```
#include <iostream>
#include "SyntaxAnalyzer.h"

int main() {
    ifstream lexemes("sourcelexemes.txt");
    SyntaxAnalyzer SyntaxAnalyzer(lexemes);
    SyntaxAnalyzer.parse();
    return 0;
}
```

```
#include "SyntaxAnalyzer.h"
#include <istream>

// erika
bool SyntaxAnalyzer::vdec() { // GOOD
    if (tokitr != tokens.end()) {
        if (*tokitr != "t_var") {
            // no variables declared
            return true;
        }
        tokitr++;lexitr++;
        int varsResult = vars();
        while(varsResult == 1) {
            varsResult= vars();
        }
        if (varsResult == 2 || varsResult == 1) {
            return true;
        }
        if (varsResult == 0) {
        }
    }
    return false;
}
bool SyntaxAnalyzer::addSymbol(string& value, string& lexeme) {
    if (!symboltable.contains(lexeme)) {
        symboltable[lexeme] = value;
        return true;
    }
    return false;
}
// evan
int SyntaxAnalyzer::vars() { // GOOD
    if(tokitr != tokens.end()) {
        if (*tokitr != "t_string" && *tokitr != "t_integer") {
            return 2;
        }
        if(*tokitr == "t_string" ||*tokitr == "t_integer") {
            string value = *lexitr; //used later for symbol table
            tokitr++; lexitr++;
            if (tokitr != tokens.end() && *tokitr == "t_id")  {
                if (!addSymbol(value, *lexitr)) {
                    return false;
                }
                // symboltable[*lexitr] = value; //adds to symbol table
                tokitr++;lexitr++;
                while (tokitr != tokens.end() && *tokitr == "s_comma") {
                    tokitr++; lexitr++;
                    if (tokitr != tokens.end() && *tokitr == "t_id") {
                        if (!addSymbol(value, *lexitr)) {
                            return false;
                        }
                        // symboltable[*lexitr] = value;
                        tokitr++; lexitr++;
```

```
                    } else {
                        return 0;
                    }
                }
                if (tokitr != tokens.end() && *tokitr == "s_semi") {
                    tokitr++ ; lexitr++;
                    return 1;
                }
            }
        }
    }
    return 0;
}

// erika
bool SyntaxAnalyzer::stmtlist() {
    if (tokitr != tokens.end()) {
        int stmtResult = stmt();
        while(stmtResult == 1) {
            stmtResult= stmt();
        }
        if (stmtResult == 2) {
            return true;
        }
        if (stmtResult == 0) {
            return false;
        }
    }
}
// mark
int SyntaxAnalyzer::stmt() {
    if (tokitr != tokens.end() ){
        if (*tokitr != "t_while" && *tokitr != "t_if" && *tokitr != "
t_output" && *tokitr != "t_input" && *tokitr != "t_id") {
            // no statement selected
            return 2;
        }
        if (whilestmt() || ifstmt() || outputstmt() || inputstmt() ||
assignstmt()) {
            return 1;
        }
        return 0;
    }
}
// mark
bool SyntaxAnalyzer::ifstmt() {
    if (tokitr != tokens.end()) {
        if (*tokitr == "t_if") {
            tokitr++;lexitr++;
            if (tokitr != tokens.end() && *tokitr == "s_lparen") {
                tokitr++;lexitr++;
                if (expr()) {
                    if (tokitr != tokens.end() && *tokitr == "s_rparen"
```

```
 ) {
                        tokitr++;lexitr++;
                        if (tokitr != tokens.end() && *tokitr == "
 s_lbrace") {
                            tokitr++;lexitr++;
                            if (stmtlist()) {
                                if (tokitr != tokens.end() && *tokitr ==
 "s_rbrace") {
                                    tokitr++;lexitr++;
                                    elsepart();
                                    return true;
                                }
                            }
                        }
                    }
                }
            }
        }
        return false;
 }
 // erika
 bool SyntaxAnalyzer::elsepart() {
        if (tokitr != tokens.end()) {
            if (*tokitr != "t_else") {
                return true;
            }
            if (*tokitr == "t_else") {
                tokitr++; lexitr++;
                if (tokitr != tokens.end() && *tokitr == "s_lbrace"){
                    tokitr++; lexitr++;
                    if (stmtlist()) {
                        if(tokitr != tokens.end() && *tokitr == "s_rbrace"
  ) {
                            tokitr++; lexitr++;
                            return true;
                        }
                    }
                }
            }
        }
        return false;
 }

 // evan
 bool SyntaxAnalyzer::whilestmt() {
        if (tokitr != tokens.end()) {
            if (tokitr != tokens.end() && *tokitr == "t_while") {
                tokitr++; lexitr++;
                if (tokitr != tokens.end() && *tokitr == "s_lparen") {
                    tokitr++; lexitr++;
                    if (tokitr != tokens.end()) {
                        if (expr()) {
```

```
                        if (tokitr != tokens.end() && *tokitr == "
s_rparen") {
                            tokitr++; lexitr++;
                            if (tokitr != tokens.end() && *tokitr == "
s_lbrace") {
                                tokitr++; lexitr++;
                                if (stmtlist()) {
                                    if (tokitr != tokens.end() && *tokitr
 == "s_rbrace") {
                                        tokitr++; lexitr++;
                                        return true;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return false;
}
// evan
bool SyntaxAnalyzer::assignstmt() {
    if (tokitr != tokens.end()) {
        if (*tokitr == "t_id" && symboltable.contains(*lexitr)) {
            tokitr++; lexitr++;
            if (tokitr != tokens.end() && *tokitr == "s_assign") {
                tokitr++; lexitr++;
                if (tokitr != tokens.end() && expr()) {
                    if (tokitr != tokens.end() && *tokitr == "s_semi") {
                        tokitr++;lexitr++;
                        return true;
                    }
                }
            }
        }
    }
    return false;
}
//erika
bool SyntaxAnalyzer::inputstmt() {
    if (tokitr != tokens.end()) {
        if (*tokitr == "t_input") {
            tokitr++; lexitr++;
            if (*tokitr == "s_lparen") {
                tokitr++; lexitr++;
                if (*tokitr == "t_id") {
                    tokitr++; lexitr++;
                    if (*tokitr == "s_rparen") {
                        tokitr++; lexitr++;
                        return true;
                    }
```

```
                }
            }
        }
    }
    return false;
}
// mark
bool SyntaxAnalyzer::outputstmt() {
    if (tokitr != tokens.end()) {
        if (*tokitr == "t_output") {
            tokitr++;lexitr++;
            if (tokitr != tokens.end() && *tokitr == "s_lparen") {
                tokitr++;lexitr++;
                if (tokitr != tokens.end()) {
                    if (*tokitr == "t_text") {
                        tokitr++;lexitr++;
                        if (tokitr != tokens.end() && *tokitr == "
s_rparen") {
                                tokitr++;lexitr++;
                                return true;
                            }
                        }
                    else if (expr()){
                        if (tokitr != tokens.end() && *tokitr == "
s_rparen") {
                            tokitr++;lexitr++;
                            return true;
                        }
                    }
                }
            }
        }
    }
    return false;
}
// evan
bool SyntaxAnalyzer::expr() {
    if (tokitr != tokens.end()) {
        if (simpleexpr()) {
            if (tokitr!= tokens.end() && !logicop()) {
                return true;
            }
            if (tokitr != tokens.end() && !simpleexpr()) {
                return false;
            }
            return true;
        }
    }
    return false;
}

//mark
bool SyntaxAnalyzer::simpleexpr() {
```

```
    if (tokitr != tokens.end()) {
        if (term()) {
            if (arithop()) {
                //case 1 term arith term
                if (term()) {
                    return true;
                }
            }
            if (relop( )) {
                //case 2 term relop term
                if (term()) {
                    return true;
                }
            }
            //if its just a term
            return true;
        }
    }
    return false;
}

//  erika
bool SyntaxAnalyzer::term() {
    if (tokitr != tokens.end()) {
        if ( (*tokitr == "t_id" && symboltable.contains(*lexitr)) || *
tokitr == "t_number" || *tokitr == "t_text" ) {
            tokitr++; lexitr++;
            return true;
        }
        if (*tokitr == "s_lparen") {
            *tokitr++; lexitr++;
            if (expr()) {
                if (*tokitr == "s_rparen") {
                    *tokitr++; lexitr++;
                    return true;
                }
            }
        }
    }
    return false;
}
//mark
bool SyntaxAnalyzer::logicop() {
    if (tokitr != tokens.end()) {
        if (*tokitr == "s_and" || *tokitr == "s_or") {
            tokitr++;lexitr++;
            return true;
        }
    }
    return false;
}

// evan
```

```
bool SyntaxAnalyzer::arithop(){
    if (tokitr != tokens.end()) {
        if (*tokitr == "s_plus" || *tokitr == "s_minus" ||  *tokitr == "
s_div") {
            tokitr++;lexitr++;
            return true;
        }
    }
    return false;
}

// erika
bool SyntaxAnalyzer::relop() {
    if (tokitr != tokens.end()) {
        if ( *tokitr == "s_lt" || *tokitr == "s_gt" || *tokitr == "s_eq"
 || *tokitr == "s_ne" ) {
            tokitr++; lexitr++;
            return true;
        }
    }
    return false;
}

SyntaxAnalyzer::SyntaxAnalyzer(istream& infile) {
    string line;
    getline(infile, line);
    while(!infile.eof()){
        // find the first space and split it
        int pos = line.find(" ");
        tokens.push_back(line.substr(0,pos));
        lexemes.push_back(line.substr(pos+3, line.length()));
        getline(infile, line);
    }
}

// pre: none
// post: The lexemes/tokens have been parsed.
// If an error occurs, a message prints indicating the token/lexeme pair
// that caused the error.
// If no error, vectors contain syntactically correct source code
bool SyntaxAnalyzer::parse() {
    tokitr = tokens.begin();
    lexitr = lexemes.begin();
    if (tokitr != tokens.end()) {
        if(vdec()){
            if (tokitr != tokens.end() && *tokitr == "t_main"){
                tokitr++; lexitr++;
                if(tokitr != tokens.end() && *tokitr == "s_lbrace"){
                    tokitr++; lexitr++;
                    if(stmtlist()){
                        if(tokitr != tokens.end() && *tokitr == "s_rbrace
"){
                            // cout << "Success" << endl;
```

```
                                return true;
                            }
                        }
                    }
                }
            }
        }
    }
    if (tokitr == tokens.end()){
        tokitr--; lexitr--;
    }
    cout << "Error reading file at: " << *lexitr << endl;
    return false;
}
```

```
#ifndef SYNTAXANALYZER_H
#define SYNTAXANALYZER_H
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <map>
using namespace std;
// Erika Mark Evan
// coding skills
   // evan - 2 , mark -3 erika - 2
// procrastination
   // erika - 3 mark - 2 evan 4
// communication
 //   erika - 4 mark - 4 evan - 5

class SyntaxAnalyzer{
private:
    vector<string> lexemes;
    vector<string> tokens;
    vector<string>::iterator lexitr;
    vector<string>::iterator tokitr;
    // map of variables and their datatype
    // i.e. sum t_integer
    map<string, string> symboltable;
    // other private methods
    bool addSymbol(string& value, string& lexeme);
    bool vdec(); // erika
    int vars(); // evan
    bool stmtlist(); // erika
    int stmt(); // mark
    bool ifstmt(); // mark
    bool elsepart(); // erika
    bool whilestmt(); // evan
    bool assignstmt(); // evan
    bool inputstmt(); // erika
    bool outputstmt(); // mark
    bool expr(); // evan
    bool simpleexpr();  // mark
    bool term(); //  erika
    bool logicop(); // mark
    bool arithop(); // evan
    bool relop(); // erika

public:
    SyntaxAnalyzer(istream& infile);
    // pre: 1st parameter consists of an open file containing a source
code's
    //  valid scanner/lexical analyzer 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.
    // If an error occurs, a message prints indicating the token/lexeme
pair
    // that caused the error.
    // If no error, vectors contain syntactically correct source code
};

#endif
```

```
t_var : var
t_integer : integer
t_id : x
s_comma : ,
t_id : y
s_semi : ;
t_string : string
t_id : w
s_comma : ,
t_id : s
s_semi : ;
t_main : main
s_lbrace : {
t_if : if
s_lparen : (1
s_lparen : (2
t_id : x
s_plus : +
t_id : y
s_rparen : )
s_gt : >
t_number : 0
s_and : and
t_id : x
s_ne : !=
t_id : y
s_rparen : )
s_lbrace : {
t_output : output
s_lparen : (
t_id : x
s_rparen : )
s_rbrace : }
s_rbrace : }
```