Bangladesh Agricultural University

Department of Computer Science and Mathematics

CSM 3222: Compiler Lab

Lab Assignment 5

Three Address 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:

4 February,2026

Contents

1 Task 1: Three Address Code for Arithmetic and Logical Operations

1.1 Objective

To generate three-address 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;

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

%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 {
        printf("%s = %s\n", $1, $3);
    }
    | ID ASSIGN expression {
        printf("%s = %s\n", $1, $3);
    }
    | ID PLUSEQ expression NEWLINE {
        char* temp = new_temp();
        printf("%s = %s + %s\n", temp, $1, $3);
```

```
57          printf("%s = %s\n", $1, temp);
58      }
59      | ID MINUSEQ expression NEWLINE {
60          char* temp = new_temp();
61          printf("%s = %s - %s\n", temp, $1, $3);
62          printf("%s = %s\n", $1, temp);
63      }
64      | ID TIMESEQ expression NEWLINE {
65          char* temp = new_temp();
66          printf("%s = %s * %s\n", temp, $1, $3);
67          printf("%s = %s\n", $1, temp);
68      }
69      | ID DIVEQ expression NEWLINE {
70          char* temp = new_temp();
71          printf("%s = %s / %s\n", temp, $1, $3);
72          printf("%s = %s\n", $1, temp);
73      }
74      | ID MODEQ expression NEWLINE {
75          char* temp = new_temp();
76          printf("%s = %s %% %s\n", temp, $1, $3);
77          printf("%s = %s\n", $1, temp);
78      }
79      | ID POWEREQ expression NEWLINE {
80          char* temp = new_temp();
81          printf("%s = %s ** %s\n", temp, $1, $3);
82          printf("%s = %s\n", $1, temp);
83      }
84      | NEWLINE
85      ;
86
87  expression:
88      expression PLUS term {
89          char* temp = new_temp();
90          printf("%s = %s + %s\n", temp, $1, $3);
91          $$ = temp;
92      }
93      | expression MINUS term {
94          char* temp = new_temp();
95          printf("%s = %s - %s\n", temp, $1, $3);
96          $$ = temp;
97      }
98      | expression OR term {
99          char* temp = new_temp();
100         printf("%s = %s || %s\n", temp, $1, $3);
101         $$ = temp;
102     }
103     | term { $$ = $1; }
104     ;
105
106 term:
107     term TIMES factor {
108         char* temp = new_temp();
109         printf("%s = %s * %s\n", temp, $1, $3);
110         $$ = temp;
111     }
112     | term DIVIDE factor {
113         char* temp = new_temp();
114         printf("%s = %s / %s\n", temp, $1, $3);
```

```
115         $$ = temp;
116     }
117     | term INTDIV factor {
118         char* temp = new_temp();
119         printf("%s = %s // %s\n", temp, $1, $3);
120         $$ = temp;
121     }
122     | term MOD factor {
123         char* temp = new_temp();
124         printf("%s = %s %% %s\n", temp, $1, $3);
125         $$ = temp;
126     }
127     | term AND factor {
128         char* temp = new_temp();
129         printf("%s = %s && %s\n", temp, $1, $3);
130         $$ = temp;
131     }
132     | term GT factor {
133         char* temp = new_temp();
134         printf("%s = %s > %s\n", temp, $1, $3);
135         $$ = temp;
136     }
137     | term LT factor {
138         char* temp = new_temp();
139         printf("%s = %s < %s\n", temp, $1, $3);
140         $$ = temp;
141     }
142     | factor { $$ = $1; }
143     ;
144
145 factor:
146     factor POWER unary {
147         char* temp = new_temp();
148         printf("%s = %s ** %s\n", temp, $1, $3);
149         $$ = temp;
150     }
151     | unary { $$ = $1; }
152     ;
153
154 unary:
155     NOT unary {
156         char* temp = new_temp();
157         printf("%s = ! %s\n", temp, $2);
158         $$ = temp;
159     }
160     | MINUS unary %prec UMINUS {
161         char* temp = new_temp();
162         printf("%s = -%s\n", temp, $2);
163         $$ = temp;
164     }
165     | primary { $$ = $1; }
166     ;
167
168 primary:
169     LPAREN expression RPAREN { $$ = $2; }
170     | ID { $$ = $1; }
171     | NUM { $$ = $1; }
172     ;
```

```
%%

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 tac_generator

# Run with input file
tac_generator.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:

```
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
```

1.7 Working Principles

The program uses a syntax-directed translation scheme to generate three-address code. The lexer tokenizes the input into operators, identifiers, and numbers. The parser builds a parse tree following operator precedence rules and generates intermediate code during parsing. Each binary operation produces a new temporary variable to store the result. Compound assignment operators are decomposed into binary operations followed by assignments. The grammar ensures proper precedence: exponentiation (highest), unary operators, multiplication/division, addition/subtraction, logical operations (lowest). Temporary variables follow the naming convention t1, t2, t3, etc., incrementing sequentially.

2 Task 2: Three Address Code for Math Functions

2.1 Objective

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

2.2 Grammar

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

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;
}
%}

%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 {
        printf("%s = %s\n", $1, $3);
    }
    | ID ASSIGN expression {
        printf("%s = %s\n", $1, $3);
    }
    | NEWLINE
    ;

expression:
    expression PLUS term {
        char* temp = new_temp();
        printf("%s = %s + %s\n", temp, $1, $3);
```

```
57          $$ = temp;
58      }
59      | expression MINUS term {
60          char* temp = new_temp();
61          printf("%s = %s - %s\n", temp, $1, $3);
62          $$ = temp;
63      }
64      | term { $$ = $1; }
65      ;
66
67  term:
68      term TIMES factor {
69          char* temp = new_temp();
70          printf("%s = %s * %s\n", temp, $1, $3);
71          $$ = temp;
72      }
73      | term DIVIDE factor {
74          char* temp = new_temp();
75          printf("%s = %s / %s\n", temp, $1, $3);
76          $$ = temp;
77      }
78      | term MOD factor {
79          char* temp = new_temp();
80          printf("%s = %s %% %s\n", temp, $1, $3);
81          $$ = temp;
82      }
83      | factor { $$ = $1; }
84      ;
85
86  factor:
87      function_call { $$ = $1; }
88      | LPAREN expression RPAREN { $$ = $2; }
89      | ID { $$ = $1; }
90      | NUM { $$ = $1; }
91      | MINUS factor {
92          char* temp = new_temp();
93          printf("%s = -%s\n", temp, $2);
94          $$ = temp;
95      }
96      ;
97
98  function_call:
99      SQRT LPAREN expression RPAREN {
100         char* temp = new_temp();
101         printf("%s = sqrt ( %s )\n", temp, $3);
102         $$ = temp;
103     }
104     | POW LPAREN expression COMMA expression RPAREN {
105         char* temp = new_temp();
106         printf("%s = pow (%s , %s)\n", temp, $3, $5);
107         $$ = temp;
108     }
109     | LOG LPAREN expression RPAREN {
110         char* temp = new_temp();
111         printf("%s = log ( %s )\n", temp, $3);
112         $$ = temp;
113     }
114     | EXP LPAREN expression RPAREN {
```

```
        char* temp = new_temp();
        printf("%s = exp ( %s )\n", temp, $3);
        $$ = temp;
    }
    | SIN LPAREN expression RPAREN {
        char* temp = new_temp();
        printf("%s = sin ( %s )\n", temp, $3);
        $$ = temp;
    }
    | COS LPAREN expression RPAREN {
        char* temp = new_temp();
        printf("%s = cos ( %s )\n", temp, $3);
        $$ = temp;
    }
    | TAN LPAREN expression RPAREN {
        char* temp = new_temp();
        printf("%s = tan ( %s )\n", temp, $3);
        $$ = temp;
    }
    | ABS LPAREN expression RPAREN {
        char* temp = new_temp();
        printf("%s = abs ( %s )\n", temp, $3);
        $$ = 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 tac_math

# Run with input file
tac_math.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:

```
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
```

2.7 Working Principles

The compiler frontend uses lexical analysis to identify mathematical function keywords (sqrt, pow, log, exp, sin, cos, tan, abs) as reserved tokens, distinguishing them from regular identifiers. The parser follows the grammar to build an expression tree, where function calls are treated as primary factors. During parsing, each function call generates a new temporary variable to store its result. Binary operations (+, -, *, /, %) follow standard precedence rules. For functions with multiple arguments like pow(x, y), the parser handles comma-separated expressions. The unary minus operator is processed separately to handle expressions like -a before passing them to functions. Three-address code is emitted in postfix order, ensuring all operands are computed before their operations.