Name: Bhavesh Kewalramani
Roll No.: A-25

Practical No. 2

Theory

YACC:

Yacc (Yet Another Compiler-Compiler) is a computer program for the Unix operating system developed by Stephen C. Johnson. It is a Look Ahead Left-to-Right (LALR) parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to Backus–Naur Form (BNF). Yacc is supplied as a standard utility on BSD and AT&T Unix. GNU-based Linux distributions include Bison, a forward-compatible Yacc replacement.

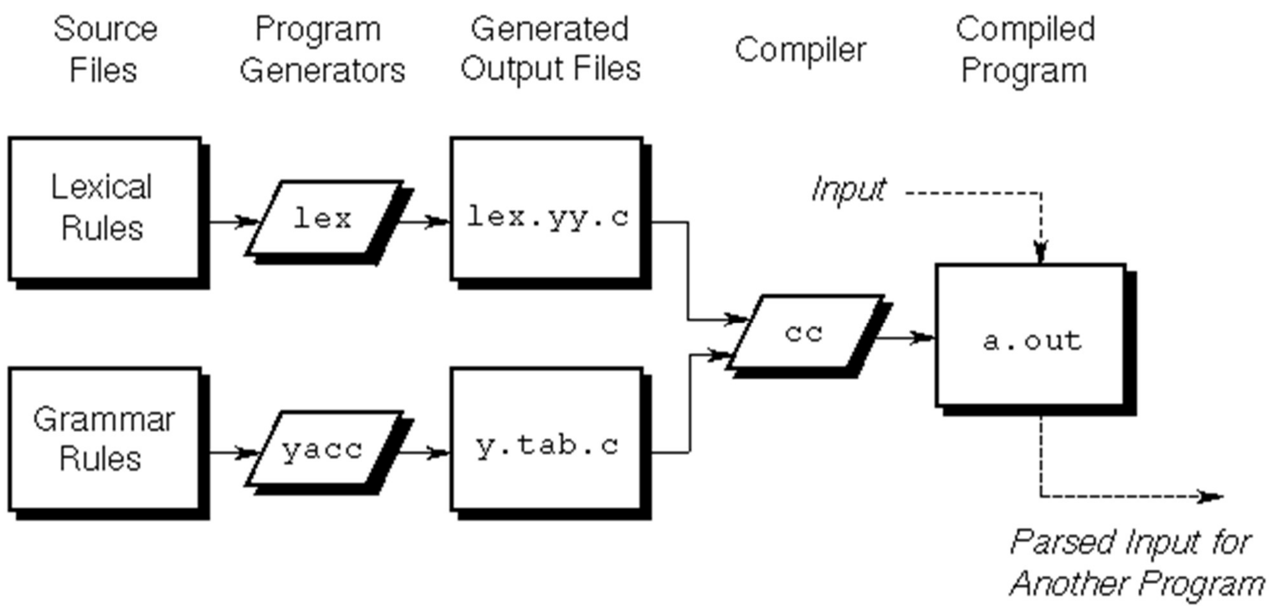
The input to Yacc is a grammar with snippets of C code (called "actions") attached to its rules. Its output is a shift-reduce parser in C that executes the C snippets associated with each rule as soon as the rule is recognized. Typical actions involve the construction of parse trees. Using an example from Johnson, if the call node (label, left, right) constructs a binary parse tree node with the specified label and children, then the rule.

recognizes summation expressions and constructs nodes for them. The special identifiers $$, $1 and $3 refer to items on the parser's stack.

Yacc produces only a parser (phrase analyzer); for full syntactic analysis this requires an external lexical analyzer to perform the first tokenization stage (word analysis), which is then followed by the parsing stage proper. Lexical analyzer generators, such as Lex or Flex are widely available. The IEEE POSIX P1003.2 standard defines the functionality and requirements for both Lex and Yacc.

Some versions of AT&T Yacc have become open source. For example, source code is available with the standard distributions of Plan 9.

Diagram 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 sections: 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
```

How the parser works?

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

Actions

With each grammar rule, you can associate actions to be performed when the rule is recognized. Actions can return values and can obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C-language statement and as such can do input and output, call subroutines, and alter arrays and variables. An action is specified by one or more statements enclosed in { and }. For example, the following two examples are grammar rules with actions:

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

and

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

The $ symbol is used to facilitate communication between the actions and the parser. The pseudo-variable $$ represents the value returned by the complete action.

For example, the action:

```
{$$ = 1;}
```

returns the value of one; in fact, that's all it does.

To obtain the values returned by previous actions and the lexical analyzer, the action can use the pseudo-variables $1, $2, ... $n. These refer to the values returned by components 1 through n of the right side of a rule, with the components being numbered from left to right. If the rule is

```
A: B C D ;
```

then $2 has the value returned by C, and $3 the value returned by D. The following rule provides a common example:

```
expr: '(' expr ')' ;
```

You would expect the value returned by this rule to be the value of the expr within the parentheses. Since the first component of the action is the literal left parenthesis, the desired logical result 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 following form frequently need not have an explicit action:

```
A : B ;
```

In previous examples, all the actions came at the end of 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 action is assumed to return a value accessible through the usual $ mechanism by the actions to the right of it. In turn, it can access the values returned by the symbols to its left. Thus, in the rule below, the effect is to set x to 1 and y to the value returned by C:

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

Actions that do not terminate a rule are handled 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 by recognizing this added rule.

YACC Declaration Summary

Declaration	**Description**
%start	Specify the grammar's start symbol
%token	Declare a terminal symbol (token type name) with no precedence or associativity specified
%type	Declare the type of semantic values for a nonterminal symbol
%right	Declare a terminal symbol (token type name) that is right-associative
%left	Declare a terminal symbol (token type name) that is left-associative
%nonassoc	Declare a terminal symbol (token type name) that is non-associative (using it in a way that would be associative is a syntax error, *Ex: x op. y op. z is syntax error)*

Practicals

Aim (I1): Write YACC specification to check syntax of a simple expression involving operators +, -, * and / and evaluate the expression.

Program:

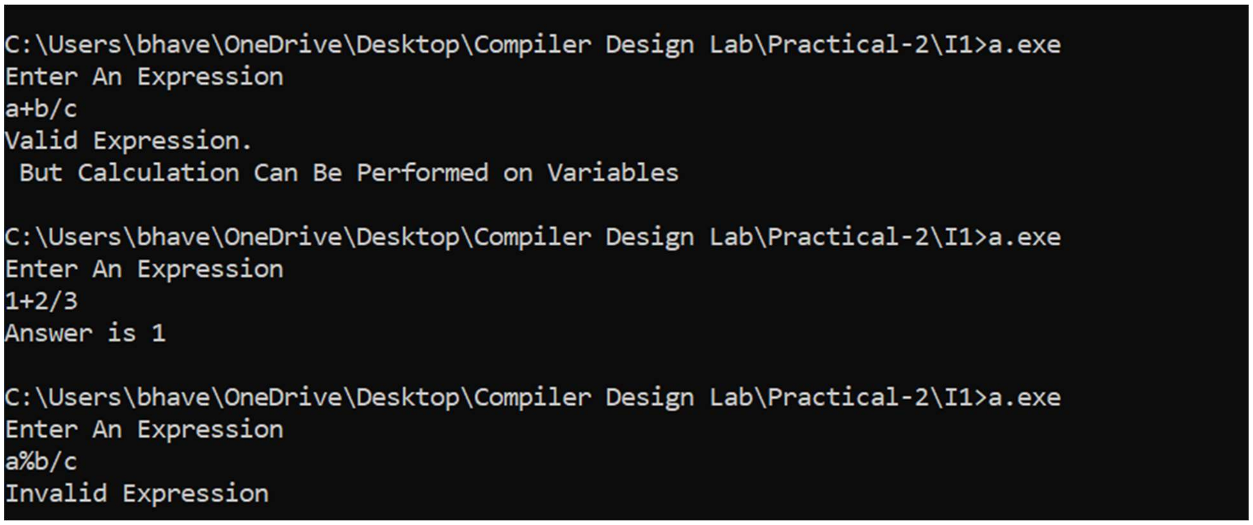
Lex Code:

```
%{
#include "y.tab.h"
%}
%%
[0-9]+ {yylval=atoi(yytext); return NUMBER;}
[a-zA-Z] {return ID;}
\n {return NL;}
. {return yytext[0];}
%%
```

YACC Code:

```
%{
#include<stdio.h>
#include<stdlib.h>
int res=0;
%}
%token NUMBER ID NL
%left '+' '-'
%left '*' '/'
%%
stmt: exp NL {printf("Answer is %d \n",$1);
               exit(0);}
|
exp1 NL {printf("Valid Expression. \n But Calculation Can Be Performed on Variables
\n");
           exit(0);}
;
exp: exp '+' exp {$$=$1+$3;}
| exp '-' exp {$$=$1-$3;}
| exp '*' exp {$$=$1*$3;}
| exp '/' exp {$$=$1/$3;}
| '(' exp ')' {$$=$1;}
| NUMBER
;
exp1: exp1 '+' exp1
| exp1 '-' exp1
| exp1 '*' exp1
| exp1 '/' exp1
| '(' exp1 ')'
| ID
;
%%
int yyerror(char *msg)
{
printf("Invalid Expression \n");
exit(0);
}
void main()
{
printf("Enter An Expression \n");
yyparse();
}
int yywrap(){ return 1;}
```

Output:



```
C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\I1>a.exe
Enter An Expression
a+b/c
Valid Expression.
 But Calculation Can Be Performed on Variables

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\I1>a.exe
Enter An Expression
1+2/3
Answer is 1

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\I1>a.exe
Enter An Expression
a%b/c
Invalid Expression
```

Aim (E1): Write YACC specification to check syntax of a simple expression involving operators +, -, * and /. Also convert the arithmetic expression to postfix.

Program:

Lex Code:

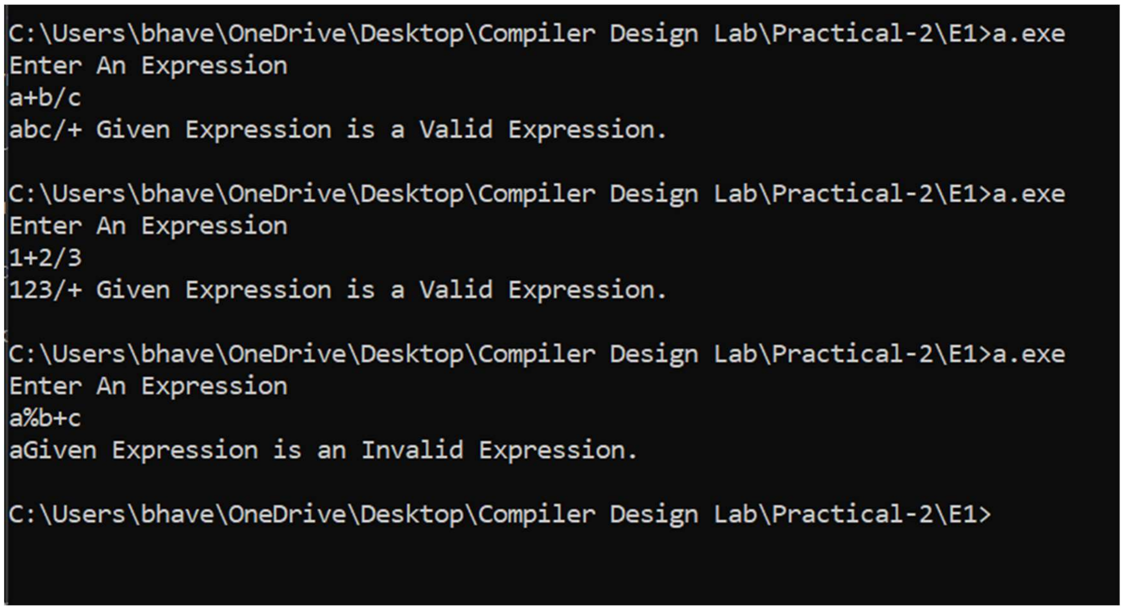
```
%{
#include"y.tab.h"
%}
%%
[a-zA-Z0-9] {yylval=int(yytext);return ID;}
\n {return NL;}
. {return yytext[0];}
%%
```

YACC Code:

```
%{
#include<stdio.h>
#include<stdlib.h>
int res=0;
%}
%token NUMBER ID NL
%left '+' '-'
%left '*' '/'
%%
stmt: exp1 NL {printf(" Given Expression is a Valid Expression.\n");
            exit(0);}
```

```
;
exp1: exp1 '+' exp1 {printf("+");}
| exp1 '-' exp1 {printf("-");}
| exp1 '*' exp1 {printf("*");}
| exp1 '/' exp1 {printf("/");}
| '(' exp1 ')'
| ID {printf("%c",(char)yylval);}
;
%%
int yyerror(char *msg)
{
printf("Given Expression is an Invalid Expression.\n");
exit(0);
}
void main()
{
printf("Enter An Expression \n");
yyparse();
}
int yywrap(){ return 1;}
```

Output:



```
C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E1>a.exe
Enter An Expression
a+b/c
abc/+ Given Expression is a Valid Expression.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E1>a.exe
Enter An Expression
1+2/3
123/+ Given Expression is a Valid Expression.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E1>a.exe
Enter An Expression
a%b+c
aGiven Expression is an Invalid Expression.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E1>
```

Aim (E2): Write a YACC specification to accept strings that starts and ends with 0 or 1.

Program:

Lex Code:

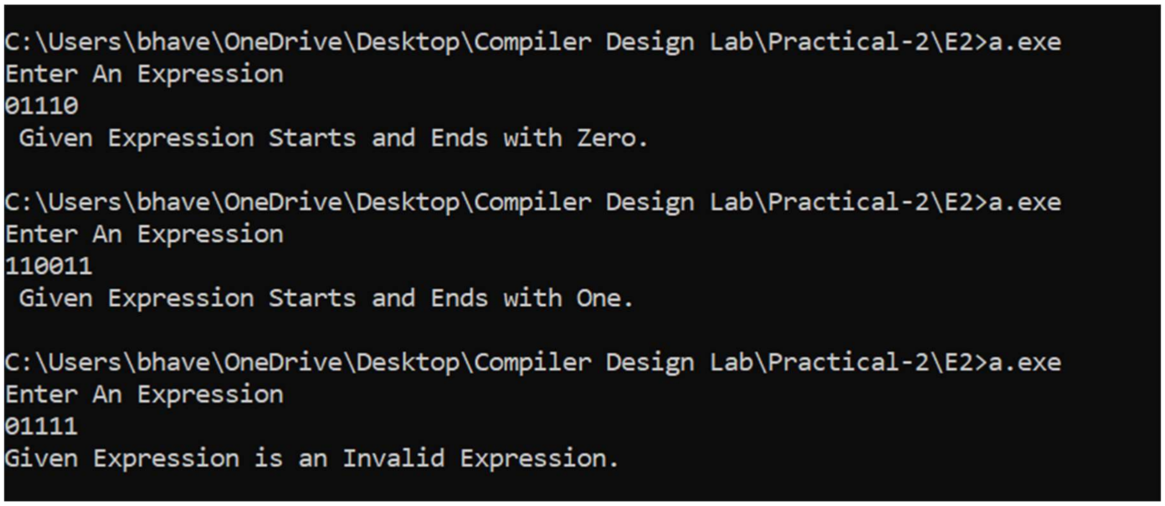
```
%{
#include "y.tab.h"
%}
%%
[0] {return ZERO;}
[1] {return ONE;}
\n {return NL;}
. {return yytext[0];}
%%
```

YACC Code:

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token ZERO ONE NL
%%
stmt:  ZERO z NL {printf(" Given Expression Starts and Ends with Zero. \n");
                    exit(0);}
|ONE y NL {printf(" Given Expression Starts and Ends with One. \n");
           exit(0);}
;
z: a z
|ZERO
;
y: a y
|ONE
;
a: ZERO
|ONE
;

%%
int yyerror(char *msg)
{
printf("Given Expression is an Invalid Expression.\n");
exit(0);
}
void main()
{
printf("Enter An Expression \n");
yyparse();
}
int yywrap(){ return 1;}
```

Output:



```
C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E2>a.exe
Enter An Expression
01110
 Given Expression Starts and Ends with Zero.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E2>a.exe
Enter An Expression
110011
 Given Expression Starts and Ends with One.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E2>a.exe
Enter An Expression
01111
Given Expression is an Invalid Expression.
```

Aim (E3): Write a YACC specification to validate the string having general form as below. Construct a proper grammar for the same and also write the corresponding LEX.:
(a) Any alphabet(s) @ any alphabet + any digit – any digit.
(b) Date
(c) Expression of the form a=b*c

Program(a):

Lex Code:

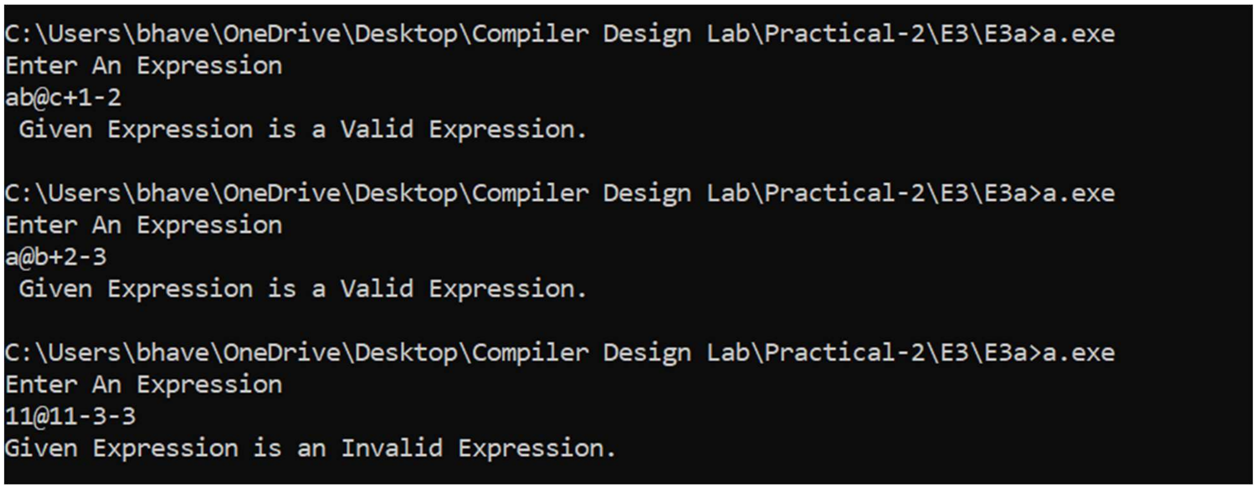
```
%{
#include "y.tab.h"
%}
%%
[a-zA-Z] {return ID1;}
[a-zA-Z]+ {return ID2;}
[0-9]+ {return NUM;}
"@" {return SP1;}
"-" {return SP2;}
"+" {return SP3;}
\n {return NL;}
```

```
. {return yytext[0];}
%%
```

YACC Code:

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token ID1 ID2 NUM SP1 SP2 SP3 NL
%%
stmt: ID2 SP1 ID1 SP3 NUM SP2 NUM NL {printf(" Given Expression is a Valid
Expression. \n");
                                        exit(0);}
|ID1 SP1 ID1 SP3 NUM SP2 NUM NL {printf(" Given Expression is a Valid
Expression. \n");
                                    exit(0);}
;
%%
int yyerror(char *msg)
{
printf("Given Expression is an Invalid Expression.\n");
exit(0);
}
void main()
{
printf("Enter An Expression \n");
yyparse();
}
int yywrap(){ return 1;}
```

Output:



```
C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E3\E3a>a.exe
Enter An Expression
ab@c+1-2
 Given Expression is a Valid Expression.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E3\E3a>a.exe
Enter An Expression
a@b+2-3
 Given Expression is a Valid Expression.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E3\E3a>a.exe
Enter An Expression
11@11-3-3
Given Expression is an Invalid Expression.
```

Program(b):

Lex Code:

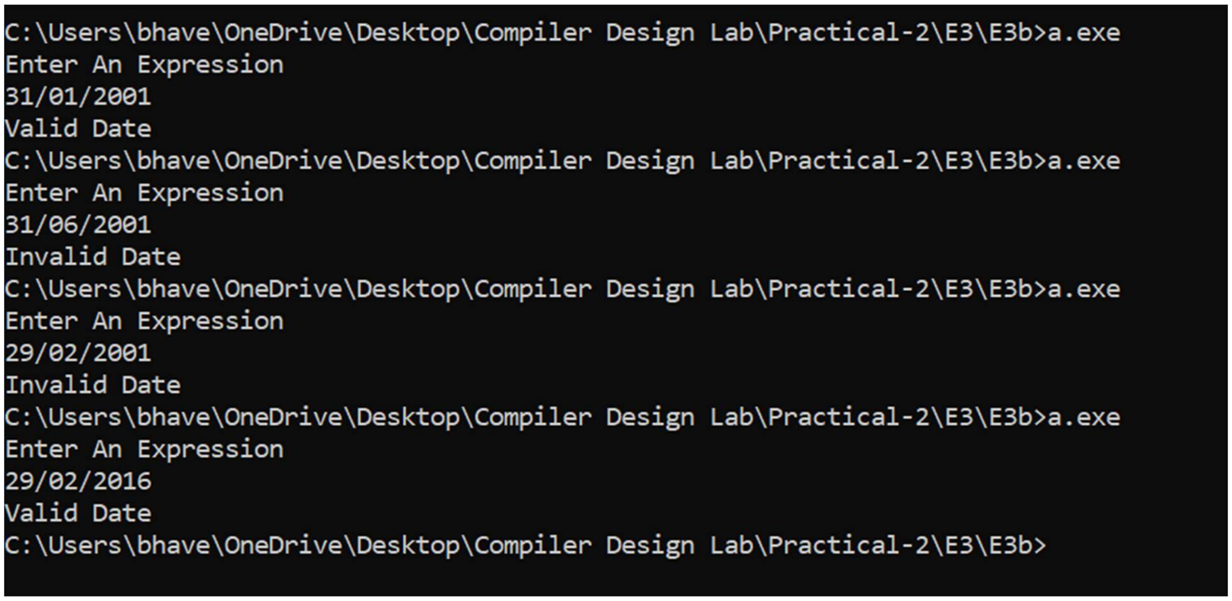
```
%{
#include "y.tab.h"
#include<stdio.h>
%}
%%
[0-2][1-9]|[3][0-1]|[2][0] {yylval=atoi(yytext); return DAYMON;}
[1-2][0-9][0-9][0-9] {yylval=atoi(yytext); return YEAR;}
"-"|"/" {return SP;}
\n {return NL;}
. {return yytext[0];}
%%
```

YACC Code:

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token DAYMON YEAR SP NL
%%
stmt: DAYMON SP DAYMON SP YEAR NL {
                                    if(($1>=1 && $1<=31) && ($3==1 || $3==3 || $3==5 ||
$3==7 || $3==8 || $3==10 || $3==12))
                                    {
                                            printf("Valid Date");
```

```
                                                exit(0);
                                        }
                                        else if(($1>=1 && $1<=30) && ($3==4 || $3==6 || $3==9 ||
$3==11))
                                        {
                                                printf("Valid Date");
                                                exit(0);
                                        }
                                        else if(($1>=1 && $1<=28) && ($3==2))
                                        {
                                                printf("Valid Date");
                                                exit(0);
                                        }
                                        else if(($1==29) && ($3==2) && (($5%4==0) &&
(($5%400==0) || ($5%100!=0))))
                                        {
                                                printf("Valid Date");
                                                exit(0);
                                        }
                                        else
                                        {
                                                printf("Invalid Date");
                                                exit(0);
                                        }
        }
        %%
        int yyerror(char *msg)
        {
        printf("Invalid Date.\n");
        exit(0);
        }
        void main()
        {
        printf("Enter An Expression \n");
        yyparse();
        }
        int yywrap(){ return 1;}
```

Output:



```
C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E3\E3b>a.exe
Enter An Expression
31/01/2001
Valid Date
C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E3\E3b>a.exe
Enter An Expression
31/06/2001
Invalid Date
C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E3\E3b>a.exe
Enter An Expression
29/02/2001
Invalid Date
C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E3\E3b>a.exe
Enter An Expression
29/02/2016
Valid Date
C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E3\E3b>
```

Program(c):

Lex Code:

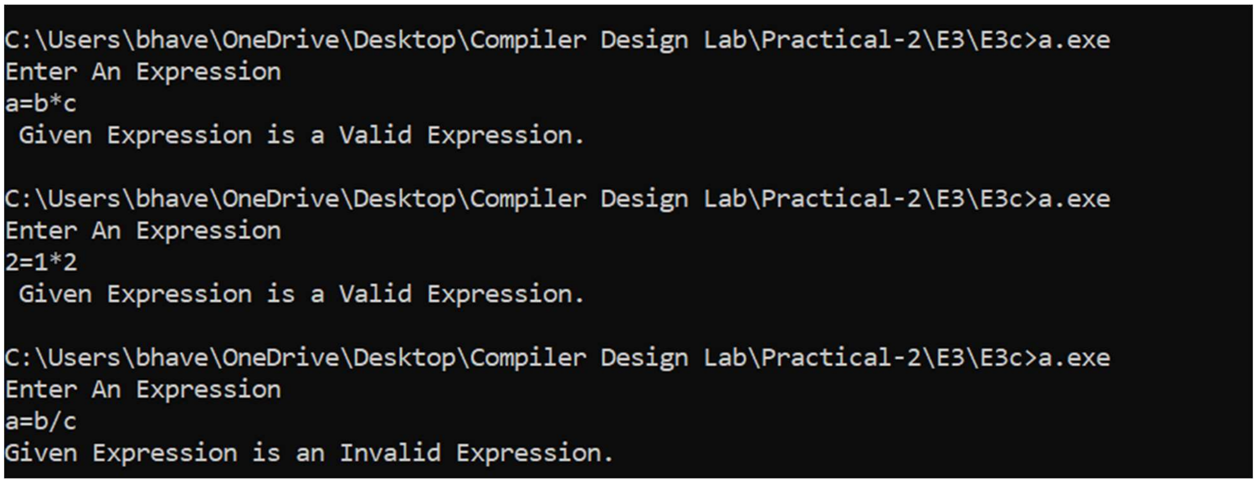
```
%{
#include "y.tab.h"
%}
%%
[a-zA-Z0-9]+ {return ID;}
"=" {return SP1;}
"*" {return SP2;}
\n {return NL;}
. {return yytext[0];}
%%
```

YACC Code:

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token ID NUM SP1 SP2 NL
%%
stmt: ID SP1 ID SP2 ID NL {printf(" Given Expression is a Valid Expression.
\n");exit(0);}
;
```

```
%%
int yyerror(char *msg)
{
printf("Given Expression is an Invalid Expression.\n");
exit(0);
}
void main()
{
printf("Enter An Expression \n");
yyparse();
}
int yywrap(){ return 1;}
```

Output:



```
C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E3\E3c>a.exe
Enter An Expression
a=b*c
 Given Expression is a Valid Expression.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E3\E3c>a.exe
Enter An Expression
2=1*2
 Given Expression is a Valid Expression.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E3\E3c>a.exe
Enter An Expression
a=b/c
Given Expression is an Invalid Expression.
```

Aim (E4): To validate syntax of following programing language construct:
Batch B1: do while loop

Program:

Lex Code:

```
%{
#include "y.tab.h"
%}
chars [a-zA-Z]
digit [0-9]
a "%d"|"%f"|"%c"|"%s"
id [a-zA-Z][a-zA-Z0-9]*
```

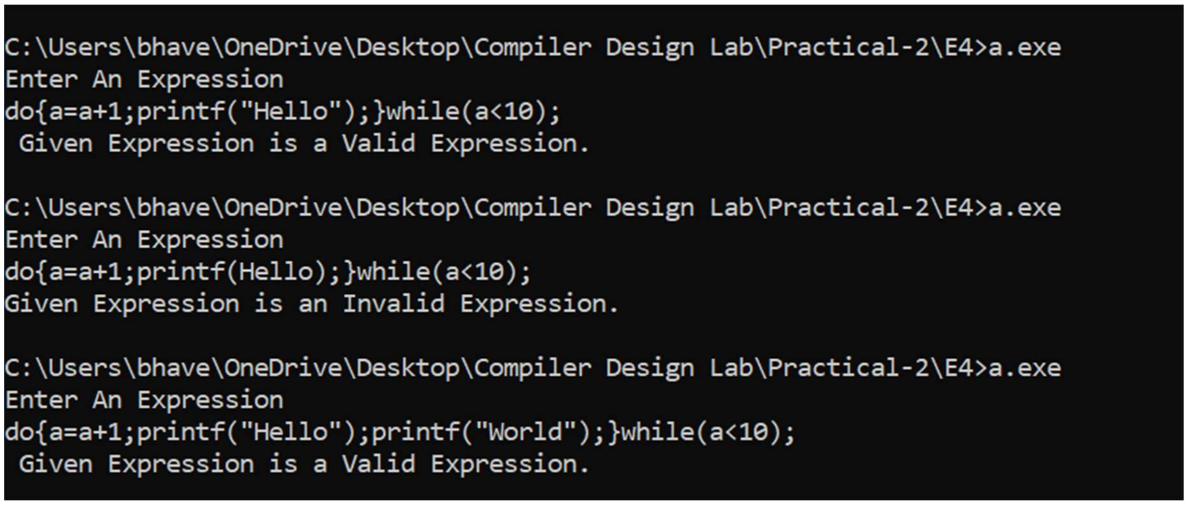
```
%%
do {return DO;}
while {return WHILE;}
{digit}+ {return NUM;}
{chars}({chars}|{digit})* {return ID;}
"<=" {return LE;}
">=" {return GE;}
"==" {return EE;}
"!=" {return NE;}
"&&" {return AND;}
"||" {return OR;}
printf\((\"({a}*|.*)*\"(,{id})*\))\; {return PF;}
\n {return NL;}
. {return yytext[0];}
%%
```

YACC Code:

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token DO WHILE ID NUM LE GE EE NE AND OR PF NL
%right '='
%left AND OR
%left '<' '>' LE GE EE NE
%left '+' '-'
%left '*' '/'
%left '!'
%%
stmt: exp NL {printf(" Given Expression is a Valid Expression. \n");exit(0);}
;
exp: DO'{'exp1'}'WHILE'('exp2')";'
;
exp1: exp1 exp1
|exp3';'
|exp1 PF
|PF
;
exp3: ID'='exp3
|exp3'+'exp3
|exp3'-'exp3
|exp3'*'exp3
|exp3'/'exp3
|exp3'<'exp3
|exp3'>'exp3
```

```
|exp3 LE exp3
|exp3 GE exp3
|exp3 EE exp3
|exp3 NE exp3
|exp3 OR exp3
|exp3 AND exp3
|ID
|NUM
;
exp2: exp3'<'exp3
|exp3'>'exp3
|exp3 LE exp3
|exp3 GE exp3
|exp3 EE exp3
|exp3 NE exp3
|exp3 OR exp3
|exp3 AND exp3
|ID
|NUM
;
%%
int yyerror(char *msg)
{
printf("Given Expression is an Invalid Expression.\n");
exit(0);
}
void main()
{
printf("Enter An Expression \n");
yyparse();
}
int yywrap(){ return 1;}
```

Output:



```
C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E4>a.exe
Enter An Expression
do{a=a+1;printf("Hello");}while(a<10);
 Given Expression is a Valid Expression.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E4>a.exe
Enter An Expression
do{a=a+1;printf(Hello);}while(a<10);
Given Expression is an Invalid Expression.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E4>a.exe
Enter An Expression
do{a=a+1;printf("Hello");printf("World");}while(a<10);
 Given Expression is a Valid Expression.
```

Aim (E5): Write YACC specification to recognize strings that can be accepted by grammar of the form: a^n b^n c, n>=1.

Program:

Lex Code:

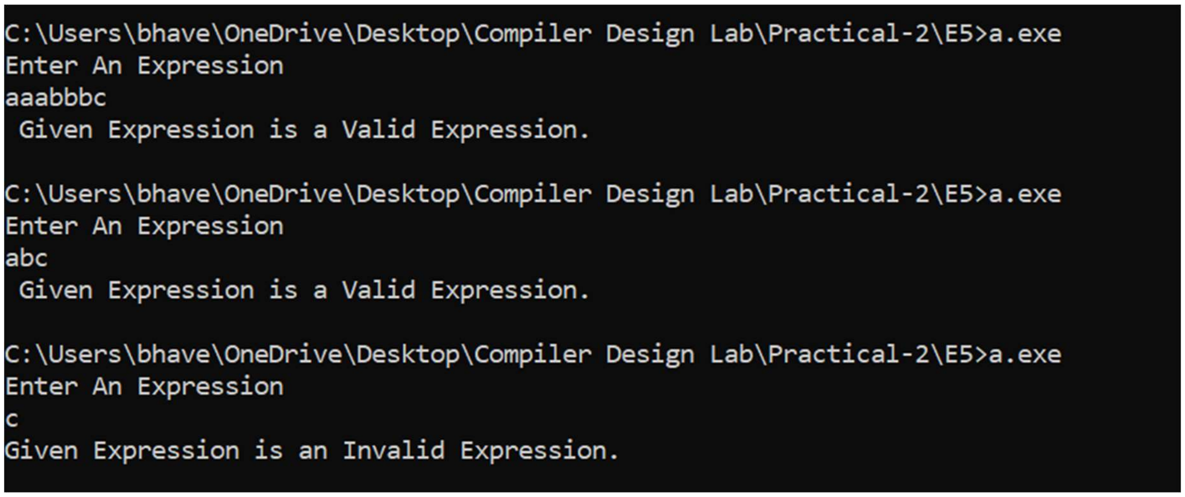
```
%{
#include "y.tab.h"
%}
%%
[aA] {return A;}
[bB] {return B;}
[cC] {return C;}
\n {return NL;}
. {return yytext[0];}
%%
```

YACC Code:

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token A B C NL
%%
stmt: A S B C NL {printf(" Given Expression is a Valid Expression. \n");
```

```
        exit(0);}
;
S: A S B
|
;
%%
int yyerror(char *msg)
{
printf("Given Expression is an Invalid Expression.\n");
exit(0);
}
void main()
{
printf("Enter An Expression \n");
yyparse();
}
int yywrap(){ return 1;}
```

Output:



```
C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E5>a.exe
Enter An Expression
aaabbbc
 Given Expression is a Valid Expression.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E5>a.exe
Enter An Expression
abc
 Given Expression is a Valid Expression.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E5>a.exe
Enter An Expression
c
Given Expression is an Invalid Expression.
```

Aim (E6): Write YACC specification to recognize strings that can be accepted by grammar of the form: {L= a^n b^2n c, n>=1 }
Program:

Lex Code:

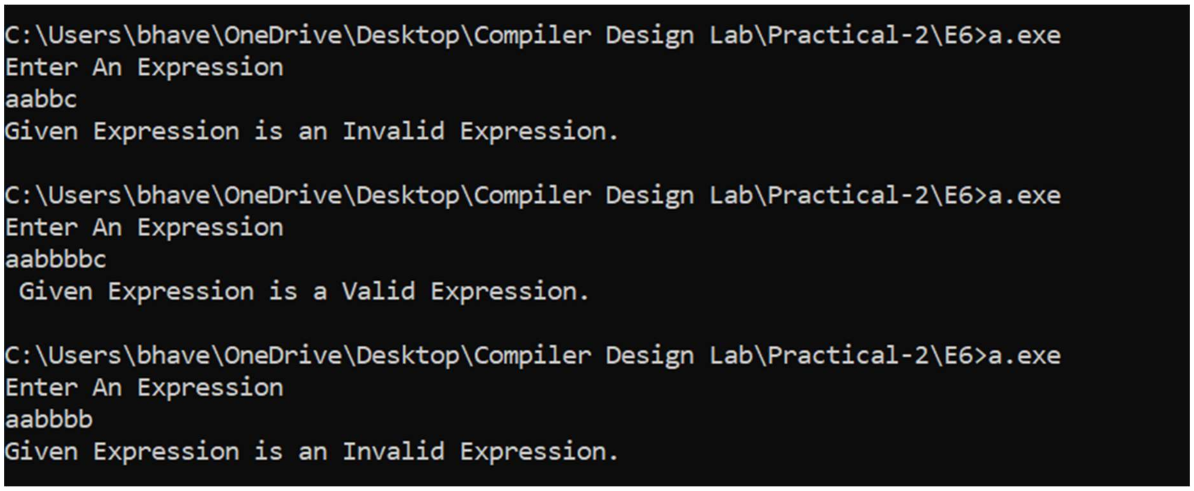
```
%{
```

```
#include "y.tab.h"
%}
%%
[aA] {return A;}
[bB] {return B;}
[cC] {return C;}
\n {return NL;}
. {return yytext[0];}
%%
```

YACC Code:

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token A B NL
%%
stmt: A S B B C NL {printf(" Given Expression is a Valid Expression. \n");
            exit(0);}
;
S: A S B B
|
;
%%
int yyerror(char *msg)
{
printf("Given Expression is an Invalid Expression.\n");
exit(0);
}
void main()
{
printf("Enter An Expression \n");
yyparse();
}
int yywrap(){ return 1;}
```

Output:



```
C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E6>a.exe
Enter An Expression
aabbc
Given Expression is an Invalid Expression.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E6>a.exe
Enter An Expression
aabbbbc
 Given Expression is a Valid Expression.

C:\Users\bhave\OneDrive\Desktop\Compiler Design Lab\Practical-2\E6>a.exe
Enter An Expression
aabbbb
Given Expression is an Invalid Expression.
```