Ahsanullah University of Science and Technology (AUST)
Department of Computer Science and Engineering

Assignment 6

Course No.: CSE4130

Course Title: Formal Languages & Compilers Lab

Date of Submission-16.08.2023

Submitted To- Mr. Md. Aminur Rahman & Iffatur Nessa.

Submitted By-

MD Shihabul Islam Shovo
190204075
Group: B1
Year- 4th
Semester- 1st
Session: Fall'22
Department- CSE

Answer:

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include<iomanip>
#include<algorithm>
#include <regex>

using namespace std;

// variable declaration
ifstream rf;
ofstream wf;
vector<string> kws = { "auto", "break", "case", "char", "const", "continue", "default",
        "do", "double", "else", "enum", "extern", "float", "for", "goto",
        "if", "inline", "int", "long", "register", "restrict", "return",
        "short", "signed", "sizeof", "static", "struct", "switch", "typedef",
        "union", "unsigned", "void", "volatile", "while", "_Bool", "_Complex",
"_Imaginary" };
vector<string> ids; //stores all the indentifier
string ops = "+-*/%=<>!|&";
string pars = "(){}[]";
string seps = ",;'\"";
string op;
char c;
string s;
struct TokenStruct { // structure of a token
   int no;
```

```
    string type;
    string value;
};
struct SymbolTable{ // structure of the symbol table
    int si_no;
    string name, id_type, data_type, scope, value;
};
vector <TokenStruct> token; // vector of TokenStruct structure
vector <SymbolTable> table; // vector of SymbolTable structure

string input_str;
string input_expr;
int idx = 0;
int position = 0;
int counter = 0;
regex identifier("^[a-zA-Z_][a-zA-Z0-9_]*$");
regex num_literal(R"(-?\d+(\.\d+)?([eE][+-]?\d+)?)");

int evaluate_statement();

int evaluate_expression();
int evaluate_term();

// User Defined function as needed
int read_file(string filename) {
    // This function will take a file name input from the user and open it in read mode
    rf.open(filename);
    if (!rf) {
```

```
        cout << "Error opening file.\n";
        return 1;
    }
    return 0;
}
void plainC() {
    // This plainC() function removes all newlines, extra spaces, and comments from a C source code file
    FILE *readFile,*writeFile;
    char c1='\0', c2 = ' ';
    int line_no = 1;
    readFile = fopen("input.c","r");
    writeFile = fopen("plainC.txt", "w");

    //If file is not created then show this message
    if(!readFile)cout<<"\nFile not found";

    /*if File is created then
     remove the spaces, empty line & comments*/

    else
    {
        c1 = fgetc(readFile); c1 = fgetc(readFile); c1 = fgetc(readFile);
        fprintf(writeFile, "%d ", line_no++);
        while((c1 = fgetc(readFile))!= EOF)
        {
            if(c1==' '){
                fputc(' ', writeFile);
                while((c1=fgetc(readFile)) == ' ');
```

```
}
if((c1=='\n')){
   fputc('\n', writeFile);
   fprintf(writeFile, "%d ", line_no++);
   continue;
}
if((c1=='/') && ((c2 = fgetc(readFile))== '/')){
   while((c1=fgetc(readFile))!='\n');
   fputc('\n', writeFile);
   fprintf(writeFile, "%d ", line_no++);
}
else if((c1=='/') && (c2=='*')){
   c2 = c1; c1 = fgetc(readFile);
   if(c1=='\n'){
      fputc('\n', writeFile);
      fprintf(writeFile, "%d ", line_no++);
   }
   while((c1!='/') && (c2 != '*')) {
      c2 = c1;
      c1 = fgetc(readFile);
      if(c1=='\n'){
         fputc('\n', writeFile);
         fprintf(writeFile, "%d ", line_no++);
      }
   }
}
else{
   fputc(c1, writeFile);
```

```
            }
            c2 = c1;
        }
    }
    fclose(readFile);
    fclose(writeFile);
}
int isoperator() {
    // isoperator() function will check for an operator
    if (ops.find(c) != string::npos) {
        op += c;
        if ((c = rf.get()) != EOF) {
            isoperator();
        }
        return 1;
    }

    if (!op.empty()) {
        rf.unget();
        return 0;
    }
    return 0;
}
int isprnorsep(string str) {
    // isprnorsep() function will check for a parenthesis or separator
    if (str.find(c) != string::npos) {
        return 1;
    }
```

```
    return 0;
}
int iskeyword() {
    for (const string& keyword : kws) {
        if (s == keyword) {
            return 1;
        }
    }
    return 0;
}
int isidentifier() {
    // isidentifier() function finds the valid keywords also label as id and if not valid then label as unkn
    for (int i = 1; i < ids.size(); i++) {
        if (s == ids[i]) {
            return 1;
        }
    }

    int len = s.length();
    if (s[0] == '_' || isalpha(s[0])) {
        for (int i = 1; i < len; i++) {
            if (s[i] == '_' || isalnum(s[i])) {
                continue;
            }
            else {
                return 0;
            }
        }
```

```
        ids.push_back(s);
        return 1;
    }
    return 0;
}
int isnumber() {
    // Check if the word is a number or not
    int len = s.length();
    int i, nflag = 0;
    for (i = 0; i < len; i++) {
        if (isdigit(s[i])) {
            nflag = 1;
        }
        else if (s[i] == '.') {
            nflag = 2;
            i++;
            break;
        }
        else {
            return 0;
        }
    }
    if (nflag == 2) {
        while (i < len) {
            if (isdigit(s[i])) {
                nflag = 1;
            }
            else {
```

```
            return 0;
          }
          i++;
        }
    }
    if (nflag == 1) {
        return 1;
    }
    return 0;
}
void insertToken(string type, string str){
    // insert tokens to a vector of structure
    TokenStruct newtoken;
    newtoken.no = token.size() + 1;
    newtoken.type = type;
    newtoken.value = str;

    token.push_back(newtoken);
}
void lexemes() {
    cout<<"Step 1: Intermediate Output: Recognized tokens in the lines of code."<<endl;
    // This function analyzes all the words and finds the lexemes
    // Read a c file to get the source code
    if (read_file("plainC.txt") != 0) {
        cout<<"Error opening file"<<endl;
    }
    wf.open("lexemes.txt");
    c = rf.get(); // to read the first line no value
```

```
   s=c; insertToken("lno", s); wf<<"[lno "<<s<<"] ";  s.clear(); cout<<c;
   while ((c = rf.get()) != EOF) {
      if(c=='\n'){
         cout<<c;
         c = rf.get(); cout<<c;
         while(isdigit(c)) { s+=c; c = rf.get(); cout<<c;}
         insertToken("lno", s);
         wf<<"[lno "<<s<<"] ";
         s.clear();
      }
      if(isspace(c)){
         cout<<c;
         continue;
      }
      // Read letters and store the word
      for (int i = 0; !isspace(c) && !isoperator() && !isprnorsep(pars) && !isprnorsep(seps); i++)
{
         // Store the letters until there is a space, operator, parenthesis, or separator
         // If isoperator() function is called, this will store the operator or consecutive operators
         // Other functi ons will only return a positive value or 1
         s += c;
         c = rf.get();
      }
      if (!s.empty()) {
         if (iskeyword()) {cout<<"kw "<<s<<" "; insertToken("kw", s); wf<<"[kw "<<s<<"] ";} //
insertToken(string , string) receives two string value and insert as token
         else if (isidentifier()) {cout<<"id "<<s<<" "; insertToken("id", s); wf<<"[id "<<s<<"] ";}
         else if (isnumber()) {cout<<"num "<<s<<" "; insertToken("num", s); wf<<"[num
"<<s<<"] ";}
      }
```

```
        if (!op.empty()) {
            // If there is an operator stored from the previous call of isoperator() function tokenize the operator
            insertToken("op", op); wf<<"[op "<<op<<"] ";
            cout<<"op "<<op<<" "; op.clear(); // clear the operator so that next time it don't contain any value if isoperator() function don't assign any value to op
        }
        else if (isprnorsep(pars)) {
            // Call the isprnorsep() function and tokenize the parenthesis
            s=c; // converts char to string
            insertToken("par", s); wf<<"[par "<<s<<"] ";
            cout<<"par "<<s<<" ";
        }
        else if (isprnorsep(seps)) {
            // Call the isprnorsep() function and tokenize the separator
            s=c;
            insertToken("sep", s); wf<<"[sep "<<s<<"] ";
            cout<<"sep "<<s<<" ";
        }
        s.clear();
    }
    rf.close();
    wf.close();
    cout<<endl;
}
//assignment 3
bool isdatatype(string str) {
    vector<string> dt = { "int", "float", "double", "char", "bool", "vector", "string" };
    for (const string& datatype : dt) {
```

```
        if (str == datatype) {
            return true;
        }
    }
    return false;
}
void search_in_table(int i, string recentScope){
    //find id the variable already exist in the symbol table and if it has assigned a value
    //then update the value in symbol table
    TokenStruct& t = token[i];
    for(auto& src: table){
        if(t.value==src.name && src.id_type=="var" && recentScope==src.scope){ // recentScope==src.scope is used to check existed variable from the same scope
            t=token[++i];
            if(t.value=="="){
                t=token[++i];
                if(t.type=="num"){
                    src.value = t.value;
                } else t = token[--i];
            }
        }
    }
}
void create_symbol_table(vector <TokenStruct> newtoken){ // instance of a vector of structure so that the main variable's values doesn't get manipulated
    // Insert new entry in the symbol table for lexemes
    string recentScope, lastScope= "Global"; // initially scope is Global
    string recentDatatype;
    int braces=0;
```

```
   for(int i=0; i<token.size(); i++){
      TokenStruct& t = newtoken[i]; // pointer t to indicate a vector of structure's (newtoken) index
      if(braces==0){ // braces value 0 means it is in Global section
         recentScope = "Global";
      }
      if(t.value == "{") {
         braces++;
         recentScope = lastScope;
      }
      else if(t.value == "}") braces--;
      SymbolTable tb;
      if(t.type =="kw" && isdatatype(t.value)){ // isdatatype() function to check if it's a data type or not
         tb.si_no = table.size() + 1; // sirial no
         recentDatatype = t.value; // this is used for next variable in a single type
         t = newtoken[++i]; // get the next token from the newtoken vector
         if(t.type == "id"){
            tb.name = t.value; // name
            tb.data_type = recentDatatype; // data type
            tb.scope = recentScope; // scope
            t = newtoken[++i];
            if(t.value == "("){
               tb.id_type = "func"; // insert id type = func
               recentScope = tb.name;
               lastScope = recentScope; // save the current scope for further use for variables in this scope
               tb.value = "\0"; // no value has for funciton
            }
```

```
            else if(t.value == "=" || t.value == ";" || t.value == ")" || t.value == ","){ // could be
x1 = 121 or x1; or f1(int x1)
               tb.id_type = "var"; // id type = var
               tb.scope = lastScope; // scope
               if(t.value == "="){ // = means a value is assigned for this variable
                  t = newtoken[++i];
                  if(t.type == "num"){ // condition to check if assigned value is a num attribute
                     tb.value = t.value; // value of the variable
                  }
                  else t = newtoken[--i]; // if the if statement is not true then go to the previous
token
               }
            }
            else t = newtoken[--i];

         } else t = newtoken[--i];
         string vname = tb.name; if(!vname.empty() ) table.push_back(tb); // push new values
in the vector
      }
      else if(t.type=="id"){
         search_in_table(i, recentScope); // update values of variables
      }
   }
}
bool searchByString(const SymbolTable& obj, const string& value) {
   return obj.name == value;
}
void free(){
   // Delete all the entry from symbol table
   if(table.size()>0){
```

```
        table.erase(table.begin(),table.end());
        cout<<"--All entry cleared successfully."<<endl<<endl;
    }
    else cout<<"--Symbol table is already empty."<<endl<<endl;
}
void lookUp(){
    // lookUp() function search for a name in the symbol table
    if(table.size()>0){
        string searchName;
        cout<<"Enter a name to search: ";
        cin>>searchName;
        auto stringResult = find_if(table.begin(), table.end(),
            [searchName](const SymbolTable& obj) { return searchByString(obj, searchName); });
        if (stringResult != table.end()) {
            cout << "--Result: The searched name's SI.No is: " << stringResult->si_no <<
endl<<endl;
        }
        else {
            cout << "--Error: Name \""<<searchName  <<"\" doesn't exist on the symbol table!" <<
endl<<endl;
        }
    }
    else cout<<"--Symbol table is empty."<<endl<<endl;
}
bool setAttribute(string iteamname){
    s.clear();
    s=iteamname;
    if(isidentifier() ){
        string idtype, datatype, scope, value="";
```

```
int sno;
cout<<"Enter attributes values: "<<endl;
cout<<"SI no: "; cin>>sno;
cout<<"Id type: "; cin>>idtype;
cout<<"Data type: "; cin>>datatype;
cout<<"Scope: "; cin>>scope;
if(idtype == "var"){
   cout<<"Value: "; cin>>value; }

SymbolTable tb;
tb.si_no = 0;
tb.name = " ";
tb.data_type = " ";
tb.id_type = " ";
tb.scope = " ";
tb.value = " ";
table.push_back(tb);
cout<<table.size()<<endl;
for(int i=table.size()-1; i>=sno; i--){
   cout<<table[i].si_no<<endl;
   table[i].si_no = table[i-1].si_no + 1;
   table[i].name = table[i-1].name;
   table[i].id_type = table[i-1].id_type;
   table[i].data_type = table[i-1].data_type;
   table[i].scope = table[i-1].scope;
   table[i].value = table[i-1].value;
}
table[sno-1].si_no = sno;
```

```
        table[sno-1].name = iteamname;
        table[sno-1].id_type = idtype;
        table[sno-1].data_type = datatype;
        table[sno-1].scope = scope;
        table[sno-1].value = value;
        return true;
    }
    else{
        cout<<"Not a valid identifier name. "<<endl;
        return false;
    }
}
void insert_item(){
    string iteamName;
    cout<<"Enter new token name to insert: ";
    cin>>iteamName;
    auto stringResult = find_if(table.begin(), table.end(),
        [iteamName](const SymbolTable& obj) { return searchByString(obj, iteamName); });
    if (stringResult != table.end()) {
        cout << "--Result: The iteam's SI.No is: " << stringResult->si_no << endl<<endl;
    }
    else {
        if(setAttribute(iteamName)) cout<<"New token inserted successfully."<<endl;;
    }
}
void displayTable(){
    if(table.size()>0){
        cout <<left<< setw(20) << "SI.No" << setw(20) << "Name" << setw(20) << "ID Type" <<
setw(20) << "Data Type" << setw(20) << "Scope" << setw(20) << "Value" << endl;
```

```
      cout<<"-----------------------------------------------------------------------------------------------------
------"<<endl;
      for(const auto& t: table){
         cout <<left<< setw(20) << t.si_no <<setw(20)<<  t.name <<setw(20)<<  t.id_type
<<setw(20)<<  t.data_type <<setw(20)<<  t.scope <<setw(20)<< t.value << endl;
      }
      cout<<endl;
   }
   else cout<<"--Symbol table is empty."<<endl<<endl;
}
void displayLexemes(){
   cout << left << setw(7) <<"No" << setw(12)<< "Type" <<setw(12)<< "Value" << "|| "
         << setw(7) <<"No" << setw(12)<< "Type" <<setw(12)<< "Value" << "|| "
         << setw(7) <<"No" << setw(12)<< "Type" <<setw(12)<< "Value" << endl;
         cout<<"-----------------------------------------------------------------------------------------
"<<endl;
   for(int i=0; i<token.size(); i++){
      TokenStruct& t = token[i];
      cout <<left<<setw(7)<< t.no <<setw(12)<<  t.type <<setw(12)<< t.value << "|| ";
      t = token[++i];
      cout << setw(7)<< t.no <<setw(12)<<  t.type <<setw(12)<< t.value << "|| ";
      t = token[++i];
      cout << setw(7)<< t.no <<setw(12)<<  t.type <<setw(12)<< t.value <<endl;
   }
   cout<<endl;
}
int userChoice(){
   int choice;
   while(1){
      cout<< "---Choose an option: " << endl
```

```
        << " 1. Insert an entry: "<<endl
        << " 2. Lookup: Search for a name on the symbol table. " << endl
        << " 3. Free: remove all entries." << endl
        << " 4. Display Symbol table" << endl
        << " 5. Display the lexemes" << endl
        << " 6. Exit" << endl
        << "\nEnter your choice: ";
        cin>>choice;
        if(choice == 1) insert_item();
        else if(choice == 2) lookUp();
        else if(choice == 3) free();
        else if(choice == 4) displayTable();
        else if(choice == 5) displayLexemes();
        else return 0;
    }
}
//assignment 5
int evaluate_statement();

int evaluate_expression();
int evaluate_term();

bool is_valid_identifier(const string& str)
{
    static regex identifier("^[a-zA-Z_][a-zA-Z0-9_]*$");
    return regex_match(str, identifier);
}
```

```
bool is_valid_num_literal(const string& str)
{
   return regex_match(str, num_literal);
}

int evaluate_expression();
int evaluate_factor()
{
   string x;
   switch (input_str[idx])
   {
      case '(':
         idx++; // Move to the next character ('(' was found)
         if (evaluate_expression() && input_str[idx] == ')')
         {
            idx++; // Move to the next character (')' was found)
            return 1;
         }
         return 0;

      default:
         while (isalnum(input_str[idx]))
         {
            x.push_back(input_str[idx]);
            idx++;
         }
         if (is_valid_identifier(x) || is_valid_num_literal(x))
         {
```

```
            return 1;
        }
        return 0;
    }
}

int evaluate_expression()
{
    int f = evaluate_factor();

    while (idx < input_str.length() && f == 1)
    {
        char op = input_str[idx];
        if (op == '*' || op == '/')
        {
            idx++; // Move to the next character (operator was found)
            int nextFactor = evaluate_factor();
            f = (nextFactor == 1) ? 1 : 0;
        }
        else if (op == '+' || op == '-')
        {
            idx++; // Move to the next character (operator was found)
            int nextTerm = evaluate_factor();
            f = (nextTerm == 1) ? 1 : 0;
        }
        else
        {
            break; // No more valid operators, exit the loop
```

```
        }
    }

    return f;
}

int evaluate_simple_expression()
{
    int f = 0;
    f = evaluate_expression();
    return f;
}

int evaluate_relop()
{
    if (input_str[idx] == '=' && input_str[idx + 1] == '=')
    {
        idx += 2; // Move to the next character (operator was found)
        return 1;
    }
    else if (input_str[idx] == '!' && input_str[idx + 1] == '=')
    {
        idx += 2; // Move to the next character (operator was found)
        return 1;
    }
    else if (input_str[idx] == '>' && input_str[idx + 1] == '=')
    {
        idx += 2; // Move to the next character (operator was found)
```

```
        return 1;
    }
    else if (input_str[idx] == '<' && input_str[idx + 1] == '=')
    {
        idx += 2; // Move to the next character (operator was found)
        return 1;
    }
    else if (input_str[idx] == '>' || input_str[idx] == '<')
    {
        idx++; // Move to the next character (operator was found)
        return 1;
    }
    else
    {
        return 0;
    }
}


int evaluate_extension()
{
    if (idx >= input_str.length())
    {
        return 1; // Expression ends here, return 1 to indicate success
    }

    int f = evaluate_relop();
    if (f == 1)
```

```
    {
        return evaluate_simple_expression() ? 1 : 0;
    }

    return 1; // No comparison operator found, return 1 to indicate success
}

int evaluate_expression_extension()
{
    int f = 0;
    f = evaluate_simple_expression();
    if (f == 1)
    {
        f = evaluate_extension();
    }
    return f;
}

int evaluate_assignment_statement()
{
    string x;

    // Parse the identifier
    while (isalnum(input_str[idx]))
    {
        x.push_back(input_str[idx]);
        idx++;
    }
```

```
    // Check if the identifier is valid
    if (!regex_match(x, identifier))
    {
        return 0; // Invalid identifier, return 0 to indicate failure
    }

    // Check for the assignment operator '='
    if (input_str[idx] == '=')
    {
        idx++; // Move to the next character (operator '=' was found)
    }
    else
    {
        return 0; // Missing assignment operator, return 0 to indicate failure
    }

    // Evaluate the expression on the right side of the assignment
    int f = evaluate_expression_extension();

    return f;
}


int evaluate_extension_1()
{
    if (idx >= input_str.length())
    {
```

```
        return 1; // Expression ends here, return 1 to indicate success
    }

    int z = idx;
    string x;

    // Parse the next word
    while (isalnum(input_str[idx]))
    {
        x.push_back(input_str[idx]);
        idx++;
    }

    if (x == "else")
    {
        idx++; // Move to the next character ('else' was found)

        // Evaluate the statement after 'else'
        if (evaluate_statement())
        {
            return 1; // The 'else' statement is valid, return 1 to indicate success
        }
    }

    idx = z; // Reset the index to the original position
    return 1; // No 'else' statement found, return 1 to indicate success
}
```

```
int evaluate_decision_statement()
{
   string x;

   // Parse the next word
   while (isalnum(input_str[idx]))
   {
      x.push_back(input_str[idx]);
      idx++;
   }

   if (x == "if")
   {
      cout<<"hdh";
      // Check for '(' after 'if'
      if (input_str[idx++] == '(')
      {
         // Evaluate the expression inside the parentheses
         if (evaluate_expression_extension())
         {
            // Check for ')' after the expression
            if (input_str[idx++] == ')')
            {
               // Evaluate the statement after the if condition
               if (evaluate_statement())
               {
                  // Evaluate the extension_1 (optional 'else' part)
```

```
                if (evaluate_extension_1())
                {
                    return 1; // The 'if' statement is valid, return 1 to indicate success
                }
            }
        }
    }
}

    return 0; // The 'if' statement is invalid or not found, return 0 to indicate failure
}



int evaluate_loop_statement()
{
    string x;

    // Parse the next word
    while (isalnum(input_str[idx]))
    {
        x.push_back(input_str[idx]);

        idx++;
    }

    if (x == "while")
    {
```

```
    // Check for '(' after 'while'
    if (input_str[idx++] == '(')
    {
        // Evaluate the expression inside the parentheses
        if (evaluate_expression_extension() && input_str[idx++] == ')')
        {
            // Evaluate the statement inside the loop
            if (evaluate_statement())
            {
                return 1; // The 'while' loop is valid, return 1 to indicate success
            }
        }
    }
}
else if (x == "for")
{
    // Check for '(' after 'for'
    if (input_str[idx++] == '(')
    {
        // Evaluate the initialization statement for the loop
        if (evaluate_assignment_statement() && input_str[idx++] == ';')
        {
            // Evaluate the expression for the loop condition
            if (evaluate_expression_extension() && input_str[idx++] == ';')
            {
                // Evaluate the update statement for the loop
                if (evaluate_assignment_statement() && input_str[idx++] == ')')
```

```
            {
                // Evaluate the statement inside the loop
                if (evaluate_statement())
                {
                    return 1; // The 'for' loop is valid, return 1 to indicate success
                }
            }
          }
        }
      }
   }

   return 0; // The loop statement is invalid or not found, return 0 to indicate failure
}



int evaluate_statement()
{
   string x1;
   int id1 = 0;
   // Parse the next word
   while (isalnum(input_str[id1]))
   {
      x1.push_back(input_str[id1]);
      id1++;
   }

   int y ;
```

```
    position = idx;

    if (evaluate_assignment_statement())
    {
        idx++;
        return 1; // Assignment statement is valid, return 1 to indicate success
    }

    idx = position; // Reset the index to the original position
   if(x1=="while"|| x1=="for") y = evaluate_loop_statement();
   if(x1=="if") y = evaluate_decision_statement();
    if(y)
    {
        return 1; // Decision or loop statement is valid, return 1 to indicate success
    }


    return 0; // Statement is invalid, return 0 to indicate failure
}

//assignment 4
void detectErrors(){
    cout<<"\nStep 2: Detected errors:"<<endl;
    int errors=1;
    string lno="0";
    int line_count = 0;
    int sb=0, sc=0, cm=0, kw=0, ifs=0, elf=0, other=0;
    ifstream mf("plainC.txt");
```

```
vector<string> input_stat;
string line;
while(getline(mf, line)){
   if(line[1]==' ')
      line.erase(0, 2);
   else line.erase(0, 3);
   input_stat.push_back(line);
   cout<<line<<endl;
}

for(int i=0; i<token.size(); i++){
   TokenStruct& t = token[i];
   if(t.type=="lno"){
      //cout << "Statement: " << input_str << endl;
      if(sb>1)
         cout<<"Error "<<errors++ <<" : Misplaced '{' at line "<<lno<<endl;
      if(sb<-1)
         cout<<"Error "<<errors++ <<" : Misplaced '}' at line "<<lno<<endl;
      if(sc>1)
            cout<<"Error "<<errors++ <<" : Duplicate \";\" at line "<<lno<<endl;
      lno = t.value;
      input_str = input_stat[line_count++];
      if (evaluate_statement());
         //cout << "Statement is correct at line "<<lno<<endl;
      else
         //cout << "Statement is incorrect at line "<<lno<<endl;
         cout<<"Error "<<errors++ <<" : Statement is incorrect at line "<<lno<<endl;
      sb=sc=cm=kw=other=0;
```

```
    }
    else {
       other++;
       if(other!=0 && t.value!=";") sc=0;
       if(other!=0 && t.value!=",") cm=0;
       if(other!=0 && t.type!="kw") kw=0;
    }
    if(t.type=="par"){
       if(t.value=="{") sb++;
       else if(t.value=="}") sb--;
    }
    else if(t.type == "sep"){
       if(t.value==";") {
          sc++;
       }
       else if(t.value==",") {
          cm++;
          if(cm>1)
             cout<<"Error "<<errors++ <<" : Duplicate \",\" at line "<<lno<<endl;
       }
    }
    else if(t.type == "kw" && t.value!="if" && t.value!="else" &&t.value!="for" &&
t.value!="while"&& t.value!="return"){
       kw++;
       if(kw>1)
          cout<<"Error "<<errors++ <<" : Duplicate keywords at line "<<lno<<endl;
    }
    else if(t.type=="kw" && t.value=="if"){
          ifs=1;
```

```
        }
        else if(t.type=="kw" && t.value=="else" && token[i+1].value=="if"){
            if(ifs==0)
                cout<<"Error "<<errors++ <<" : Unmatched 'else if' at line "<<lno<<endl;
            i++; ifs=0; elf=1;
        }
        else if(t.type=="kw" && t.value=="else"){
            if(ifs==0 && elf==0)
                cout<<"Error "<<errors++ <<" : Unmatched 'else' at line "<<lno<<endl;
            ifs=elf=0;
        }}}
// Main function
int main() {
    plainC(); //at line no: 61 //removes all the extra space and comments and adds line no.
    lexemes(); //at line no: 215 //analysis for lexemes and assignment 4 step 1 is in function lexemes.
    create_symbol_table(token); //at line no: 300 // create the symbol table for the lexemes.
    userChoice(); //at line no: 464 //Gives users to choose from some options.

    detectErrors(); //at Line no: 825 // detect errors for syntax and statements.

    return 0;
}
//assignment 1 starts from line no 61
//assignment 2 starts from line no 215
//assignment 3 starts from line no 274
//assignment 4 starts from lone no 825
//assignment 5 starts from line no 485
```