(works)

Write a Program to perform lexical analysis on int a, b, c;

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

int main()
{
   string line = "int a, b, c;";
   vector<string> tokens;
   string token = "";
   bool is_declaring = false;

   for (int i = 0; i < line.length(); i++) {
      if (line[i] == ' ' || line[i] == '\t' || line[i] == '\n') { // skip whitespace
         continue;
      }
      else if (line[i] == ',') { // end of variable declaration
         is_declaring = true;
         if (token != "") { // check for empty token
            tokens.push_back(token);
            token = "";
         }
      }
      else if (line[i] == ';') { // end of statement
         is_declaring = false;
```

```
            if (token != "") { // check for empty token
                tokens.push_back(token);
                token = "";
            }
        }
        else { // add character to current token
            token += line[i];
            if (!is_declaring) { // if not declaring, add token to vector and reset it
                tokens.push_back(token);
                token = "";
            }
        }
    }

    // output the tokens
    cout << "Tokens: ";
    for (int i = 0; i < tokens.size(); i++) {
        cout << tokens[i] << " ";
    }
    cout << endl;

    return 0;
}
```

(works)

Write a Program to display Quadruples in Three Address Code

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

struct Quadruple {
   string op;
   string arg1;
   string arg2;
   string result;
};

int main()
{
   vector<Quadruple> quadruples = {
      {"+", "a", "b", "t1"},
      {"*", "t1", "c", "t2"},
      {"-", "t2", "d", "e"},
      {"/", "e", "f", "g"}
   };

   // display the quadruples
   cout << "Quadruples in Three Address Code:" << endl;
   for (int i = 0; i < quadruples.size(); i++) {
      cout << i << ": " << quadruples[i].op << " ";
      if (quadruples[i].arg1 != "") {
```

```
            cout << quadruples[i].arg1 << " ";
        }
        if (quadruples[i].arg2 != "") {
            cout << quadruples[i].arg2 << " ";
        }
        cout << quadruples[i].result << endl;
    }

    return 0;
}
```

(works)

Write a Program to implement constant propagation in code optimization

```
#include <iostream>
#include <string>
#include <vector>
#include <unordered_map>
using namespace std;

struct Expression {
   string op;
   string arg1;
   string arg2;
   string result;
};

unordered_map<string, int> constants = {
   {"a", 5},
   {"b", 10},
   {"c", 2}
};

int main()
{
   vector<Expression> expressions = {
      {"+", "a", "b", "t1"},
      {"*", "t1", "c", "t2"},
      {"-", "t2", "b", "t3"},
      {"/", "t3", "c", "t4"}
   };
```

```
    // perform constant propagation
    for (int i = 0; i < expressions.size(); i++) {
        Expression expr = expressions[i];
        if (constants.count(expr.arg1)) {
            expr.arg1 = to_string(constants[expr.arg1]);
        }
        if (constants.count(expr.arg2)) {
            expr.arg2 = to_string(constants[expr.arg2]);
        }
        if (expr.op == "+" && constants.count(expr.arg1) && constants.count(expr.arg2)) {
            expressions[i].result = to_string(constants[expr.arg1] + constants[expr.arg2]);
        }
        else if (expr.op == "-" && constants.count(expr.arg1) && constants.count(expr.arg2)) {
            expressions[i].result = to_string(constants[expr.arg1] - constants[expr.arg2]);
        }
        else if (expr.op == "*" && constants.count(expr.arg1) && constants.count(expr.arg2)) {
            expressions[i].result = to_string(constants[expr.arg1] * constants[expr.arg2]);
        }
        else if (expr.op == "/" && constants.count(expr.arg1) && constants.count(expr.arg2)) {
            expressions[i].result = to_string(constants[expr.arg1] / constants[expr.arg2]);
        }
    }
    // display the optimized code
    cout << "Optimized Code:" << endl;
    for (int i = 0; i < expressions.size(); i++) {
        cout << expressions[i].result << " = " << expressions[i].arg1 << " " << expressions[i].op
<< " " << expressions[i].arg2 << endl;
    }
   return 0;
}
```

(works)

Write a Program to implement Recursive MACROS

```
public class RecursiveMacro {

   // define function to implement macro
   public static int macro(int n) {
      if(n == 0 || n == 1) {
         return 1;
      } else {
         return macro(n-1) + macro(n-2);
      }
   }

   public static void main(String[] args) {
      // define macro
      int fib = macro(10);

      // print macro value
      System.out.println("Fibonacci number: " + fib);
   }
}
```

(works)

Write a Program to display MNT and MDT for given ALP

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
using namespace std;

int main() {
   vector<string> mnt; // Macro Name Table
   vector<string> mdt; // Macro Definition Table

   // Open the input file
   ifstream inputFile("input.asm");

   // Read the input file line by line
   string line;
   while (getline(inputFile, line)) {
      // Check if the line is a macro definition
      if (line.find("MACRO") != string::npos) {
         // Extract the macro name
         string macroName = line.substr(0, line.find(" "));

         // Add the macro name to MNT and get its index
         int mntIndex = mnt.size();
```

```
            mnt.push_back(macroName);

            // Read the macro definition
            string macroDefinition;
            while (getline(inputFile, line) && line != "MEND") {
                macroDefinition += line + "\n";
            }

            // Add the macro definition to MDT
            mdt.push_back(macroDefinition);

            // Print the macro definition with its index
            cout << mntIndex << "\t" << macroDefinition;
        }
    }

    // Print MNT
    cout << "\nMacro Name Table (MNT):\n";
    for (int I = 0; I < mnt.size(); i++) {
        cout << I << "\t" << mnt[i] << endl;
    }

    // Print MDT
    cout << "\nMacro Definition Table (MDT):\n";
    for (int I = 0; I < mdt.size(); i++) {
        cout << I << "\t" << mdt[i] << endl;
```

```
    }

    // Close the input file
    inputFile.close();

    return 0;
}
```

input.asm

```
MOV AX, 0
MOV BX, 0
MACRO ADD_VALUES
    ADD AX, #1
    ADD BX, #2
MEND
ADD_VALUES
ADD_VALUES
MOV CX, AX
MOV DX, BX
```

(work)

Write a Program to implement parameterized MACROS

```
#include <iostream>

#define SQUARE(x) ((x) * (x))
#define CUBE(x) ((x) * (x) * (x))

int main() {
  int a = 5;
  std::cout << “Square of “ << a << “ is “ << SQUARE(a) << std::endl;
  std::cout << “Cube of “ << a << “ is “ << CUBE(a) << std::endl;

  // parameterized macros
  #define POWER(x, n) ({ \
    int result = 1; \
    for (int I = 0; I < (n); i++) { \
      result *= (x); \
    } \
    result; \
  })

  int b = 2;
  int c = 3;
  std::cout << b << “ to the power of “ << c << “ is “ << POWER(b, c) << std::endl;

  return 0;
}
```

(works)

write a program to implement code movement in code optimization

```
#include <iostream>

int compute(int a, int b, int c, int d) {
    int x = a + b;
    int y = c + d;
    int z = x * y;
    return z;
}

int optimized_compute(int a, int b, int c, int d) {
    int x = a + b;
    int y = c + d;
    int z = x * y;
    return z;
}

int main() {
    int a = 2, b = 3, c = 4, d = 5;

    // Original computation
    int z1 = compute(a, b, c, d);
    std::cout << "Original computation result: " << z1 << std::endl;
```

```
    // Optimized computation
    int z2 = optimized_compute(a, b, c, d);
    std::cout << "Optimized computation result: " << z2 << std::endl;

    return 0;
}
```

display symbol table after pass 1 of assembly for given ALP

They will give ALP and from that ALP we have to display symbol table

You can see youtube video to learn to draw Symbol Table for ALP

(works)

Write a Program to implement nested MACROS

```
#include <stdio.h>

#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define ABS(x) ((x) < 0 ? -(x) : (x))
#define MAX_ABS(a, b) MAX(ABS(a), ABS(b))

int main() {
  int x = -5;
  int y = 10;
  printf("The maximum absolute value of %d and %d is %d\n", x, y, MAX_ABS(x, y));

  return 0;
}
```

(works)

Write a program to implement common sub expression elimination in code optimization

```
#include <iostream>
#include <unordered_map>
#include <string>

using namespace std;

string eliminate_common_subexpressions(string code) {
   // Initialize an empty unordered_map to store expressions and their computed values
   unordered_map<string, string> computed_values;

   // Initialize an empty string to store the optimized code
   string optimized_code = "";

   // Split the code into individual statements using newline character as delimiter
   size_t pos = 0;
   string delimiter = "\n";
   while ((pos = code.find(delimiter)) != string::npos) {
      string statement = code.substr(0, pos);
      code.erase(0, pos + delimiter.length());

      // Check if the statement is an assignment statement
      if (statement.find("=") != string::npos) {
         // Split the statement into left-hand side and right-hand side using the equal sign as
delimiter
         size_t equal_pos = statement.find("=");
         string lhs = statement.substr(0, equal_pos);
```

```
        string rhs = statement.substr(equal_pos + 1);

        // Check if the right-hand side is already computed and stored in the unordered_map
        if (computed_values.find(rhs) != computed_values.end()) {
            // Use the computed value instead of computing the expression again
            optimized_code += lhs + " = " + computed_values[rhs] + ";\n";
        } else {
            // Compute the value of the expression and store it in the unordered_map
            optimized_code += statement + "\n";
            computed_values[rhs] = lhs;
        }
    } else {
        // The statement is not an assignment statement, so it doesn't have any sub-expressions to eliminate
        optimized_code += statement + "\n";
    }
  }

  return optimized_code;
}

int main() {
  // Example code to optimize
  string code = "a = 2 + 3;\nb = a * 4;\nc = a + b;\nd = a + b;\n";

  // Optimize the code by eliminating common sub-expressions
  string optimized_code = eliminate_common_subexpressions(code);

  // Print the optimized code
```

```
    cout << optimized_code;

    return 0;
}
```

(works)

Write a program to implement Recursive Descent Parsing Techniques

```
#include <stdio.h>
#include <string.h>

#define SUCCESS 1
#define FAILED 0

int E(), Edash(), T(), Tdash(), F();

const char *cursor;
char string[64];

int main()
{
        puts("Enter the string");
        // scanf("%s", string);
        sscanf("i+(i+i)*i", "%s", string);
        cursor = string;
        puts("");
        puts("Input             Action");
        puts("--------------------------------");

        if (E() && *cursor == '\0') {
                puts("--------------------------------");
```

```
                puts("String is successfully parsed");
                return 0;
        } else {
                puts("------------------------------");
                puts("Error in parsing String");
                return 1;
        }
}

int E()
{
        printf("%-16s E -> T E'\n", cursor);
        if (T()) {
                if (Edash())
                        return SUCCESS;
                else
                        return FAILED;
        } else
                return FAILED;
}

int Edash()
{
        if (*cursor == '+') {
                printf("%-16s E' -> + T E'\n", cursor);
                cursor++;
```

```
                if (T()) {
                        if (Edash())
                                return SUCCESS;
                        else
                                return FAILED;
                } else
                        return FAILED;
        } else {
                printf("%-16s E' -> $\n", cursor);
                return SUCCESS;
        }
}

int T()
{
        printf("%-16s T -> F T'\n", cursor);
        if (F()) {
                if (Tdash())
                        return SUCCESS;
                else
                        return FAILED;
        } else
                return FAILED;
}

int Tdash()
```

```
{
        if (*cursor == '*') {
                printf("%-16s T' -> * F T'\n", cursor);
                cursor++;
                if (F()) {
                        if (Tdash())
                                return SUCCESS;
                        else
                                return FAILED;
                } else
                        return FAILED;
        } else {
                printf("%-16s T' -> $\n", cursor);
                return SUCCESS;
        }
}

int F()
{
        if (*cursor == '(') {
                printf("%-16s F -> ( E )\n", cursor);
                cursor++;
                if (E()) {
                        if (*cursor == ')') {
                                cursor++;
                                return SUCCESS;
```

```
                } else
                        return FAILED;
            } else
                return FAILED;
        } else if (*cursor == 'i') {
            cursor++;
            printf("%-16s F -> i\n", cursor);
            return SUCCESS;
        } else
            return FAILED;
}
```

(works)

Write a Program to display triples in three address code

```
#include <iostream>
#include <vector>
using namespace std;

struct Triple {
   string op;
   string arg1;
   string arg2;
};

void display_tac(vector<Triple> triples) {
   for (int i = 0; i < triples.size(); i++) {
      cout << i+1 << ". ";
      cout << triples[i].op << " ";
      cout << triples[i].arg1 << " ";
      cout << triples[i].arg2 << endl;
   }
}

int main() {
   vector<Triple> triples = {{"+", "a", "b"}, {"=", "c", "1"}, {"*", "d", "e"}};
   display_tac(triples);
   return 0;
}
```

(works)

Write a Program to remove left recursion

```
#include<iostream>
#include<string>
using namespace std;
int main()
{  string ip,op1,op2,temp;
   int sizes[10] = {};
   char c;
   int n,j,l;
   cout<<"Enter the Parent Non-Terminal : ";
   cin>>c;
   ip.push_back(c);
   op1 += ip + "\'->";
   ip += "->";
   op2+=ip;
   cout<<"Enter the number of productions : ";
   cin>>n;
   for(int i=0;i<n;i++)
   {   cout<<"Enter Production "<<i+1<<" : ";
      cin>>temp;
      sizes[i] = temp.size();
```

```
        ip+=temp;
        if(i!=n-1)
            ip += "|";
    }
    cout<<"Production Rule : "<<ip<<endl;
    for(int i=0,k=3;i<n;i++)
    {
        if(ip[0] == ip[k])
        {
            cout<<"Production "<<i+1<<" has left
recursion."<<endl;
            if(ip[k] != '#')
            {
                for(l=k+1;l<k+sizes[i];l++)
                    op1.push_back(ip[l]);
                k=l+1;
                op1.push_back(ip[0]);
                op1 += "\'|";
            }
        }
        else
        {
```

```
        cout<<"Production "<<i+1<<" does not have left
recursion."<<endl;
        if(ip[k] != '#')
        {
           for(j=k;j<k+sizes[i];j++)
              op2.push_back(ip[j]);
           k=j+1;
           op2.push_back(ip[0]);
           op2 += "\'|";
        }
        else
        {
           op2.push_back(ip[0]);
           op2 += "\'";
        }}}
   op1 += "#";
   cout<<op2<<endl;
   cout<<op1<<endl;
   return 0;}
```

OUTPUT:

Enter the Parent Non-Terminal: E
Enter the number of productions : 3
Enter Production 1: E+T
Enter Production 2: T
Enter Production 3: #
Production Rule : E->E+T|T|#
Production 1 has left recursion.
Production 2 does not have left recursion.
Production 3 does not have left recursion.
E->TE'|E'

E'->+TE'|#

(works)

Write a Program to perform lexical analysis on c = a*b+d

```
#include <iostream>
#include <string>
#include <sstream>

int main() {
  std::string input = "c = a*b+d";
  std::stringstream ss(input);
  std::string token;

  while (ss >> token) {
    if (token == "c") {
      std::cout << "Identifier: " << token << std::endl;
    } else if (token == "=") {
      std::cout << "Assignment operator" << std::endl;
    } else if (token == "a" || token == "b" || token == "d") {
      std::cout << "Identifier: " << token << std::endl;
    } else if (token == "*" || token == "+") {
      std::cout << "Arithmetic operator" << std::endl;
    } else {
      std::cout << "Invalid token: " << token << std::endl;
      return 1;
    }
  }

  return 0;
}
```

(works)

Write a Program to implement any parsing techniques

Recursive descent parser

```
#include <stdio.h>
#include <string.h>

#define SUCCESS 1
#define FAILED 0

int E(), Edash(), T(), Tdash(), F();

const char *cursor;
char string[64];

int main()
{
        puts("Enter the string");
        // scanf("%s", string);
        sscanf("i+(i+i)*i", "%s", string);
        cursor = string;
        puts("");
        puts("Input             Action");
        puts("--------------------------------");

        if (E() && *cursor == '\0') {
                puts("--------------------------------");
```

```
                puts("String is successfully parsed");
                return 0;
        } else {
                puts("------------------------------");
                puts("Error in parsing String");
                return 1;
        }
}

int E()
{
        printf("%-16s E -> T E'\n", cursor);
        if (T()) {
                if (Edash())
                        return SUCCESS;
                else
                        return FAILED;
        } else
                return FAILED;
}

int Edash()
{
        if (*cursor == '+') {
                printf("%-16s E' -> + T E'\n", cursor);
                cursor++;
```

```
                if (T()) {
                        if (Edash())
                                return SUCCESS;
                        else
                                return FAILED;
                } else
                        return FAILED;
        } else {
                printf("%-16s E' -> $\n", cursor);
                return SUCCESS;
        }
}

int T()
{
        printf("%-16s T -> F T'\n", cursor);
        if (F()) {
                if (Tdash())
                        return SUCCESS;
                else
                        return FAILED;
        } else
                return FAILED;
}

int Tdash()
```

```
{
        if (*cursor == '*') {
                printf("%-16s T' -> * F T'\n", cursor);
                cursor++;
                if (F()) {
                        if (Tdash())
                                return SUCCESS;
                        else
                                return FAILED;
                } else
                        return FAILED;
        } else {
                printf("%-16s T' -> $\n", cursor);
                return SUCCESS;
        }
}

int F()
{
        if (*cursor == '(') {
                printf("%-16s F -> ( E )\n", cursor);
                cursor++;
                if (E()) {
                        if (*cursor == ')') {
                                cursor++;
                                return SUCCESS;
```

```
                } else
                        return FAILED;
            } else
                return FAILED;
        } else if (*cursor == 'i') {
            cursor++;
            printf("%-16s F -> i\n", cursor);
            return SUCCESS;
        } else
            return FAILED;
}
```

(works)

Write a Program to implement Conditional MACROS

```
#include <iostream>

#define PI 3.14159
#define SQUARE(x) ((x) * (x))

#define DEBUG 1

int main() {
   double radius = 2.5;
   double area = PI * SQUARE(radius);

   #if DEBUG
   std::cout << "The area of a circle with radius " << radius << " is " << area << std::endl;
   #endif

   #ifndef DEBUG
   std::cout << area << std::endl;
   #endif

   return 0;
}
```

(works)

Write a program to find FIRST and FOLLOW of given grammar(Note epsilon is equivalent to 0)

```
// C program to calculate the First and
// Follow sets of a given grammar
#include <ctype.h>
#include <stdio.h>
#include <string.h>

// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);

// Function to calculate First
void findfirst(char, int, int);

int count, n = 0;

// Stores the final result
// of the First Sets
char calc_first[10][100];

// Stores the final result
// of the Follow Sets
char calc_follow[10][100];
int m = 0;
```

```
// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char** argv)
{
        int jm = 0;
        int km = 0;
        int i, choice;
        char c, ch;
        count = 8;

        // The Input grammar
        strcpy(production[0], "X=TnS");
        strcpy(production[1], "X=Rm");
        strcpy(production[2], "T=q");
        strcpy(production[3], "T=#");
        strcpy(production[4], "S=p");
        strcpy(production[5], "S=#");
        strcpy(production[6], "R=om");
        strcpy(production[7], "R=ST");
```

```
int kay;
char done[count];
int ptr = -1;

// Initializing the calc_first array
for (k = 0; k < count; k++) {
        for (kay = 0; kay < 100; kay++) {
                calc_first[k][kay] = '!';
        }
}
int point1 = 0, point2, xxx;

for (k = 0; k < count; k++) {
        c = production[k][0];
        point2 = 0;
        xxx = 0;

        // Checking if First of c has
        // already been calculated
        for (kay = 0; kay <= ptr; kay++)
                if (c == done[kay])
                        xxx = 1;

        if (xxx == 1)
                continue;
```

```
// Function call
findfirst(c, 0, 0);
ptr += 1;

// Adding c to the calculated list
done[ptr] = c;
printf("\n First(%c) = { ", c);
calc_first[point1][point2++] = c;

// Printing the First Sets of the grammar
for (i = 0 + jm; i < n; i++) {
        int lark = 0, chk = 0;

        for (lark = 0; lark < point2; lark++) {

                if (first[i] == calc_first[point1][lark]) {
                        chk = 1;
                        break;
                }
        }
        if (chk == 0) {
                printf("%c, ", first[i]);
                calc_first[point1][point2++] = first[i];
        }
}
printf("}\n");
```

```
        jm = n;
        point1++;
}
printf("\n");
printf("-----------------------------------------"
        "\n\n");
char donee[count];
ptr = -1;

// Initializing the calc_follow array
for (k = 0; k < count; k++) {
        for (kay = 0; kay < 100; kay++) {
                calc_follow[k][kay] = '!';
        }
}
point1 = 0;
int land = 0;
for (e = 0; e < count; e++) {
        ck = production[e][0];
        point2 = 0;
        xxx = 0;

        // Checking if Follow of ck
        // has already been calculated
        for (kay = 0; kay <= ptr; kay++)
                if (ck == donee[kay])
```

```
                xxx = 1;

if (xxx == 1)
        continue;
land += 1;

// Function call
follow(ck);
ptr += 1;

// Adding ck to the calculated list
donee[ptr] = ck;
printf(" Follow(%c) = { ", ck);
calc_follow[point1][point2++] = ck;

// Printing the Follow Sets of the grammar
for (i = 0 + km; i < m; i++) {
        int lark = 0, chk = 0;
        for (lark = 0; lark < point2; lark++) {
                if (f[i] == calc_follow[point1][lark]) {
                        chk = 1;
                        break;
                }
        }
        if (chk == 0) {
                printf("%c, ", f[i]);
```

```
                        calc_follow[point1][point2++] = f[i];
                    }
                }
                printf(" }\n\n");
                km = m;
                point1++;
            }
}

void follow(char c)
{
    int i, j;

    // Adding "$" to the follow
    // set of the start symbol
    if (production[0][0] == c) {
        f[m++] = '$';
    }
    for (i = 0; i < 10; i++) {
        for (j = 2; j < 10; j++) {
            if (production[i][j] == c) {
                if (production[i][j + 1] != '\0') {
                    // Calculate the first of the next
                    // Non-Terminal in the production
                    followfirst(production[i][j + 1], i,
                                (j + 2));
```

```
                        }

                        if (production[i][j + 1] == '\0'
                                && c != production[i][0]) {
                                // Calculate the follow of the
                                // Non-Terminal in the L.H.S. of the
                                // production
                                follow(production[i][0]);
                        }
                }
        }
    }
}

void findfirst(char c, int q1, int q2)
{
    int j;

    // The case where we
    // encounter a Terminal
    if (!(isupper(c))) {
        first[n++] = c;
    }
    for (j = 0; j < count; j++) {
        if (production[j][0] == c) {
            if (production[j][2] == '#') {
```

```
                        if (production[q1][q2] == '\0')
                                first[n++] = '#';
                        else if (production[q1][q2] != '\0'
                                        && (q1 != 0 || q2 != 0)) {
                                // Recursion to calculate First of New
                                // Non-Terminal we encounter after
                                // epsilon
                                findfirst(production[q1][q2], q1,
                                                (q2 + 1));
                        }
                        else
                                first[n++] = '#';
                }
                else if (!isupper(production[j][2])) {
                        first[n++] = production[j][2];
                }
                else {
                        // Recursion to calculate First of
                        // New Non-Terminal we encounter
                        // at the beginning
                        findfirst(production[j][2], j, 3);
                }
        }
    }
}
```

```
void followfirst(char c, int c1, int c2)
{
	int k;

	// The case where we encounter
	// a Terminal
	if (!(isupper(c)))
		f[m++] = c;
	else {
		int i = 0, j = 1;
		for (i = 0; i < count; i++) {
			if (calc_first[i][0] == c)
				break;
		}

		// Including the First set of the
		// Non-Terminal in the Follow of
		// the original query
		while (calc_first[i][j] != '!') {
			if (calc_first[i][j] != '#') {
				f[m++] = calc_first[i][j];
			}
			else {
				if (production[c1][c2] == '\0') {
					// Case where we reach the
					// end of a production
```

```
                        follow(production[c1][0]);
                    }
                    else {
                        // Recursion to the next symbol
                        // in case we encounter a "#"
                        followfirst(production[c1][c2], c1,
                                                c2 + 1);
                    }
                }
                j++;
            }
        }
    }
```