**Name: Chetan Pardhi**

**Roll No.: 22**

**Practical No. 2**

**Theory**

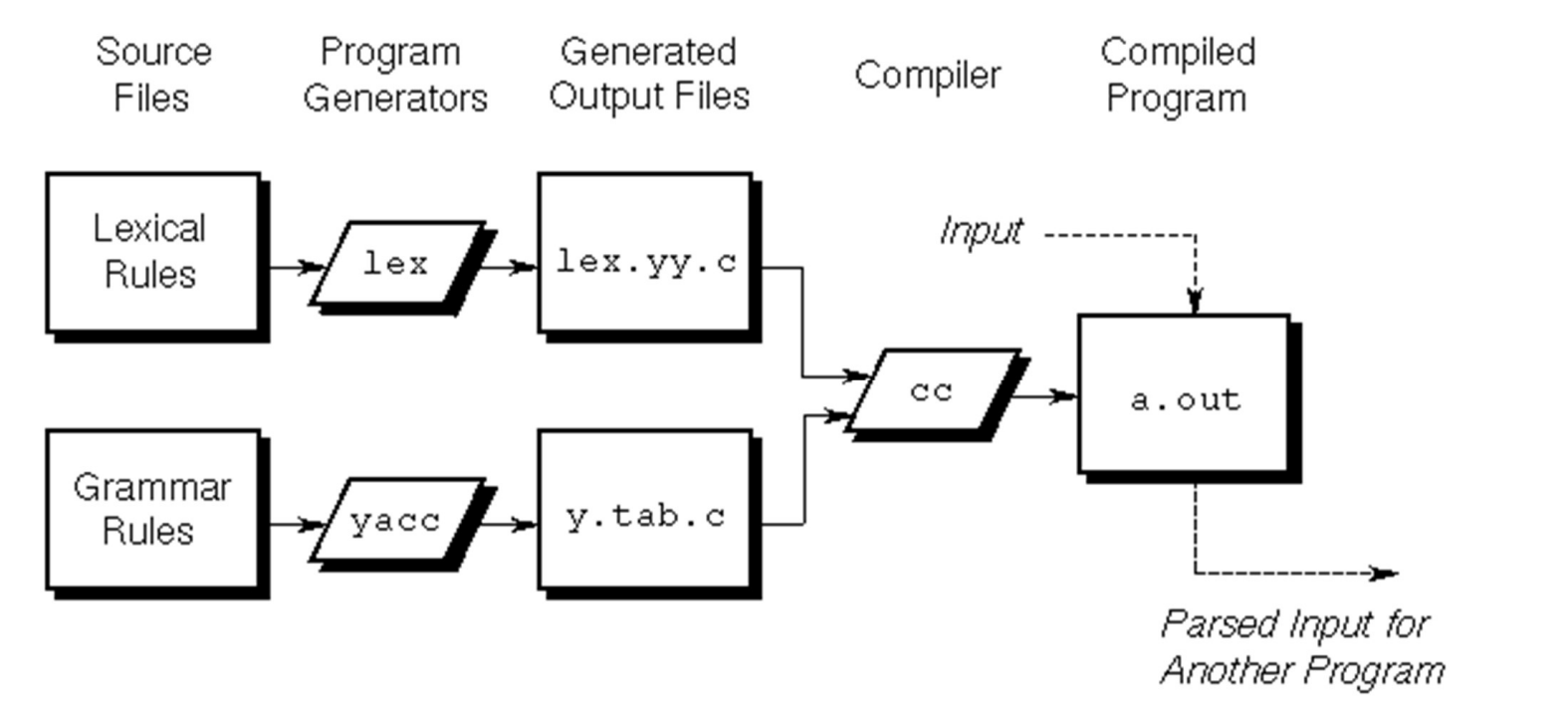
**YACC:**

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 forwardcompatible 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 Specification**

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:

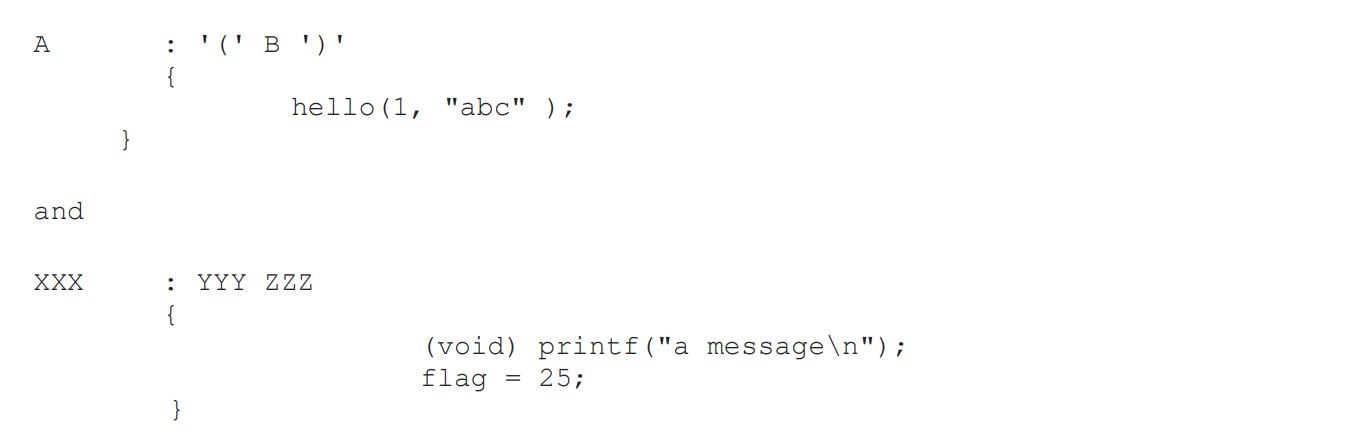
**1.** Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.

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**

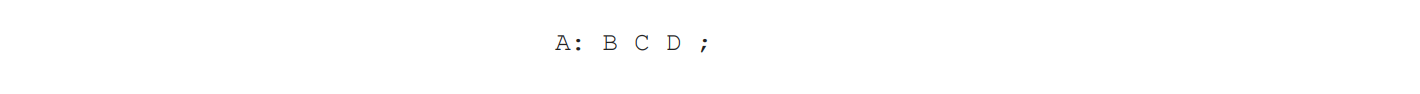
With each grammar rule, you can associate actions to be performed when the rule is recognized. Actions can return values and can obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

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:



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

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

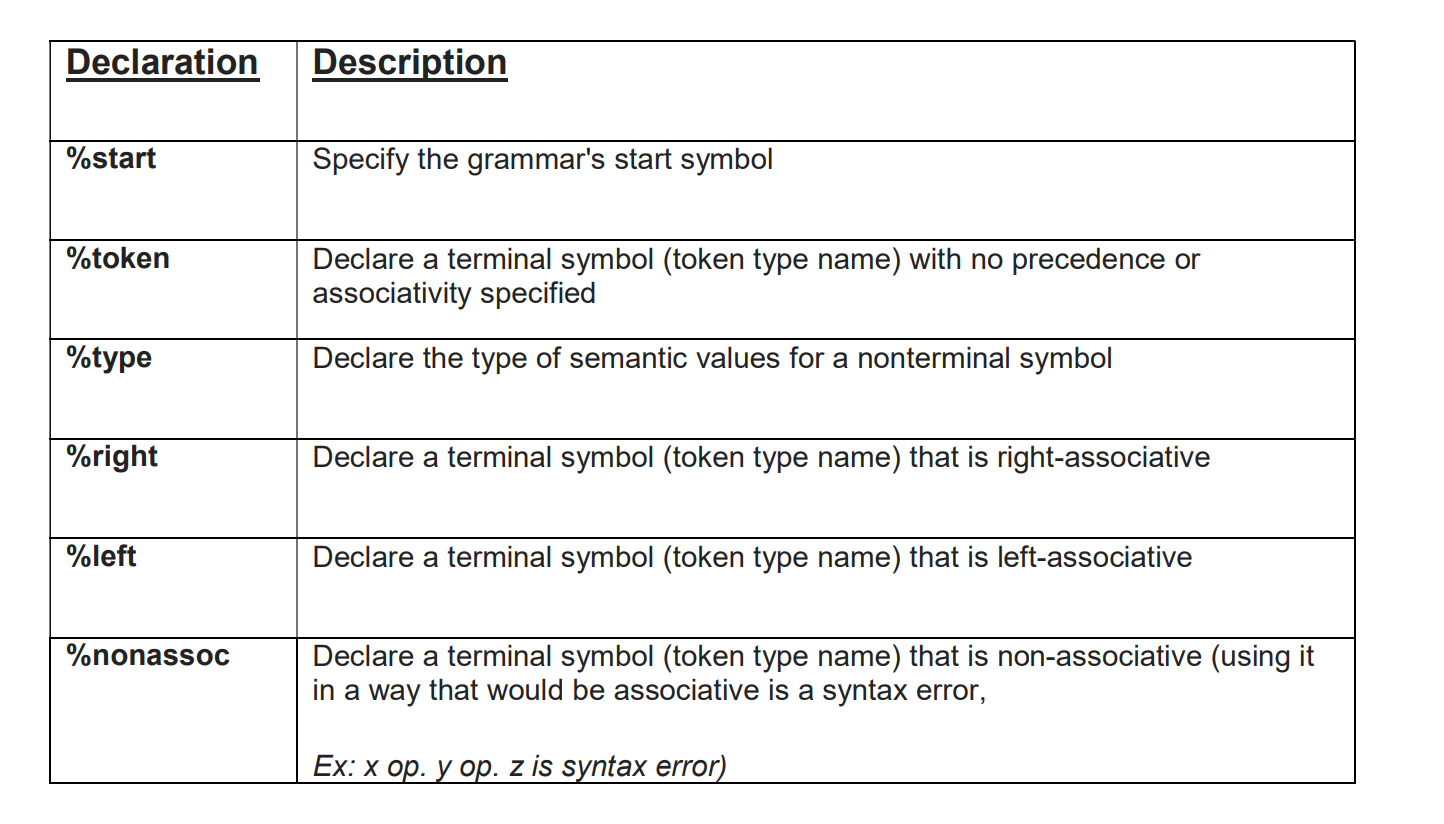


You would expect the value returned by this rule to be the value of the expr within the parentheses. Since the first component of the action is the literal left parenthesis, the desired logical result can be indicated by:



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**



**Practical No. L1**

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

**Program:**

**Lex :**

%{

#include"y.tab.h"

extern char \*yytext;

%}

%%

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

[a-zA-Z] {yylval = yytext[0]; return ID;}

. return NL;

%%

**Yacc :**

%{

#include<stdio.h>

#include<stdlib.h>

int yyerror(char\* msg);

int yylex(void);

%}

%token NUMBER ID NL

%left '+' '-'

%left '\*' '/'

%%

stmt: exp NL{printf("\nExpression is valid.\n");

exit(0);}

;

exp: exp '+' exp ;

| exp '\*' exp ;

| exp '-' exp ;

| exp '/' exp ;

| '(' exp ')'

| NUMBER ;

| ID ;

;

%%

int main(){

printf("Enter an Expression: ");

yyparse();

}

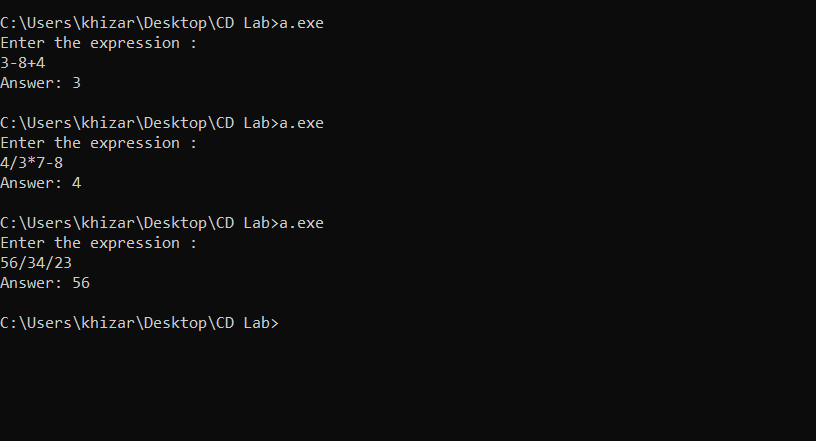
int yyerror (char \*msg) {

printf ("Invalid Expression.", msg);

}

int yywrap(){ return 1;}

**Output:**



**Practical No. E1**

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

**Program:**

**Lex :**

%{

#include"y.tab.h"

extern char \*yytext;

%}

%%

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

[a-zA-Z] {yylval = yytext[0]; return ID;}

\n      return NL;

.       return yytext[0];

%%

**Yacc**:

%{

#include<stdio.h>

#include<stdlib.h>

int yyerror(char\* *msg*);

int yylex(void);

%}

%token *NUMBER* *ID* *NL*

%left '+' '-'

%left '\*' '/'

%%

stmt:  *exp* {printf("\nExpression is valid.\n");

            exit(0);}

;

exp:  *exp* '+' *exp* {printf("+");}

|   *exp* '\*' *exp* {printf("\*");}

|   *exp* '-' *exp* {printf("-");}

|   *exp* '/' *exp* {printf("/");}

|   '(' *exp* ')'

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

|   *ID*     {printf("%c", (char)yylval);}

;

%%

int main(){

printf("Enter an Expression: ");

yyparse();

}

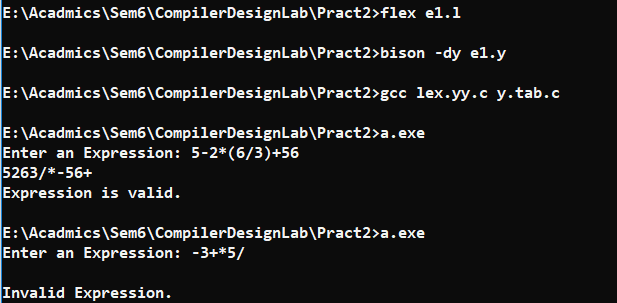
int yyerror (char \**msg*) {

    printf ("\nInvalid Expression.", msg);

}

int yywrap(){ return 1;}

**Output:**



**Practical No. E2**

**Aim:** Write a YACC specification to accept strings that starts and ends with 0 or 1

**Program:**

**Lex:**

**%{**

**#include "y.tab.h"**

**%}**

**%%**

**[0][0-9a-zA-Z]\*[0] return ZERO;**

**[1][0-9a-zA-Z]\*[1] return ONE;**

**\n return NL;**

**. {return yytext[0];}**

**%%**

**YACC:**

%{

#include<stdio.h>

#include<stdlib.h>

int yyerror(char\* msg);

int yylex(void);

int res=0;

%}

%token ZERO ONE NL

%%

stmt: ZERO NL {printf("Expression starts and ends with 0.\n");

exit(0);}

|ONE NL {printf("Expression starts and ends with 1.\n");

exit(0);}

%%

int yyerror(char \*msg){

printf("Expression does not start and end with 0 or 1.\n");

exit(0);

}

void main(){

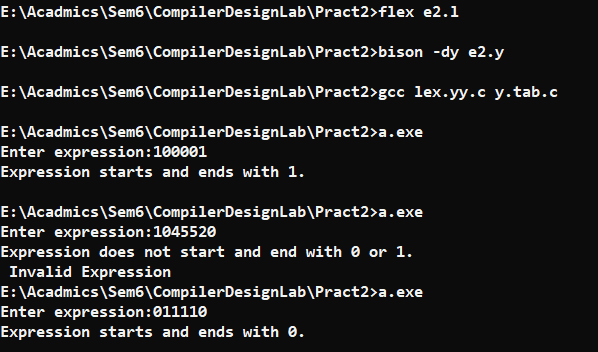
printf("Enter expression:");

yyparse();

}

int yywrap(){return 1;}

**Output:**



**Practical No. E3**

**Aim:** 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

1. **Program:**

**Lex :**

%{

#include "y.tab.h"

%}

%%

[a-zA-Z] return B;

[a-zA-Z]+ return A;

[0-9] return NUM;

.       return yytext[0];

%%

**Yacc :**

%{

#include<stdio.h>

#include<stdlib.h>

int yyerror(char\* *msg*);

int yylex(void);

%}

%token *A* *B* *NUM*

%%

stmt: *A* '@' *B* '+' *NUM* '-' *NUM* {printf("\nExpression is valid.\n");

            exit(0);}

;

%%

int yyerror (char \**msg*) {

    printf ("\nExpression is invalid.\n", msg);

}

int main(){

printf("Enter an Expression: ");

yyparse();

}

int yywrap(){ return 1;}

**Output:**

1. **Program :**

**Lex :**

%{

#include "y.tab.h"

%}

%%

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

\n return NL;

. {return yytext[0];}

%%

**Yacc :**

%{

#include<stdio.h>

#include<stdlib.h>

int yyerror(char\* *msg*);

int yylex(void);

%}

%token *NUM* *NL*

%left '/'

%%

%%

int main(){

printf("Enter an Expression: ");

yyparse();

}

int yyerror (char \**msg*) {

    printf ("\n%s", msg);

}

int yywrap(){ return 1;}

1. **Program :**

**Lex :**

%{

#include "y.tab.h"

%}

%%

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

. {return yytext[0];}

%%

**Yacc :**

%{

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int yyerror(char\* *msg*);

int yylex(void);

%}

%token *NUM*

%%

stmt : *A* {printf("\nValid EXPRESSION : Accepted\n"); exit(0);}

A : *NUM* '=' *NUM* '\*' *NUM*

;

%%

int yyerror(char \**msg*){

    printf("Invalid Expression.\n");

    exit(0);

}

void main()

{

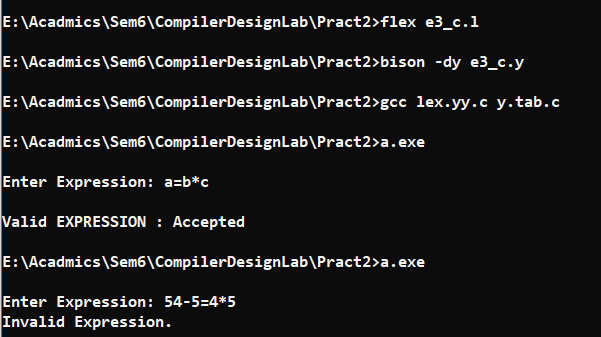
    printf("\nEnter Expression: ");

    yyparse();

}

int yywrap(){return (1);}

**Output :**



**Practical No. E4**

**Aim: To validate syntax of following programing language construct:**

**Batch B1: do while loop**

**Batch B2: while loop**

**Batch B3: if else statement**

**Batch B4: for loop**

**Program:**

**Lex :**

alpha [A-Za-z]

digit [0-9]

%%

[\t \n]

for return FOR;

{digit}+ return NUM;

{alpha}({alpha}|{digit})\* return ID;

"<=" return LE;

">=" return GE;

"==" return EQ;

"!=" return NE;

"||" return OR;

"&&" return AND;

. return yytext[0];

%%

**Yacc:**

%{

#include <stdio.h>

#include <stdlib.h>

%}

%token ID NUM FOR LE GE EQ NE OR AND

%right "="

%left OR AND

%left '>' '<' LE GE EQ NE

%left '+' '-'

%left '\*' '/'

%right UMINUS

%left '!'

%%

S : ST {printf("Input accepted\n"); exit(0);}

ST : FOR '(' E ';' E2 ';' E ')' DEF

;

DEF : '{' BODY '}'

| E';'

| ST

|

;

BODY : BODY BODY

| E ';'

| ST

|

;

E : ID '=' E

| E '+' E

| E '-' E

| E '\*' E

| E '/' E

| E '<' E

| E '>' E

| E LE E

| E GE E

| E EQ E

| E NE E

| E OR E

| E AND E

| E '+' '+'

| E '-' '-'

| ID

| NUM

;

E2 : E'<'E

| E'>'E

| E LE E

| E GE E

| E EQ E

| E NE E

| E OR E

| E AND E

;

%%

int main() {

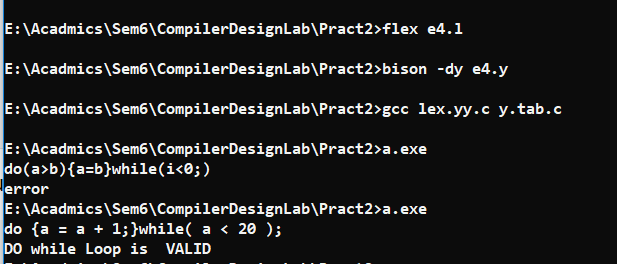
printf("Enter the expression:\n");

yyparse();

}

**Output:**

**Practical No. E5**



**Aim: Write YACC specification to recognize strings that can be accepted by grammar of the form:**

**a n b n c, n&gt;=1**

**Program:**

**Lex :**

%{

#include "y.tab.h"

%}

%%

[aA] return A;

[bB] return B;

[cC] return C;

\n return NL;

. return yytext[0];

%%

**Yacc :**

%{

#include<stdio.h>

#include<stdlib.h>

int yyerror(char\* *msg*);

int yylex(void);

%}

%token *A* *B* *C* *NL*

%%

stmt: *A* *S* *B* *C* *NL* {printf("\nExpression is accepted.\n");

            exit(0);}

;

S: *A* *S* *B*

|

;

%%

int yyerror (char \**msg*) {

    printf ("\nExpression is invalid.\n", msg);

}

int main(){

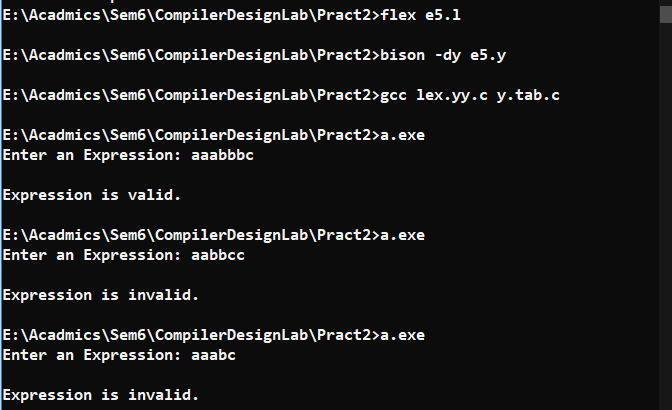
printf("Enter an Expression: ");

yyparse();

}

int yywrap(){ return 1;}

**Output:**



**Practical No. E6**

**Aim: Write YACC specification to recognize strings that can be accepted by grammar of the form: {L= an b2n c, n>=1}**

**Program:**

**Lex :**

%{

#include "y.tab.h"

%}

%%

[aA] return A;

[bB] return B;

[cC] return C;

\n return NL;

. return yytext[0];

%%

**Yacc :**

%{

#include<stdio.h>

#include<stdlib.h>

int yyerror(char\* *msg*);

int yylex(void);

%}

%token *A* *B* *C* *NL*

%%

stmt: *A* *S* *B* *B* *C* *NL* {printf("\nExpression is valid.\n");

            exit(0);}

;

S:*A* *S* *B* *B*

|

;

%%

int yyerror (char \**msg*) {

    printf ("\nExpression is invalid.\n", msg);

}

int main(){

printf("Enter an Expression: ");

yyparse();

}

int yywrap(){ return 1;}

**Output:**

