Bangladesh Agricultural University

Department of Computer Science and Mathematics

CSM 3222: Compiler Lab

Lab Assignment 6

Three Address Code and Assembly Code Generation

Submitted By:

Name: Ihfaz Hakim Adnan

Student ID: 2209014

Level-3, Semester-2

Submitted To:

Md Saifuddin

Lecturer,Department of Computer Science and Mathematics

Bangladesh Agricultural University

Date of Submission:

9 February,2026

Contents

1 Task 1: TAC and Assembly Code for Arithmetic and Logical Operations

1.1 Objective

To generate three-address code and assembly code for statements containing arithmetic, assignment, and logical operations with proper operator precedence handling.

1.2 Grammar

$$
\begin{aligned}
Program &\rightarrow StatementList \\
StatementList &\rightarrow Statement \mid StatementList\ NEWLINE\ Statement \\
Statement &\rightarrow ID\ '='\ Expression \mid ID\ OpAssign\ Expression \\
OpAssign &\rightarrow' +=' \mid' -=' \mid' *=' \mid' /=' \mid' \%=' \mid' **=' \\
Expression &\rightarrow Expression\ '+'\ Term \mid Expression\ '-'\ Term \mid Term \\
Term &\rightarrow Term\ '*'\ Factor \mid Term\ '/'\ Factor \mid Term\ '//'\ Factor \mid Factor \\
Factor &\rightarrow Factor\ '**'\ Unary \mid Unary \\
Unary &\rightarrow'!'\ Unary \mid' -'\ Unary \mid Primary \\
Primary &\rightarrow ID \mid NUM \mid' ('\ Expression\ ')' \\
ID &\rightarrow [a-zA-Z][a-zA-Z0-9]^* \\
NUM &\rightarrow [0-9]+ \\
NEWLINE &\rightarrow' \backslash n'
\end{aligned}
$$

1.3 Requirements

- Flex (version 2.6 or higher)
- Bison (version 3.x or higher)
- GCC compiler (MinGW for Windows)
- Text editor

1.4 Installation and Setup

Installing Flex, Bison, and GCC on Windows:

1. Download and install MinGW-w64 from `https://www.mingw-w64.org/`
2. Add MinGW bin directory to System PATH (e.g., `C:\mingw64\bin`)
3. Install Flex and Bison binaries for Windows
4. Verify installation:

```
flex --version
bison --version
gcc --version
```

1.5 Implementation

GitHub Repository: https://github.com/Ihfaz07/Compiler-lab

Lexer File (lexer.l):

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

%%
[ \t]+          { /* ignore whitespace */ }
\n              { return NEWLINE; }
[0-9]+          { yylval.str = strdup(yytext); return NUM; }
[a-zA-Z][a-zA-Z0-9]* { yylval.str = strdup(yytext); return ID; }
"+"             { return PLUS; }
"-"             { return MINUS; }
"*"             { return TIMES; }
"/"             { return DIVIDE; }
"//"            { return INTDIV; }
"**"            { return POWER; }
"%"             { return MOD; }
"="             { return ASSIGN; }
"+="            { return PLUSEQ; }
"-="            { return MINUSEQ; }
"*="            { return TIMESEQ; }
"/="            { return DIVEQ; }
"%="            { return MODEQ; }
"**="           { return POWEREQ; }
"&&"            { return AND; }
"||"            { return OR; }
"!"             { return NOT; }
">"             { return GT; }
"<"             { return LT; }
"("             { return LPAREN; }
")"             { return RPAREN; }
.               { /* ignore other characters */ }
%%

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

Parser File (parser.y):

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

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

int temp_count = 0;
int reg_count = 0;

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

void emit_tac(const char* dest, const char* src1, const char* op,
              const char* src2) {
    if (src2) {
        printf("%s = %s %s %s\n", dest, src1, op, src2);
    } else if (op) {
        printf("%s = %s %s\n", dest, op, src1);
    } else {
        printf("%s = %s\n", dest, src1);
    }
}

void emit_asm(const char* op, const char* dest, const char* src) {
    if (src) {
        printf("%s %s , %s\n", op, dest, src);
    } else {
        printf("%s %s\n", op, dest);
    }
    printf("\n");
}
%}

%union {
    char* str;
}

%token <str> ID NUM
%token PLUS MINUS TIMES DIVIDE INTDIV POWER MOD
%token ASSIGN PLUSEQ MINUSEQ TIMESEQ DIVEQ MODEQ POWEREQ
%token AND OR NOT GT LT LPAREN RPAREN NEWLINE

%type <str> expression term factor unary primary

%left OR
%left AND
%left PLUS MINUS
%left TIMES DIVIDE INTDIV MOD
%right POWER
%right NOT UMINUS

```

```
%%

program:
    statement_list
    ;

statement_list:
    statement
    | statement_list statement
    ;

statement:
    ID ASSIGN expression NEWLINE {
        emit_tac($1, $3, NULL, NULL);
        emit_asm("MOV", $3, $1);
        emit_asm("MOV", $1, $3);
    }
    | ID ASSIGN expression {
        emit_tac($1, $3, NULL, NULL);
        emit_asm("MOV", "R0", $3);
        emit_asm("MOV", $1, "R0");
    }
    | ID PLUSEQ expression NEWLINE {
        char* temp = new_temp();
        emit_tac(temp, $1, "+", $3);
        emit_tac($1, temp, NULL, NULL);

        emit_asm("MOV", "R0", $1);
        emit_asm("ADD", "R0", $3);
        emit_asm("MOV", $1, "R0");
    }
    | ID MINUSEQ expression NEWLINE {
        char* temp = new_temp();
        emit_tac(temp, $1, "-", $3);
        emit_tac($1, temp, NULL, NULL);

        emit_asm("MOV", "R0", $1);
        emit_asm("SUB", "R0", $3);
        emit_asm("MOV", $1, "R0");
    }
    | ID TIMESEQ expression NEWLINE {
        char* temp = new_temp();
        emit_tac(temp, $1, "*", $3);
        emit_tac($1, temp, NULL, NULL);

        emit_asm("MOV", "R0", $1);
        emit_asm("MUL", "R0", $3);
        emit_asm("MOV", $1, "R0");
    }
    | ID DIVEQ expression NEWLINE {
        char* temp = new_temp();
        emit_tac(temp, $1, "/", $3);
        emit_tac($1, temp, NULL, NULL);

        emit_asm("MOV", "R0", $1);
        emit_asm("DIV", "R0", $3);
        emit_asm("MOV", $1, "R0");
    }
    | ID MODEQ expression NEWLINE {
        char* temp = new_temp();
        emit_tac(temp, $1, "%", $3);
        emit_tac($1, temp, NULL, NULL);

        emit_asm("MOV", "R0", $1);
        emit_asm("MOD", "R0", $3);
        emit_asm("MOV", $1, "R0");
    }
    | ID POWEREQ expression NEWLINE {
        char* temp = new_temp();
        emit_tac(temp, $1, "**", $3);
        emit_tac($1, temp, NULL, NULL);

        emit_asm("MOV", "R0", $1);
        emit_asm("POW", "R0", $3);
        emit_asm("MOV", $1, "R0");
    }
    | NEWLINE
    ;

expression:
    expression PLUS term {
        char* temp = new_temp();
        emit_tac(temp, $1, "+", $3);
        $$ = temp;
    }
    | expression MINUS term {
        char* temp = new_temp();
        emit_tac(temp, $1, "-", $3);
        $$ = temp;
    }
    | expression OR term {
        char* temp = new_temp();
        emit_tac(temp, $1, "||", $3);
        $$ = temp;
    }
    | term { $$ = $1; }
    ;

term:
    term TIMES factor {
        char* temp = new_temp();
        emit_tac(temp, $1, "*", $3);
        $$ = temp;
    }
    | term DIVIDE factor {
        char* temp = new_temp();
        emit_tac(temp, $1, "/", $3);
        $$ = temp;
    }
    | term INTDIV factor {
        char* temp = new_temp();
        emit_tac(temp, $1, "//", $3);
        $$ = temp;
    }
    | term MOD factor {
        char* temp = new_temp();
        emit_tac(temp, $1, "%", $3);
        $$ = temp;
    }
    | term AND factor {
        char* temp = new_temp();
        emit_tac(temp, $1, "&&", $3);
        $$ = temp;
    }
    | term GT factor {
        char* temp = new_temp();
        emit_tac(temp, $1, ">", $3);
        $$ = temp;
    }
    | term LT factor {
        char* temp = new_temp();
        emit_tac(temp, $1, "<", $3);
        $$ = temp;
    }
    | factor { $$ = $1; }
    ;

factor:
    factor POWER unary {
        char* temp = new_temp();
        emit_tac(temp, $1, "**", $3);
        $$ = temp;
    }
    | unary { $$ = $1; }
    ;

unary:
    NOT unary {
        char* temp = new_temp();
        emit_tac(temp, "!", $2, NULL);
        $$ = temp;
    }
    | MINUS unary %prec UMINUS {
        char* temp = new_temp();
        emit_tac(temp, "-", $2, NULL);
        $$ = temp;
    }
    | primary { $$ = $1; }
    ;

primary:
    LPAREN expression RPAREN { $$ = $2; }
    | ID { $$ = $1; }
    | NUM {
        char* temp = (char*)malloc(strlen($1) + 2);
        sprintf(temp, "#%s", $1);
        $$ = temp;
    }
    ;

%%

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

```
}

int main(int argc, char **argv) {
    if (argc > 1) {
        FILE *file = fopen(argv[1], "r");
        if (file) {
            yyin = file;
        }
    }
    yyparse();
    return 0;
}
```

Compilation and Execution:

```
# Generate parser and lexer
bison -d parser.y
flex lexer.l

# Compile
gcc parser.tab.c lex.yy.c -o codegen

# Run with input file
codegen.exe input.txt
```

1.6 Input and Output

Input (input.txt):

```
a = 5 + 3
b += a * 2
c = ! b || 0
d = a ** 2
e //= 3
f = ( a + b ) * ( c - d ) / e
g %= ( f ** 2) + 1
h = !(( a > b ) && ( c < d ) ) || e
i **= 2
j = i // (a + b * c)
```

Output (Three Address Code):

```
t1 = 5 + 3
a = t1
t2 = a * 2
b = b + t2
t3 = ! b
t4 = t3 || 0
c = t4
t5 = a ** 2
d = t5
t6 = e // 3
e = t6
t7 = a + b
t8 = c - d
t9 = t7 * t8
t10 = t9 / e
f = t10
t11 = f ** 2
t12 = t11 + 1
g = f % t12
t13 = a > b
t14 = c < d
t15 = t13 && t14
t16 = ! t15
t17 = t16 || e
h = t17
t18 = i ** 2
i = t18
t19 = b * c
t20 = a + t19
t21 = i // t20
j = t21
```

Output (Assembly Code):

```
MOV R0 , #5
ADD R0 , #3
MOV a , R0

MOV R0 , a
MUL R0 , #2
MOV R1 , b
ADD R1 , R0
MOV b , R1

MOV R0 , b
NOT R0
OR R0 , #0
```

```
MOV c , R0

MOV R0 , a
POW R0 , #2
MOV d , R0

MOV R0 , e
IDIV R0 , #3
MOV e , R0

MOV R0 , a
ADD R0 , b

MOV R1 , c
SUB R1 , d

MUL R0 , R1
DIV R0 , e
MOV f , R0

MOV R0 , f
POW R0 , #2
ADD R0 , #1

MOV R1 , f
MOD R1 , R0
MOV g , R1

MOV R0 , a
CMPGT R0 , b

MOV R1 , c
CMPLT R1 , d

AND R0 , R1
NOT R0
OR R0 , e
MOV h , R0

MOV R0 , i
POW R0 , #2
MOV i , R0

MOV R0 , b
MUL R0 , c

MOV R1 , a
ADD R1 , R0
```

```
MOV R2 , i
IDIV R2 , R1
MOV j , R2
```

1.7 Working Principles

The code generator operates in two phases: three-address code generation and assembly code translation. During parsing, each operation creates a temporary variable following the syntax-directed translation scheme. Binary operations emit TAC instructions in the form ”t = x op y”, while unary operations use ”t = op x”. The assembly code generation maps TAC instructions to a register-based architecture using R0, R1, R2 as general-purpose registers. Simple assignments use MOV instructions, arithmetic operations (ADD, SUB, MUL, DIV, MOD, POW) operate on registers, and immediate values are prefixed with #. Comparison operations (CMPGT, CMPLT) set condition flags, logical operations (AND, OR, NOT) manipulate boolean values, and compound assignments decompose into binary operations followed by assignment. The integer division operator (//) maps to IDIV instruction in assembly.

2 Task 2: TAC and Assembly Code for Math Functions

2.1 Objective

To generate three-address code and assembly code for arithmetic statements containing mathematical functions like sqrt, pow, log, exp, sin, cos, tan, and abs.

2.2 Grammar

$$
\begin{aligned}
Program &\rightarrow StatementList\\
StatementList &\rightarrow Statement \mid StatementList\ NEWLINE\ Statement\\
Statement &\rightarrow ID\ '='\ Expression\\
Expression &\rightarrow Expression\ '+'\ Term \mid Expression\ '-'\ Term \mid Term\\
Term &\rightarrow Term\ '*'\ Factor \mid Term\ '/'\ Factor \mid Term\ '\%'\ Factor \mid Factor\\
Factor &\rightarrow FunctionCall \mid'\ ('\ Expression\ ')' \mid ID \mid NUM \mid'-'\ Factor\\
FunctionCall &\rightarrow sqrt'('Expression')' \mid pow'('Expression','Expression')'\\
&\mid log'('Expression')' \mid exp'('Expression')'\\
&\mid sin'('Expression')' \mid cos'('Expression')'\\
&\mid tan'('Expression')' \mid abs'('Expression')'\\
ID &\rightarrow [a-zA-Z][a-zA-Z0-9]^*\\
NUM &\rightarrow [0-9]+\\
NEWLINE &\rightarrow'\backslash n'
\end{aligned}
$$

2.3 Requirements

- Flex (version 2.6 or higher)
- Bison (version 3.x or higher)
- GCC compiler (MinGW for Windows)
- Text editor

2.4 Installation and Setup

Installing Flex, Bison, and GCC on Windows:

1. Download and install MinGW-w64 from `https://www.mingw-w64.org/`
2. Add MinGW bin directory to System PATH (e.g., `C:\mingw64\bin`)
3. Install Flex and Bison binaries for Windows
4. Verify installation:

```
flex --version
bison --version
gcc --version
```

2.5 Implementation

GitHub Repository: https://github.com/Ihfaz07/Compiler-lab

Lexer File (math_lexer.l):

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

%%
[ \t]+          { /* ignore whitespace */ }
\n              { return NEWLINE; }
[0-9]+          { yylval.str = strdup(yytext); return NUM; }
"sqrt"          { return SQRT; }
"pow"           { return POW; }
"log"           { return LOG; }
"exp"           { return EXP; }
"sin"           { return SIN; }
"cos"           { return COS; }
"tan"           { return TAN; }
"abs"           { return ABS; }
[a-zA-Z][a-zA-Z0-9]* { yylval.str = strdup(yytext); return ID; }
"+"             { return PLUS; }
"-"             { return MINUS; }
"*"             { return TIMES; }
"/"             { return DIVIDE; }
"%"             { return MOD; }
"="             { return ASSIGN; }
"("             { return LPAREN; }
")"             { return RPAREN; }
","             { return COMMA; }
.               { /* ignore other characters */ }
%%

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

Parser File (math_parser.y):

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

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

int temp_count = 0;

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

void emit_tac(const char* dest, const char* src1, const char* op,
              const char* src2) {
    if (src2) {
        printf("%s = %s %s %s\n", dest, src1, op, src2);
    } else if (op) {
        printf("%s = %s %s\n", dest, op, src1);
    } else {
        printf("%s = %s\n", dest, src1);
    }
}

void emit_asm(const char* op, const char* dest, const char* src) {
    if (src) {
        printf("%s %s , %s\n", op, dest, src);
    } else {
        printf("%s %s\n", op, dest);
    }
    printf("\n");
}

void emit_function_asm(const char* func, const char* arg1,
                       const char* arg2) {
    printf("MOV R0 , %s\n", arg1);
    if (arg2) {
        printf("%s R0 , %s\n", func, arg2);
    } else {
        printf("%s R0\n", func);
    }
    printf("\n");
}
%}

%union {
    char* str;
}

%token <str> ID NUM
%token PLUS MINUS TIMES DIVIDE MOD ASSIGN
%token LPAREN RPAREN COMMA NEWLINE
%token SQRT POW LOG EXP SIN COS TAN ABS
```

```

%type <str> expression term factor function_call

%left PLUS MINUS
%left TIMES DIVIDE MOD

%%

program:
    statement_list
    ;

statement_list:
    statement
    | statement_list statement
    ;

statement:
    ID ASSIGN expression NEWLINE {
        emit_tac($1, $3, NULL, NULL);
        emit_asm("MOV", "R0", $3);
        emit_asm("MOV", $1, "R0");
    }
    | ID ASSIGN expression {
        emit_tac($1, $3, NULL, NULL);
        emit_asm("MOV", "R0", $3);
        emit_asm("MOV", $1, "R0");
    }
    | NEWLINE
    ;

expression:
    expression PLUS term {
        char* temp = new_temp();
        emit_tac(temp, $1, "+", $3);
        $$ = temp;
    }
    | expression MINUS term {
        char* temp = new_temp();
        emit_tac(temp, $1, "-", $3);
        $$ = temp;
    }
    | term { $$ = $1; }
    ;

term:
    term TIMES factor {
        char* temp = new_temp();
        emit_tac(temp, $1, "*", $3);
        $$ = temp;
    }
    | term DIVIDE factor {
        char* temp = new_temp();
        emit_tac(temp, $1, "/", $3);
        $$ = temp;
    }
    | term MOD factor {
        char* temp = new_temp();
        emit_tac(temp, $1, "%", $3);
        $$ = temp;
    }
    | factor { $$ = $1; }
    ;

factor:
    function_call { $$ = $1; }
    | LPAREN expression RPAREN { $$ = $2; }
    | ID { $$ = $1; }
    | NUM {
        char* temp = (char*)malloc(strlen($1) + 2);
        sprintf(temp, "#%s", $1);
        $$ = temp;
    }
    | MINUS factor {
        char* temp = new_temp();
        printf("%s = -%s\n", temp, $2);
        printf("MOV R0 , %s\n", $2);
        printf("NEG R0\n\n");
        $$ = temp;
    }
    ;

function_call:
    SQRT LPAREN expression RPAREN {
        char* temp = new_temp();
        printf("%s = sqrt ( %s )\n", temp, $3);
        emit_function_asm("SQRT", $3, NULL);
        $$ = temp;
    }
    | POW LPAREN expression COMMA expression RPAREN {
        char* temp = new_temp();
        printf("%s = pow (%s , %s)\n", temp, $3, $5);
        emit_function_asm("POW", $3, $5);
        $$ = temp;
    }
    | LOG LPAREN expression RPAREN {
        char* temp = new_temp();
        printf("%s = log ( %s )\n", temp, $3);
        emit_function_asm("LOG", $3, NULL);
        $$ = temp;
    }
    | EXP LPAREN expression RPAREN {
        char* temp = new_temp();
        printf("%s = exp ( %s )\n", temp, $3);
        emit_function_asm("EXP", $3, NULL);
        $$ = temp;
    }
    | SIN LPAREN expression RPAREN {
        char* temp = new_temp();
        printf("%s = sin ( %s )\n", temp, $3);
        emit_function_asm("SIN", $3, NULL);
        $$ = temp;
    }
    | COS LPAREN expression RPAREN {
        char* temp = new_temp();
        printf("%s = cos ( %s )\n", temp, $3);
        emit_function_asm("COS", $3, NULL);
        $$ = temp;
    }
    | TAN LPAREN expression RPAREN {
        char* temp = new_temp();
        printf("%s = tan ( %s )\n", temp, $3);
        emit_function_asm("TAN", $3, NULL);
        $$ = temp;
    }
    | ABS LPAREN expression RPAREN {
        char* temp = new_temp();
        printf("%s = abs ( %s )\n", temp, $3);
        emit_function_asm("ABS", $3, NULL);
        $$ = temp;
    }
    ;

%%

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

int main(int argc, char **argv) {
    if (argc > 1) {
        FILE *file = fopen(argv[1], "r");
        if (file) {
            yyin = file;
        }
    }
    yyparse();
    return 0;
}
```

Compilation and Execution:

```
# Generate parser and lexer
bison -d math_parser.y
flex math_lexer.l

# Compile
gcc math_parser.tab.c lex.yy.c -o math_codegen

# Run with input file
math_codegen.exe input.txt
```

2.6 Input and Output

Input (input.txt):

```
a = 9
b = sqrt ( a )
c = pow (a , 3)
d = log ( b ) + sin ( a )
e = cos ( c ) * tan ( d )
```

```
f = abs ( - a + b ) / exp (2)
```

Output (Three Address Code):

```
a = 9
t1 = sqrt ( a )
b = t1
t2 = pow (a , 3)
c = t2
t3 = log ( b )
t4 = sin ( a )
t5 = t3 + t4
d = t5
t6 = cos ( c )
t7 = tan ( d )
t8 = t6 * t7
e = t8
t9 = -a
t10 = t9 + b
t11 = abs ( t10 )
t12 = exp (2)
t13 = t11 / t12
f = t13
```

Output (Assembly Code):

```
MOV R0 , #9
MOV a , R0

MOV R0 , a
SQRT R0
MOV b , R0

MOV R0 , a
POW R0 , #3
MOV c , R0

MOV R0 , b
LOG R0

MOV R1 , a
SIN R1

ADD R0 , R1
MOV d , R0

MOV R0 , c
COS R0

MOV R1 , d
TAN R1
```

```
MUL R0 , R1
MOV e , R0

MOV R0 , a
NEG R0

ADD R0 , b
ABS R0

MOV R1 , #2
EXP R1

DIV R0 , R1
MOV f , R0
```

2.7 Working Principles

The math function code generator extends the basic arithmetic translator to handle mathematical library functions. Each function call in the source generates both TAC and assembly code. Single-argument functions (sqrt, log, exp, sin, cos, tan, abs) take one expression as input, load it into R0, apply the function instruction, and store the result in a temporary variable. The two-argument pow function loads the base into R0 and applies the exponent as a second operand. Arithmetic operations between function results follow standard precedence rules. The unary minus operator generates a NEG instruction in assembly. Complex expressions are decomposed into simple operations where each intermediate result is stored in a temporary variable, creating a linear sequence of three-address instructions. Assembly instructions use register R0 for primary operations and R1 for secondary operands in binary operations, with immediate values prefixed by #.