//Project 2 part 2

//written by Jacob Clarke and Mitchell Cummins

//code provided by Dr. St. Clair

#include <cstdlib>

#include <iostream>

#include <fstream>

#include <vector>

#include <map>

#include <string>

using namespace std;

ofstream out("parseout.txt");

class SyntaxAnalyzer{

private:

vector<string> lexemes;

vector<string> tokens;

vector<string>::iterator lexitr;

vector<string>::iterator tokitr;

// map of variables to datatype (i.e. sum t\_integer)

map<string, string> symboltable;

// other private methods

bool vdec();

int vars();

bool stmtlist();

int stmt();

bool ifstmt();

bool elsepart();

bool whilestmt();

bool assignstmt();

bool inputstmt();

bool outputstmt();

bool expr();

bool simpleexpr();

bool term();

bool arithop();

bool relop();

public:

SyntaxAnalyzer(istream& infile);

// pre: 1st parameter consists of an open file containing a source code's

// valid scanner output. This data must be in the form

// token : lexeme

// post: the vectors have been populated

bool parse();

// pre: none

// post: The lexemes/tokens have been parsed and the symboltable created.

// If an error occurs, a message prints indicating the token/lexeme pair

// that caused the error. If no error occurs, the symboltable contains all

// variables and datatypes.

};

SyntaxAnalyzer::SyntaxAnalyzer(istream& infile){

string line, tok, lex;

int pos;

getline(infile, line);

while(!infile.eof()){

pos = line.find(":");

tok = line.substr(0, pos-1);

lex = line.substr(pos+2, line.length() - (pos +2));

cout << pos << tok << " " << lex << endl;

tokens.push\_back(tok);

lexemes.push\_back(lex);

getline(infile, line);

}

tokitr = tokens.begin();

lexitr = lexemes.begin();

}

bool SyntaxAnalyzer::parse(){

if (!vdec()){

return false;

}

if (\*tokitr != "t\_begin"){

return false;

}

tokitr++;

lexitr++;

if (tokitr!=tokens.end() && stmtlist()){

if (tokitr!=tokens.end() && \*tokitr == "t\_end"){

tokitr++; lexitr++;

if (tokitr==tokens.end()){

out << "Valid source code file" << endl;

return true;

}

}

}

else{

return true;

}

}

//vdec method checks for valid vdec grammar

//Depending on vars grammar, result will be an error, done, or continues

bool SyntaxAnalyzer::vdec(){

if (\*tokitr != "t\_var"){ //null variable declaration list

return true;

}

else{

tokitr++; lexitr++;

int result = 0; // 0 - valid, 1 - done, 2 - error

result = vars();

if (result == 2){ //vars == 2 means error

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

while (result == 0){

if (tokitr!=tokens.end())

result = vars(); // parse vars //?

}

if (result == 1){

return true;

}

else{

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

}

}

//Vars method checks for valid Vars grammar

//Determines Type of variable, checks for valid id, and semicolon

int SyntaxAnalyzer::vars(){

int result = 0; // 0 - valid, 1 - done, 2 - error

string temp;

if (\*tokitr == "t\_boolean"){ //check for boolean type

temp = "t\_boolean";

tokitr++; lexitr++;

}

else if (\*tokitr == "t\_integer"){ //check for integer type

temp = "t\_integer";

tokitr++; lexitr++;

}

else{

return 1;

}

bool semihit = false;

while (tokitr != tokens.end() && result == 0 && !semihit){

if (\*tokitr == "t\_id"){

symboltable[\*lexitr] = temp;

tokitr++; lexitr++;

if (tokitr != tokens.end() && \*tokitr == "s\_comma"){

tokitr++; lexitr++;

}

else if (tokitr != tokens.end() && \*tokitr == "s\_semi"){

semihit = true;

tokitr++; lexitr++;

}

else{

result = 2;

}

}

else{

result = 2;

}

}

return result;

}

//StmtList method checks for valid StmtList grammar

//Checks stmt validity in order to be valid stmtlist

bool SyntaxAnalyzer::stmtlist(){

int result = stmt();

while (result == 1){

result = stmt();

}

if (result == 0){

return false;

}

else{

return true;

}

}

//stmt method checks for valid stmt grammar

//checks that stmt is either if, while, assign, input, and output

//can be null

int SyntaxAnalyzer::stmt(){

if (\*tokitr == "t\_if"){

tokitr++; lexitr++;

if (ifstmt()){

return 1;

}

else{

return 0;

}

}

else if (\*tokitr == "t\_while"){

tokitr++; lexitr++;

if (whilestmt()){

return 1;

}

else{

return 0;

}

}

else if (\*tokitr == "t\_id"){ // assignment starts with identifier

tokitr++; lexitr++;

if (assignstmt()){

return 1;

}

else {

return 0;

}

}

else if (\*tokitr == "t\_input"){

tokitr++; lexitr++;

if (inputstmt()) {

return 1;

}

else {

return 0;

}

}

else if (\*tokitr == "t\_output"){

tokitr++; lexitr++;

if (outputstmt()) {

return 1;

}

else {

return 0;

}

}

return 2; //stmtlist can be null

}

//ifstmt method checks for valid ifstmt grammar

//checks for (), expr, stmtlist, else, and end if

bool SyntaxAnalyzer::ifstmt(){

// we will write this together in class

if(\*tokitr != "s\_lparen"){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

tokitr++; lexitr++;

if(!expr()){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

if(\*tokitr != "s\_rparen"){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

tokitr++; lexitr++;

if(\*tokitr != "t\_then"){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

tokitr++; lexitr++;

if(!stmtlist()){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

if(!elsepart()){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

if(\*tokitr != "t\_end"){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

tokitr++; lexitr++;

if(\*tokitr != "t\_if"){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

tokitr++; lexitr++;

return true;

}

//elsepart method checks for valid elsepart method

//checks for else, and stmt list, can be null

bool SyntaxAnalyzer::elsepart(){

if (\*tokitr == "t\_else"){

tokitr++; lexitr++;

if (stmtlist()){ //could just return stmtlist

return true;

}

else{

return false;

}

}

return true; // elsepart can be null

}

//whilestmt method checks for valid whilestmt

//checks for (), expr, loop, stmtlist, and end loop

bool SyntaxAnalyzer::whilestmt(){

// write this function

if(\*tokitr != "s\_lparen"){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

tokitr++; lexitr++;

if(!expr()){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

if(\*tokitr != "s\_rparen"){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

tokitr++; lexitr++;

if(\*tokitr != "t\_loop"){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

tokitr++; lexitr++;

if(!stmtlist()){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

if(\*tokitr != "t\_end"){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

tokitr++; lexitr++;

if(\*tokitr != "t\_loop"){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

tokitr++; lexitr++;

return true;

}

//assignstmt method checks for valid assignstmt grammar

//checks for assign, expr, and semi

bool SyntaxAnalyzer::assignstmt(){

// write this function

if(\*tokitr != "s\_assign"){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

tokitr++; lexitr++;

if(!expr()){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

if(\*tokitr != "s\_semi"){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

tokitr++; lexitr++;

return true;

}

//inputstmt method checks for valid inputstmt grammar

//checks for (), and id

bool SyntaxAnalyzer::inputstmt(){

if (\*tokitr == "s\_lparen"){

tokitr++; lexitr++;

if (\*tokitr == "t\_id"){

tokitr++; lexitr++;

if (\*tokitr == "s\_rparen"){

tokitr++; lexitr++;

return true;

}

}

}

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

//outputstmt method checks for valid outputstmt grammar

//checks for () and expr or string

bool SyntaxAnalyzer::outputstmt(){

// write this function

if(\*tokitr != "s\_lparen"){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

tokitr++; lexitr++;

if((!expr()) && (\*tokitr!= "t\_string")){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

if(\*tokitr != "s\_rparen"){

out << "Error at "<< \*tokitr<< " : " << \*lexitr << endl;

return false;

}

tokitr++; lexitr++;

return true;

}

//expr method checks for valid expr grammar

//checks for simpleexpr, relop and simpleexpr

bool SyntaxAnalyzer::expr(){

if (simpleexpr()){

if (relop()){

if (simpleexpr()){

return true;

}

else{

return false;

}

}

else{

return true;

}

}

else{

return false;

}

}

//simpleexpr method checks for valid simpleexpr grammar

//checks for term, arithop and term

bool SyntaxAnalyzer::simpleexpr(){

// write this function

if(!term()){

return false;

}

while(arithop()){

if(!term())

{

return false;

}

}

return true;

}

//term method checks for valid term grammar

//checks for terminals and expr

bool SyntaxAnalyzer::term(){

if ((\*tokitr == "t\_number")

|| (\*tokitr == "t\_false")

|| (\*tokitr == "t\_true")

|| (\*tokitr == "t\_id")){

tokitr++; lexitr++;

return true;

}

else{

if (\*tokitr == "s\_lparen"){

tokitr++; lexitr++;

if (expr())

if (\*tokitr == "s\_rparen"){

tokitr++; lexitr++;

return true;

}

}

}

return false;

}

//arithop method checks for valid arithop grammar

bool SyntaxAnalyzer::arithop(){

if ((\*tokitr == "s\_mult") || (\*tokitr == "s\_plus") || (\*tokitr == "s\_minus")

|| (\*tokitr == "s\_div") || (\*tokitr == "s\_mod") || (\*tokitr == "t\_and")

|| (\*tokitr == "t\_or")){

tokitr++; lexitr++;

return true;

}

else{

return false;

}

}

//relop method checks for valid relop grammar

bool SyntaxAnalyzer::relop(){

if ((\*tokitr == "t\_lt") || (\*tokitr == "t\_gt") || (\*tokitr == "t\_ge")

|| (\*tokitr == "t\_eq") || (\*tokitr == "t\_ne") || (\*tokitr == "t\_le")){

tokitr++; lexitr++;

return true;

}

else{

return false;

}

}

int main(){

ifstream infile("datain.txt");

if (!infile){

cout << "error opening input file - token/lexeme pairs" << endl;

}

SyntaxAnalyzer parser(infile);

parser.parse();

infile.close();

return 1;

}