EX NO :1	USING THE LEX TOOL, DEVELOP A LEXICAL ANALYZER TO RECOGNIZE A FEW PATTERNS IN C. (EX. IDENTIFIERS, CONSTANTS, COMMENTS, OPERATORS ETC.). CREATE A SYMBOL TABLE, WHILE RECOGNIZING IDENTIFIERS.

AIM

To develop a lexical analyzer using the LEX tool that recognizes basic patterns in C, including keywords, operators, numbers, and identifiers, and to create a symbol table for identifiers

ALGORITHM

1. Start the program and include necessary header files (`stdio.h, stdlib.h, string.h`).
2. Define a symbol table using a 2D character array and a counter for storing identifiers.
3. Write a function `insertSymbol()` to add new identifiers into the symbol table if they are not already present.
4. Write a function `printSymbolTable()` to display all identifiers collected.
5. Specify patterns in LEX:
    - Keywords: int, float, char
    - Operators: =, +, -, *, /
    - Numbers: Integer and floating-point
    - Identifiers: Alphanumeric strings starting with a letter or underscore
6. Skip whitespaces and handle unknown characters.
7. Define `yywrap()` to terminate scanning.
8. Call `yylex()` in `main()` to start scanning input.
9. Print the symbol table after scanning is complete.

PROGRAM

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int count = 0;
char symtab[100][100];

void insertSymbol(char *sym) {
    for (int i = 0; i < count; i++)
        if (!strcmp(symtab[i], sym)) return;
    strcpy(symtab[count++], sym);
}

void printSymbolTable() {
    printf("\nSymbol Table:\n");
    for (int i = 0; i < count; i++)
        printf("%s\n", symtab[i]);
}
%}
%%
"int"|"float"|"char"    { printf("Keyword: %s\n", yytext); }
"="|"+"|"-"|"*"|"/"     { printf("Operator: %s\n", yytext); }
[0-9]+\.[0-9]+          { printf("Float Number: %s\n", yytext); }
[0-9]+                  { printf("Integer Number: %s\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]*  { printf("Identifier: %s\n", yytext); insertSymbol(yytext); }
[ \t\n]+                { /* skip whitespace */ }
.                       { printf("Unknown token: %s\n", yytext); }
%%
int yywrap(void) { return 1; }

int main() {
    printf("Enter your C code (Ctrl+Z to end input):\n\n");
    yylex();
    printSymbolTable();
    return 0;
}
```

PROCEDURE

1. Create a file named `lexical.l` and write the program.
2. Open the command prompt and navigate to the folder containing `lexical.l`.
3. Run `Win_flex` to generate the C source file:
   - `win_flex lexical.l`

4. Compile the generated source code using GCC:
   - `gcc lex.yy.c -o lexical.exe`

5. Run the program and provide C code input (from a file or manually):
   - `lexical.exe < input.c`

FOR THE INPUT FILE INPUT.C:

```
int main() {
    int i = 0;
    float x = 3.5;
    i = i + 1;
}
```

OUTPUT

RESULT

Thus, the lexical analyzer successfully identifies keywords, operators, numbers, and identifiers, and creates a symbol table for user-defined variables

EX NO: 2	USING THE WINFLEX TOOL TO DEVELOP A LEXICAL ANALYZER

AIM

To implement a complete lexical analyzer using the LEX tool that recognizes C language tokens

ALGORITHM

1. Start the program and include required header files `(stdio.h, stdlib.h, string.h)`.
2. Create a symbol table as a 2D array to store unique user-defined identifiers.
3. Define a function `insertSymbol()` to add identifiers to the symbol table only if they are not already present and not a keyword.
4. Define a function `printSymbolTable()` to display the collected identifiers at the end of lexical analysis.
5. Write token patterns in LEX for:
   - Keywords: int, float, char, if, else, while, for, do, switch, case, return, main
   - Operators: `=, +, -, *, /`
   - Relational operators: `==, !=, <, >, <=, >=`
   - Special symbols: `(, ), {, }, ;, ,`
   - Identifiers: `[a-zA-Z_][a-zA-Z0-9_]*`
   - Constants: Integer and floating-point numbers
   - Strings: Enclosed in double quote
6. Ignore whitespaces and newline characters
7. Define `yywrap()` to signal the end of input.
8. Call `yylex()` in `main()` to start the lexical analysis.
9. Display the recognized tokens and the final symbol table.

PROGRAM

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

int count = 0;
char symtab[100][100];

void insertSymbol(char *sym) {
    if (!strcmp(sym,"int") || !strcmp(sym,"float") || !strcmp(sym,"char") ||
        !strcmp(sym,"if") || !strcmp(sym,"else") || !strcmp(sym,"while") ||
        !strcmp(sym,"for") || !strcmp(sym,"do") || !strcmp(sym,"switch") ||
        !strcmp(sym,"case") || !strcmp(sym,"return") || !strcmp(sym,"main") ||
        !strcmp(sym,"printf") || !strcmp(sym,"scanf"))
        return;
    for (int i = 0; i < count; i++)
        if (!strcmp(symtab[i], sym)) return;
    strcpy(symtab[count++], sym);
}

void printSymbolTable() {
    printf("\nSymbol Table:\n");
    for (int i = 0; i < count; i++)
        printf("%s\n", symtab[i]);
}
%}
%%
"int"|"float"|"char"|"if"|"else"|"while"|"for"|"do"|"switch"|"case"|"return"|"main" {
printf("Keyword: %-10s\n", yytext); }
"=="|"!="|"<="|">="|"<"|">" { printf("Relational Operator: %-5s\n", yytext); }
"="|"+"|"-"|"*"|"/" { printf("Operator: %-5s\n", yytext); }
"(" { printf("Special Symbol: ( \n"); }
")" { printf("Special Symbol: ) \n"); }
"{" { printf("Special Symbol: { \n"); }
"}" { printf("Special Symbol: } \n"); }
";" { printf("Special Symbol: ; \n"); }
"," { printf("Special Symbol: , \n"); }
[0-9]+\.[0-9]+ { printf("Float Number: %-10s\n", yytext); }
```

```
[0-9]+         { printf("Integer Number: %-10s\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]* { printf("Identifier: %-10s\n", yytext); insertSymbol(yytext); }
\"([^\"\n]|\\.)*\" { printf("String: %s\n", yytext); }
[ \t\n]+ { }
. { printf("Unknown token: %s\n", yytext); }
%%

int yywrap(void) { return 1; }

int main() {
    printf("Enter your C code (Ctrl+Z to end input):\n\n");
    yylex();
    printSymbolTable();
    return 0;
}
```

PROCEDURE

1. Create a file named `lexer.l` and write the program.
2. Open the command prompt and navigate to the folder containing `lexer.l`.
3. Run `Win_flex` to generate the C source file:
   - `win_flex lexer.l`
4. Compile it using GCC:
   - `gcc lex.yy.c -o lexer.exe`
5. Create a test input file input.c with sample C code
6. Run the executable and redirect input from the file:
   - `lexer.exe < input.c`

OUTPUT

RESULT

Thus, the complete lexical analyzer was implemented successfully using LEX tool WINFLEX

EX NO: 3(A)	RECOGNIZE A VALID ARITHMETIC EXPRESSION

AIM

To write a LEX and YACC program that recognizes valid arithmetic expressions involving the operators +, -, *, and / and evaluates them

ALGORITHM

Lexical Analysis (LEX)

- Identify numbers (integer and float).
- Recognize operators +, -, *, / and parentheses (,).

Parsing (YACC)

- Define grammar rules for arithmetic expressions:

```
expr: expr + expr
    | expr - expr
    | expr * expr
    | expr / expr
    | ( expr )
    | number
```

- For each production, evaluate the expression recursively.

Execution

- Read an arithmetic expression from input.
- Tokenize the input using LEX.
- Parse using YACC rules and compute the result.
- Print the final result.

PROCEDURE / PROGRAM

1. Write the Lexical Analyzer (`arith_lex.l`)

```
%{

#include "arith_parser.tab.h"

#include <stdlib.h>

%}

%%

[0-9]+\.[0-9]+  { yylval.fval = atof(yytext); return FLOAT_CONST; }

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

"("             { return LPAREN; }

")"             { return RPAREN; }

"+"             { return PLUS; }

"-"             { return MINUS; }

"*"             { return MUL; }

"/"             { return DIV; }

[ \t\n]+        { /* skip whitespace */ }

.               { return yytext[0]; }

%%

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

2. Write the Parser `(arith_parser.y)`

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

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

%union {
    float fval;
}

%token <fval> INT_CONST FLOAT_CONST
%token PLUS MINUS MUL DIV LPAREN RPAREN

%type <fval> expr

%left PLUS MINUS
%left MUL DIV

%%
program:
      expr { printf("Result: %f\n", $1); }
    ;

expr:
      expr PLUS expr     { $$ = $1 + $3; }
    | expr MINUS expr    { $$ = $1 - $3; }
    | expr MUL expr      { $$ = $1 * $3; }
    | expr DIV expr      { $$ = $1 / $3; }
    | LPAREN expr RPAREN { $$ = $2; }
```

```
    | INT_CONST        { $$ = $1; }
    | FLOAT_CONST      { $$ = $1; }
    ;

%%

int main() {
    printf("Enter arithmetic expression:\n");
    printf("-------------------------------------------------\n");
    printf("1. Type your arithmetic expression.\n");
    printf("2. Press Enter.\n");
    printf("3. Then press Ctrl+Z and again press Enter to show the result.\n");
    printf("-------------------------------------------------\n");
    yyparse();
    return 0;
}

int yyerror(const char *s) {
    fprintf(stderr, "Syntax Error: %s\n", s);
    return 0;
}
```

3. Input.txt:

- `3 + 5 * (2 - 1)`

4.. Compile and Run

- `win_bison -d -v arith_parser.y`
- `win_flex arith_lex.l`
- `gcc lex.yy.c arith_parser.tab.c -o arith.exe`
- `arith.exe < input.txt`

OUTPUT

RESULT

The program successfully recognizes and evaluates arithmetic expressions with `+`, `-`, `*`, and `/` operators.

EX NO: 3(B)	RECOGNIZE A VALID VARIABLE NAME

AIM

To write a LEX and YACC program that recognizes valid variable names in C language — variable names that start with a letter or underscore (_), followed by any combination of letters, digits, or underscores.

ALGORITHM

LEX Phase

- Identify and return tokens for identifiers matching the pattern: `[a-zA-Z_][a-zA-Z0-9_]*`
- Ignore whitespace and invalid tokens.

YACC Phase

- Define grammar rules that accept a valid identifier and reject invalid ones.
- Display whether the entered identifier is valid or not.

PROCEDURE / PROGRAM

1. Write the Lexical Analyzer `(var_lex.l)`

```
%{

#include "var_parser.tab.h"

#include <string.h>

%}

%%

[a-zA-Z_][a-zA-Z0-9_]*   { return IDENTIFIER; }

[ \t\n]+                 { /* ignore spaces */ }

.                        { return INVALID; }

%%

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

2. YACC Program (var_parser.y)

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

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

%token IDENTIFIER INVALID

%%

start:
      IDENTIFIER { printf("Valid variable name.\n"); }
    | INVALID    { printf("Invalid variable name.\n"); }
    ;

%%

int main() {
    printf("Enter a variable name: ");
    yyparse();
    return 0;
}

int yyerror(const char *s) {
    printf("Error: %s\n", s);
    return 0;
}
```

3.Compilation Steps

- `win_bison -d -v var_parser.y`
- `win_flex var_lex.l`
- `gcc lex.yy.c var_parser.tab.c -o variable.exe`
- `variable.exe`

OUTPUT

RESULT

The program successfully recognizes `valid and invalid variable names` as per C language naming conventions.

EX NO: 3(C)	RECOGNIZE CONTROL STRUCTURES

AIM

To write a LEX and YACC program that recognizes control structures such as: `if, else, for, while, do, switch`

ALGORITHM

LEX Phase

- Detect keywords like if, else, for, while, do, switch.
- Return a specific token for each keyword.
- Ignore whitespaces and other identifiers.

YACC Phase

- Define grammar rules to match these control structures.
- When a token is recognized, print which control structure it is.

PROCEDURE / PROGRAM

1. LEX Program (`control_lex.l`)

```
%{

#include "control_parser.tab.h"

%}

%%

"if"        { return IF; }

"else"      { return ELSE; }

"for"       { return FOR; }

"while"     { return WHILE; }

"do"        { return DO; }

"switch"    { return SWITCH; }
```

```
[ \t\n]+    { /* skip spaces */ }

.           { return OTHER; }

%%

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

2. YACC Program (control_parser.y)

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

int yylex(void);
int yyerror(const char *s);
%}
%token IF ELSE FOR WHILE DO SWITCH OTHER
%%
start:
      controls
    ;

controls:
      control
    | controls control
    ;

control:
      IF      { printf("Control Structure: IF\n"); }
    | ELSE    { printf("Control Structure: ELSE\n"); }
    | FOR     { printf("Control Structure: FOR\n"); }
    | WHILE   { printf("Control Structure: WHILE\n"); }
    | DO      { printf("Control Structure: DO\n"); }
    | SWITCH  { printf("Control Structure: SWITCH\n"); }
    | OTHER   { /* ignore non-control words */ }
    ;
```

```
%%

int main() {
    printf("Instructions:\n");
    printf("-------------------------------------------------\n");
    printf("1. Create a file named 'input.c' in the same folder.\n");
    printf("2. Write your control instrucations code inside 'input.c'.\n");
    printf("3. Save the file.\n");
    printf("4. Run this program using the command:\n");
    printf("       control.exe < input.c\n");
    printf("5. The program will read from 'input.c' and display the result.\n");
    printf("-------------------------------------------------\n");
    yyparse();
    return 0;
}

int yyerror(const char *s) {
    printf("Error: %s\n", s);
    return 0;
}
```

3. Input.txt

```
if (x > 0) {
    while (x < 10) {
        x++;
    }
} else {
    do {
        x--;
    } while (x > 0);
}
```

4. Compilation Steps

- `win_bison -d -v control_parser.y`
- `win_flex control_lex.l`
- `gcc lex.yy.c control_parser.tab.c -o control.exe`
- `control.exe`

OUTPUT

RESULT

The LEX and YACC program successfully recognizes and displays `control structures` present in the given C code.

EX NO: 3(D)	IMPLEMENTATION OF CALCULATOR USING LEX TOOL (WIN_FLEX) AND YACC (WIN_BISON)

AIM

To develop a calculator using LEX and YACC that can:

- Recognize and evaluate arithmetic expressions.
- Handle variable assignments and retrievals.
- Recognize basic C control structures syntax.

ALGORITHM

Lexical Analysis (LEX)

- Identify keywords `(if, while, for, return).`
- Recognize identifiers, numbers (integer and float), operators, and special symbols.
- Store identifiers in a symbol table for later reference.

Parsing and Evaluation (YACC)

- Define grammar for expressions and statements.
- For arithmetic expressions, recursively evaluate using rules for `+, -, *, /.`
- For assignment statements, store values in the symbol table.
- Recognize control structures syntax (if, else, while, for).

Execution

- Read input C-like code.
- Tokenize using the lexical analyzer.
- Parse using the grammar rules, evaluate expressions, and update the symbol table.
- Print expression results and display the symbol table.

PROGRAMS

lexical3.l

```
%{
#include "parser.tab.h"
#include <stdlib.h>
#include <string.h>
%}
%%
"if"            { return IF; }
"else"          { return ELSE; }
"while"         { return WHILE; }
"for"           { return FOR; }
"return"        { return RETURN; }

[0-9]+\.[0-9]+  { yylval.fval = atof(yytext); return FLOAT_CONST; }
[0-9]+          { yylval.fval = atoi(yytext); return INT_CONST; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.sval = strdup(yytext); return ID; }
"="             { return ASSIGN; }
"+"             { return PLUS; }
"-"             { return MINUS; }
"*"             { return MUL; }
"/"             { return DIV; }
"("             { return LPAREN; }
")"             { return RPAREN; }
"{"             { return LBRACE; }
"}"             { return RBRACE; }
";"             { return SEMICOLON; }
","             { return COMMA; }
"=="            { return EQ; }
"!="            { return NE; }
">"             { return GT; }
"<"             { return LT; }
">="            { return GE; }
"<="            { return LE; }

[ \t\n]+        ;   // skip whitespace
.               { return yytext[0]; }
%%
int yywrap(void) { return 1; }
```

symbol_table.h

```
#ifndef SYMBOL_TABLE_H
#define SYMBOL_TABLE_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Var {
    char *name;
    float value;
    struct Var *next;
} Var;

Var *symbol_table = NULL;
// Lookup a variable
Var* lookup_var(const char *name) {
    Var *current = symbol_table;
    while (current != NULL) {
        if (strcmp(current->name, name) == 0) return current;
        current = current->next;
    }
    return NULL;
}
void set_var(const char *name, float value) {
    Var *v = lookup_var(name);
    if (v) {
        v->value = value;
    } else {
        v = (Var*)malloc(sizeof(Var));
        v->name = strdup(name);
        v->value = value;
        v->next = symbol_table;
        symbol_table = v;
    }
}
```

```
float get_var_value(const char *name) {
    Var *v = lookup_var(name);
    if (v) return v->value;
    printf("Warning: variable %s not initialized. Using 0.\n", name);
    return 0.0;
}

#endif
```

parser.y

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

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

%union {
    float fval;
    char *sval;
}

%token <fval> INT_CONST FLOAT_CONST
%token <sval> ID
%token IF ELSE WHILE FOR RETURN
%token ASSIGN PLUS MINUS MUL DIV LPAREN RPAREN LBRACE RBRACE SEMICOLON COMMA
%token EQ NE GT LT GE LE

%type <fval> expr statement
%left PLUS MINUS
```

```
%left MUL DIV

%%

program:
    /* empty */
  | program statement
  ;

statement:
      expr SEMICOLON                     { printf("Result: %f\n", $1); }
    | ID ASSIGN expr SEMICOLON           { set_var($1, $3); free($1); }
    | IF LPAREN expr RPAREN statement_block { if($3) ; }
    | IF LPAREN expr RPAREN statement_block ELSE statement_block { if($3) ; else ; }
    | WHILE LPAREN expr RPAREN statement_block { while($3) ; }
    | FOR LPAREN statement expr SEMICOLON expr RPAREN statement_block { ; }
    ;

statement_block:
      LBRACE program RBRACE
    | statement
    ;

expr:
      expr PLUS expr      { $$ = $1 + $3; }
    | expr MINUS expr     { $$ = $1 - $3; }
    | expr MUL expr       { $$ = $1 * $3; }
    | expr DIV expr       { $$ = $1 / $3; }
    | expr GT expr       { $$ = $1 > $3; }
    | expr LT expr       { $$ = $1 < $3; }
    | expr GE expr       { $$ = $1 >= $3; }
    | expr LE expr       { $$ = $1 <= $3; }
    | expr EQ expr       { $$ = $1 == $3; }
    | expr NE expr       { $$ = $1 != $3; }

    | LPAREN expr RPAREN  { $$ = $2; }
```

```
    | ID                  { $$ = get_var_value($1); free($1); }
    | INT_CONST           { $$ = $1; }
    | FLOAT_CONST         { $$ = $1; }
    ;


%%

int main(void) {
    printf("Instructions:\n");
    printf("-------------------------------------------------\n");
    printf("1. Create a file named 'input.c' in the same folder.\n");
    printf("2. Write your control instrucations code inside 'input.c'.\n");
    printf("3. Save the file.\n");
    printf("4. Run this program using the command:\n");
    printf("       control.exe < input.c\n");
    printf("5. The program will read from 'input.c' and display the result.\n");
    printf("-------------------------------------------------\n");

    return yyparse();
}

int yyerror(const char *s) {
    fprintf(stderr, "Syntax Error: %s\n", s);
    return 0;
}
```

input.c

```
x = 10;
y = 5 * 2;
x + y;
if (x > 5) { y = y + 1; }
y;
```

PROCEDURE

1. Write the lexical analyzer `(lexical3.l)` with:
   - Rules for keywords, identifiers, numbers, operators, relational operators, and special symbols.
   - A symbol table to store identifiers.
2. Write the parser `(parser.y)` with:
   - Grammar rules for arithmetic expressions.
   - Grammar rules for assignment and control statements.
   - Evaluation actions for arithmetic expressions.
   - Functions to insert, update, and retrieve variables from the symbol table.
3. Include the symbol table header `(symbol_table.h)` to manage identifiers and values.
4. Compile using `WinBison` and `WinFlex`:
   - `win_bison -d -v parser.y`
   - `win_flex lexical3.l`
   - `gcc lex.yy.c parser.tab.c -o calc.exe`
5. Run the executable and input C-like statements for evaluation:

- `calc.exe < input.c`

OUTPUT

RESULT

Thus, the calculator using LEX and YACC is implemented successfully.

It evaluates arithmetic expressions, handles variable assignments, and recognizes control structures

EX NO: 4	GENERATE THREE ADDRESS CODE FOR A SIMPLE PROGRAM USING LEX AND YACC.

AIM

To develop a program using LEX and YACC that generates `Three Address Code (TAC)` for simple C-like statements such as arithmetic expressions, assignments, and conditional (if / if-else) statements.

ALGORITHM

1. **Start** the program and define tokens for identifiers, numbers, operators, and keywords using **LEX**.
2. **Use YACC** to define grammar rules for:
   - Arithmetic expressions
   - Assignment statements
   - Conditional (if and if-else) statements
3. For each grammar reduction, generate appropriate TAC instructions.
4. Maintain counters for temporary variables (t1, t2, …) and labels (L1, L2, …).
5. Print TAC code sequentially as parsing progresses.
6. **End** after successfully generating all intermediate code.

PROGRAM

tacgen.l

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

```
%%
"if"        { return IF; }
"else"      { return ELSE; }
[0-9]+      { yylval.str = strdup(yytext); return NUM; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); return ID; }
"=="        { return EQ; }
"!="        { return NE; }
">="        { return GE; }
"<="        { return LE; }
">"         { return GT; }
"<"         { return LT; }
"="         { return ASSIGN; }
"+"         { return PLUS; }
"-"         { return MINUS; }
"*"         { return MUL; }
"/"         { return DIV; }
"("         { return LPAREN; }
")"         { return RPAREN; }
";"         { return SEMI; }
[ \t\n]+    ;  // ignore whitespace
.           ;
%%

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

tacgen.y

```
%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int tempCount = 1, labelCount = 1;
```

```
char* newTemp();
char* newLabel();

void emit(const char *s);

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

%union {
    char *str;
}

%token <str> ID NUM
%token IF ELSE
%token ASSIGN PLUS MINUS MUL DIV
%token EQ NE LT GT LE GE
%token LPAREN RPAREN SEMI
%type <str> expr stmt condition

%left PLUS MINUS
%left MUL DIV
%left EQ NE LT GT LE GE
%%
program:
    /* empty */
  | program stmt
  ;
stmt:
    ID ASSIGN expr SEMI
        {
            printf("%s = %s\n", $1, $3);
        }
```

```
  | IF LPAREN condition RPAREN stmt
        {
            char *L1 = newLabel(), *L2 = newLabel();
            printf("if %s goto %s\n", $3, L1);
            printf("goto %s\n", L2);
            printf("%s:\n", L1);
            // then-part already printed
            printf("%s:\n", L2);
        }
  | IF LPAREN condition RPAREN stmt ELSE stmt
        {
            char *L1 = newLabel(), *L2 = newLabel(), *L3 = newLabel();
            printf("if %s goto %s\n", $3, L1);
            printf("goto %s\n", L2);
            printf("%s:\n", L1);
            // then-part
            printf("%s:\n", L2);
            // else-part
            printf("%s:\n", L3);
        }
  ;

condition:
    expr LT expr    { $$ = newTemp(); printf("%s = %s < %s\n", $$, $1, $3); }
  | expr GT expr    { $$ = newTemp(); printf("%s = %s > %s\n", $$, $1, $3); }
  | expr LE expr    { $$ = newTemp(); printf("%s = %s <= %s\n", $$, $1, $3); }
  | expr GE expr    { $$ = newTemp(); printf("%s = %s >= %s\n", $$, $1, $3); }
  | expr EQ expr    { $$ = newTemp(); printf("%s = %s == %s\n", $$, $1, $3); }
  | expr NE expr    { $$ = newTemp(); printf("%s = %s != %s\n", $$, $1, $3); }
  ;
expr:
    expr PLUS expr  { $$ = newTemp(); printf("%s = %s + %s\n", $$, $1, $3); }
  | expr MINUS expr { $$ = newTemp(); printf("%s = %s - %s\n", $$, $1, $3); }
  | expr MUL expr   { $$ = newTemp(); printf("%s = %s * %s\n", $$, $1, $3); }
```

```
  | expr DIV expr   { $$ = newTemp(); printf("%s = %s / %s\n", $$, $1, $3); }
  | LPAREN expr RPAREN { $$ = $2; }
  | ID              { $$ = $1; }
  | NUM             { $$ = $1; }
  ;
%%
char* newTemp() {
    char *buf = malloc(8);
    sprintf(buf, "t%d", tempCount++);
    return buf;
}

char* newLabel() {
    char *buf = malloc(8);
    sprintf(buf, "L%d", labelCount++);
    return buf;
}

int yyerror(const char *s) {
    printf("Error: %s\n", s);
    return 0;
}

int main() {
    printf("Instructions:\n");
    printf("--------------------------------------------------\n");
    printf("1. Create a file named 'input.c' in the same folder.\n");
    printf("2. Write your instruction code inside 'input.c'.\n");
    printf("3. Save the file.\n");
    printf("4. Run this program using the command:\n");
    printf("       tacgen.exe < input.c\n");
    printf("5. The program will read from 'input.c' and display the result.\n");
    printf("--------------------------------------------------\n");
    yyparse();
    return 0;
}
```

Input.txt

```
x = a + b * c;

if (x < 10) y = x + 1;

else y = x - 1;
```

PROCEDURE

1. Create the LEX and YACC files (`tacgen.l`, `tacgen.y`).
2. Run the following commands in terminal:
   - `win_bison -d -v tacgen.y`
   - `win_flex tacgen.l`
   - `gcc lex.yy.c tacgen.tab.c -o tacgen.exe`
   - `tacgen.exe < input.txt`
3. Enter test code in C-like syntax.
4. Observe the generated three address code printed as output.

OUTPUT

RESULT

Thus, a `LEX and YACC-based program` was successfully developed to generate `Three Address Code` for simple C-like statements including arithmetic, assignment, and conditional expressions.

EX NO: 5	IMPLEMENTATION OF TYPE CHECKING USING LEX AND YACC

AIM

To design and implement a type checking program using Lex and Yacc that verifies type compatibility in variable declarations, assignments, and arithmetic expressions.

ALGORITHM

1. **Start** the program and initialize a symbol table to store variable names and their types.
2. **Lexical Analyzer (Lex):**
   - Recognize tokens such as keywords (int, float), identifiers, operators, numbers, and special symbols.
   - Return token types to Yacc for syntactic and semantic analysis.
3. **Syntax Analyzer (Yacc):**
   - Define grammar rules for declarations, assignments, and expressions.
   - On encountering a declaration:
     - Insert the variable name and its data type into the symbol table.
   - On encountering an assignment:
     - Retrieve variable types from the symbol table.
     - Check if both sides of the assignment are type-compatible.
     - If not, report a **type mismatch error**.
4. **Expression Evaluation:**
   - Determine the result type of arithmetic expressions based on operand types (e.g., int + float → float).
   - Propagate type errors if any operand is undeclared or incompatible.
5. **Display** meaningful messages for valid and invalid type usages.
6. **Stop.**

PROGRAM:

typecheck.l

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

%%
"int"      { return INT; }
"float"    { return FLOAT; }
[0-9]+"."[0-9]+   { yylval.fval = atof(yytext); return FLOAT_NUM; }
[0-9]+            { yylval.ival = atoi(yytext); return INT_NUM; }
[a-zA-Z_][a-zA-Z0-9_]*  { yylval.str = strdup(yytext); return ID; }
"="        { return ASSIGN; }
";"        { return SEMI; }
[+\-*/]    { return OP; }
[ \t\n]+   { /* skip whitespace */ }
.          { /* ignore invalid characters */ }
%%
int yywrap() { return 1; }
```

typecheck.y

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int yylex(void);
int yyerror(const char *s);
struct symtab {
    char name[20];
    char type[10];
} table[50];
```

```
int count = 0;

int lookup(char *name) {
    for (int i = 0; i < count; i++)
        if (strcmp(table[i].name, name) == 0)
            return i;
    return -1;
}

void insert(char *name, char *type) {
    if (lookup(name) == -1) {
        strcpy(table[count].name, name);
        strcpy(table[count].type, type);
        count++;
    } else {
        printf("Error: variable '%s' redeclared.\n", name);
    }
}

char* getType(char *t1, char *t2) {
    if (strcmp(t1, t2) == 0) return t1;
    if ((strcmp(t1, "float") == 0 && strcmp(t2, "int") == 0) ||
        (strcmp(t1, "int") == 0 && strcmp(t2, "float") == 0))
        return "float";
    return "error";
}
%}

%union {
    int ival;
    float fval;
    char *str;
}
```

```
%token <str> ID
%token <str> INT FLOAT
%token <ival> INT_NUM
%token <fval> FLOAT_NUM
%token ASSIGN SEMI OP

%type <str> expr type

%%

program : program stmt
        | stmt
        ;

stmt : type ID SEMI
        { insert($2, $1); }

     | ID ASSIGN expr SEMI
        {
          int pos = lookup($1);
          if (pos == -1)
              printf("Error: variable '%s' not declared.\n", $1);
          else {
              char *result = getType(table[pos].type, $3);
              if (strcmp(result, "error") == 0)
                  printf("Type Error: cannot assign %s to %s.\n", $3, table[pos].type);
              else
                  printf("Assignment valid: %s = %s\n", $1, $3);
          }
        }
     ;
```

```
type : INT   { $$ = "int"; }
     | FLOAT { $$ = "float"; }
     ;

expr : expr OP expr
        {
           $$ = getType($1, $3);
           if (strcmp($$, "error") == 0)
               printf("Type Error in expression.\n");
        }

     | ID
        {
           int pos = lookup($1);
           if (pos == -1) {
               printf("Error: variable '%s' undeclared.\n", $1);
               $$ = "error";
           } else
               $$ = table[pos].type;
        }

     | INT_NUM   { $$ = "int"; }
     | FLOAT_NUM { $$ = "float"; }
     ;

%%

int main() {
    printf("Enter declarations and expressions:\n");
    yyparse();
    return 0;
}
```

```
int yyerror(const char *s) {
    fprintf(stderr, "Syntax Error: %s\n", s);
    return 0;
}
```

input.c

```
int a;

float b;

a = 5;

b = a + 3.5;

a = b + 2;

c = a + 1;
```

PROCEDURE:

1. Open terminal or command prompt in the directory containing both files.
2. Run the following commands:
   - `win_bison -d -v typecheck.y`
   - `win_flex typecheck.l`
   - `gcc lex.yy.c typecheck.tab.c -o typecheck.exe`
3. Enter declarations and expressions as input.
4. Observe whether type checking passes or fails for each statement.

OUTPUT

RESULT

The program to perform `type checking using Lex and Yacc` was successfully implemented.

EX NO: 6	IMPLEMENT SIMPLE CODE OPTIMIZATION TECHNIQUES

AIM

To implement simple code optimization techniques such as `Constant Folding, Strength Reduction`, and `Algebraic Transformation` using LEX and YACC.

ALGORITHM

Algorithm:

1. **Start** the program and define the grammar using YACC to recognize arithmetic expressions.
2. **Use LEX** to tokenize identifiers, numbers, and operators (+, -, *, /, =).
3. **Build an expression tree** where each node represents an operation or operand.
4. **Apply constant folding:**
    - Evaluate expressions where both operands are constants (e.g., 2 + 3 → 5).
5. **Apply strength reduction:**
    - Replace costly operations with cheaper equivalents (e.g., x * 2 → x << 1).
6. **Apply algebraic transformation:**
    - Simplify expressions using rules like

        x + 0 = x

        x * 1 = x

        x * 0 = 0
7. **Generate optimized expression** and display both **original** and **optimized** results.
8. **Stop**.

PROGRAM

optimizer_opt.l

```
%{
#include "optimizer_opt.tab.h"
#include <string.h>
#include <stdlib.h>

typedef struct Node Node;
%}

%%

[0-9]+                  { yylval.num = atoi(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]*  { strcpy(yylval.id, yytext); return ID; }

"="     { return ASSIGN; }
"+"     { return PLUS; }
"-"     { return MINUS; }
"*"     { return MUL; }
"/"     { return DIV; }
";"     { return SEMI; }

[ \t\n]+    ;  // Ignore whitespace
.           { return yytext[0]; }

%%

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

optimizer_opt.y

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

// Forward declarations
typedef struct Node Node;
Node* makeNode(char op, Node* left, Node* right);
Node* makeLeaf(int value);
Node* makeVariable(char* id);
void optimize(Node* node);
void printOptimized(Node* node);
void freeTree(Node* node);
int isPowerOfTwo(int n);

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

%union {
    int num;
    char id[32];
    struct Node* node;
}

%token <num> NUMBER
%token <id> ID
%token ASSIGN PLUS MINUS MUL DIV SEMI
%type <node> expr

%left PLUS MINUS
%left MUL DIV

%%
```

```
program:
    statement
;

statement:
    ID ASSIGN expr SEMI
    {
        printf("\n=== Optimization Results ===\n");
        printf("Original Code:   %s = ", $1);
        printOptimized($3);
        printf(";\n");

        optimize($3);

        printf("Optimized Code:  %s = ", $1);
        printOptimized($3);
        printf(";\n");

        freeTree($3);
    }
;

expr:
      expr PLUS expr   { $$ = makeNode('+', $1, $3); }
    | expr MINUS expr  { $$ = makeNode('-', $1, $3); }
    | expr MUL expr    { $$ = makeNode('*', $1, $3); }
    | expr DIV expr    { $$ = makeNode('/', $1, $3); }
    | NUMBER           { $$ = makeLeaf($1); }
    | ID               { $$ = makeVariable($1); }  // ADDED: Variable support
;

%%

// ---------- Node structure definition ----------
struct Node {
    char op;
    int value;
```

```
    char id[32];
    int is_var;  // ADDED: Flag to indicate if this is a variable
    struct Node *left, *right;
};

// ---------- Utility Functions ----------
int isPowerOfTwo(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}

int logBaseTwo(int n) {
    int log = 0;
    while (n > 1) {
        n >>= 1;
        log++;
    }
    return log;
}

// ---------- Node creation ----------
Node* makeNode(char op, Node* left, Node* right) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->op = op;
    node->left = left;
    node->right = right;
    node->id[0] = '\0';
    node->value = 0;
    node->is_var = 0;
    return node;
}

Node* makeLeaf(int value) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->op = '\0';
    node->left = node->right = NULL;
    node->value = value;
    node->id[0] = '\0';
```

```
    node->is_var = 0;
    return node;
}

// ADDED: Function to create variable nodes
Node* makeVariable(char* id) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->op = '\0';
    node->left = node->right = NULL;
    node->value = 0;
    strcpy(node->id, id);
    node->is_var = 1;  // Mark as variable
    return node;
}

// ---------- Optimization ----------
void optimize(Node* node) {
    if (!node) return;

    // Optimize children first (post-order traversal)
    if (node->left) optimize(node->left);
    if (node->right) optimize(node->right);

    // Skip if already a leaf node (constant or variable)
    if (node->op == '\0') return;

    // TECHNIQUE 1: CONSTANT FOLDING
    if (node->left && node->left->op == '\0' && !node->left->is_var &&
        node->right && node->right->op == '\0' && !node->right->is_var) {

        int left_val = node->left->value;
        int right_val = node->right->value;
        int result = 0;
        int valid = 1;

        switch (node->op) {
            case '+':
```

```
            result = left_val + right_val;
            printf("// Constant folding: %d + %d = %d\n", left_val, right_val, result);
            break;
        case '-':
            result = left_val - right_val;
            printf("// Constant folding: %d - %d = %d\n", left_val, right_val, result);
            break;
        case '*':
            result = left_val * right_val;
            printf("// Constant folding: %d * %d = %d\n", left_val, right_val, result);
            break;
        case '/':
            if (right_val != 0) {
                result = left_val / right_val;
                printf("// Constant folding: %d / %d = %d\n", left_val, right_val, result);
            } else {
                printf("// Warning: Division by zero! Using 0 as result.\n");
                result = 0;
            }
            break;
        default:
            valid = 0;
    }

    if (valid) {
        // Convert this node to a leaf
        node->op = '\0';
        node->value = result;
        node->is_var = 0;
        free(node->left);
        free(node->right);
        node->left = node->right = NULL;
        return;
    }
}

// TECHNIQUE 2: STRENGTH REDUCTION
```

```
if (node->op == '*') {
    // Multiplication by powers of 2 (right side)
    if (node->right && node->right->op == '\0' && !node->right->is_var &&
        isPowerOfTwo(node->right->value)) {
        int power = node->right->value;
        int shift = logBaseTwo(power);
        printf("// Strength reduction: *%d -> <<%d\n", power, shift);
    }
    // Multiplication by powers of 2 (left side)
    else if (node->left && node->left->op == '\0' && !node->left->is_var &&
             isPowerOfTwo(node->left->value)) {
        int power = node->left->value;
        int shift = logBaseTwo(power);
        printf("// Strength reduction: %d* -> <<%d\n", power, shift);
    }

    // Multiplication by 2 (with variable on left)
    if (node->left && node->left->is_var &&
        node->right && node->right->op == '\0' && !node->right->is_var &&
        node->right->value == 2) {
        printf("// Strength reduction: %s * 2 -> %s + %s\n",
               node->left->id, node->left->id, node->left->id);
    }
    // Multiplication by 2 (with variable on right)
    else if (node->right && node->right->is_var &&
             node->left && node->left->op == '\0' && !node->left->is_var &&
             node->left->value == 2) {
        printf("// Strength reduction: 2 * %s -> %s + %s\n",
               node->right->id, node->right->id, node->right->id);
    }
}
else if (node->op == '/') {
    // Division by powers of 2
    if (node->right && node->right->op == '\0' && !node->right->is_var &&
        isPowerOfTwo(node->right->value)) {
        int power = node->right->value;
        int shift = logBaseTwo(power);
```

```
            printf("// Strength reduction: /%d -> >>%d\n", power, shift);
        }
    }

    // TECHNIQUE 3: ALGEBRAIC TRANSFORMATIONS
    if (node->op == '+') {
        // x + 0 = x (0 on right)
        if (node->right && node->right->op == '\0' && !node->right->is_var &&
            node->right->value == 0) {
            printf("// Algebraic simplification: x + 0 -> x\n");
            Node* temp = node->left;
            *node = *temp;
            free(temp);
            return;
        }
        // 0 + x = x (0 on left)
        else if (node->left && node->left->op == '\0' && !node->left->is_var &&
                 node->left->value == 0) {
            printf("// Algebraic simplification: 0 + x -> x\n");
            Node* temp = node->right;
            *node = *temp;
            free(temp);
            return;
        }
    }
    else if (node->op == '-') {
        // x - 0 = x
        if (node->right && node->right->op == '\0' && !node->right->is_var &&
            node->right->value == 0) {
            printf("// Algebraic simplification: x - 0 -> x\n");
            Node* temp = node->left;
            *node = *temp;
            free(temp);
            return;
        }
        // x - x = 0 (same variable on both sides)
        else if (node->left && node->left->is_var &&
```

```c
            node->right && node->right->is_var &&
            strcmp(node->left->id, node->right->id) == 0) {
        printf("// Algebraic simplification: %s - %s -> 0\n",
               node->left->id, node->right->id);
        node->op = '\0';
        node->value = 0;
        node->is_var = 0;
        free(node->left);
        free(node->right);
        node->left = node->right = NULL;
        return;
    }
}
else if (node->op == '*') {
    // x * 0 = 0 (0 on left)
    if (node->left && node->left->op == '\0' && !node->left->is_var &&
        node->left->value == 0) {
        printf("// Algebraic simplification: 0 * x -> 0\n");
        node->op = '\0';
        node->value = 0;
        node->is_var = 0;
        free(node->left);
        free(node->right);
        node->left = node->right = NULL;
        return;
    }
    // x * 0 = 0 (0 on right)
    else if (node->right && node->right->op == '\0' && !node->right->is_var &&
             node->right->value == 0) {
        printf("// Algebraic simplification: x * 0 -> 0\n");
        node->op = '\0';
        node->value = 0;
        node->is_var = 0;
        free(node->left);
        free(node->right);
        node->left = node->right = NULL;
        return;
```

```
    }
    // x * 1 = x (1 on right)
    else if (node->right && node->right->op == '\0' && !node->right->is_var &&
             node->right->value == 1) {
        printf("// Algebraic simplification: x * 1 -> x\n");
        Node* temp = node->left;
        *node = *temp;
        free(temp);
        return;
    }
    // 1 * x = x (1 on left)
    else if (node->left && node->left->op == '\0' && !node->left->is_var &&
             node->left->value == 1) {
        printf("// Algebraic simplification: 1 * x -> x\n");
        Node* temp = node->right;
        *node = *temp;
        free(temp);
        return;
    }
}
else if (node->op == '/') {
    // 0 / x = 0 (x ≠ 0)
    if (node->left && node->left->op == '\0' && !node->left->is_var &&
        node->left->value == 0) {
        printf("// Algebraic simplification: 0 / x -> 0\n");
        node->op = '\0';
        node->value = 0;
        node->is_var = 0;
        free(node->left);
        free(node->right);
        node->left = node->right = NULL;
        return;
    }
    // x / 1 = x
    else if (node->right && node->right->op == '\0' && !node->right->is_var &&
             node->right->value == 1) {
        printf("// Algebraic simplification: x / 1 -> x\n");
```

```
            Node* temp = node->left;
            *node = *temp;
            free(temp);
            return;
        }
        // x / x = 1 (same variable on both sides)
        else if (node->left && node->left->is_var &&
                 node->right && node->right->is_var &&
                 strcmp(node->left->id, node->right->id) == 0) {
            printf("// Algebraic simplification: %s / %s -> 1\n",
                   node->left->id, node->right->id);
            node->op = '\0';
            node->value = 1;
            node->is_var = 0;
            free(node->left);
            free(node->right);
            node->left = node->right = NULL;
            return;
        }
    }
}

// ---------- Print ----------
void printOptimized(Node* node) {
    if (!node) return;
    if (node->op == '\0') {
        if (node->is_var) {
            printf("%s", node->id);
        } else {
            printf("%d", node->value);
        }
    } else {
        printf("(");
        printOptimized(node->left);
        printf(" %c ", node->op);
        printOptimized(node->right);
        printf(")");
```

```
    }
}

// ---------- Memory cleanup ----------
void freeTree(Node* node) {
    if (!node) return;
    freeTree(node->left);
    freeTree(node->right);
    free(node);
}

int main() {
    printf("=== Advanced Expression Optimizer ===\n");
    printf("Supports: Constant Folding, Strength Reduction, Algebraic Transformations\n");
    printf("Enter assignment statements (e.g., x = 2 + 3 * 4; or y = a * 8;)\n");
    printf("Press Ctrl+C to exit\n\n");

    while(1) {
        printf("> ");
        if (yyparse() != 0) {
            break; // Exit on parsing error
        }
    }

    return 0;
}

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

node.h:

```c
#ifndef NODE_H
#define NODE_H

// Simple node structure for expression tree
typedef struct Node {
    char op;
    int value;
    char id[32];
    struct Node *left, *right;
} Node;

// Function declarations
Node* makeNode(char op, Node* left, Node* right);
Node* makeLeaf(int value);
void optimize(Node* node);
void printOptimized(Node* node);
void freeTree(Node* node);

#endif
```

PROCEDURE

1. Write the Lex specification `(optimizer_opt.l)`.
2. Write the YACC grammar `(optimizer_opt.y)`.
3. Run commands:
   - `win_bison -d -v optimizer_opt.y`
   - `win_flex optimizer_opt.l`
   - `gcc optimizer_opt.tab.c lex.yy.c -o optimizer_opt.exe`
   - `optimizer_opt.exe`
4. Give input expression when prompted.
5. Observe optimization steps in output.

OUTPUT

RESULT

The Successfully implemented basic `code optimization techniques` like `Constant Folding, Strength Reduction`, and `Algebraic Transformation` using `LEX` and `YACC`

EX NO: 7	IMPLEMENT BACK-END OF THE COMPILER FOR WHICH THE THREE ADDRESS CODE IS GIVEN AS INPUT AND THE 8086 ASSEMBLY LANGUAGE CODE IS PRODUCED AS OUTPUT.

AIM

To design and implement the back-end of a compiler that takes `Three Address Code (TAC)` as input and generates equivalent `8086 assembly language code` as output.

ALGORITHM

Main Algorithm:

1. **Start**
2. **Initialize** symbol table and TAC storage
3. **Read** Three Address Code statements sequentially
4. **Parse** each TAC statement into its components (result, arg1, op, arg2)
5. **Build Symbol Table** by extracting all variables and temporaries
6. **Generate Data Segment** with variable declarations
7. **Generate Code Segment** with program initialization
8. **For each TAC instruction:**
    - **If** assignment operation → Generate MOV instructions
    - **If** arithmetic operation → Generate appropriate arithmetic instructions
    - **If** conditional operation → Generate comparison and jump instructions
9. **Generate** program termination code
10. **Output** complete 8086 assembly program
11. **Stop**

PROGRAM:

tac_to_8086.l:

```
%{
#include "tac_to_8086.tab.h"
#include <string.h>
#include <ctype.h>
%}
%%
[a-zA-Z_][a-zA-Z0-9_]*  {
    strcpy(yylval.strval, yytext);
    return ID;
}
[0-9]+                  {
    strcpy(yylval.strval, yytext);
    return NUMBER;
}
"+"|"-"|"*"|"/"         {
    strcpy(yylval.strval, yytext);
    return OPERATOR;
}
"=="|"!="|"<"|">"|"<="|">=" {
    strcpy(yylval.strval, yytext);
    return RELOP;
}
"="                     {
    strcpy(yylval.strval, yytext);
    return ASSIGNOP;
}
\n                      { return NEWLINE; }
[ \t]+                  ; /* Skip whitespace */
.                       { return yytext[0]; }
```

```
%%

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

tac_to_8086.y:

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

#define MAX_TAC 100
#define SYMBOL_TABLE_SIZE 50

// Structure for Three Address Code
typedef struct {
    char result[20];
    char arg1[20];
    char arg2[20];
    char op[10];
} TAC;

// Structure for Symbol Table
typedef struct {
    char name[20];
    int offset;
} Symbol;

TAC tac_codes[MAX_TAC];
Symbol symbol_table[SYMBOL_TABLE_SIZE];
int tac_count = 0;
int symbol_count = 0;
int current_offset = 0;

void add_symbol(char* name);
```

```
int find_symbol(char* name);
void generate_8086_code();
void process_assignment(int index);
void process_arithmetic(int index);
void process_conditional(int index);

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

%union {
    char strval[50];
}

%token <strval> ID NUMBER OPERATOR ASSIGNOP RELOP
%token NEWLINE

%%

program:
    lines
;

lines:
    lines line
    | line
;

line:
    assignment NEWLINE
    | arithmetic NEWLINE
    | conditional NEWLINE
    | NEWLINE
;

assignment:
    ID ASSIGNOP ID {
        strcpy(tac_codes[tac_count].result, $1);
        strcpy(tac_codes[tac_count].arg1, $3);
```

```
        strcpy(tac_codes[tac_count].op, "=");
        tac_count++;

        add_symbol($1);
        add_symbol($3);
    }
    | ID ASSIGNOP NUMBER {
        strcpy(tac_codes[tac_count].result, $1);
        strcpy(tac_codes[tac_count].arg1, $3);
        strcpy(tac_codes[tac_count].op, "=");
        tac_count++;

        add_symbol($1);
    }
;

arithmetic:
    ID ASSIGNOP ID OPERATOR ID {
        strcpy(tac_codes[tac_count].result, $1);
        strcpy(tac_codes[tac_count].arg1, $3);
        strcpy(tac_codes[tac_count].arg2, $5);
        strcpy(tac_codes[tac_count].op, $4);
        tac_count++;

        add_symbol($1);
        add_symbol($3);
        add_symbol($5);
    }
    | ID ASSIGNOP ID OPERATOR NUMBER {
        strcpy(tac_codes[tac_count].result, $1);
        strcpy(tac_codes[tac_count].arg1, $3);
        strcpy(tac_codes[tac_count].arg2, $5);
        strcpy(tac_codes[tac_count].op, $4);
        tac_count++;

        add_symbol($1);
        add_symbol($3);
    }
    | ID ASSIGNOP NUMBER OPERATOR ID {
```

```
        strcpy(tac_codes[tac_count].result, $1);
        strcpy(tac_codes[tac_count].arg1, $3);
        strcpy(tac_codes[tac_count].arg2, $5);
        strcpy(tac_codes[tac_count].op, $4);
        tac_count++;

        add_symbol($1);
        add_symbol($5);
    }
;

conditional:
    ID RELOP ID {
        strcpy(tac_codes[tac_count].result, "IF");
        strcpy(tac_codes[tac_count].arg1, $1);
        strcpy(tac_codes[tac_count].arg2, $3);
        strcpy(tac_codes[tac_count].op, $2);
        tac_count++;

        add_symbol($1);
        add_symbol($3);
    }
;

%%

// Add symbol to symbol table if not exists
void add_symbol(char* name) {
    // Skip numbers and temporary variables starting with 't'
    if (name[0] >= '0' && name[0] <= '9') return;

    for (int i = 0; i < symbol_count; i++) {
        if (strcmp(symbol_table[i].name, name) == 0) {
            return;
        }
    }

    if (symbol_count < SYMBOL_TABLE_SIZE) {
        strcpy(symbol_table[symbol_count].name, name);
```

```
        symbol_table[symbol_count].offset = current_offset;
        current_offset += 2; // Each variable takes 2 bytes in 8086
        symbol_count++;
    }
}

// Find symbol in symbol table
int find_symbol(char* name) {
    for (int i = 0; i < symbol_count; i++) {
        if (strcmp(symbol_table[i].name, name) == 0) {
            return symbol_table[i].offset;
        }
    }
    return -1;
}

// Generate 8086 Assembly Code
void generate_8086_code() {
    printf("\n=== GENERATED 8086 ASSEMBLY CODE ===\n\n");

    // Data Segment
    printf("DATA SEGMENT\n");
    for (int i = 0; i < symbol_count; i++) {
        printf("    %s DW ?\n", symbol_table[i].name);
    }
    printf("DATA ENDS\n\n");

    // Code Segment
    printf("CODE SEGMENT\n");
    printf("    ASSUME CS:CODE, DS:DATA\n");
    printf("START:\n");
    printf("    MOV AX, DATA\n");
    printf("    MOV DS, AX\n\n");

    // Process each TAC instruction
    for (int i = 0; i < tac_count; i++) {
        printf("; TAC: %s = %s %s %s\n",
               tac_codes[i].result, tac_codes[i].arg1,
               tac_codes[i].op, tac_codes[i].arg2);
```

```
        if (strcmp(tac_codes[i].op, "=") == 0) {
            process_assignment(i);
        }
        else if (strcmp(tac_codes[i].result, "IF") == 0) {
            process_conditional(i);
        }
        else {
            process_arithmetic(i);
        }
        printf("\n");
    }

    // Program end
    printf("    MOV AH, 4CH\n");
    printf("    INT 21H\n");
    printf("CODE ENDS\n");
    printf("END START\n");
}

void process_assignment(int index) {
    char* result = tac_codes[index].result;
    char* arg1 = tac_codes[index].arg1;

    // Check if arg1 is a number
    if (arg1[0] >= '0' && arg1[0] <= '9') {
        printf("    MOV %s, %s\n", result, arg1);
    } else {
        printf("    MOV AX, %s\n", arg1);
        printf("    MOV %s, AX\n", result);
    }
}

void process_arithmetic(int index) {
    char* result = tac_codes[index].result;
    char* arg1 = tac_codes[index].arg1;
    char* arg2 = tac_codes[index].arg2;
    char* op = tac_codes[index].op;
```

```
// Load first operand
if (arg1[0] >= '0' && arg1[0] <= '9') {
    printf("    MOV AX, %s\n", arg1);
} else {
    printf("    MOV AX, %s\n", arg1);
}

// Perform operation
if (strcmp(op, "+") == 0) {
    if (arg2[0] >= '0' && arg2[0] <= '9') {
        printf("    ADD AX, %s\n", arg2);
    } else {
        printf("    ADD AX, %s\n", arg2);
    }
}
else if (strcmp(op, "-") == 0) {
    if (arg2[0] >= '0' && arg2[0] <= '9') {
        printf("    SUB AX, %s\n", arg2);
    } else {
        printf("    SUB AX, %s\n", arg2);
    }
}
else if (strcmp(op, "*") == 0) {
    if (arg2[0] >= '0' && arg2[0] <= '9') {
        printf("    MOV BX, %s\n", arg2);
    } else {
        printf("    MOV BX, %s\n", arg2);
    }
    printf("    MUL BX\n");
}
else if (strcmp(op, "/") == 0) {
    if (arg2[0] >= '0' && arg2[0] <= '9') {
        printf("    MOV BX, %s\n", arg2);
    } else {
        printf("    MOV BX, %s\n", arg2);
    }
    printf("    XOR DX, DX\n");
    printf("    DIV BX\n");
}
```

```
    // Store result
    printf("    MOV %s, AX\n", result);
}

void process_conditional(int index) {
    char* arg1 = tac_codes[index].arg1;
    char* arg2 = tac_codes[index].arg2;
    char* op = tac_codes[index].op;

    static int label_count = 0;
    label_count++;

    printf("    MOV AX, %s\n", arg1);
    printf("    CMP AX, %s\n", arg2);

    if (strcmp(op, "==") == 0) {
        printf("    JNE LABEL_%d\n", label_count);
    }
    else if (strcmp(op, "!=") == 0) {
        printf("    JE LABEL_%d\n", label_count);
    }
    else if (strcmp(op, "<") == 0) {
        printf("    JGE LABEL_%d\n", label_count);
    }
    else if (strcmp(op, ">") == 0) {
        printf("    JLE LABEL_%d\n", label_count);
    }
    else if (strcmp(op, "<=") == 0) {
        printf("    JG LABEL_%d\n", label_count);
    }
    else if (strcmp(op, ">=") == 0) {
        printf("    JL LABEL_%d\n", label_count);
    }

    printf("    ; True branch code here\n");
    printf("LABEL_%d:\n", label_count);
    printf("    ; False branch code here\n");
}
```

```
void print_tac() {
    printf("\n=== THREE ADDRESS CODE ===\n");
    for (int i = 0; i < tac_count; i++) {
        if (strcmp(tac_codes[i].result, "IF") == 0) {
            printf("%d: IF %s %s %s GOTO L\n",
                   i, tac_codes[i].arg1, tac_codes[i].op, tac_codes[i].arg2);
        } else {
            printf("%d: %s = %s %s %s\n",
                   i, tac_codes[i].result, tac_codes[i].arg1,
                   tac_codes[i].op, tac_codes[i].arg2);
        }
    }
}

int main() {
    printf("=== Three Address Code to 8086 Assembly Compiler ===\n");
    printf("Enter TAC statements (one per line):\n");
    printf("Format: result = arg1 op arg2\n");
    printf("        or if a relop b\n");
    printf("Operators: +, -, *, /, =, ==, !=, <, >, <=, >=\n");
    printf("Example: t1 = a + b\n");
    printf("         if a < b\n");
    printf("Type 'END' to finish input\n\n");

    yyparse();

    print_tac();
    generate_8086_code();

    return 0;
}

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

INPUT.TAC:

```
t1 = b * c

t2 = a + t1

x = t2

if x < 10 goto L1

goto L2

L1:

t3 = x + 1

y = t3

goto L3

L2:

t4 = x - 1

y = t4

L3:
```

PROCEDURCE / COMPILATION PROCESS:

Execute these commands in sequence:

- win_bison -d tac_to_8086.y
- win_flex tac_to_8086.l
- gcc lex.yy.c tac_to_8086.tab.c -o tac_to_8086.exe

Input Three Address Code

Run the program and enter TAC statements in these formats:

Supported Formats:

- Simple Assignment: variable = constant or variable = variable
- Arithmetic Operations: result = arg1 operator arg2
- Conditional Statements: if variable relop variable

OUTPUT

RESULT

Thus, the back-end of the compiler was successfully implemented to generate equivalent `8086 assembly language code` from the given `Three Address Code.`