

**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