MANOJ RL 220701161

EXP NO: 08 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:
   - o ID: Identifiers (variables) are strings starting with a letter and followed by letters or digits (e.g., a, b, result). o NUMBER: Constants (e.g., 1, 5, 100).
   - o OPERATOR: Arithmetic operators (+, -, *, /). o ASSIGNMENT: Assignment operator (=). o PARENTHESIS: Parentheses for grouping ((and)).
   - o WHITESPACE: Spaces, tabs, and newline characters (which should be ignored).
2. Write Regular Expressions for the Tokens:
   - o ID -> [a-zA-Z_][a-zA-Z0-9_]* o NUMBER -> [0-9]+ o OPERATOR -> [\+\-*/] o ASSIGN -> "=" o PAREN -> [\(\)]
   - o WHITESPACE -> [\t\n]+ (skip whitespace)
3. Action on Tokens:
   - o 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
     CopyEdi
     t 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:

- o Keep a counter (temp_count) for generating unique temporary variable names
  (t0, t1, t2, ...).
- o Each time a new operation is encountered, increment the temp_count to generate a new temporary variable.

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

PROGRAM:

3address.l

```
%{
#include "3address.tab.h"
#include <string.h>
#include <stdlib.h>
%}

ID    [a-zA-Z_][a-zA-Z0-9_]*
```

```
NUM    [0-9]+

%%

{ID}    { yylval.str = strdup(yytext); return ID; }
{NUM}   { yylval.str = strdup(yytext); return NUM; }
"="     { return '='; }
";"     { return ';'; }
"("     { return '('; }
")"     { return ')'; }
"+"     { return '+'; }
"-"     { return '-'; }
"*"     { return '*'; }
"/"     { return '/'; }
[ \t\n] ;  // skip whitespace

%%

int yywrap() {
return 1; }
```

3address.y

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

int tempCount = 0;

char* createTemp() {     char* temp =
(char*)malloc(10);     sprintf(temp,
"t%d", tempCount++);     return
temp;
}

void yyerror(const char* s);
int yylex();
%}

%union {
   char* str;
}

%token <str> ID NUM
```

```
%type <str> expr

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

%%

stmt:
      ID '=' expr ';' {
printf("%s = %s\n", $1, $3);
      }
    ;

expr:
      expr '+' expr {
          char* temp = createTemp();
printf("%s = %s + %s\n", temp, $1, $3);
          $$ = temp;
      }
    | expr '-' expr {           char* temp =
createTemp();           printf("%s = %s -
%s\n", temp, $1, $3);
          $$ = temp;
      }
    | expr '*' expr {           char* temp =
createTemp();           printf("%s = %s *
%s\n", temp, $1, $3);
          $$ = temp;
}
    | expr '/' expr {           char* temp =
createTemp();           printf("%s = %s /
%s\n", temp, $1, $3);
          $$ = temp;
      }
    | '(' expr ')' {
$$ = $2;
      }
    | ID {
          $$ = strdup($1);
      }
    | NUM {
          $$ = strdup($1);
      }
    ;
```

```
%%

void yyerror(const char* s) {
printf("Syntax Error: %s\n", s);
}

int main() {     printf("Enter an arithmetic
expression :\n");     yyparse();     return 0;
}
```

OUTPUT :

yacc -d expr.y lex expr.l gcc
y.tab.c lex.yy.c -o expr_parser
./expr_parse
r 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.