EXP NO: 8 **DATE:**

GENERATE THREE ADDRESS CODE FOR A SIMPLE PROGRAM USING LEX AND YACC

AIM:

To design and implement a **LEX and YACC** program that generates **three-address code (TAC)** for a simple arithmetic expression or program. The program will:

- Recognize **expressions** like addition, subtraction, multiplication, and division.
- Generate **three-address code** that represents the operations in a way that could be directly translated into assembly code or intermediate code for a compiler.

ALGORITHM:

1. Lexical Analysis (LEX) Phase:

Input: A string containing an arithmetic expression (e.g., a = b + c * d;).

Output: A stream of tokens such as identifiers (variables), numbers (constants), operators, and special characters (like =, ;, (), etc.).

1. **Define the Token Patterns:**
    - **ID:** Identifiers (variables) are strings starting with a letter and followed by letters or digits (e.g., a, b, result).
    - **NUMBER:** Constants (e.g., 1, 5, 100).
    - **OPERATOR:** Arithmetic operators (+, -, *, /).
    - **ASSIGNMENT:** Assignment operator (=).
    - **PARENTHESIS:** Parentheses for grouping ((and)).
    - **WHITESPACE:** Spaces, tabs, and newline characters (which should be ignored).
2. **Write Regular Expressions for the Tokens:**
    - ID -> [a-zA-Z_][a-zA-Z0-9_]*
    - NUMBER -> [0-9]+
    - OPERATOR -> [\+\-*/]
    - ASSIGN -> "="
    - PAREN -> [\(\)]
    - WHITESPACE -> [\t\n]+ (skip whitespace)
3. **Action on Tokens:**
    - When a token is matched, pass it to **YACC** using yylval to store the token values.

2. Syntax Analysis and TAC Generation (YACC) Phase:

Input: Tokens provided by the **LEX** lexical analyzer.

Output: Three-address code for the given arithmetic expression.

1. **Define Grammar Rules:**
    - **Assignment:**

      bash
      CopyEdit
      ```
      statement: ID '=' expr
      ```

      This means an expression is assigned to a variable.

    - **Expressions:**

      bash
      CopyEdit
      ```
      expr: expr OPERATOR expr
      ```

      An expression can be another expression with an operator (+, -, *, /).

      bash
      CopyEdit
      ```
      expr: NUMBER
      expr: ID
      expr: '(' expr ')'
      ```

2. **Three-Address Code Generation:**
    - For every arithmetic operation, generate a temporary variable (e.g., t1, t2, etc.) to hold intermediate results.
    - For a = b + c, generate:

      ini
      CopyEdit
      ```
      t1 = b + c
      a = t1
      ```

    - For a = b * c + d, generate:

      ini
      CopyEdit
      ```
      t1 = b * c
      t2 = t1 + d
      a = t2
      ```

3. **Temporary Variable Management:**
    - Keep a counter (temp_count) for generating unique temporary variable names (t0, t1, t2, ...).
    - Each time a new operation is encountered, increment the temp_count to generate a new temporary variable.
4. **Rule Actions:**
    - When a rule is matched (e.g., expr OPERATOR expr), generate the TAC and assign temporary variables for intermediate results.

Detailed Algorithm:

1. **Initialize Lexical Analyzer:**
   - Define the token patterns for ID, NUMBER, OPERATOR, ASSIGN, PAREN, and WHITESPACE.
2. **Define the Syntax Grammar:**
   - Define grammar rules for:
     - **Assignments:** ID = expr
     - **Expressions:** expr -> expr OPERATOR expr, expr -> NUMBER, expr -> ID, expr -> (expr)
3. **Token Matching:**
   - **LEX:** Match input characters against the defined regular expressions for tokens.
   - **YACC:** Use the tokens to parse and apply grammar rules.
4. **TAC Generation:**
   - **For Assignment:**
     - Upon parsing ID = expr, generate a temporary variable for the result of expr and assign it to the variable ID.
   - **For Arithmetic Operations:**
     - For each operator (e.g., +, -, *, /), generate temporary variables for intermediate calculations.
5. **Output TAC:**
   - Print the generated three-address code, with each expression and its intermediate results represented by temporary variables.

PROGRAM:

LEX file (expr.l)

```
%{
#include "y.tab.h"
%}

%%
[0-9]+      { yylval.str = strdup(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); return ID; }
[+\-*/=()]  { return yytext[0]; }
[ \t\n]     { /* Ignore whitespace */ }
.           { printf("Unexpected character: %s\n", yytext); }
%%
```

YACC Program expr.y

```
%{
#include  <stdio.h>
#include <stdlib.h>
#include <string.h>
int temp_count = 0;
```

```
char* new_temp() {
   char* temp = (char*)malloc(8);
   sprintf(temp, "t%d", temp_count++);
   return temp;
}

void emit(char* result, char* op1, char op, char* op2) {
   printf("%s = %s %c %s\n", result, op1, op, op2);
}

void emit_assign(char* id, char* expr) {
   printf("%s = %s\n", id, expr);
}
%}

%union {
   char* str;
}

%token <str> ID NUMBER
%type <str> expr term factor

%left '+' '-'
%left '*' '/'

%%
statement   : ID '=' expr { emit_assign($1, $3); }
            ;

expr        : expr '+' term { $$ = new_temp(); emit($$, $1, '+', $3); }
            | expr '-' term { $$ = new_temp(); emit($$, $1, '-', $3); }
            | term          { $$ = $1; }
            ;

term        : term '*' factor { $$ = new_temp(); emit($$, $1, '*', $3); }
            | term '/' factor { $$ = new_temp(); emit($$, $1, '/', $3); }
            | factor          { $$ = $1; }
            ;

factor      : '(' expr ')' { $$ = $2; }
            | NUMBER       { $$ = $1; }
            | ID           { $$ = $1; }
            ;
%%

int main() {
   yyparse();
   return 0;
}
```

```
void yyerror(const char* s) {
   fprintf(stderr, "Error: %s\n", s);
}
```

OUTPUT :

yacc -d expr.y

lex expr.l

gcc y.tab.c lex.yy.c -o expr_parser

./expr_parser

a = b * c + d;

t0 = b * c

t1 = t0 + d

a = t1

Implementation	
Output/Signature	

RESULT:

Thus the process effectively tokenizes the input, parses it according to defined grammar rules, and generates the corresponding Three-Address Code, facilitating further compilation or interpretation stages.