SSBT's College of Engineering & Technology, Bambhori, Jalgaon

Department of Computer Engineering

Name: _	Date of Performance: _ _ /_ _/20_ _
Class: B.E. Computer	Date of Completion: _ _ /_ _/20_ _
Division :	Grade:
Batch:	
Roll No:	Sign. of Teacher with Date:
Subject: Compiler Design Lab	

Experiment No. 3

Aim: To Implement a Calculator using LEX and YACC.

1. Objective: Student able to understand implementation of grammar rule by using a Calculator implementation through LEX and YACC.

2. **Background:**

Yacc takes a grammar that you specify and writes a parser that recognizes valid "sentences" in that grammar. We use the term "sentence" here in a fairly general way-for a C language grammar the sentences are syntactically valid C programs.

a grammar is a series of rules that the parser uses to recognize syntactically valid input. For example, here is a version of the grammar we'll use later in this chapter to build a calculator.

Statement →*NAME* = expression

Expression →*NUMBER* + *NUMBER* | *NUMBER* -*NUMBER*

The vertical bar, "|", means there are two possibilities for the same symbol,

i.e., an expression can be either an addition or a subtraction. The symbol to the left of the →is known as the left-hand side of the rule, often abbreviated LHS, and the symbols to the right are the right-hand side, usually abbreviated RHS. Several rules may have the same left-hand side; the vertical bar is just a short hand for this. Symbols that actually appear in the input and are returned by the lexer are terminal symbols or tokens, while those that appear on the left-hand side of some rule are non-terminal symbols or non-terminals. Terminal and non-terminal symbols must be different; it is an error to write a rule with a token on the left side.

3. Pre-lab Task:

A yacc grammar has the same **three-part** structure as a lex specification. (Lex copied its structure from yacc.) The first section, the **definition section**, handles control information for the yacc-generated parser (from here on we will call it the parser), and generally sets up the execution environment in which the parser will operate. The second section contains the **rules for the parser**, and the third section is **C code** copied verbatim into the generated C program.

Definition Section

The definition section includes declarations of the tokens used in the grammar,the types of values used on the parser stack, and other odds and ends.It can also include a literal block, C code enclosed in %{ %} lines. We start our first parser by declaring two symbolic tokens.

```
 %{
        #include<stdio.h>

%}

%union {
        double dval;
        int vblno;
        }

%token <vblno> NAME
%token <dval> NUMBER
%left '+''-'
%left '*''/'
%nonassoc uniminus
%type <dval> expression
```

You can use single quoted characters as tokens without declaring them, sowe don't need to declare "=", "+", or "-".

Rules Section

The rules section simply consists of a list of grammar rules in much thesame format as we used above. Since **ASCII** keyboards don't have a → key, we use a colon between the left- and right-hand sides of a rule, and we put a semicolon at the end of each rule:

```
%%
statement:NAME '=' expression {$1=$3;}
         | expression {printf("%f",$1);}
         ;
expression: expression '+' expression {$$=$1+$3;}
          | expression '-' expression {$$=$1-$3;}
          | expression '*' expression {$$=$1*$3;}
          | expression '/' expression {
                                  if($3==0.0)
                                  {
                                          yyerror("divide by zero error");
                                  }
                                  else
                                          {$$=$1/$3;}
                                         }
           |'-'expression %prec uniminus {$$=-$2;}
          | '('expression')'{$$=$2;}
          | NUMBER{$$=$1;}
          ;
%%
```

C code

The parser reduces a rule, it executes user C code associated withthe rule, known as the rule's *action.* The action appears in braces after theend of the rule, before the semicolon or vertical bar. The action code can refer to the values of the right-hand side symbols as $1, **$2,** . . . , and can set the value of the left-hand side by setting **$$.** In our parser, the value of an *expression* symbol is the value of the expression it represents. We add some code to evaluate and print expressions, bringing our grammar up to that used:

```
int main()
{
        yyparse();
        return(0);
}
void yyerror(char *s)
{
        printf("%s",s);
}
```

The rules that build an expression compute the appropriate values, and the rule that recognizes an expression as a statement prints out the result. Inthe expression building rules, the first and second numbers' values are **$1**and **$3,** respectively. The operator's value would be ***$2,*** although in this grammar the operators do not have interesting values. The action on the last rule is not strictly necessary, since the default action that yacc performs after every reduction, before running any explicit action code, assigns the value $1 to $$.

The Lexer

To try out our parser, we need a lexer to feed it tokens. The parser is the higher level routine, and calls the lexer   yylex() whenever it needs a token from the input. As soon as the lexer finds a token of interest to the parser, it returns to the parser, returning the token code as the value. Yacc defines the token names in the parser as C preprocessor names in *y.tab.h* (or some similar name on MS-DOS systems) so the lexer can use them.

```
%{
#include<math.h>
#include"y.tab.h"

%}
%%
[0-9]+ {yylval.dval=atoi(yytext); return NUMBER;}
[\t] ;
\n  return(0);
. return yytext[0];
%%
```

4. In-lab Task:

1. **Definition section**
2. **Rules for the parser**
3. **C Code**
4. **lexer**

5. Post-lab Task:

Outcomes:

. . . .

. . . .

. . . .

. . . .

Question:

Write a grammar for implement arithmetic calculator.