LAB SESSION 9: YACC: INTERMEDIATE CODE GENERATION

AIM: To implement a YACC program to generate three address intermediate code.

PROBLEM DEFINITION: Design and implement a YACC program that can:

1. Parse arithmetic expressions involving:
    a. Integer constants and variables
    b. Binary operators: +, -, *, /
    c. Parentheses for grouping
2. Parse while loops of the form:

```
while (condition) {
    statements
}
```

    Where condition is a relational expression using operators <, <=, >, >=, ==, != and statements may include assignments and nested arithmetic expressions.
3. Generate Three-Address Code (TAC) or quadruples for both arithmetic expressions and while loops.

THEORY: Intermediate code generation is an essential phase in a compiler, acting as a bridge between the high-level source code and the machine code. The intermediate code (IC) is a low-level, machine-independent representation of the program that makes it easier to optimize and translate for different target architectures. Unlike source code, which is closely tied to human-readable constructs, IC abstracts away hardware details while retaining enough structure to facilitate further processing, such as code optimization and target code generation.

One of the most common forms of intermediate code is the Three-Address Code (TAC). TAC represents computations in a form where each instruction contains at most three operands, typically written as x = y op z. Here, x is the target variable or a temporary variable used for intermediate results, y and z are operands (which may be variables, constants, or temporaries), and op is an operator such as addition, subtraction, multiplication, division, or relational operators. For example, the expression x = (a + b) * (c - d) can be translated into TAC as: t1 = a + b, t2 = c - d, t3 = t1 * t2, and x = t3. Temporary variables make it possible to break down complex expressions into simple, manageable steps.

TAC is also used to represent control flow explicitly, which is particularly useful for loops and conditional statements. In the case of a while loop, labels and conditional jumps are introduced to control the flow of execution. For instance, the high-level loop while (i <= 10) { sum = sum + i; i = i + 1; } can be represented in TAC as: L1: t1 = i <= 10, if t1 == 0 goto L2, t2 = sum + i, sum = t2, t3 = i + 1, i = t3, goto L1, L2:. Here, L1 marks the start of the loop and L2 the exit point, while temporary variables (t1, t2, t3) store intermediate results of computations.

The process of generating TAC is closely tied to the syntax-directed translation approach, where semantic actions are associated with grammar rules in a parser, typically using tools like YACC. These semantic actions are executed when a grammar rule is matched during parsing. They allow the compiler to assign temporary variables, generate TAC instructions for expressions and assignments, handle labels for control structures, and manage the

symbol table for variable declarations and usage. By using semantic actions, the compiler can generate intermediate code dynamically as it parses the input program.

TAC offers several advantages in the compilation process. First, it simplifies optimization because each intermediate instruction is simple and explicit, making it easier to identify common subexpressions, move code outside loops, or eliminate redundant calculations. Second, it provides a clear mapping to target machine instructions, facilitating the final code generation. Finally, it makes the representation of expressions, assignments, and control flow explicit, which aids in semantic analysis and further transformations in the compiler.

Three-Address Code is a powerful and widely used form of intermediate code that allows compilers to represent arithmetic expressions, assignments, and control structures like loops in a simple, machine-independent format. Understanding TAC and its generation through semantic actions is crucial for anyone studying compiler design, as it provides the foundation for code optimization and efficient target code generation. By systematically translating expressions and loops into TAC, a compiler can ensure correctness while preparing for subsequent stages of compilation.

In addition to basic arithmetic expressions and simple loops, generating intermediate code efficiently requires careful consideration of expression evaluation order and operator precedence. When expressions involve multiple operators, especially with mixed precedence and associativity, the compiler must generate TAC in a sequence that respects the original semantics of the program. This often involves creating a postfix or abstract syntax tree (AST) representation of the expression first, and then traversing it to emit TAC instructions in a systematic manner. Such an approach ensures that temporary variables are allocated correctly, and intermediate computations are not overwritten before they are used, which is essential for maintaining program correctness.

When dealing with nested loops or nested control structures, the management of labels becomes more complex. Each loop or conditional construct typically requires unique start and exit labels to avoid conflicts, and compilers often maintain a label stack to handle this. This allows TAC generation to be modular: inner loops can generate their own set of labels independently of outer loops, and goto statements can be resolved accurately. Nested loops also highlight the importance of efficient temporary variable management, as deeply nested expressions can result in a large number of intermediate values.

Short-circuit evaluation is another important consideration in TAC generation for logical and relational expressions, particularly when used in loop conditions or if statements. For example, in a compound condition like while (a < b && c != d), the compiler must ensure that the second part of the condition (c != d) is evaluated only if the first part (a < b) is true. TAC generation can achieve this by using conditional jumps to skip unnecessary evaluations, improving both efficiency and correctness of the generated code.

Intermediate code also plays a critical role in compiler optimizations. Once TAC is generated, it can be analyzed for redundant calculations, common subexpressions, and dead code elimination. For instance, if the same arithmetic expression appears multiple times in a loop, TAC generation combined with temporary variables makes it straightforward to compute the value once and reuse it. Similarly, optimization algorithms can rearrange TAC instructions to reduce the number of temporary variables, minimize jumps, or reorder independent computations for better performance on the target machine.

Memory management for temporaries is another consideration in intermediate code generation. Since TAC introduces many temporary variables for intermediate results, an

efficient compiler must reuse temporary variables whenever possible to reduce memory footprint. This involves tracking which temporaries are no longer needed and can be safely reassigned, a concept closely related to register allocation in later stages of compilation. Proper management of temporaries not only conserves memory but also simplifies the mapping of TAC to machine registers during final code generation.

Furthermore, TAC is instrumental in representing complex control flow constructs, such as nested while loops combined with break and continue statements. By generating labels for each control entry and exit point, the compiler can accurately implement the semantics of these statements without modifying the original AST. This label-based approach makes it easier to implement additional control structures in the future, such as for loops, do-while loops, and multi-branch conditionals, using a consistent intermediate representation.

Finally, TAC serves as a foundation for machine-independent optimizations. Because TAC abstracts away from hardware details, transformations can be applied uniformly across different architectures. For example, loop unrolling, strength reduction, and inlining can be performed at the TAC level before translating to the target code. This separation of concerns—where TAC handles program semantics independently of the target machine—enables compiler designers to write more modular and maintainable optimization passes, ultimately producing more efficient executable programs.

PROGRAM:

Lex file:

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

%%
[ \t\r\n]+          ;
"while"             { return WHILE; }
"<="                { return LE; }
">="                { return GE; }
"=="                { return EQ; }
"!="                { return NE; }
"<"                 { return LT; }
">"                 { return GT; }
[0-9]+              { yylval.str = strdup(yytext); return NUM; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); return ID; }
"="                 { return '='; }
";"                 { return ';'; }
"("                 { return '('; }
")"                 { return ')'; }
"{"                 { return '{'; }
```

```
"}"             { return '}'; }
"+"             { return '+'; }
"-"             { return '-'; }
"*"             { return '*'; }
"/"             { return '/'; }
.               { printf("Unknown character: %s\n", yytext); }

%%

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

YACC file:

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

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

static int temp_count = 0;
static int label_count = 0;

char *newtemp() {
   char buf[32];
   sprintf(buf, "t%d", ++temp_count);
   return strdup(buf);
}

char *newlabel() {
   char buf[32];
   sprintf(buf, "L%d", ++label_count);
   return strdup(buf);
}

/* Simple label stack to support nested loops */
#define MAX_LABELS 256
static char *label_stack[MAX_LABELS];
static int label_top = 0;
void push_label(char *l) { if (label_top < MAX_LABELS) label_stack[label_top++] = l; }
char *pop_label() { if (label_top>0) return label_stack[--label_top]; return NULL; }
```

```
/* Exit label stack to ensure condition jump is placed before body */
static char *exit_stack[MAX_LABELS];
static int exit_top = 0;
void push_exit(char *l) { if (exit_top < MAX_LABELS) exit_stack[exit_top++] = l; }
char *pop_exit() { if (exit_top>0) return exit_stack[--exit_top]; return NULL; }

void emit(const char *fmt, ...) {
   va_list ap;
   va_start(ap, fmt);
   vprintf(fmt, ap);
   printf("\n");
   va_end(ap);
}

%}

%union { char *str; }

%token <str> ID NUM
%token WHILE
%token LT LE GT GE EQ NE

%left '+' '-'
%left '*' '/'
%right UMINUS

%type <str> expr cond relop

%%

program:
    stmt_list
   ;

stmt_list:
    /* empty */
   | stmt_list stmt
   ;

stmt:
    assignment
```

```
  | while_stmt
  ;

assignment:
    ID '=' expr ';' {
      emit("%s = %s", $1, $3);
      free($1); free($3);
    }
  ;

/* mid-rule action to emit start label before condition code is generated */
while_stmt:
    WHILE { char *s = newlabel(); push_label(s); printf("%s:\n", s); }
    '(' cond ')' { char *e = newlabel(); push_exit(e); emit("if %s == 0 goto %s", $4, e);
free($4); }
    '{' stmt_list '}' {
      char *start = pop_label();
      char *exit = pop_exit();
      /* loop back after body */
      emit("goto %s", start);
      printf("%s:\n", exit);
      free(start); free(exit);
    }
  ;

cond:
    expr relop expr {
      char *tmp = newtemp();
      emit("%s = %s %s %s", tmp, $1, $2, $3);
      free($1); free($2); free($3);
      $$ = tmp;
    }
  ;

relop:
    LT { $$ = strdup("<"); }
  | LE { $$ = strdup("<="); }
  | GT { $$ = strdup(">"); }
  | GE { $$ = strdup(">="); }
  | EQ { $$ = strdup("=="); }
  | NE { $$ = strdup("!="); }
  ;
```

```
expr:
      expr '+' expr { char *t=newtemp(); emit("%s = %s + %s", t, $1, $3); free($1);
free($3); $$ = t; }
    | expr '-' expr { char *t=newtemp(); emit("%s = %s - %s", t, $1, $3); free($1);
free($3); $$ = t; }
    | expr '*' expr { char *t=newtemp(); emit("%s = %s * %s", t, $1, $3); free($1);
free($3); $$ = t; }
    | expr '/' expr { char *t=newtemp(); emit("%s = %s / %s", t, $1, $3); free($1);
free($3); $$ = t; }
   | '(' expr ')'  { $$ = $2; }
   | '-' expr %prec UMINUS { char *t=newtemp(); emit("%s = - %s", t, $2); free($2); $$
= t; }
   | ID { $$ = $1; }
   | NUM { $$ = $1; }
   ;

%%

int main(int argc, char **argv) {
   printf("--- Three Address Code (TAC) output ---\n");
   if (yyparse() == 0) {
      printf("--- End of TAC ---\n");
   }
   return 0;
}

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

Make file:

```
all: tac

tac: y.tab.c lex.yy.c
   @echo "Building tac..."
   gcc -o tac y.tab.c lex.yy.c

y.tab.c y.tab.h: grammar.y
   bison -d -y grammar.y
```

```
lex.yy.c: lexer.l y.tab.h
   flex lexer.l

clean:
   rm -f y.tab.c y.tab.h lex.yy.c tac y.output
```

all_tests.tst:

```
x = 42;

x = 1 + 2 * 3;

x = (1 + 2) * 3;

a = 10;
x = -a;
y = - ( -5 );

a = 10;
b = 6;
c = 2;
z = a - b / c;

i = 0;
while (i < 3) {
   i = i + 1;
}

i = 0;
sum = 0;
while (i <= 4) {
   sum = sum + i * 2;
   i = i + 2;
}

n = 3;
while (n > 0) {
   n = n - 1;
}

a = 5;
b = 0;
```

```
while (a + b >= 5) {
   a = a - 1;
}

x = 0;
y = 0;
while (x == y) {
   x = x + 1;
}

x = 5;
while (x != 0) {
   x = x / 2;
}

i = 0;
j = 0;
while (i < 3) {
   while (j < 2) {
      j = j + 1;
   }
   i = i + 1;
   j = 0;
}
```

OUTPUT:

```
PS C:\Users\Joseph\Desktop\compiler design\expt9> Set-Location -Path "c:\Users\Joseph\Desktop\compiler design\expt9"; gcc -o tac y.tab.c lex.yy.c; Get-Content .\tests\all_tests.tst | .\tac.exe
--- Three Address Code (TAC) output ---
x = 42
t1 = 2 * 3
t2 = 1 + t1
x = t2
t3 = 1 + 2
t4 = t3 * 3
x = t4
a = 10
t5 = - a
x = t5
t6 = - 5
```

```
t7 = - t6
y = t7
a = 10
b = 6
c = 2
t8 = b / c
t9 = a - t8
z = t9
i = 0
L1:
t10 = i < 3
if t10 == 0 goto L2
t11 = i + 1
i = t11
goto L1
L2:
i = 0
sum = 0
L3:
t12 = i <= 4
if t12 == 0 goto L4
t13 = i * 2
t14 = sum + t13
sum = t14
t15 = i + 2
i = t15
goto L3
L4:
n = 3
L5:
t16 = n > 0
if t16 == 0 goto L6
t17 = n - 1
n = t17
goto L5
L6:
a = 5
b = 0
L7:
t18 = a + b
t19 = t18 >= 5
if t19 == 0 goto L8
```

```
t20 = a - 1
a = t20
goto L7
L8:
x = 0
y = 0
L9:
t21 = x == y
if t21 == 0 goto L10
t22 = x + 1
x = t22
goto L9
L10:
x = 5
L11:
t23 = x != 0
if t23 == 0 goto L12
t24 = x / 2
x = t24
goto L11
L12:
i = 0
j = 0
L13:
t25 = i < 3
if t25 == 0 goto L14
L15:
t26 = j < 2
if t26 == 0 goto L16
t27 = j + 1
j = t27
goto L15
L16:
t28 = i + 1
i = t28
j = 0
goto L13
L14:
--- End of TAC ---
```

CONCLUSION:

In this lab expt , we consolidated all individual test programs into a single input file (all_tests.tst), built the TAC generator from the existing Bison/YACC and Flex outputs, and executed the full suite in native PowerShell without any script files by piping input (Get-Content … | .\tac.exe), verifying that the grammar correctly handles operator precedence, parentheses, unary minus, division/subtraction interactions, while-loops (including nesting), and relational operators (==, !=, <, <=, >, >=). Key learnings: understanding how lexical tokens and parser actions cooperate to emit consistent three-address code with deterministic temporaries and labels; appreciating the value of a simple Makefile and pre-generated parser/lexer sources for fast rebuilds; recognizing PowerShell's I/O differences from cmd/sh (prefer piping over < redirection in this context); and seeing how combining small, focused tests into one stream provides a repeatable, end-to-end validation of code generation and control-flow handling suitable for automated checks in future labs.