

# **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

<b>1</b>	<b>Task 1: TAC and Assembly Code for Arithmetic and Logical Operations</b>	<b>2</b>
1.1	Objective . . . . .	2
1.2	Grammar . . . . .	2
1.3	Requirements . . . . .	2
1.4	Installation and Setup . . . . .	2
1.5	Implementation . . . . .	3
1.6	Input and Output . . . . .	8
1.7	Working Principles . . . . .	11
<b>2</b>	<b>Task 2: TAC and Assembly Code for Math Functions</b>	<b>12</b>
2.1	Objective . . . . .	12
2.2	Grammar . . . . .	12
2.3	Requirements . . . . .	12
2.4	Installation and Setup . . . . .	12
2.5	Implementation . . . . .	13
2.6	Input and Output . . . . .	17
2.7	Working Principles . . . . .	20

# 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

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

## 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):

```
1 %{
2 #include "parser.tab.h"
3 #include <string.h>
4 %}
5
6 %%
7 [ \t]+      { /* ignore whitespace */ }
8 \n          { return NEWLINE; }
9 [0-9]+      { yylval.str = strdup(yytext); return NUM; }
10 [a-zA-Z][a-zA-Z0-9]* { yylval.str = strdup(yytext); return ID; }
11 "+"        { return PLUS; }
12 "-"        { return MINUS; }
13 "*"        { return TIMES; }
14 "/"        { return DIVIDE; }
15 "//"       { return INTDIV; }
16 "**"       { return POWER; }
17 "%"       { return MOD; }
18 "="      { return ASSIGN; }
19 "+="      { return PLUSEQ; }
20 "-="      { return MINUSEQ; }
21 "*="      { return TIMESEQ; }
22 "/="      { return DIVEQ; }
23 "%="      { return MODEQ; }
24 "**="      { return POWEREQ; }
25 "&&"      { return AND; }
26 "||"      { return OR; }
27 "!"       { return NOT; }
28 ">"       { return GT; }
29 "<"       { return LT; }
30 "("       { return LPAREN; }
31 ")"       { return RPAREN; }
32 .         { /* ignore other characters */ }
33 %%
34
35 int yywrap() { return 1; }
```

## Parser File (parser.y):

```
1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  extern int yylex();
7  void yyerror(const char *s);
8
9  int temp_count = 0;
10 int reg_count = 0;
11
12 char* new_temp() {
13     char* temp = (char*)malloc(10);
14     sprintf(temp, "t%d", ++temp_count);
15     return temp;
16 }
17
18 void emit_tac(const char* dest, const char* src1, const char* op,
19              const char* src2) {
20     if (src2) {
21         printf("%s = %s %s %s\n", dest, src1, op, src2);
22     } else if (op) {
23         printf("%s = %s %s\n", dest, op, src1);
24     } else {
25         printf("%s = %s\n", dest, src1);
26     }
27 }
28
29 void emit_asm(const char* op, const char* dest, const char* src) {
30     if (src) {
31         printf("%s %s , %s\n", op, dest, src);
32     } else {
33         printf("%s %s\n", op, dest);
34     }
35     printf("\n");
36 }
37 %}
38
39 %union {
40     char* str;
41 }
42
43 %token <str> ID NUM
44 %token PLUS MINUS TIMES DIVIDE INTDIV POWER MOD
45 %token ASSIGN PLUSEQ MINUSEQ TIMESEQ DIVEQ MODEQ POWEREQ
46 %token AND OR NOT GT LT LPAREN RPAREN NEWLINE
47
48 %type <str> expression term factor unary primary
49
50 %left OR
51 %left AND
52 %left PLUS MINUS
53 %left TIMES DIVIDE INTDIV MOD
54 %right POWER
55 %right NOT UMINUS
56
```

```

57 %%
58
59 program:
60     statement_list
61     ;
62
63 statement_list:
64     statement
65     | statement_list statement
66     ;
67
68 statement:
69     ID ASSIGN expression NEWLINE {
70         emit_tac($1, $3, NULL, NULL);
71         emit_asm("MOV", $3, $1);
72         emit_asm("MOV", $1, $3);
73     }
74     | ID ASSIGN expression {
75         emit_tac($1, $3, NULL, NULL);
76         emit_asm("MOV", "R0", $3);
77         emit_asm("MOV", $1, "R0");
78     }
79     | ID PLUSEQ expression NEWLINE {
80         char* temp = new_temp();
81         emit_tac(temp, $1, "+", $3);
82         emit_tac($1, temp, NULL, NULL);
83
84         emit_asm("MOV", "R0", $1);
85         emit_asm("ADD", "R0", $3);
86         emit_asm("MOV", $1, "R0");
87     }
88     | ID MINUSEQ expression NEWLINE {
89         char* temp = new_temp();
90         emit_tac(temp, $1, "-", $3);
91         emit_tac($1, temp, NULL, NULL);
92
93         emit_asm("MOV", "R0", $1);
94         emit_asm("SUB", "R0", $3);
95         emit_asm("MOV", $1, "R0");
96     }
97     | ID TIMESEQ expression NEWLINE {
98         char* temp = new_temp();
99         emit_tac(temp, $1, "*", $3);
100        emit_tac($1, temp, NULL, NULL);
101
102        emit_asm("MOV", "R0", $1);
103        emit_asm("MUL", "R0", $3);
104        emit_asm("MOV", $1, "R0");
105    }
106    | ID DIVEQ expression NEWLINE {
107        char* temp = new_temp();
108        emit_tac(temp, $1, "/", $3);
109        emit_tac($1, temp, NULL, NULL);
110
111        emit_asm("MOV", "R0", $1);
112        emit_asm("DIV", "R0", $3);
113        emit_asm("MOV", $1, "R0");
114    }

```

```

115 | ID MODEQ expression NEWLINE {
116 |     char* temp = new_temp();
117 |     emit_tac(temp, $1, "%", $3);
118 |     emit_tac($1, temp, NULL, NULL);
119 |
120 |     emit_asm("MOV", "R0", $1);
121 |     emit_asm("MOD", "R0", $3);
122 |     emit_asm("MOV", $1, "R0");
123 | }
124 | ID POWEREQ expression NEWLINE {
125 |     char* temp = new_temp();
126 |     emit_tac(temp, $1, "**", $3);
127 |     emit_tac($1, temp, NULL, NULL);
128 |
129 |     emit_asm("MOV", "R0", $1);
130 |     emit_asm("POW", "R0", $3);
131 |     emit_asm("MOV", $1, "R0");
132 | }
133 | NEWLINE
134 | ;
135
136 expression:
137     expression PLUS term {
138 |         char* temp = new_temp();
139 |         emit_tac(temp, $1, "+", $3);
140 |         $$ = temp;
141 |     }
142 |     expression MINUS term {
143 |         char* temp = new_temp();
144 |         emit_tac(temp, $1, "-", $3);
145 |         $$ = temp;
146 |     }
147 |     expression OR term {
148 |         char* temp = new_temp();
149 |         emit_tac(temp, $1, "||", $3);
150 |         $$ = temp;
151 |     }
152 |     term { $$ = $1; }
153 | ;
154
155 term:
156     term TIMES factor {
157 |         char* temp = new_temp();
158 |         emit_tac(temp, $1, "*", $3);
159 |         $$ = temp;
160 |     }
161 |     term DIVIDE factor {
162 |         char* temp = new_temp();
163 |         emit_tac(temp, $1, "/", $3);
164 |         $$ = temp;
165 |     }
166 |     term INTDIV factor {
167 |         char* temp = new_temp();
168 |         emit_tac(temp, $1, "//", $3);
169 |         $$ = temp;
170 |     }
171 |     term MOD factor {
172 |         char* temp = new_temp();

```



```

173     emit_tac(temp, $1, "%", $3);
174     $$ = temp;
175 }
176 | term AND factor {
177     char* temp = new_temp();
178     emit_tac(temp, $1, "&&", $3);
179     $$ = temp;
180 }
181 | term GT factor {
182     char* temp = new_temp();
183     emit_tac(temp, $1, ">", $3);
184     $$ = temp;
185 }
186 | term LT factor {
187     char* temp = new_temp();
188     emit_tac(temp, $1, "<", $3);
189     $$ = temp;
190 }
191 | factor { $$ = $1; }
192 ;
193
194 factor:
195     factor POWER unary {
196         char* temp = new_temp();
197         emit_tac(temp, $1, "**", $3);
198         $$ = temp;
199     }
200     | unary { $$ = $1; }
201 ;
202
203 unary:
204     NOT unary {
205         char* temp = new_temp();
206         emit_tac(temp, "!", $2, NULL);
207         $$ = temp;
208     }
209     | MINUS unary %prec UMINUS {
210         char* temp = new_temp();
211         emit_tac(temp, "-", $2, NULL);
212         $$ = temp;
213     }
214     | primary { $$ = $1; }
215 ;
216
217 primary:
218     LPAREN expression RPAREN { $$ = $2; }
219     | ID { $$ = $1; }
220     | NUM {
221         char* temp = (char*)malloc(strlen($1) + 2);
222         sprintf(temp, "%s", $1);
223         $$ = temp;
224     }
225 ;
226
227 %%
228
229 void yyerror(const char *s) {
230     fprintf(stderr, "Error: %s\n", s);

```

```

231 }
232
233 int main(int argc, char **argv) {
234     if (argc > 1) {
235         FILE *file = fopen(argv[1], "r");
236         if (file) {
237             yyin = file;
238         }
239     }
240     yyparse();
241     return 0;
242 }

```

### 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

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

### 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):

```
1 %{
2 #include "math_parser.tab.h"
3 #include <string.h>
4 %}
5
6 %%
7 [ \t]+      { /* ignore whitespace */ }
8 \n          { return NEWLINE; }
9 [0-9]+      { yylval.str = strdup(yytext); return NUM; }
10 "sqrt"      { return SQRT; }
11 "pow"       { return POW; }
12 "log"       { return LOG; }
13 "exp"       { return EXP; }
14 "sin"       { return SIN; }
15 "cos"       { return COS; }
16 "tan"       { return TAN; }
17 "abs"       { return ABS; }
18 [a-zA-Z][a-zA-Z0-9]* { yylval.str = strdup(yytext); return ID; }
19 "+"         { return PLUS; }
20 "-"         { return MINUS; }
21 "*"         { return TIMES; }
22 "/"         { return DIVIDE; }
23 "%"         { return MOD; }
24 "="        { return ASSIGN; }
25 "("         { return LPAREN; }
26 ")"         { return RPAREN; }
27 ",", "      { return COMMA; }
28 .           { /* ignore other characters */ }
29 %%
30
31 int yywrap() { return 1; }
```

## Parser File (math\_parser.y):

```
1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  extern int yylex();
7  void yyerror(const char *s);
8
9  int temp_count = 0;
10
11 char* new_temp() {
12     char* temp = (char*)malloc(10);
13     sprintf(temp, "t%d", ++temp_count);
14     return temp;
15 }
16
17 void emit_tac(const char* dest, const char* src1, const char* op,
18             const char* src2) {
19     if (src2) {
20         printf("%s = %s %s %s\n", dest, src1, op, src2);
21     } else if (op) {
22         printf("%s = %s %s\n", dest, op, src1);
23     } else {
24         printf("%s = %s\n", dest, src1);
25     }
26 }
27
28 void emit_asm(const char* op, const char* dest, const char* src) {
29     if (src) {
30         printf("%s %s , %s\n", op, dest, src);
31     } else {
32         printf("%s %s\n", op, dest);
33     }
34     printf("\n");
35 }
36
37 void emit_function_asm(const char* func, const char* arg1,
38                      const char* arg2) {
39     printf("MOV R0 , %s\n", arg1);
40     if (arg2) {
41         printf("%s R0 , %s\n", func, arg2);
42     } else {
43         printf("%s R0\n", func);
44     }
45     printf("\n");
46 }
47 %}
48
49 %union {
50     char* str;
51 }
52
53 %token <str> ID NUM
54 %token PLUS MINUS TIMES DIVIDE MOD ASSIGN
55 %token LPAREN RPAREN COMMA NEWLINE
56 %token SQRT POW LOG EXP SIN COS TAN ABS
```



```

57
58 %type <str> expression term factor function_call
59
60 %left PLUS MINUS
61 %left TIMES DIVIDE MOD
62
63 %%
64
65 program:
66     statement_list
67     ;
68
69 statement_list:
70     statement
71     | statement_list statement
72     ;
73
74 statement:
75     ID ASSIGN expression NEWLINE {
76         emit_tac($1, $3, NULL, NULL);
77         emit_asm("MOV", "R0", $3);
78         emit_asm("MOV", $1, "R0");
79     }
80     | ID ASSIGN expression {
81         emit_tac($1, $3, NULL, NULL);
82         emit_asm("MOV", "R0", $3);
83         emit_asm("MOV", $1, "R0");
84     }
85     | NEWLINE
86     ;
87
88 expression:
89     expression PLUS term {
90         char* temp = new_temp();
91         emit_tac(temp, $1, "+", $3);
92         $$ = temp;
93     }
94     | expression MINUS term {
95         char* temp = new_temp();
96         emit_tac(temp, $1, "-", $3);
97         $$ = temp;
98     }
99     | term { $$ = $1; }
100     ;
101
102 term:
103     term TIMES factor {
104         char* temp = new_temp();
105         emit_tac(temp, $1, "*", $3);
106         $$ = temp;
107     }
108     | term DIVIDE factor {
109         char* temp = new_temp();
110         emit_tac(temp, $1, "/", $3);
111         $$ = temp;
112     }
113     | term MOD factor {
114         char* temp = new_temp();

```

```

115         emit_tac(temp, $1, "%", $3);
116         $$ = temp;
117     }
118     | factor { $$ = $1; }
119     ;
120
121 factor:
122     function_call { $$ = $1; }
123     | LPAREN expression RPAREN { $$ = $2; }
124     | ID { $$ = $1; }
125     | NUM {
126         char* temp = (char*)malloc(strlen($1) + 2);
127         sprintf(temp, "%s", $1);
128         $$ = temp;
129     }
130     | MINUS factor {
131         char* temp = new_temp();
132         printf("%s = -%s\n", temp, $2);
133         printf("MOV R0 , %s\n", $2);
134         printf("NEG R0\n\n");
135         $$ = temp;
136     }
137     ;
138
139 function_call:
140     SQRT LPAREN expression RPAREN {
141         char* temp = new_temp();
142         printf("%s = sqrt ( %s )\n", temp, $3);
143         emit_function_asm("SQRT", $3, NULL);
144         $$ = temp;
145     }
146     | POW LPAREN expression COMMA expression RPAREN {
147         char* temp = new_temp();
148         printf("%s = pow (%s , %s)\n", temp, $3, $5);
149         emit_function_asm("POW", $3, $5);
150         $$ = temp;
151     }
152     | LOG LPAREN expression RPAREN {
153         char* temp = new_temp();
154         printf("%s = log ( %s )\n", temp, $3);
155         emit_function_asm("LOG", $3, NULL);
156         $$ = temp;
157     }
158     | EXP LPAREN expression RPAREN {
159         char* temp = new_temp();
160         printf("%s = exp ( %s )\n", temp, $3);
161         emit_function_asm("EXP", $3, NULL);
162         $$ = temp;
163     }
164     | SIN LPAREN expression RPAREN {
165         char* temp = new_temp();
166         printf("%s = sin ( %s )\n", temp, $3);
167         emit_function_asm("SIN", $3, NULL);
168         $$ = temp;
169     }
170     | COS LPAREN expression RPAREN {
171         char* temp = new_temp();
172         printf("%s = cos ( %s )\n", temp, $3);

```

```

173     emit_function_asm("COS", $3, NULL);
174     $$ = temp;
175 }
176 | TAN LPAREN expression RPAREN {
177     char* temp = new_temp();
178     printf("%s = tan ( %s )\n", temp, $3);
179     emit_function_asm("TAN", $3, NULL);
180     $$ = temp;
181 }
182 | ABS LPAREN expression RPAREN {
183     char* temp = new_temp();
184     printf("%s = abs ( %s )\n", temp, $3);
185     emit_function_asm("ABS", $3, NULL);
186     $$ = temp;
187 }
188 ;
189
190 %%
191
192 void yyerror(const char *s) {
193     fprintf(stderr, "Error: %s\n", s);
194 }
195
196 int main(int argc, char **argv) {
197     if (argc > 1) {
198         FILE *file = fopen(argv[1], "r");
199         if (file) {
200             yyin = file;
201         }
202     }
203     yyparse();
204     return 0;
205 }

```

### 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 )

```

$$f = \text{abs} ( - a + b ) / \exp (2)$$

### 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 #.