LAB MANUAL

Name: SHAH HETVI

Enrollment No.: 180850131016

Class: CSE sem. 7

Subject: Compiler Design

Subject Code: 3170701

HJD INSTITUTE OF TECHNICAL EDUCATION AND RESEARCH, KERA
COMPUTER ENGINEERING DEPARTMENT

HJD INSTITUTE OF TECHNICAL
EDUCATION AND RESEARCH, KERA

Approved by AICTE and affiliated to
Gujarat Technological University

CERTIFICATE

This is to certify that **Miss SHAH HETVI Enrollment No 180850131016** of programme **BE 7th SEM Computer Engineering** has satisfactorily completed **her** term work in **Compiler Design(3170701)** for the term ending in 2021-2022.

Miss. Priyanka Ahir
(Lab In-charge)

Mr. Vishal Bhimani
(Head Of Department)

Practical - 1

Aim: To study about LEX, YACC and FLEX.

LEX a Lexer Generator

LEX is officially known as a "Lexical Analyzer".

Its 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 notinteresting on its own. Instead, we are interested in the whole word.

Spaces and newlines are completely uninteresting, and we want to ignore themcompletely, unless they appear within quotes "like this".

All of these things are handled by the Lexical Analyzer.

During the first phase the compiler reads the input and converts strings in thesource to tokens.

With regular expressions we can specify patterns to lex so it can generate a codethat 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 bythe use 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 scan for identifiers. Lex will read this pattern and procedure 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.

Form of a LEX specification file:

Definitions

%%

Rules

%%

User code

The definitions section: "name definition"

The rules section: "pattern action"

The user code section: "yylex() routine"

Yacc a Parser Generator:

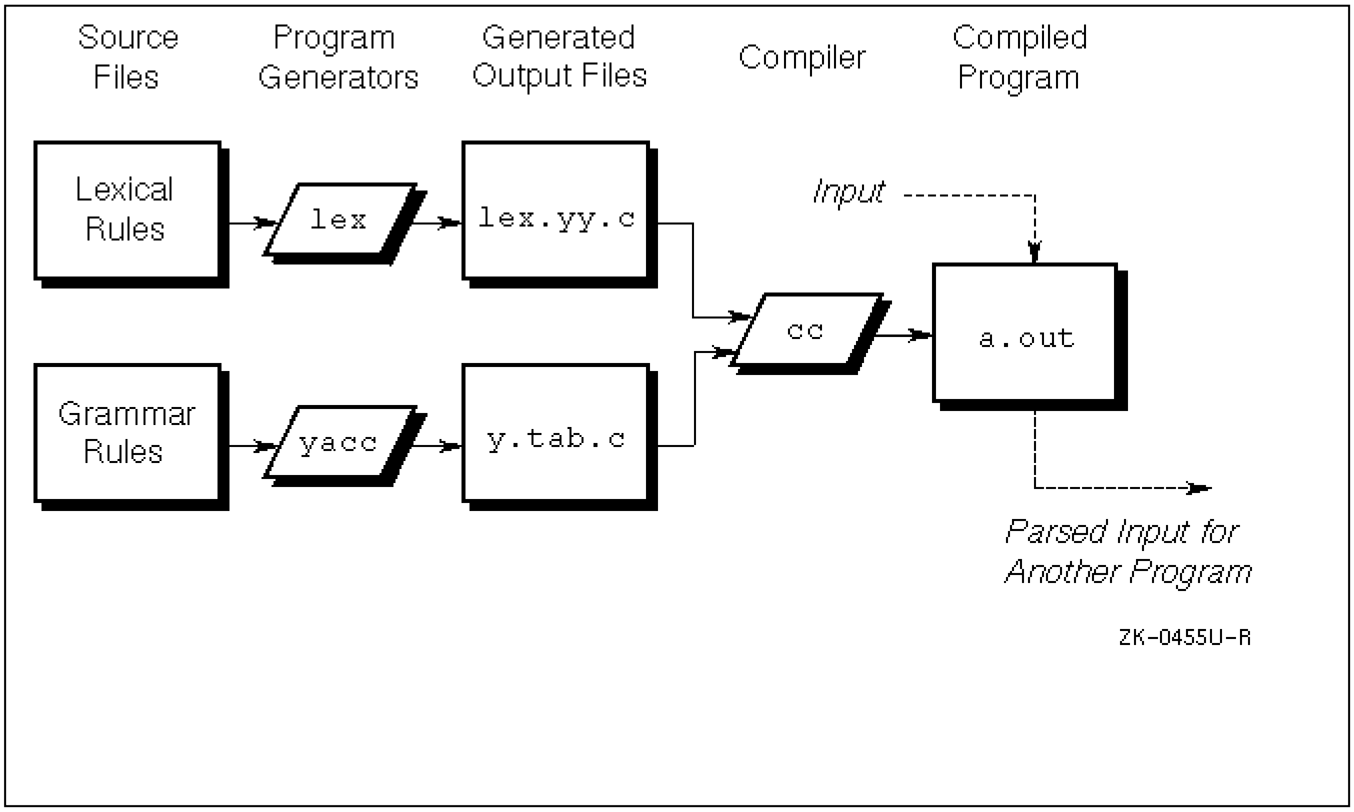
Yacc is officially known as a "parser". Its job is to analyze the structure of the input stream. In the course of its normal work, the parser also verifies that the input is syntactically sound. Consider again the example of a C-compiler. In the C-language, a word can be a function name or a variable, 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. For example, a C program may contain something like:

```
{
 int int;
 int = 33;
 printf("int: %d\n",int);
}
```

In this case, the lexical analyzer would have broken the input stream into a series of "tokens", like this:

```
{
Int;
Int=33;
Printf(" int:%d\n",int);
}
```

Note that the lexical analyzer has already determined that where the keyword int appears within quotes, it is really just part of a literal 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.



Form of a yacc specification file:

Declaration

%%

Grammar rules

Programs

Yacc : Grammar Rules

Terminals (tokens):

• Names must be declared:

• %token name 1 name 2 ...

•Any name not declared as a token in the declarations section is assumed to be a nonterminal.

Start Symbol :

• may be declared, via : %start name

• if not declared explicitly, defaults to the nonterminal on the LHS of the first grammar rule listed.

Production :

A grammar production A->B1|B2|Bn is written as :

A:B1|B2|Bn

Example:

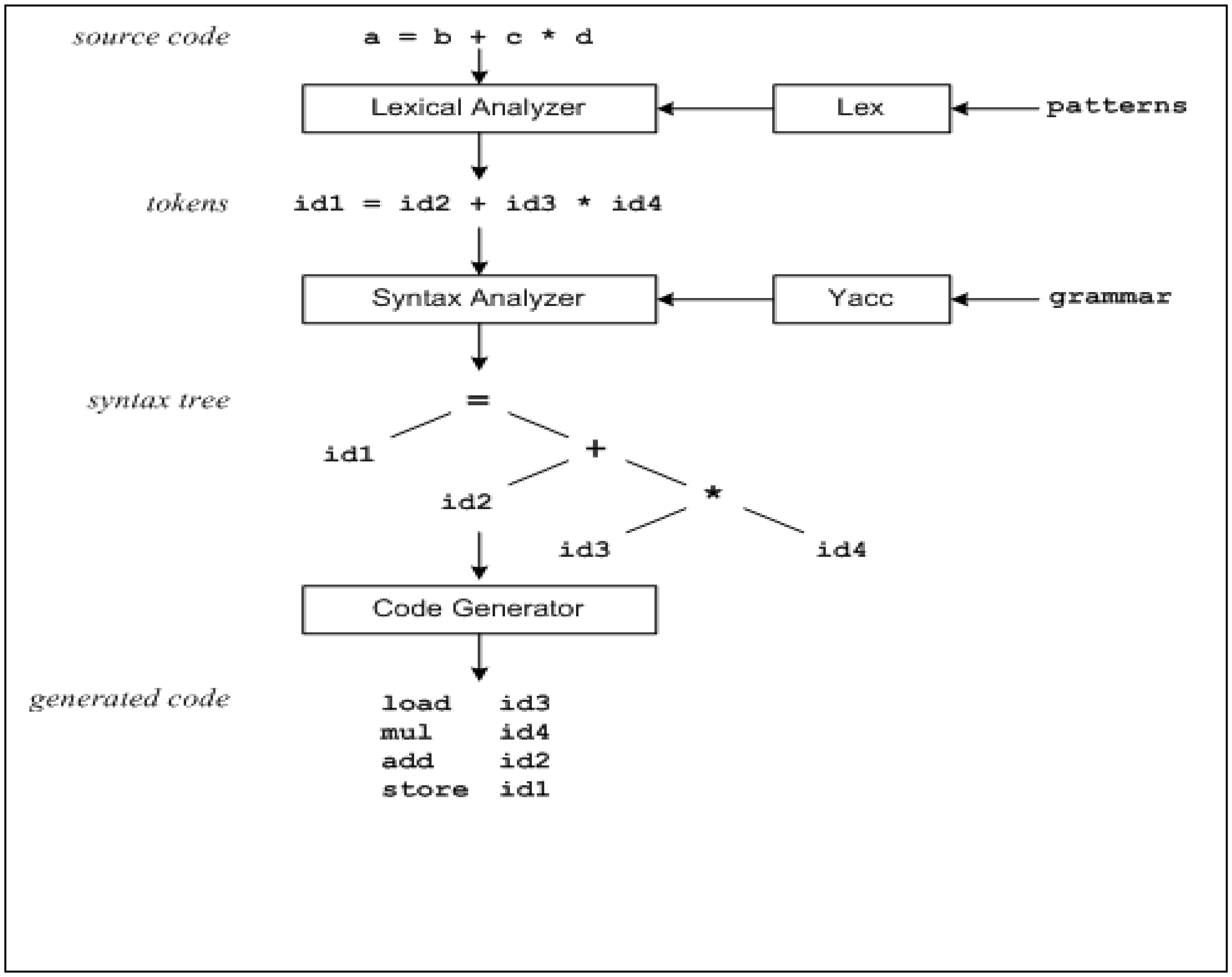
stmt : KEYWORD_IF(' expr ') stmt;

Communication between Scanner and Parser :

Pattern in bellow figure is a file you create with a text editor. Lex will read your patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in input, based on your patterns, and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing.

When the lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type (integer to real) and location of each variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.

The grammar in the below diagram is a text file you create with a text editor Yacc will read your grammar and generate C code for a syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree.



Previous figure illustrates the file naming conventions used by lex and yacc. We'll assume our goal is to write a BASIC compiler. First, we need to specify all pattern matching rules for lex (bas.1) and grammar rules for yace (bas.y). Commands to create our compiler, bas.exe, are listed below:

yace -d bas.y   # create y.tab.h, y.tab.c

lex bas.1   #create lex.yy.c

cc lex.yy.c y.tab.c -obas.exe   #compile/link

Yacc reads the grammar descriptions in bas.y and generates a syntax analyzer (parser), that includes function yyparse, in file y.tab.c. Included in file bas.y are token declarations. The -d option causes yacc to generate definitions for tokens and place them in file y.tab.h. Lex reads the pattern descriptions in bas.l, includes file y.tab.h, and generates a lexical analyzer, that includes function yylex, in file lex.yy.c.

Finally, the lexer and parser are compiled and linked together to create executable bas.exe. From main we call yyparse to run the compiler. Function yyparse automatically calls yylex to obtain each token.

FLEX

FLEX is an open source version of LEX. Flex is an open source program designed to automatically and quickly generate scanners, also known as tokenizers, which recognize lexical patterns in text. Flex is an acronym that stands for "fast lexical analyzer generator. It is a free alternative to Lex, the standard lexical analyzer generator in Unix-based systems. Flex was originally written in the C programming language by Vern Paxson in 1987. Lex is proprietary but versions based on the original code are available as open source. FLEX (Fast LEXical analyzer generator) is a tool for generating scanners. First, FLEX reads a specification of a scanner either from an input file * lex, or from standard input, and it generates as output a C source file lex.yy.c. Then, lex.yy.c is compiled and linked with the "-Ifl" library to produce an executable a.out. Finally, a.out analyzes its input stream and transforms it into a sequence of tokens.

Practical - 2

Aim: LEX program to count the number of comment lines in a given C program. Also eliminate them and copy that program into separate file.

Lex File:

```
%{
int c=0;
%}
T(*)\n {c++;fprintf(yyout, "");}
 "/"(."\n)*"*/" {c++;fprintf(yyout, ""); }
%%
int main()
{     yyin=fopen("pr2.c","r");
   yylex();
   printf("comment=%d\n",c);
   return 0;     }
```

C FILE:

```
 #include<stdio.h>
int main()
printf("hello good morning");
//comment 1
 /*comment 2
Ghg
```

```
 hhtdty
fjkgh
dhjlkdfgi
dkljsg
kjarht*/
return 0; }
```

Output:

```
number of comments are = 2
```

Practical - 3

Aim: LEX Program to Count no. of Lines, Words, Letters and Special Characters from given input.

Lex File:

```
%{
 int s letter=0,c_letter=0,words=0, line=0,num=0,special _c=0,total=0;
%}
%%
\n {line++; words++; }
 [\t’’ ] {words++; }
 [a-z] (s_letter++; }
 [A-Z] {c_letter++;}
 [0-9] {num++; }
 {special_c++;)
%%
int main() {
yyin=fopen(“j.text","r");
yylex();
 total=s_letter+c_letter+num+special_c;
printf(" This File contains ...");
printf("\n\t%d lines", lines);
printf("\n\t%d words",words);
printf("\n\t%d small letters", s_letters);

printf("\n\t%d capital letters",c_letters);
printf("\n\t%d digits", num);

printf("\n\t%d special characters",spl_char);
```

Text file:

```
This is my 1st lex program!!!
                  Cheers!! It works!!:)
```

Output:

```
This file contains…

2 lines

9 words

30 small letters

3 capital letters

1 digits

9 special characters
```

Practical - 4

Aim: C Program to check that given input in Identifier, keyword, Operator or Digit.

C FILE:

```
#module<stdio.h>

#include<ctype.h>

#module< string.h>

int main()

{

chat s[20].c;

printf("Enter Input: ");

scanf("%s",s);

 c= [0];

 if (isalpha(c)) {

if(strcmp(s,"do")==0 ||strcmp(s,"for") ==0 ||strcmp(s, "if")==0)||strcmp(s,
"main") =0)

{

printf("Input is Keyword"); }

else {printf("Input is Identifire");  }   }

else if(stremp(s,"+") ==0||stremp(s, "*" )== 0) ||strcmp(s, "/")==0)

{     printf("Input is Operator"); }

else if(isdigit(c))
```

```
{
printf("Input is Number");
}
 else
{
printf("Invalid Input");
}
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
```

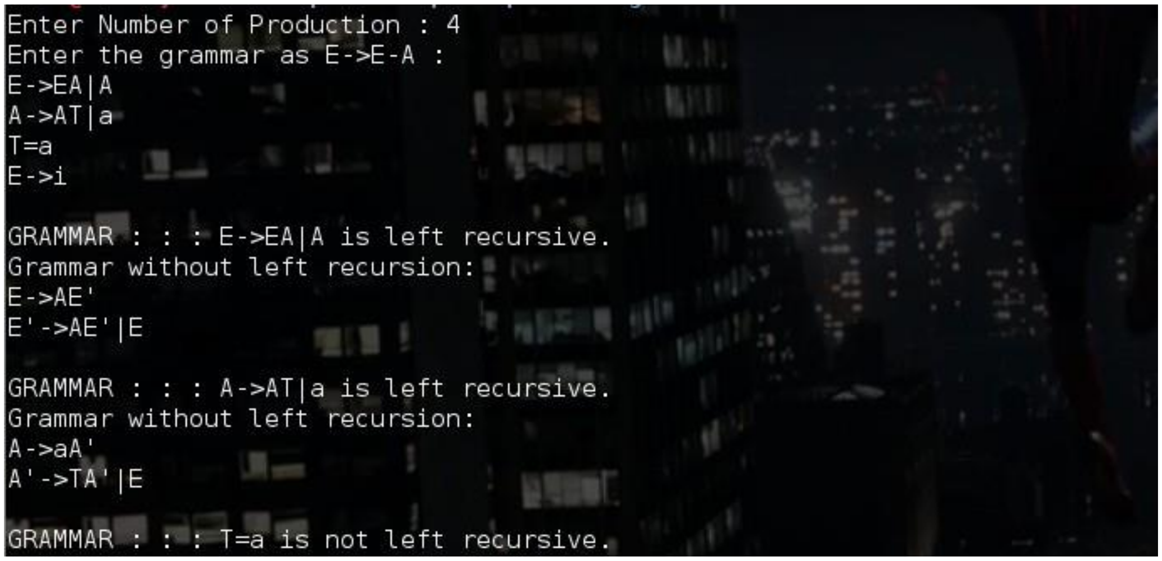
Practical - 5

Aim: C Program to remove Left Recursion from given grammar Production.

C File:

```
#include<stdio.h>
 int main()  {
int count=4,1,j,p;
 char af[10].pr.alpha[10].beta[10];
 printf("Enter production:");
 scanf("%s",a);
 if (a[0]=a[31])
{
printf("\n Grammar is left recursive");
 pr=a[3];
for(i=4 j=0;a[i]!="|”; i++)  {
alpha[j]=a[i];
j++;
count++; }
alpha[j]='\0';
for(j=count+1,p=0;a[j]!='\0';j++)  {
beta[p]=a[j];  }
betalj+1]='\0';
 printf("\n after removing left recursion");
 printf("\n% c>",pr);

for(i=0;beta[i]!='\0';i++) {

printf("%c”,beta[i]);    printf("%c"",pr);

printf("\n %c'-&gt;".pr); for(i=0:alpha[i]!='\0';i++)

printf("%c".alpha[i]);

printf("%c "le",pr);

else

printf("\n No left recursion");

return 0;
```

Output:



```
Enter Number of Production : 4
Enter the grammar as E->E-A :
E->EA|A
A->AT|a
T=a
E->i

GRAMMAR : : : E->EA|A is left recursive.
Grammar without left recursion:
E->AE'
E'->AE'|E

GRAMMAR : : : A->AT|a is left recursive.
Grammar without left recursion:
A->aA'
A'->TA'|E

GRAMMAR : : : T=a is not left recursive.
```

Practical - 6

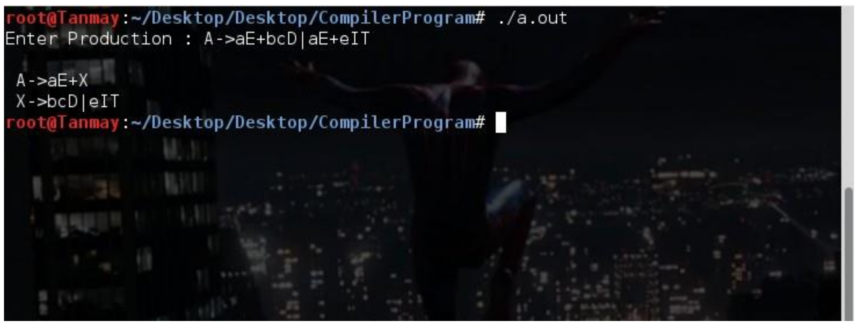
Aim: C Program to Left Factor the given grammar production.

C File:

```
 #include
 int main()  {
char a[2],b[2],c[20],alpha[20], beta 1[20],beta2[20];
int count=0,i,j,k,q,w,e,al=0, a2=0;
printf("Enter Production:");
 scanf("%s",a);
for(i=0; a[i]!='/'; i++) {
count++; }
if(a[3]=a[count+1])  {
printf("\n Grammar is not Left Factored”);
 for(i=3.j=0;a[i]!="|';i++,j++)  {
b[j]=a[i];  }
b[i]='\0';
for(i=count+1,k=0;a[i]!='\0';i++,k++) {
c[k]=a[i];  }
c[k]='\0';
 for(j=0,k=0,q=0;b[j]=c[k]; j++,k++,q++) {
alpha[q]=b[j];
a1++;
```

```
 a2++;
}
for(j=a1,w=0;b[j]!='\0';j++,w++) {
beta 1[w]=b[j];
 }
Beta1[w]='\0';
 for(k=a2,e=0;c[k]!='\0';k++,e++)
{
 beta 2[e]=c[k]; }
 beta 2[e]='\0';
 printf("\n%c->%s%c'",a[0],alpha,a[0]);
printf("\n %c'>;%s|%s", a[0],beta 1,beta2); }
else {
printf("\n Grammar is already left factored"); }
return 0; }
```

Output:



```
root@Tanmay:~/Desktop/Desktop/CompilerProgram# ./a.out
Enter Production : A->aE+bcD|aE+eIT

 A->aE+X
 X->bcD|eIT
root@Tanmay:~/Desktop/Desktop/CompilerProgram#
```

Practical - 7

Aim: YACC Program to Check Valid Identifier.

Lex File:

```
% {
#include"y.tab.h"
%}
%%
[0-9]+ {return DIGIT;}
[a-zA-Z]+ {return LETTER;}
%%
```

Yacc File:

```
%{
#include<stdio.h>
%}
%token LETTER DIGIT
%%
variable: LETTER|LETTER rest;
 rest: LETTER rest|DIGIT rest|LETTER DIGIT;
%%
int main() {
printf("Enter Number of variable:");
```

 yyparse();

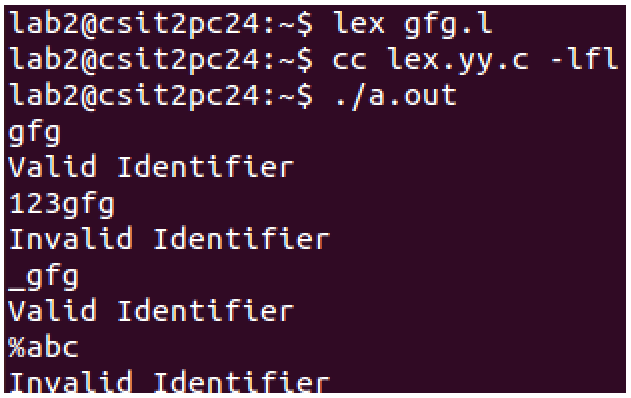
 printf("\n Valid Variable");

return 0;

int yyerror(char *s){ printf("Invalid Variablle”);

exit(0);

Output:



```
lab2@csit2pc24:~$ lex gfg.l
lab2@csit2pc24:~$ cc lex.yy.c -lfl
lab2@csit2pc24:~$ ./a.out
gfg
Valid Identifier
123gfg
Invalid Identifier
_gfg
Valid Identifier
%abc
Invalid Identifier
```

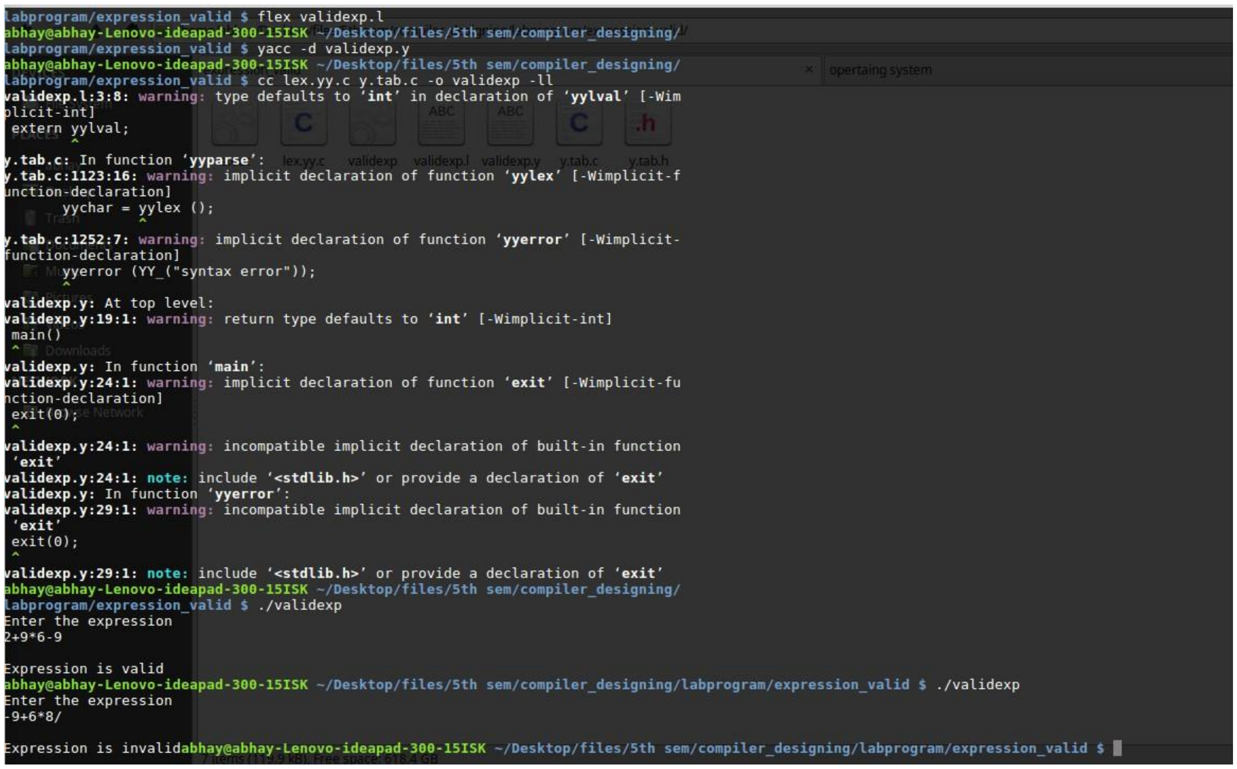
Practical - 8

Aim: YACC Program to Check Validity of arithmetic expression.

```
%{
#include"y.tab.h"
extern yylval;
%}

/* defined section */
%%
[0-9]+ {yylval=atoi(yytext); return NUMBER;}   //this is send to the yacc
code as token INTEGER
[a-zA-Z]+ {return ID;}                         //this is send to the yacc
code as token  ID
[\t]+ ;
\n {return 0;}
. {return yytext[0];}
%%


%{
#include<stdio.h>
%}

//definition section



%token NUMBER ID                  // token from lex file
%left '+' '-'                      // left associative
%left '*' '/'
%%

expr: expr '+' expr                       // grammer production rule
```

```
    |expr '-' expr
    |expr '*' expr
    |expr '/' expr
    |'-'NUMBER
    |'-'ID
    |'('expr')'
    |NUMBER
    |ID
    ;
%%

//main function

main()
{
printf("Enter the expression\n");
yyparse();
printf("\nExpression is valid\n");
exit(0);
}

//if error occured

int yyerror(char *s)
{
printf("\nExpression is invalid");
exit(0);
}
```

Output:



```
labprogram/expression_valid $ flex validexp.l
abhay@abhay-Lenovo-ideapad-300-15ISK ~/Desktop/files/5th sem/compiler_designing/
labprogram/expression_valid $ yacc -d validexp.y
abhay@abhay-Lenovo-ideapad-300-15ISK ~/Desktop/files/5th sem/compiler_designing/
labprogram/expression_valid $ cc lex.yy.c y.tab.c -o validexp -ll
validexp.l:3:8: warning: type defaults to ‘int’ in declaration of ‘yylval’ [-Wim
plicit-int]
 extern yylval;
        ^
y.tab.c: In function ‘yyparse’:
y.tab.c:1123:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-f
unction-declaration]
       yychar = yylex ();
                ^
y.tab.c:1252:7: warning: implicit declaration of function ‘yyerror’ [-Wimplicit-
function-declaration]
       yyerror (YY_("syntax error"));
       ^
validexp.y: At top level:
validexp.y:19:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main()
 ^
validexp.y: In function ‘main’:
validexp.y:24:1: warning: implicit declaration of function ‘exit’ [-Wimplicit-fu
nction-declaration]
 exit(0);
 ^
validexp.y:24:1: warning: incompatible implicit declaration of built-in function
 ‘exit’
validexp.y:24:1: note: include ‘<stdlib.h>’ or provide a declaration of ‘exit’
validexp.y: In function ‘yyerror’:
validexp.y:29:1: warning: incompatible implicit declaration of built-in function
 ‘exit’
 exit(0);
 ^
validexp.y:29:1: note: include ‘<stdlib.h>’ or provide a declaration of ‘exit’
abhay@abhay-Lenovo-ideapad-300-15ISK ~/Desktop/files/5th sem/compiler_designing/
labprogram/expression_valid $ ./validexp
Enter the expression
2+9*6-9

Expression is valid
abhay@abhay-Lenovo-ideapad-300-15ISK ~/Desktop/files/5th sem/compiler_designing/labprogram/expression_valid $ ./validexp
Enter the expression
-9+6*8/

Expression is invalidabhay@abhay-Lenovo-ideapad-300-15ISK ~/Desktop/files/5th sem/compiler_designing/labprogram/expression_valid $ 
```

Practical - 09

Aim: YACC Program to check given string belongs to given grammar or not.

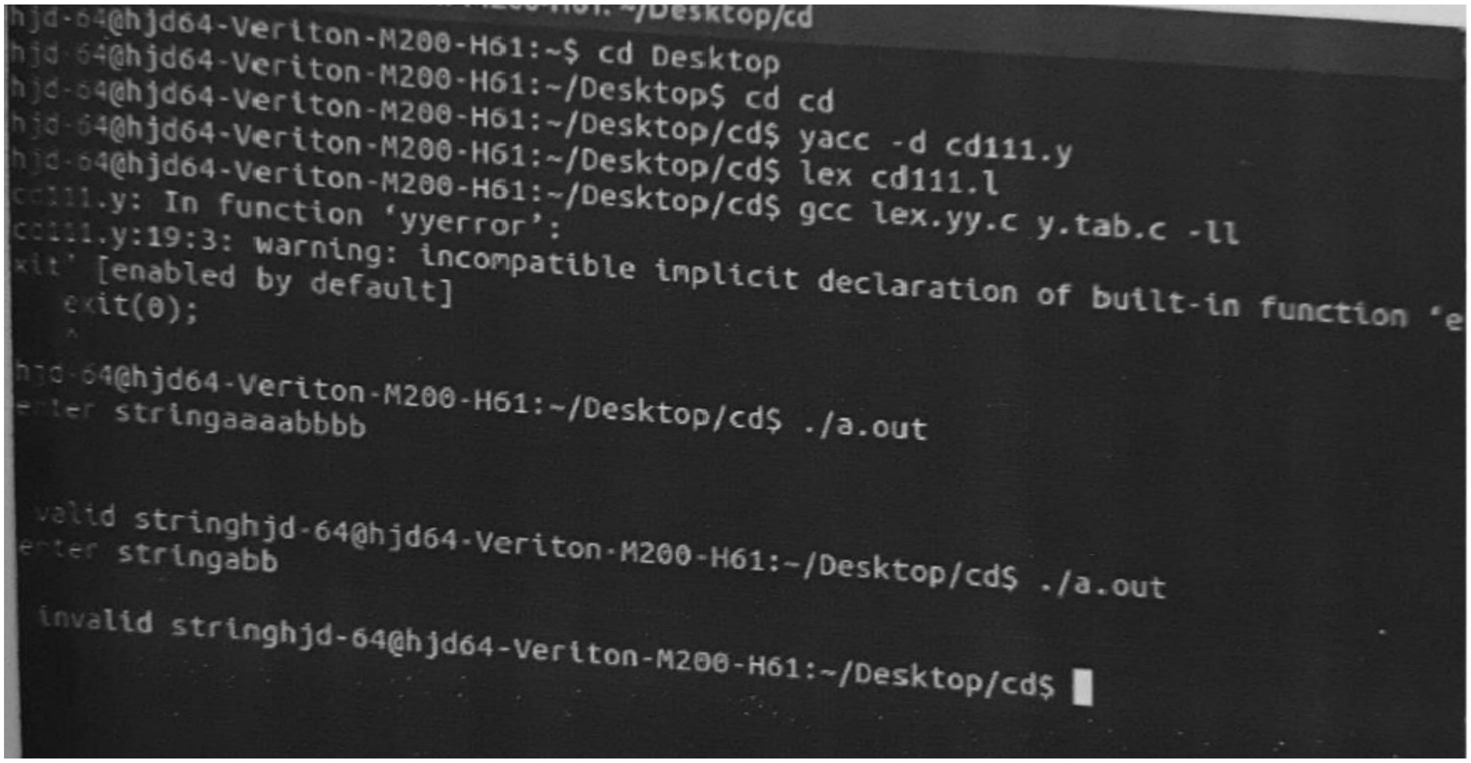
Lex File:

```
%{
#include"y.tab.h"
%}
% %
(a) return A;
 [b] return B;
%%
```

Yacc File:

```
 %{
 #include<stdio.h>
 %}
 %token AB
%%
S:A S b;
 %%
 int main() {
printf("Enter string \n");
yyparse();

printf("\n valid string");

return 0;   }

yyerror(char *S)      {

printf("\n invalid string");  }
```

Output:



```
hjd-64@hjd64-Veriton-M200-H61:~/Desktop/cd
hjd-64@hjd64-Veriton-M200-H61:~$ cd Desktop
hjd-64@hjd64-Veriton-M200-H61:~/Desktop$ cd cd
hjd-64@hjd64-Veriton-M200-H61:~/Desktop/cd$ yacc -d cd111.y
hjd-64@hjd64-Veriton-M200-H61:~/Desktop/cd$ lex cd111.l
hjd-64@hjd64-Veriton-M200-H61:~/Desktop/cd$ gcc lex.yy.c y.tab.c -ll
cd111.y: In function 'yyerror':
cd111.y:19:3: warning: incompatible implicit declaration of built-in function 'e
xit' [enabled by default]
   exit(0);
   ^

hjd-64@hjd64-Veriton-M200-H61:~/Desktop/cd$ ./a.out
enter stringaaaabbbb


valid stringhjd-64@hjd64-Veriton-M200-H61:~/Desktop/cd$ ./a.out
enter stringabb

invalid stringhjd-64@hjd64-Veriton-M200-H61:~/Desktop/cd$
```

Practical - 10

Aim: Create Yacc and Lex specification files are used to generate a calculate or which acceptingtegerand float type arguments.

Lex File:

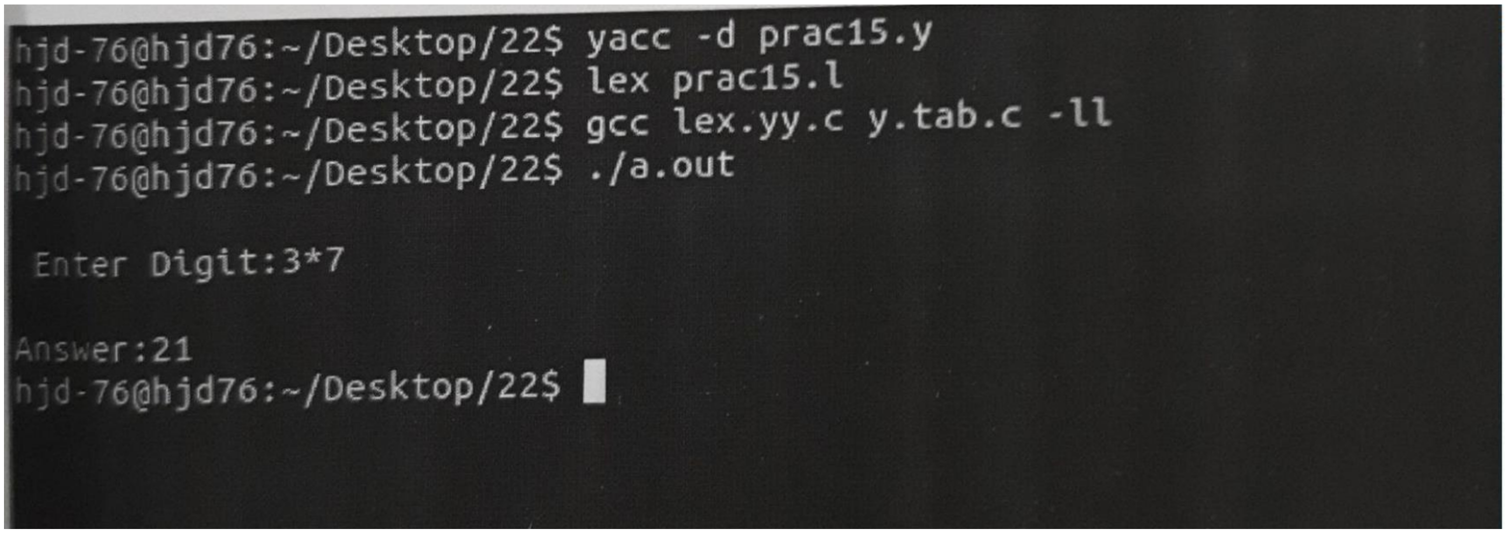
```
%{
#include"y.tab.h"
 #include<stdio.h>
 %}
%%
[0-9]+ {yylval=atoi(yytext); return DIGIT;}
{return yytext[0];}
%%
```

Yacc File:

```
 %{
 #include<stdio.h>
 #include<stdlib.h>
 % }
%token DIGIT
%left’+’_’ ’ *’ ‘/’ ‘%’
 %%
state:expr {printf("Answer:%d\n",$$);};
```

```
 expr:DIGIT|expr'+'expr{$$=$1+$3;}|expr'-'expr{$$=$1-
$3;}|expr'*'expr{$$=$1*$3;}|expr'/'expr{$$=$1/$3;};

%%

int main(){

printf("\n Enter Digit:");

 yyparse();

return 0;

int yyerror(char *s)

{

printf("\n Invalid Digit");

exit(0);

}
```

Output:



```
hjd-76@hjd76:~/Desktop/22$ yacc -d prac15.y
hjd-76@hjd76:~/Desktop/22$ lex prac15.l
hjd-76@hjd76:~/Desktop/22$ gcc lex.yy.c y.tab.c -ll
hjd-76@hjd76:~/Desktop/22$ ./a.out

 Enter Digit:3*7

Answer:21
hjd-76@hjd76:~/Desktop/22$ 
```