EXP NO:7 **DATE:**

RECOGNIZE A VALID CONTROL STRUCTURES SYNTAX OF C LANGUAGE (FOR LOOP, WHILE LOOP, IF-ELSE, IF-ELSE-IF, SWITCH CASE, ETC.,

AIM:

To design and implement a LEX and YACC program that recognizes the syntax of common control structures in C programming, including:
For loop
While loop
If-else
If-else-if
Switch-case

ALGORITHM:

LEX (Lexical Analyzer)

1. Start
2. Define token patterns for:
   - o Keywords (e.g., if, else, for, while, switch, case)
   - o Identifiers (variable names)
   - o Operators (arithmetic and relational)o Parentheses ((), {}, etc.)
   - o Semicolon (;)
3. Pass recognized tokens to YACC for syntax validation.
4. End

YACC (Syntax Analyzer)

1. Start
2. Define grammar rules for:
   - o For loop: for(initialization; condition; increment) { ... }
   - o While loop: while(condition) { ... }
   - o If-else: if(condition) { ... } else { ... }
   - o If-else-if: if(condition) { ... } else if(condition) { ... } else { ... }
   - o Switch-case: switch(expression) { case value: ... default: ... }
3. Parse the input expression and validate the syntax of the control structures.
4. Print appropriate messages for valid or invalid control structure syntax.
5. End

PROGRAM:

LEX File (control_structures.l):
```
%{
#include "y.tab.h"
%}
%%
"if" { return IF; }
"else" { return ELSE; }
"for" { return FOR; }
"while" { return WHILE; }
"switch" { return SWITCH; }
"case" { return CASE; }
[a-zA-Z_][a-zA-Z0-9_]* { return IDENTIFIER; }
"=="|"!="|"<="|">="|"<"|">" { return REL_OP; }
"+"|"-"|"*"|"/" { return ARITH_OP; }
```

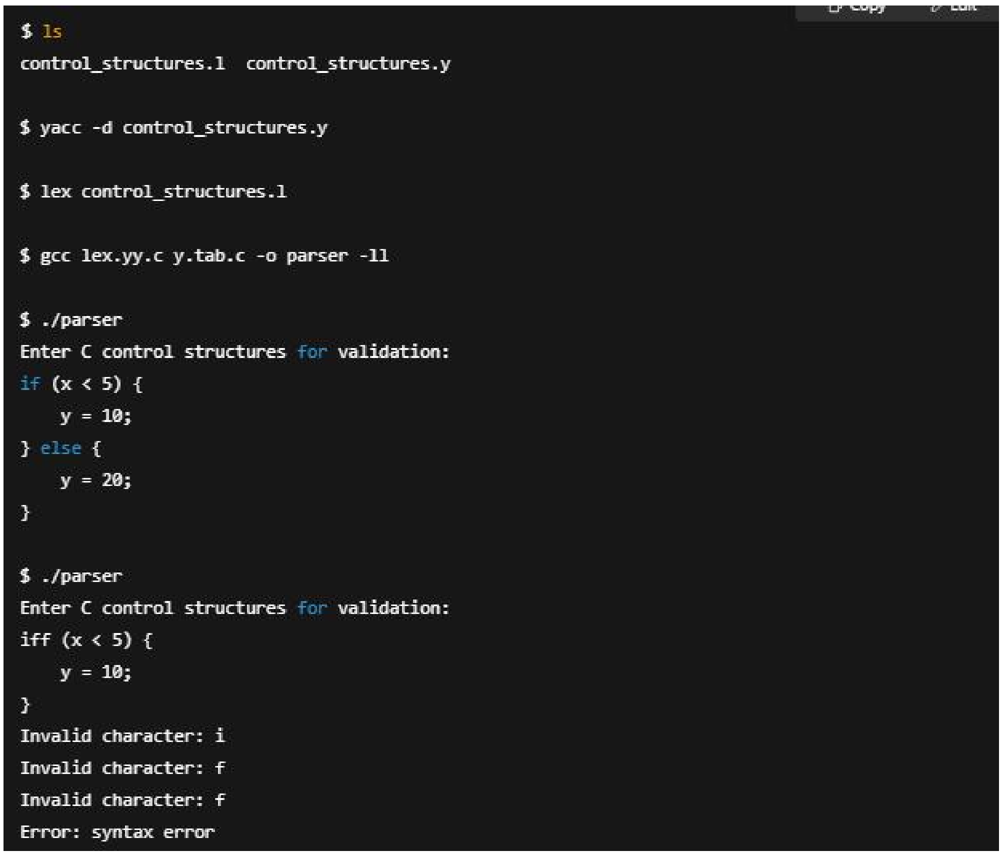
```
"(" { return LPAREN; }
")" { return RPAREN; }
"{" { return LBRACE; }
"}" { return RBRACE; }
";" { return SEMICOLON; }
[ \t\n] ; /* Ignore whitespace */
. { printf("Invalid character: %s\n", yytext); }
%%
int yywrap() {
return 1;
}
```

YACC File (control_structures.y)

```
%{
#include <stdio.h>
#include <stdlib.h>
void yyerror(const char *s);
int yylex(void);
%}
%token IF ELSE FOR WHILE SWITCH CASE IDENTIFIER REL_OP ARITH_OP
%token LPAREN RPAREN LBRACE RBRACE SEMICOLON
%start program
%%
program:
statement
| program statement
;
statement:
if_statement
| for_loop
| while_loop
| switch_case
;
if_statement:
IF LPAREN condition RPAREN LBRACE statements RBRACE
| IF LPAREN condition RPAREN LBRACE statements RBRACE ELSE LBRACE
statements RBRACE
;
for_loop:
FOR LPAREN assignment SEMICOLON condition SEMICOLON assignment RPAREN
LBRACE statements RBRACE
;
while_loop:
WHILE LPAREN condition RPAREN LBRACE statements RBRACE
;
switch_case:
SWITCH LPAREN expression RPAREN LBRACE case_statements RBRACE
;
case_statements:
CASE expression COLON statements
| case_statements CASE expression COLON statements
| case_statements DEFAULT COLON statements
;
condition:
IDENTIFIER REL_OP IDENTIFIER
| IDENTIFIER REL_OP NUMBER
| NUMBER REL_OP IDENTIFIER
| NUMBER REL_OP NUMBER
```

```
;
assignment:
IDENTIFIER '=' expression
;
expression:
IDENTIFIER
| NUMBER
| expression ARITH_OP expression
;
statements:
statement
| statements statement
;
%%
void yyerror(const char *s) {
fprintf(stderr, "Error: %s\n", s);
}
int main() {
printf("Enter C control structures for validation:\n");
yyparse();
return 0;
}
```

OUTPUT:



```
$ ls
control_structures.l  control_structures.y

$ yacc -d control_structures.y

$ lex control_structures.l

$ gcc lex.yy.c y.tab.c -o parser -ll

$ ./parser
Enter C control structures for validation:
if (x < 5) {
    y = 10;
} else {
    y = 20;
}

$ ./parser
Enter C control structures for validation:
iff (x < 5) {
    y = 10;
}
Invalid character: i
Invalid character: f
Invalid character: f
Error: syntax error
```

RESULT:

Thus the above program to recognize a valid control structures syntax of c language (for loop, while loop, if-else, if-else-if, switch case as been implemented and executed successfully with LEX and YACC.

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:
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:
o Assignment:
bash
CopyEdit
statement: ID '=' expr
This means an expression is assigned to a variable.
o 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:
o For every arithmetic operation, generate a temporary variable (e.g., t1, t2, etc.) to hold intermediate results.
o For a = b + c, generate:
ini
CopyEdit
```
t1 = b + c
a = t1
```
o 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:

LEX file (expr.l)

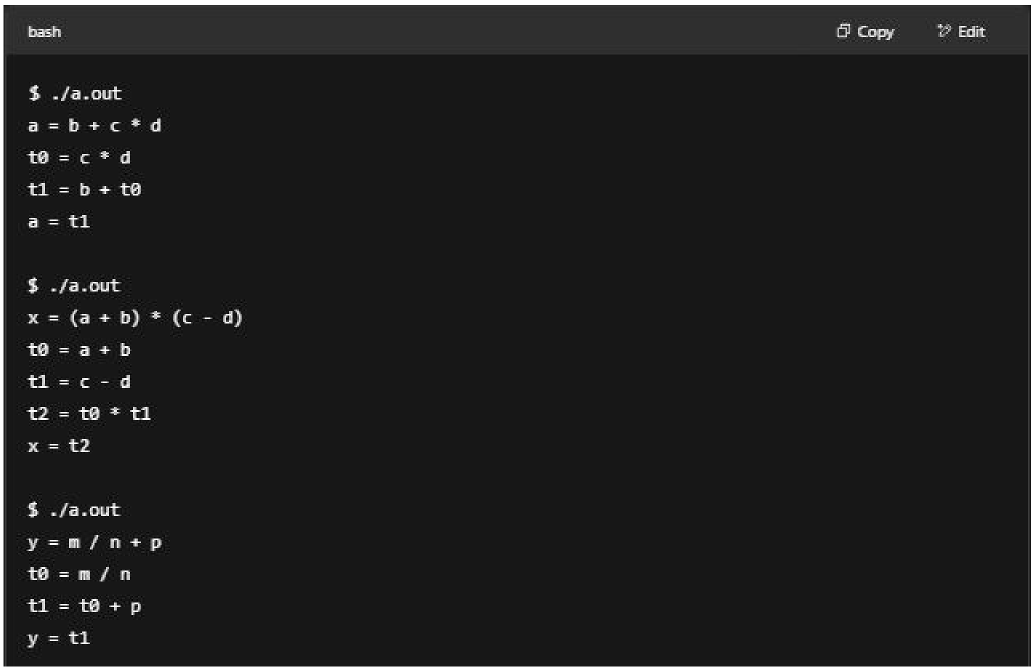
```
%{
#include "y.tab.h"
%}
%%
[0-9]+ { yylval.str = strdup(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); return ID; }
[+\-*/=()] { return yytext[0]; }
[ \t\n] { /* Ignore whitespace */ }
. { printf("Unexpected character: %s\n", yytext); }
%%
```

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:



```
$ ./a.out
a = b + c * d
t0 = c * d
t1 = b + t0
a = t1

$ ./a.out
x = (a + b) * (c - d)
t0 = a + b
t1 = c - d
t2 = t0 * t1
x = t2

$ ./a.out
y = m / n + p
t0 = m / n
t1 = t0 + p
y = t1
```

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.

EXP NO:9 **DATE:**

DEVELOP THE BACK-END OF A COMPILER THAT TAKES THREE-ADDRESSCODE (TAC) AS INPUT AND GENERATES CORRESPONDING 8086 ASSEMBLY LANGUAGE CODE AS OUTPUT.

AIM:

To design and implement the back-end of a compiler that takes three-address code (TAC) as input and produces 8086 assembly language code as output. The three-address code is an intermediate representation used by compilers to break down expressions and operations, while the 8086 assembly code is a machine-level representation of the program that can be executed by a processor.

ALGORITHM:

1. Parse the Three-Address Code (TAC):
Input: Three-Address Code, which is an intermediate representation. For example:
```
t0 = b + c
t1 = t0 * d
a = t1
```
Output: 8086 assembly language code. For example:
```
MOV AX, [b] ; Load b into AX
ADD AX, [c] ; Add c to AX
MOV [t0], AX ; Store result in t0
```

2. Process Each TAC Instruction:
1. Initialize Registers:
o Set up the registers in 8086 assembly (e.g., AX, BX, CX, etc.) for storing intermediate results and final outputs.
o Maintain a temporary register counter for naming temporary variables in TAC (e.g., t0, t1).

2. For each TAC instruction, based on its operation:
o Identify the components: operands and operator.
o Choose an appropriate register (AX, BX, etc.) for storing intermediate results.
o If the operation involves multiple operands or temporary variables, map them to registers.

3. Translating TAC to 8086 Assembly:
 Addition/Subtraction (e.g., t0 = b + c):
o Load operands into registers and perform the operation:

```
MOV AX, [b] ; Load b into AX
ADD AX, [c] ; Add c to AX
MOV [t0], AX ; Store result in t0
```

 Multiplication (e.g., t1 = t0 * d):
o Load operands into registers and perform the operation:
```
MOV AX, [t0] ; Load t0 into AX
MOV BX, [d] ; Load d into BX
MUL BX ; Multiply AX by BX (result in AX)
MOV [t1], AX ; Store result in t1
```

 Assignment (e.g., a = t1):
o Move the value from a temporary variable to the target variable:
```
MOV [a], [t1] ; Move value of t1 into a
```

Division (e.g., t2 = b / c):
o Division is a bit more complex due to the 8086's limitations with the DIV instruction. For example, the result might need to be stored in AX or DX:AX (if it's a 32-bit result):

```
MOV AX, [b] ; Load b into AX
MOV DX, 0 ; Clear DX (important for division)
MOV BX, [c] ; Load c into BX
DIV BX ; AX = AX / BX (quotient in AX, remainder in DX)
MOV [t2], AX ; Store quotient in t2
```

4. Manage Memory and Registers:
Variables: Variables like a, b, c are stored in memory, so you will use memory addressing modes such as [variable_name] to access them.
Temporary Variables: Temporary variables like t0, t1, t2, etc., are stored in registers (AX, BX, etc.) or memory if there are more variables than registers available.

5. Handle Control Flow (Optional):
If the TAC contains control structures (such as loops, if-else statements, or function calls), you will need to generate labels and jump instructions in 8086 assembly.
If Statements: For example, if (x > 0) { y = 1; } could generate:

```
MOV AX, [x]
CMP AX, 0
JG positive_case ; Jump if greater
JMP end_if
```

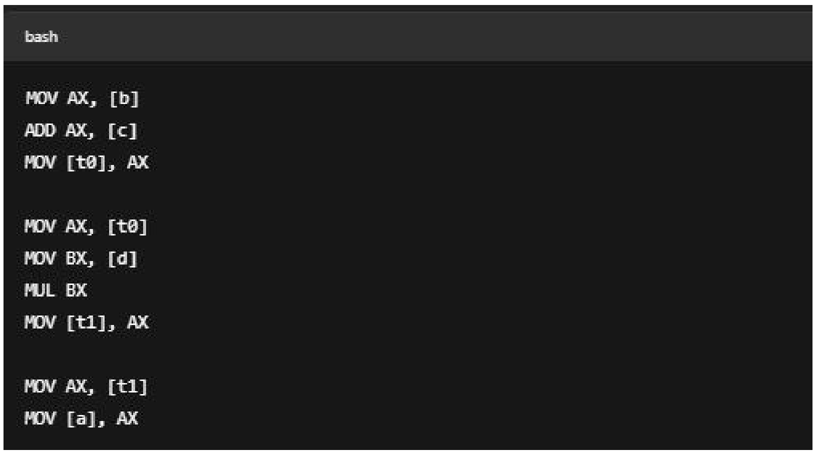
PROGRAM:

```
#include <stdio.h>
#include <string.h>
void generateAssembly(const char* tac) {
char result[10], op1[10], op[2], op2[10];
// Parse the TAC instruction
sscanf(tac, "%s = %s %s %s", result, op1, op, op2);
// Generate assembly code based on the operator
if (strcmp(op, "+") == 0) {
printf("MOV AX, [%s]\n", op1);
printf("ADD AX, [%s]\n", op2);
printf("MOV [%s], AX\n", result);
} else if (strcmp(op, "-") == 0) {
printf("MOV AX, [%s]\n", op1);
printf("SUB AX, [%s]\n", op2);
printf("MOV [%s], AX\n", result);
} else if (strcmp(op, "*") == 0) {
printf("MOV AX, [%s]\n", op1);
printf("MOV BX, [%s]\n", op2);
printf("MUL BX\n");
printf("MOV [%s], AX\n", result);
} else if (strcmp(op, "/") == 0) {
printf("MOV AX, [%s]\n", op1);
printf("MOV BX, [%s]\n", op2);
printf("DIV BX\n");
printf("MOV [%s], AX\n", result);
} else {
printf("Unsupported operation: %s\n", op);
```

```
}
}
int main() {
const char* tacInstructions[] = {
```

```
"t0 = b + c",
"t1 = t0 * d",
"a = t1"
};
int numInstructions = sizeof(tacInstructions) / sizeof(tacInstructions[0]);
for (int i = 0; i < numInstructions; i++) {
generateAssembly(tacInstructions[i]);
printf("\n");
}
return 0;
}
```

OUTPUT:



```
MOV AX, [b]
ADD AX, [c]
MOV [t0], AX

MOV AX, [t0]
MOV BX, [d]
MUL BX
MOV [t1], AX

MOV AX, [t1]
MOV [a], AX
```

RESULT:

Thusthe above example provides a foundational approach to converting TAC to 8086 assembly using C. For a complete compiler back-end, you would need to handle additional aspects such as register allocation, memory management, and more complex control flow constructs.

EXP NO:10 **DATE:**

GENERATE THREE ADDRESS CODES FOR A GIVEN EXPRESSION(ARITHMETIC EXPRESSION, FLOW OF CONTROL)

AIM:

The aim is to generate Three-Address Code (TAC) for a given arithmetic expression and flow of control (e.g., if-else, loops). TAC is an intermediate representation used in compilers to simplify the task of code generation. It consists of simple instructions that make it easier to translate into machine-level code.
For example, for an arithmetic expression a = b + c * d, the TAC would break it down into simpler steps, using temporary variables to hold intermediate results.

ALGORITHM:

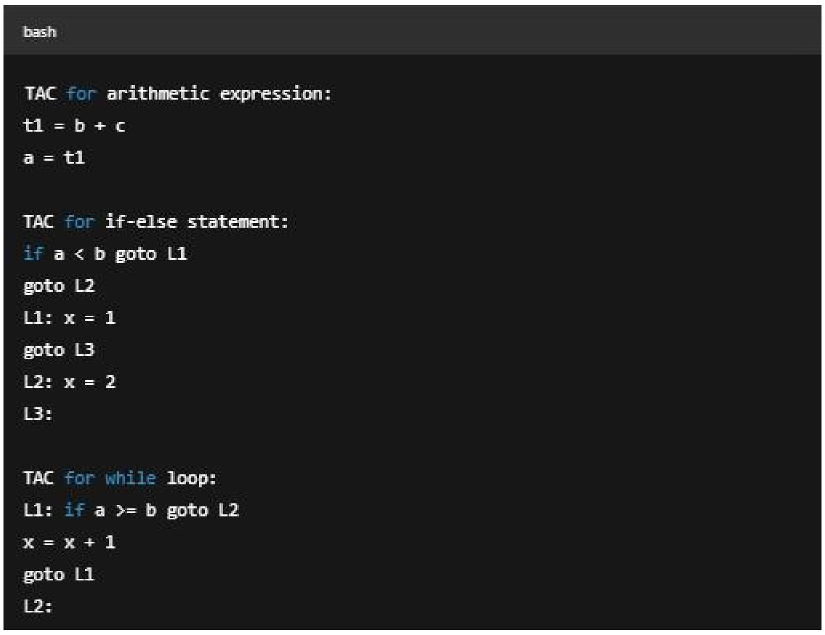
- The expression is read from the file using a file pointer
- Each string is read and the total no. of strings in the file is calculated.
- Each string is compared with an operator; if any operator is seen then the previous string and next string are concatenated and stored in a first temporary value and the three address code expression is printed
- Suppose if another operand is seen then the first temporary value is concatenated to the next string using the operator and the expression is printed.
- The final temporary value is replaced to the left operand value.

PROGRAM:

```
#include <stdio.h>
#include <string.h>
// Function to generate TAC for arithmetic expressions
void generateArithmeticTAC(const char* expr) {
// Example: a = b + c * d
// Expected TAC:
// t1 = c * d
// t2 = b + t1
// a = t2
char result[10], op1[10], op[2], op2[10];
sscanf(expr, "%s = %s %s %s", result, op1, op, op2);
if (strcmp(op, "+") == 0 || strcmp(op, "-") == 0) {
printf("t1 = %s %s %s\n", op1, op, op2);
printf("%s = t1\n", result);
} else if (strcmp(op, "*") == 0 || strcmp(op, "/") == 0) {
printf("t1 = %s %s %s\n", op1, op, op2);
printf("%s = t1\n", result);
} else {
printf("Unsupported operation: %s\n", op);
}
```

```
}
// Function to generate TAC for if-else statements
void generateIfElseTAC(const char* condition, const char* trueStmt, const char* falseStmt)
{
// Example:
// if (a < b) x = 1; else x = 2;
// Expected TAC:
// if a < b goto L1
// goto L2
// L1: x = 1
// goto L3
// L2: x = 2
// L3:
printf("if %s goto L1\n", condition);
printf("goto L2\n");
printf("L1: %s\n", trueStmt);
printf("goto L3\n");
printf("L2: %s\n", falseStmt);
printf("L3:\n");
}
// Function to generate TAC for while loops
void generateWhileLoopTAC(const char* condition, const char* body) {
// Example:
// while (a < b) { x = x + 1; }
// Expected TAC:
// L1: if a >= b goto L2
// x = x + 1
// goto L1
// L2:
printf("L1: if %s goto L2\n", condition);
printf("%s\n", body);
printf("goto L1\n");
printf("L2:\n");
}
int main() {
// Example usage:
printf("TAC for arithmetic expression:\n");
generateArithmeticTAC("a = b + c");
printf("\nTAC for if-else statement:\n");
generateIfElseTAC("a < b", "x = 1", "x = 2");
printf("\nTAC for while loop:\n");
generateWhileLoopTAC("a >= b", "x = x + 1");
return 0;
}
```

OUTPUT:



```
TAC for arithmetic expression:
t1 = b + c
a = t1

TAC for if-else statement:
if a < b goto L1
goto L2
L1: x = 1
goto L3
L2: x = 2
L3:

TAC for while loop:
L1: if a >= b goto L2
x = x + 1
goto L1
L2:
```

RESULT:

Thus the above program is the simplified example and a complete implementation and it would need to handle more complex expressions, nested control structures, and ensure proper parsing of the input.

EXP NO:11 **DATE:**

IMPLEMENT CODE OPTIMIZATION TECHNIQUES LIKE DEAD CODE AND COMMON EXPRESSION ELIMINATION

AIM:

The aim is to implement code optimization techniques such as Dead Code Elimination (DCE) and Common Subexpression Elimination (CSE) on an intermediate representation of a program (such as Three-Address Code (TAC)). These optimization techniques help reduce the size of the code, improve runtime performance, and eliminate redundant computations during the compilation process.

ALGORITHM:

● Start
● Create the input file which contains three address code.
● Open the file in read mode.
● If the file pointer returns NULL, exit the program else go to 5.
● Scan the input symbol from left to right.
● Store the first expression in a string.
● Compare the string with the other expressions in the file.
● If there is a match, remove the expression from the input file.
● Perform these steps 5-8 for all the input symbols in the file.
● Scan the input symbol from the file from left to right.
● Get the operand before the operator from the three address code.
● Check whether the operand is used in any other expression in the three address code.
● If the operand is not used, then eliminate the complete expression from the three address code else go to 14.
● Perform steps 11 to 13 for all the operands in the three address code till end of the file is reached.
● Stop.

PROGRAM:

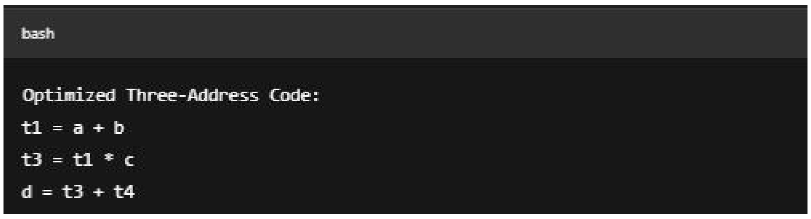
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_CODE_LINES 100
#define MAX_LINE_LENGTH 100
#define MAX_VAR_LENGTH 20
typedef struct {
char lhs[MAX_VAR_LENGTH];
char op1[MAX_VAR_LENGTH];
char operator;
char op2[MAX_VAR_LENGTH];
int isDead;
} TAC;

// Function to parse a TAC line
void parseTACLine(char *line, TAC *tac) {
sscanf(line, "%s = %s %c %s", tac->lhs, tac->op1, &tac->operator, tac->op2);
tac->isDead = 0;
}
// Function to perform Dead Code Elimination
void performDCE(TAC tac[], int n) {
```

```
int used[MAX_CODE_LINES] = {0};
// Mark variables that are used
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
if (strcmp(tac[i].lhs, tac[j].op1) == 0 || strcmp(tac[i].lhs, tac[j].op2) == 0) {
used[i] = 1;
break;
}
}
}
// Eliminate dead code
for (int i = 0; i < n; i++) {
if (!used[i]) {
tac[i].isDead = 1;
}
}
}
// Function to perform Common Subexpression Elimination
void performCSE(TAC tac[], int n) {
for (int i = 0; i < n; i++) {
if (tac[i].isDead) continue;
for (int j = i + 1; j < n; j++) {
if (tac[j].isDead) continue;
if (strcmp(tac[i].op1, tac[j].op1) == 0 &&
strcmp(tac[i].op2, tac[j].op2) == 0 &&
tac[i].operator == tac[j].operator) {
// Replace the second occurrence with the first
strcpy(tac[j].op1, tac[i].lhs);
tac[j].operator = '\0';
strcpy(tac[j].op2, "");
tac[j].isDead = 1;
}
}
}
}
// Function to print the optimized TAC
void printOptimizedTAC(TAC tac[], int n) {
printf("Optimized Three-Address Code:\n");
for (int i = 0; i < n; i++) {
if (!tac[i].isDead) {
printf("%s = %s", tac[i].lhs, tac[i].op1);
if (tac[i].operator != '\0') {
printf(" %c %s", tac[i].operator, tac[i].op2);
}
printf("\n");
}
}
}
int main() {
char *code[] = {
"t1 = a + b",
"t2 = a + b",
"t3 = t1 * c",
"t4 = t2 * c",
"d = t3 + t4",
"e = t5 - t6"
};
```

```
int n = sizeof(code) / sizeof(code[0]);
TAC tac[MAX_CODE_LINES];
// Parse the TAC lines
for (int i = 0; i < n; i++) {
parseTACLine(code[i], &tac[i]);
}
// Perform Common Subexpression Elimination
performCSE(tac, n);
// Perform Dead Code Elimination
performDCE(tac, n);
// Print the optimized TAC
printOptimizedTAC(tac, n);
return 0;
}
```

OUTPUT:



```
bash

Optimized Three-Address Code:
t1 = a + b
t3 = t1 * c
d = t3 + t4
```

RESULT:

Thus The Above Program To Implement Code Optimization Techniques Like Dead Code And Common Expression Elimination Is Executed And Implemented Successfully.

EXP NO : 12 **DATE :**

IMPLEMENT CODE OPTIMIZATION TECHNIQUES COPY PROPAGATION

AIM:

The aim is to implement code optimization techniques like Dead Code Elimination (DCE) and Common Subexpression Elimination (CSE) to improve the efficiency and performance of a program. These techniques are applied to intermediate code (e.g., Three-Address Code or TAC) during the compilation process.

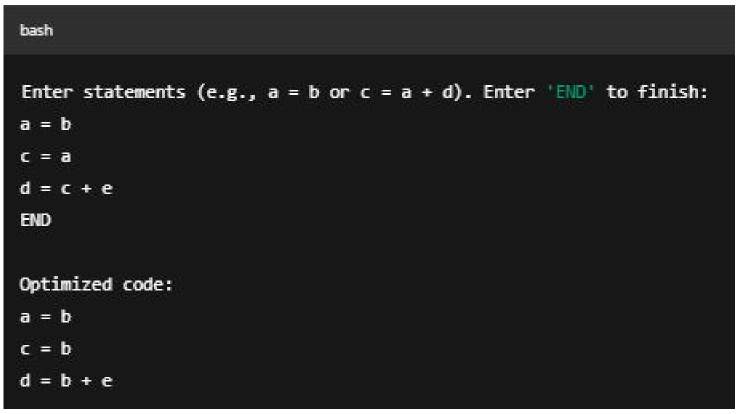
ALGORITHM:

The desired header files are declared.
The two file pointers are initialized one for reading the C program from the file and one for writing the converted program with constant folding
The file is read and checked if there are any digits or operands present.
If there is, then the evaluations are to be computed in switch case and stored.
Copy the stored data to another file.
Print the copied data file.

PROGRAM:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_LINES 100
#define MAX_LENGTH 50
typedef struct {
char var[MAX_LENGTH];
char value[MAX_LENGTH];
int is_direct_assignment;
} Statement;
void apply_copy_propagation(Statement statements[], int count) {
for (int i = 0; i < count; i++) {
if (statements[i].is_direct_assignment) {
char *lhs = statements[i].var;
char *rhs = statements[i].value;
for (int j = i + 1; j < count; j++) {
if (statements[j].is_direct_assignment) {
if (strcmp(statements[j].value, lhs) == 0) {
strcpy(statements[j].value, rhs);
}
} else {
char *pos = strstr(statements[j].value, lhs);
if (pos != NULL) {
char temp[MAX_LENGTH];
strcpy(temp, pos + strlen(lhs));
*pos = '\0';
strcat(statements[j].value, rhs);
strcat(statements[j].value, temp);
}
}
}
}
}
}
int main() {
Statement statements[MAX_LINES];
int count = 0;
printf("Enter statements (e.g., a = b or c = a + d). Enter 'END' to finish:\n");
char line[MAX_LENGTH];
while (fgets(line, sizeof(line), stdin)) {
if (strncmp(line, "END", 3) == 0) break;
line[strcspn(line, "\n")] = 0; // Remove newline character
char *equals = strchr(line, '=');
if (equals != NULL) {
*equals = '\0';
strcpy(statements[count].var, line);
strcpy(statements[count].value, equals + 1);
statements[count].is_direct_assignment = (strchr(equals + 1, '+') == NULL &&
strchr(equals + 1, '-') == NULL &&
strchr(equals + 1, '*') == NULL &&
strchr(equals + 1, '/') == NULL);
count++;
}
}
apply_copy_propagation(statements, count);
printf("\nOptimized code:\n");
for (int i = 0; i < count; i++) {
if (!(statements[i].is_direct_assignment && statements[i].value[0] == '\0')) {
printf("%s = %s\n", statements[i].var, statements[i].value);
}
}
return 0;
}
```

OUTPUT:



```bash
Enter statements (e.g., a = b or c = a + d). Enter 'END' to finish:
a = b
c = a
d = c + e
END

Optimized code:
a = b
c = b
d = b + e
```

RESULT:
Thus the above to implement code optimization techniques for copy propagation is executed successfully.