .

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

%start symbol

The end of the input to the parser is signaled by a special token, called the endmarker. If the tokens up to,

but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it accepts the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as ``end-of-file'' or ``end-of-record''.

**Actions**

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces ``{'' and ``}''. For example,

A : '(' B ')' { hello( 1, "abc" ); }

and

XXX : YYY ZZZ { printf("a message\n"); flag = 25; }

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol ``dollar sign'' ``$'' is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable ``$$'' to some value. For example, an action that does nothing but return the value 1 is

{ $$ = 1; }

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

A : B C D ;

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

expr : '(' expr ')' ;

The value returned by this rule is usually the value of the expr in parentheses. This can be indicated by

expr : '(' expr ')' { $$ = $2 ; }

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

A : B ;

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

A : B { $$ = 1; } C { x = $2; y = $3; } ;

the effect is to set x to 1, and y to the value returned by C.

Actions that do not terminate a rule are actually han- dled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

$ACT : /\* empty \*/ { $$ = 1; } ;

A : B $ACT C { x = $2; y = $3; } ;

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function node, written so that the call

node( L, n1, n2 )

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

expr : expr '+' expr { $$ = node( '+', $1, $3 ); }

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks ``%{'' and ``%}''. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

%{ int variable = 0; %}

could be placed in the declarations section, making variable accessible to all of the actions. The Yacc parser uses only names beginning in ``yy''; the user should avoid such names.

**To compile and run Yacc program:-**

* First compile lex program ( lex programname.l)
* Then compile Yacc program ( yacc programname.y)
* gcc y.tab.c
* ./a.out

**Code with Comments:-**

**calc.lex file**

%{

   /\* 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];

%%

intyywrap()

{

return 1;

}

**Calc.yacc file:**

%{

#include<stdio.h>

Void yyerror(char \*c);

%}

%token NUM;

%start S;

%%

S:E {$$=$1;};

E:E'+'T {$$=$1+$3;};

E:T {$$=$1;};

T:T'\*'F {$$=$1\*$3;};

T:F {$$=$1;};

F:NUM{$$=$1;};

%%

#include "lex.yy.c"

Void yyerror(char \*s)

{

fprintf(stderr,"%s",s);

}

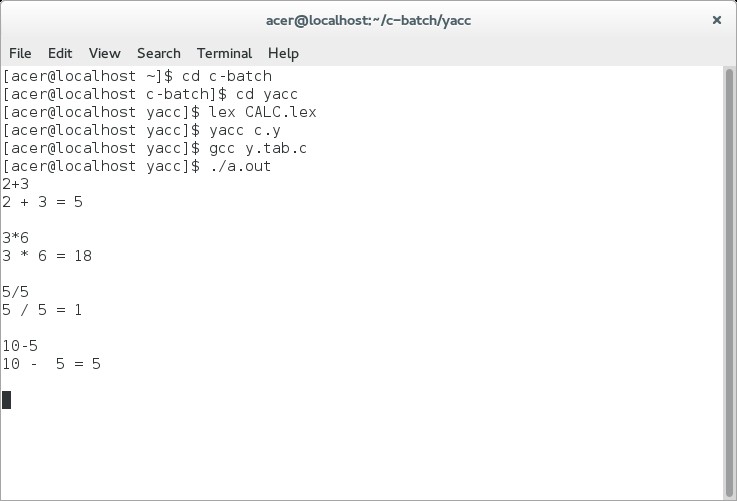
void main()

{

yyparse();

}

**Output:**

****

**Aim: Write a Yacc Program to convert from infix to postfix.**

**Code with Comment:-**

### gram.l

%{

#include"y.tab.h"

extern int yylval;

%}

%%

[0-9]+ {yylval=atoi(yytext); return NUM;}

\n return 0;

. return \*yytext;

%%

int yywrap(){

return 1;

}

### gram.y

%{

#include<stdio.h>

%}

%token NUM

%left '+' '-'

%left '\*' '/'

%right NEGATIVE

%%

S: E {printf("\n");}

;

E: E '+' E {printf("+");}

| E '\*' E {printf("\*");}

| E '-' E {printf("-");}

| E '/' E {printf("/");}

| '(' E ')'

| '-' E %prec NEGATIVE {printf("-");}

| NUM {printf("%d", yylval);}

;

%%

int main(){

yyparse();

}

int yyerror (char \*msg) {

return printf ("error YACC: %s\n", msg);

}

**Output:-**

yacc -d gram.y

flex gram.l

cc lex.yy.c y.tab.c

./a.out

example: 2+6\*2-5/3 --> output: 262\*+53/-