Compiler Design Lab Assignment - 5

Name: Apurba Koirala

22BCE3799

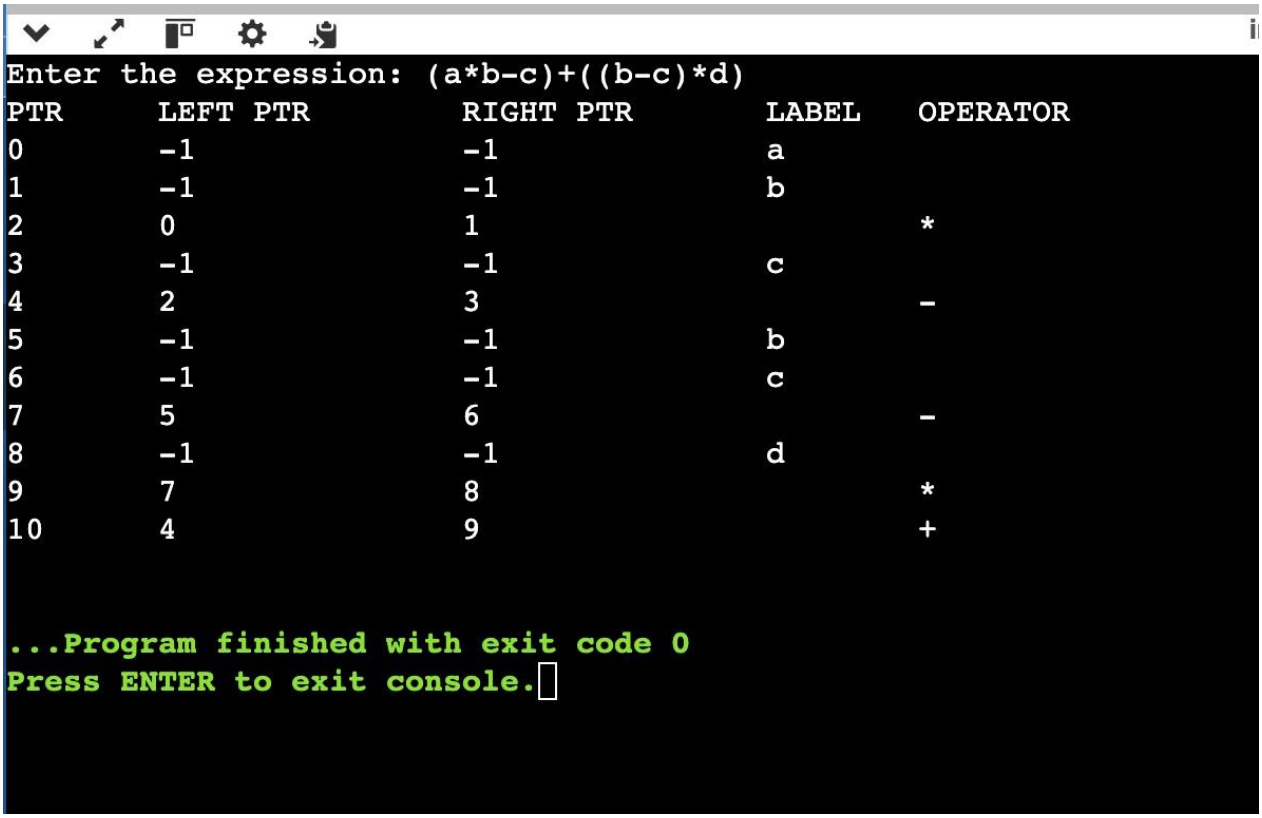
Q1. To write a C program to construct of DAG(Directed Acyclic Graph)

Code:

```
#include <iostream>
#include <stack>
#include <map>
#include        <vector>
#include       <cctype>
using namespace std;
struct Node { int
left, right; char
label;
char op;
};
vector<Node> dag; map<char,
int> node_map; void
printDAG() {
cout << "PTR\tLEFT PTR\tRIGHT PTR\tLABEL\tOPERATOR\n";
for (int i = 0; i < dag.size(); ++i) {
cout << i << "\t" << dag[i].left << "\t\t" << dag[i].right << "\t\t"
<< dag[i].label << "\t" << dag[i].op << "\n";
}
}
int main() { string expression;
cout << "Enter the expression:
"; cin >> expression; stack<int>
operandStack; stack<char>
operatorStack; for (char ch :
expression) { if (isalnum(ch)) {
Node node = {-1,
-1, ch, '\0'};
int index = dag.size();
dag.push_back(node); node_map[ch]
= index;
operandStack.push(index);
} else if (ch == '(') {
operatorStack.push(ch);
} else if (ch == ')') {
while (!operatorStack.empty() && operatorStack.top() != '(') {
char op = operatorStack.top(); operatorStack.pop(); int right
```

```
= operandStack.top(); operandStack.pop(); int left =
operandStack.top();
operandStack.pop();
Node node = {left, right, '\0', op};
int index = dag.size(); dag.push_back(node);
operandStack.push(index);
}
operatorStack.pop(); // Pop the '('
} else {
while (!operatorStack.empty() && operatorStack.top() != '(') {
char op = operatorStack.top(); operatorStack.pop(); int right
= operandStack.top(); operandStack.pop(); int left =
operandStack.top();
operandStack.pop();
Node node = {left, right, '\0', op};
int index = dag.size(); dag.push_back(node);
operandStack.push(index);
}
operatorStack.push(ch);
}
}
while (!operatorStack.empty()) {
char op = operatorStack.top();
operatorStack.pop(); int right =
operandStack.top();
operandStack.pop(); int left =
operandStack.top();
operandStack.pop();
Node node = {left, right, '\0', op};
int index = dag.size(); dag.push_back(node);
operandStack.push(index); }
printDAG(); return
0;
}
```

Output:



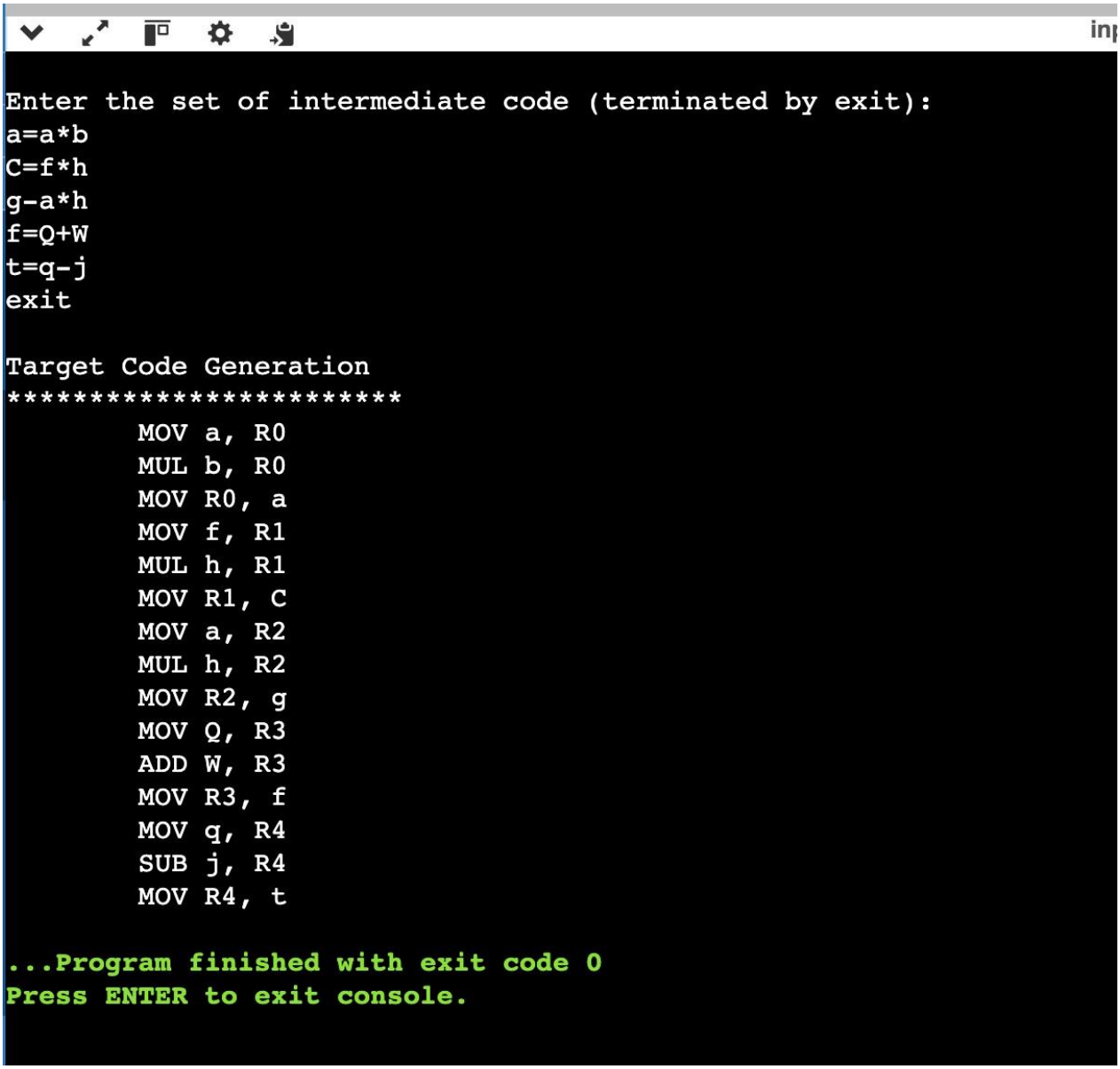
Q2. Write code to implement the back end of the compiler which takes the three-address code and produces the 8086 assembly language instructions that can be assembled and run using a 8086 assembler. The target assembly instructions can be simple move, add, sub, jump. Also, simple addressing modes are used.

CODE:

```
#include <stdio.h>
#include <string.h>
#include <conio.h> void
main() {
char icode[10][30], str[20], opr[10];
int i = 0;
// Accepting intermediate code input from user
printf("\nEnter the set of intermediate code (terminated by exit):\n");
do { scanf("%s"
, icode[i]);
} while (strcmp(icode[i++], "exit") != 0);
// Start target code generation
printf("\nTarget Code Generation");
```

```
printf("\n************************"); //
Reset i for target code generation
i = 0;
// Generate target code based on intermediate code do
{
strcpy(str, icode[i]);
// Handle the operator in the intermediate code
switch (str[3]) {
case '+': strcpy(opr,
"ADD "); break;
case '-': strcpy(opr,
"SUB "); break;
case '*': strcpy(opr,
"MUL "); break;
case '/': strcpy(opr,
"DIV "); break;
default:
strcpy(opr, "UNKNOWN ");
break;
}
// Printing assembly instructions based on the intermediate code
printf("\n\tMOV %c, R%d"
, str[2], i); // Move operand into a register printf("\n\t%s%c,
R%d"
, opr, str[4], i); // Perform the operation printf("\n\tMOV
R%d, %c"
, i, str[0]); // Move the result to the left
} while (strcmp(icode[++i], "exit") != 0); // Loop until 'exit' is encountered getch();
// Wait for user input (useful in DOS-based systems)
}
```

OUTPUT:



Q3. Program to recognize a valid variable which starts with a Letter followed by any number of letters or digits.

CODE:
////////// Lex Code:

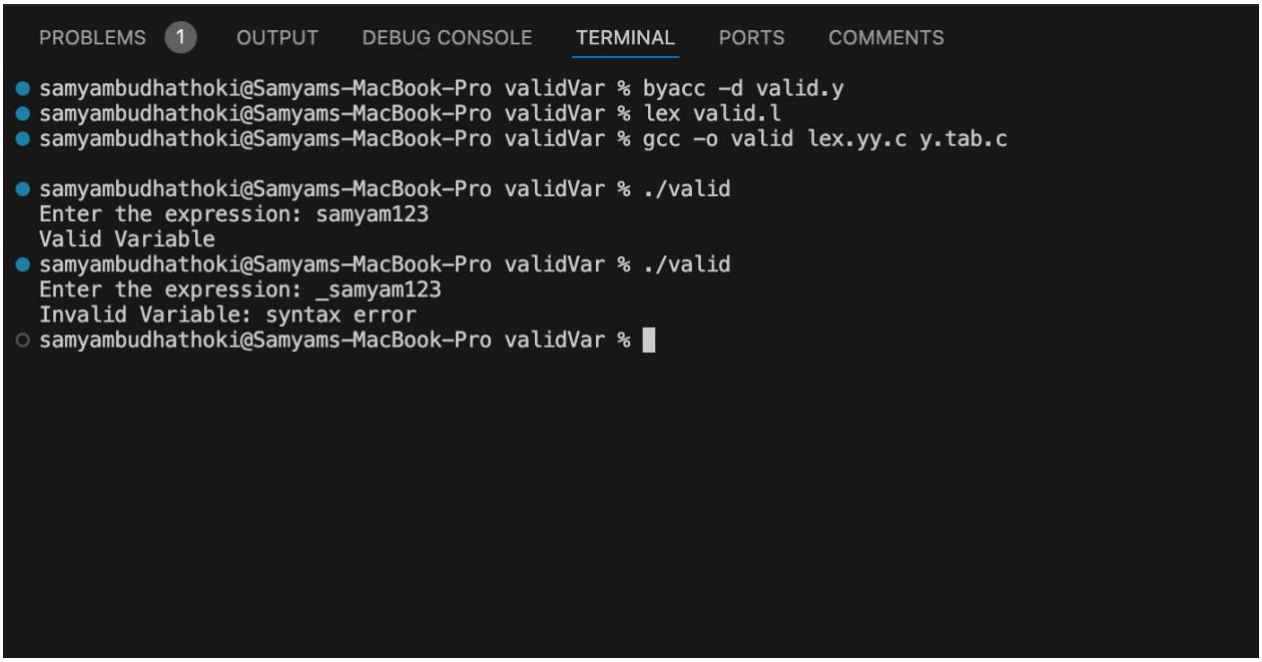
```
%{
#include "y.tab.h"
%}
```

```
%%
[a-zA-Z] { return ALPHA; }
[0-9]+   { return NUMBER; }
"\n"    { return ENTER; }
.      { return ER; }
%%
int yywrap() { // Explicit return type
   return 1;
}
```

////////////////// YACC Code:

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ALPHA NUMBER ENTER ER
%%
var: v ENTER {
printf("Valid Variable\n");
   exit(0);
}
v: ALPHA exp1
exp1: ALPHA exp1
| NUMBER exp1
   | /* empty */
%%
void yyerror(const char *s) { // Explicit return type and parameter
   printf("Invalid Variable: %s\n", s);
}

int main() {
   printf("Enter the expression: ");
yyparse();
   return 0; // Ensure main returns an int
}
```

OUTPUT:



Q4. Program to recognize a valid arithmetic expression that uses operator +, - , * and /.

CODE:

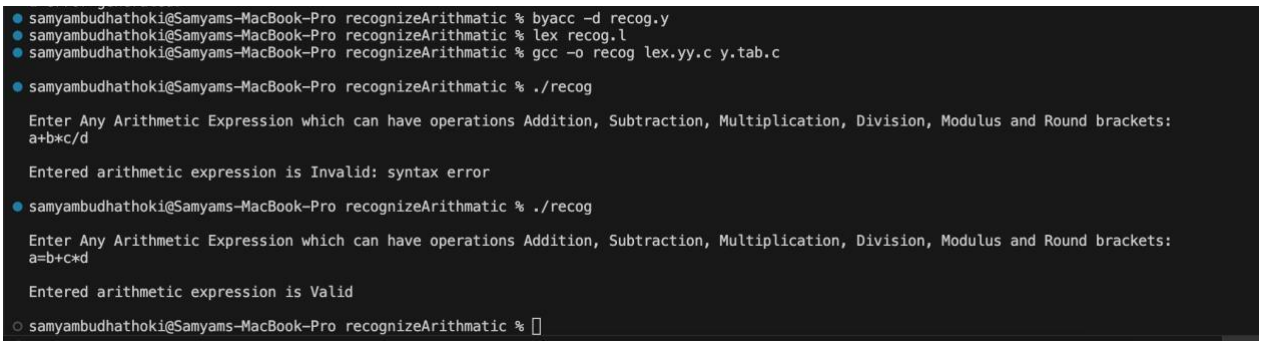
//////////////////Lex CODE:

```
%{
#include       <stdio.h>
#include        "y.tab.h"
extern YYSTYPE yylval;
%}

%%
[a-zA-Z]+   { yylval.var = strdup(yytext); return VARIABLE; }
[0-9]+      { yylval.num = atoi(yytext); return NUMBER; }
[\t]        ; // ignore tabs
[\n]        { return 0; } // newline (end of input)
.           { return yytext[0]; } // return any other character as is
%%
```

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

//////////////YACC CODE:

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

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

%}

%union {
   int num;
char *var;
}

%token <num> NUMBER
%token <var> VARIABLE

%left '+' '-'
%left '*' '/' '%'
%left '(' ')'

%%

S: VARIABLE '=' E {
   printf("\nEntered arithmetic expression is Valid\n\n");
}
;

E: E '+' E
 | E '-' E
 | E '*' E
 | E '/' E
 | E '%' E
```

```
 | '(' E ')'
 | NUMBER
 | VARIABLE
 ;

%%

int main() {
   printf("\nEnter Any Arithmetic Expression which can have operations
Addition, Subtraction, Multiplication, Division, Modulus and Round
brackets:\n");    yyparse();
   return 0;
}

void yyerror(const char *s) {
   printf("\nEntered arithmetic expression is Invalid: %s\n\n", s);
}
```

OUTPUT:



Program to implement simple calculator using Lex and YACC in LLVM.

CODE:

Lexical Analyzer:

1. calc.l:

```
%{
```

```
#include<stdio.h
>              #include
"y.tab.h"  extern
int yylval;   %}


%%
[0-9]+ {
   yylval=atoi(yytext);     return NUMBER;


           }
[\t] ;


[\n] return 0;


. return yytext[0];


%%


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

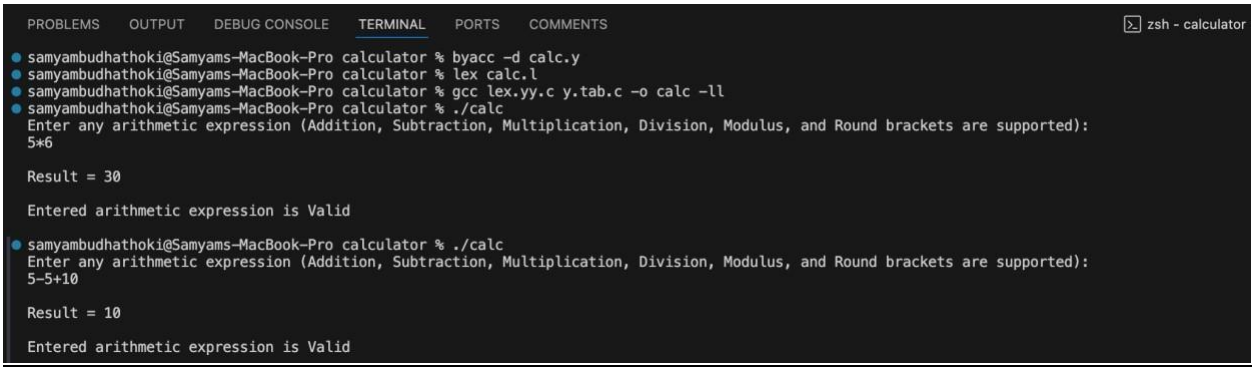
Parser Source code:

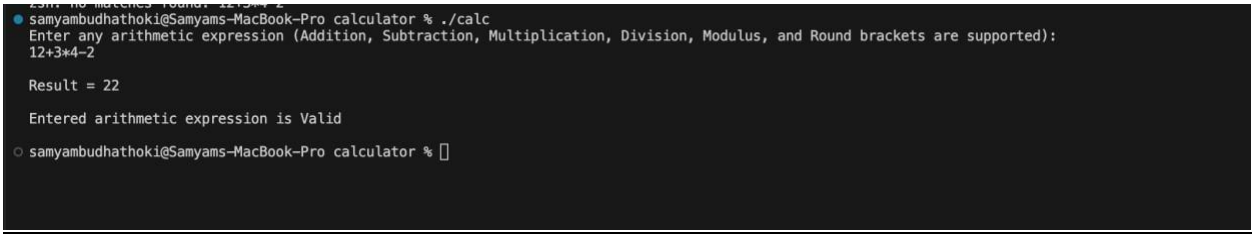
2. calc.y:

```
%{
#include <stdio.h>
#include <stdlib.h> // Required for atoi
int flag = 0;
%}

%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'



%%
ArithmeticExpression:
   E {
      printf("\nResult = %d\n", $1);
return 0;
   }
;

E:
   E '+' E { $$ = $1 + $3; }
   | E '-' E { $$ = $1 - $3; }
   | E '*' E { $$ = $1 * $3; }
   | E '/' E { $$ = $1 / $3; }
   | E '%' E { $$ = $1 % $3; }
```

```
   | '(' E ')' { $$ = $2; }
   | NUMBER { $$ = $1; }
;

%%

int main() {
   printf("Enter any arithmetic expression (Addition, Subtraction, Multiplication,
Division,  Modulus, and Round brackets are supported):\n");    yyparse();    if (flag == 0)
printf("\nEntered arithmetic expression is Valid\n\n");    return 0;
}

void yyerror(const char *msg) {
printf("\nError: %s\n", msg);
   flag = 1;
}
```

Output:


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
COMMENTS
zsh - calculator
samyambudhathoki@Samyams-MacBook-Pro calculator % byacc -d calc.y
samyambudhathoki@Samyams-MacBook-Pro calculator % lex calc.l
samyambudhathoki@Samyams-MacBook-Pro calculator % gcc lex.yy.c y.tab.c -o calc -ll
samyambudhathoki@Samyams-MacBook-Pro calculator % ./calc
Enter any arithmetic expression (Addition, Subtraction, Multiplication, Division, Modulus, and Round brackets are supported):
5*6
Result = 30
Entered arithmetic expression is Valid
samyambudhathoki@Samyams-MacBook-Pro calculator % ./calc
Enter any arithmetic expression (Addition, Subtraction, Multiplication, Division, Modulus, and Round brackets are supported):
5-5+10
Result = 10
Entered arithmetic expression is Valid


samyambudhathoki@Samyams-MacBook-Pro calculator % ./calc
Enter any arithmetic expression (Addition, Subtraction, Multiplication, Division, Modulus, and Round brackets are supported):
12+3*4-2
Result = 22
Entered arithmetic expression is Valid
samyambudhathoki@Samyams-MacBook-Pro calculator %