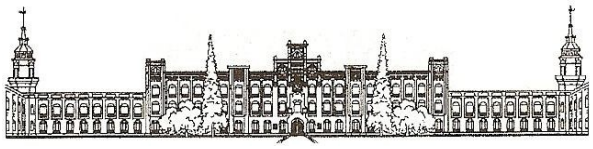

Міністерство освіти і науки України Національний
технічний університет України

«Київський політехнічний інститут»

з дисципліни «Основи проектування трансляторів»

Виконав студент групи: КВ-22

ПІБ: Крутогуз Максим Ігорович

Перевірив:

_____

Київ 2025

Мета розрахунково-графічної роботи

Метою лабораторної роботи «Розробка генератора коду» є засвоєння теоретичного матеріалу та набуття практичного досвіду і практичних навичок розробки генераторів коду.

Постановка задачі

1. Разробити програму генератора коду (ГК) для подмножини мови програмування SIGNAL, заданої за варіантом.

2. Програма генератора коду має забезпечувати:

- читання дерева розбору та таблиць, створених синтаксичним аналізатором, що було розроблено в розрахунково-графічній роботі;
- виявлення семантичних помилок;
- генерацію коду та/або побудову внутрішніх таблиць для генерації коду.

3. Зкомпонувати повний компілятор, що складається з розроблених раніше лексичного та синтаксичного аналізаторів і генератора коду, який забезпечує наступне:

- генерацію коду та/або побудову внутрішніх таблиць для генерації коду;
- формування лістингу вхідної програми з повідомленнями пролексичні, синтаксичні та семантичні помилки.

Варіант 12

```
1.    <signal-program> --> <program>
2.    <program> --> PROCEDURE <procedure-identifier>
                    <parameters-list> ; <block> ;
3.    <block> --> <declarations> BEGIN <statements-list>
                    END
4.    <declarations> --> <label-declarations>
5.    <label-declarations> --> LABEL <unsigned-integer>
                    <labels-list>; |
                    <empty>
6.    <labels-list> --> , <unsigned-integer> <labels-list>
                    |
                    <empty>
7.    <parameters-list> --> ( <variable-identifier>
                    <identifiers-list> ) |
                    <empty>
8.    <identifiers-list> --> , <variable-identifier>
                    <identifiers-list> |
                    <empty>
9.    <statements-list> --> <statement> <statements-list>
                    |
                    <empty>
10.   <statement> --> <unsigned-integer> : <statement> |
                    GOTO <unsigned-integer> ; |
                    RETURN ; |
                    ;  |
                    ($ <assembly-insert-file-identifier> $)
11.   <variable-identifier> --> <identifier>
12.   <procedure-identifier> --> <identifier>
13.   <assembly-insert-file-identifier> --> <identifier>
14.   <identifier> --> <letter><string>
15.   <string> --> <letter><string> |
                    <digit><string> |
                    <empty>
16.   <unsigned-integer> --> <digit><digits-string>
17.   <digits-string> --> <digit><digits-string> |
                    <empty>
18.   <digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
19.   <letter> --> A | B | C | D | ... | Z
```

Рисунок 1 — Варіант завдання

Тестування

True тест 1:

```
PROCEDURE GREET (NAME, MESSAGE, AGE, JOB);
LABEL 10, 23;
BEGIN
    23: RETURN;
    ($
    MOV EAX, 2
    INT 10h
    $)
    RETURN;
    10: GOTO 23;
    GOTO 10;
END;
```

```
GREET:
    push      rbp
    mov       rbp, rsp
    mov       DWORD PTR [rbp-20], edi
    mov       DWORD PTR [rbp-24], esi
    mov       DWORD PTR [rbp-28], edx
    mov       DWORD PTR [rbp-32], ecx
  23:
    pop       rbp
    ret
    mov eax, 2
    int 10h
    pop       rbp
    ret
  10:
    jump      23
    jump      10
    pop       rbp
    ret
```

```
GREET:
    push      rbp
    mov       rbp, rsp
    mov       DWORD PTR [rbp-20], edi
    mov       DWORD PTR [rbp-24], esi
    mov       DWORD PTR [rbp-28], edx
    mov       DWORD PTR [rbp-32], ecx
  23:
    pop       rbp
    ret
    mov eax, 2
    int 10h
    pop       rbp
    ret
  10:
    jump      23
    jump      10
    pop       rbp
    ret
```

False-тест 1:

```
PROCEDURE GREET (NAME, MESSAGE, AGE, JOB);
LABEL 10, 23, 11;
BEGIN
    23: RETURN;
    ($
    MOV EAX, 2
    INT 10h
    $)
    RETURN;
    10: GOTO 23;
    GOTO 10;
END;
```

```
There are 1 errors in the program:
Semantic error: Unused declared label: 11
```

True тест 2:

```
PROCEDURE GREET;
LABEL 10, 23, 11;
BEGIN
    23: 11: RETURN;
    ($
    MOV EAX, 2
    INT 10h
    $)
    10: GOTO 23;
    GOTO 10;
END;
```

```
GREET:
    push      rbp
    mov       rbp, rsp
  23:
  11:
    pop       rbp
    ret
    mov eax, 2
    int 10h
  10:
    jump      23
    jump      10
    pop       rbp
    ret
```

False-тест 2:

```
PROCEDURE GREET;
LABEL 10, 23, 11;
BEGIN
    23: 11: RETURN;
    ($
    MOV EAX, 2
    INT 10h
    $)
    12: 10: GOTO 23;
    GOTO 10;
END;
```

```
There are 1 errors in the program:
Semantic error: Used undeclared label: 12
```

[Github]

```
J:.
├───.vscode
├───bin
└───src
    ├───DataManagement
    └───Utilities
```

Код програми:

```cpp
// App.cpp
#include "DataManagement/Global.hpp"
#include "Controller.hpp"
#include "ViewStream.hpp"
#include <string>

int main() {
    ViewStream vs;
    Controller("J:/Repositories/University/Translators/Lab2/bin/semantic2f.s",
vs);
    return 0;
}


// Controller.cpp
#include "Controller.hpp"
#include "Utilities/Parser.hpp"
#include "Utilities/CodeGenerator.hpp"
```

```
Controller::Controller(string filepath, ViewStream vs) :
_vs(vs),
_it(InformationTables(vs)),
_em(ErrorManager(vs))
{
    this->run(filepath);

    // _it.outputAllTables();

    Parser parser(_it, _em, filepath);
    parser.parse();
    Tree* root = parser.getRoot();
    if (root) {
        // root->print();
    }

    CodeGenerator cg(_it, _em, "output.asm");

    cg.run(root);

    _em.output();
}

void Controller::run(string filepath) {
    ifstream inputFile(filepath);
    ostringstream buffer;

    try {
        if (inputFile.is_open()) {
            char symbol;
            int currentRow = 1;
            int currentCol = 1;
            bool isReadingAllowed = true;
            bool isLoopActive;
            while (!isReadingAllowed || inputFile.get(symbol)) {
                if (inputFile.eof()) {
                    break;
                }
                isReadingAllowed = true;
                switch (SYMBOL_CATEGORIES[(short)symbol]) {
                    case WHITESPACE:
                        switch (symbol)
                        {
                            case '\n':
                                currentCol = 1;
                                currentRow++;
                                break;
                            case '\t':
                                currentCol += 4;
                                break;
                            default:
                                currentCol++;
                                break;
                        }
                        break;
                    case CONSTANT_START:
                        do {
                        buffer << symbol;
                        } while (inputFile.get(symbol) &&
SYMBOL_CATEGORIES[(short)symbol] == CONSTANT_START);
                        _it.addConstant(buffer.str());
```

```
                        _it.prosessToken(NUMBER, buffer.str(), currentRow,
currentCol);
                        currentCol += buffer.str().length();
                        buffer.str("");
                        isReadingAllowed = false;
                        break;
                    case IDENTIFIER_OR_KEYWORD_START:
                        do {
                            buffer << symbol;

                        } while (inputFile.get(symbol) &&
(SYMBOL_CATEGORIES[(short)symbol] == CONSTANT_START ||
SYMBOL_CATEGORIES[(short)symbol] == IDENTIFIER_OR_KEYWORD_START));
                        if (_it.isKeyword(buffer.str())) {
                            _it.prosessToken(KEYWORD, buffer.str(), currentRow,
currentCol);
                        } else {
                            _it.addIdentifier(buffer.str());
                            _it.prosessToken(IDENTIFIER, buffer.str(),
currentRow, currentCol);
                        }
                        currentCol += buffer.str().length();
                        buffer.str("");
                        isReadingAllowed = false;
                        break;
                    case UNIQUE_SEPARATORS:
                        buffer << symbol;
                        _it.prosessToken(SEPARATOR, buffer.str(), currentRow,
currentCol);
                        buffer.str("");
                        currentCol++;
                        break;
                    case AMBIGUES_SEPARATORS:
                        buffer << symbol;
                        inputFile.get(symbol);
                        currentCol++;
                        short state;
                        isLoopActive = true;
                        int startRow;
                        int startCol;
                        switch (symbol) {
                            case '*':
                                buffer.str("");
                                state = COM;
                                while (isLoopActive) {
                                    if (!inputFile.get(symbol)) {
                                        throw
Utilities::getErrorMessage(filepath, currentRow, currentCol, "Not closed
commentary", "");
                                    }

                                    switch (symbol) {
                                        case '\t':
                                            currentCol += 4;
                                            break;
                                        case '\n':
                                            currentCol = 1;
                                            currentRow++;
                                            break;
                                        default:
                                            currentCol++;
```

```
                                    }
                                    switch (state) {
                                        case COM:
                                            if (symbol == '*') {
                                                state = ECOM;
                                            }
                                            break;
                                        case ECOM:
                                            if (symbol == ')') {
                                                isLoopActive = false;
                                            } else {
                                                state = COM;
                                            }
                                    }
                                }
                                break;
                            case '$':
                                buffer.str("");
                                state = AI;
                                startRow = currentRow;
                                startCol = currentCol + 1;
                                while (isLoopActive) {
                                    if (!inputFile.get(symbol)) {
                                        throw
Utilities::getErrorMessage(filepath, currentRow, currentCol, "Not closed
assembly insertion", "");
                                    }

                                    switch (symbol) {
                                        case '\t':
                                            currentCol += 4;
                                            break;
                                        case '\n':
                                            currentCol = 1;
                                            currentRow++;
                                            break;
                                        default:
                                            currentCol++;
                                    }
                                    bool isDollarMissed = false;
                                    switch (state) {
                                        case AI:
                                            if (symbol == '$') {
                                                state = EAI;
                                                isDollarMissed = true;
                                            } else {
                                                buffer << symbol;
                                            }
                                            break;
                                        case EAI:
                                            if (symbol == ')') {
                                                isLoopActive = false;

_it.prosessToken(ASSEMBLY_INSERTION, buffer.str(), startRow, startCol);
                                                buffer.str("");

                                            } else {
                                                state = AI;
                                                if (isDollarMissed) {
                                                    buffer << '$';
                                                    isDollarMissed = false;
```

```
                                        }
                                        if (symbol == '$') {
                                            isDollarMissed = true;
                                            state = EAI;
                                        } else {
                                            buffer << symbol;
                                        }
                                    }
                                }
                            }
                            break;
                        default:
                            _it.prosessToken(SEPARATOR, buffer.str(),
currentRow, currentCol - 1);
                            buffer.str("");
                            // currentCol++;
                            isReadingAllowed = false;
                            break;
                    }
                    break;

                case PROHIBITED_CHARACTER:
                default:

_em.addCompilingError(Utilities::getErrorMessage(filepath, currentRow,
currentCol, "Used prohibited character:", string(1, (char)symbol)));
                    currentCol++;
                    break;
                }
            }
        } else {
            throw string("Reading file error");
        }
    } catch (string erorrMessage) {
        _em.addProgramError(erorrMessage);
    }
}


// Controller.hpp
#include <fstream>
#include <iostream>
#include <string>
#include <sstream>
#include "DataManagement/Global.hpp"
#include "ViewStream.hpp"
#include "DataManagement/InformationTables.hpp"
#include "Utilities/ErrorManager.hpp"

#ifndef CONTROLLER_HPP
#define CONTROLLER_HPP

using namespace std;

class Controller {
    public:
    Controller(string filepath, ViewStream vs);
    void run(string);

    private:
    ViewStream _vs;
```

```
    InformationTables _it;
    ErrorManager _em;
};

#endif


// ViewStream.cpp
#include "ViewStream.hpp"

ViewStream& ViewStream::operator<<(string data) {
        cout << data;
        return *this;
}

ViewStream& ViewStream::operator<<(const char data) {
        cout << data;
        return *this;
}

ViewStream& ViewStream::operator<<(int data) {
        cout << data;
        return *this;
}

// template <typename T>
// ViewStream& ViewStream::operator<<(T data) {
//     cout << data;
//     return *this;
// }

// template ViewStream& ViewStream::operator<<(ostream& data);
// template ViewStream& ViewStream::operator<<(istream& data);
// template ViewStream& ViewStream::operator<<(ofstream& data);
// template ViewStream& ViewStream::operator<<(ifstream& data);


// ViewStream.hpp
#include <sstream>
#include <iostream>
#include <string>

#ifndef VIEWSTEAM_HPP
#define VIEWSTEAM_HPP

using namespace std;

class ViewStream {
    public:
    // std::ostringstream buffer;

    ViewStream& operator<<(string data);
    ViewStream& operator<<(const char data);
    ViewStream& operator<<(int data);

    // template <typename T>
    // ViewStream& operator<<(T data);
};

#endif
```

```cpp
// Global.cpp
#include "Global.hpp"

int SYMBOL_CATEGORIES[] = {
    PROHIBITED_CHARACTER, PROHIBITED_CHARACTER, PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER, PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    WHITESPACE,
    WHITESPACE,
    WHITESPACE,
    WHITESPACE,
    WHITESPACE,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    WHITESPACE,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    AMBIGUES_SEPARATORS,
    UNIQUE_SEPARATORS,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    UNIQUE_SEPARATORS,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    CONSTANT_START,
    CONSTANT_START,
    CONSTANT_START,
    CONSTANT_START,
    CONSTANT_START,
    CONSTANT_START,
    CONSTANT_START,
    CONSTANT_START,
    CONSTANT_START,
    CONSTANT_START,
    UNIQUE_SEPARATORS,
```

```
UNIQUE_SEPARATORS,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
IDENTIFIER_OR_KEYWORD_START,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
PROHIBITED_CHARACTER,
```

```
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
    PROHIBITED_CHARACTER,
};

map<short, string> TOKEN_MAP = {
    {NUMBER, "Number"},
    {KEYWORD, "Keyword"},
    // {CONSTANT, "Constant"},
    {IDENTIFIER, "Identifier"}
};


map<int, string> PARAM_REG_MAP = {
    {0, "edi"},
    {1, "esi"},
    {2, "edx"},
    {3, "ecx"},
    {4, "r8d"},
    {5, "r9d"}
};
```

```
// Global.hpp
#include <map>
#include <string>

#ifndef GLOBAL_HPP
#define GLOBAL_HPP

using namespace std;

extern int SYMBOL_CATEGORIES[];

extern map<short, string> TOKEN_MAP;

extern map<int, string> PARAM_REG_MAP;

enum tokensType {
    NUMBER,
    KEYWORD,
    IDENTIFIER,
    SEPARATOR,
    ASSEMBLY_INSERTION,
    KEYWORD_OR_IDENDIFIER,
    END_OF_FILE
};

enum category {
    WHITESPACE,
    CONSTANT_START,
    IDENTIFIER_OR_KEYWORD_START,
    UNIQUE_SEPARATORS,
    AMBIGUES_SEPARATORS,
    PROHIBITED_CHARACTER
```

```
};

enum state {
    CNS,
    IDN,
    BCAI,
    COM,
    AI,
    ECOM,
    EAI,
    OUT
};



#endif



// InformationTables.cpp
#include "InformationTables.hpp"

InformationTables::InformationTables(ViewStream& vs) :
_nextNumberInOneCharacterSeperatorTable(0),
_nextNumberInMultiCharacterSeperatorTable(301),
_nextNumberInKeywordTable(601),
_nextNumberInConstantTable(1001),
_nextNumberInIdentifierTable(1000001),
_nextNumberInAssemblyInsertionTable(2000001),
_vs(vs),
_currentTokenIndex(0)
{
    initializeTables();
    // We can estimate size of file and reserve size for optimization
    _tokenTable.reserve(100000);
}

ViewStream& InformationTables::getViewStream() {
    return _vs;
}

void InformationTables::initializeTables() {
    initializeOneCharacterSeparatorTable();
    initializeMultiCharecterSeparatorTable();
    initializeKeywordTable();
}

void InformationTables::initializeOneCharacterSeparatorTable() {
    _oneCharacterSeparatorTable[";"] = 0;
    _oneCharacterSeparatorTable[","] = 1;
    _oneCharacterSeparatorTable["("] = 2;
    _oneCharacterSeparatorTable[")"] = 3;
    _oneCharacterSeparatorTable[":"] = 4;
    _nextNumberInOneCharacterSeperatorTable = 5;
}

void InformationTables::initializeMultiCharecterSeparatorTable() {
    _multiCharecterSeparatorTable["(*"] =
_nextNumberInMultiCharacterSeperatorTable++;
```

```
    _multiCharecterSeparatorTable["*)"] =
_nextNumberInMultiCharacterSeperatorTable++;
    _multiCharecterSeparatorTable["($"] =
_nextNumberInMultiCharacterSeperatorTable++;
    _multiCharecterSeparatorTable["$)"] =
_nextNumberInMultiCharacterSeperatorTable++;
}

void InformationTables::initializeKeywordTable() {
    _keywordTable["PROCEDURE"] = _nextNumberInKeywordTable++;
    _keywordTable["BEGIN"] = _nextNumberInKeywordTable++;
    _keywordTable["END"] = _nextNumberInKeywordTable++;
    _keywordTable["LABEL"] = _nextNumberInKeywordTable++;
    _keywordTable["GOTO"] = _nextNumberInKeywordTable++;
    _keywordTable["RETURN"] = _nextNumberInKeywordTable++;
}

void InformationTables::outputAllTables() {
    if (_oneCharacterSeparatorTable.size() > 0) {
        outputOneCharacterSeparatorTable();
        _vs << '\n';
    }
    if (_multiCharecterSeparatorTable.size() > 0) {
        outputMultiCharecterSeparatorTable();
        _vs << '\n';
    }
    if (_keywordTable.size() > 0) {
        outputKeywordTable();
        _vs << '\n';
    }
    if (_constantTable.size() > 0) {
        outputConstantTable();
        _vs << '\n';
    }
    if (_identifierTable.size() > 0) {
        outputIdentifierTable();
        _vs << '\n';
    }
    if (_asssemblyInsertionTable.size() > 0) {
        outputAssemblyInsertionTable();
        _vs << '\n';
    }
    if (_tokenTable.size() > 0) {
        outputTokenTable();
        _vs << '\n';
    }
}

void InformationTables::outputTokenTable() {
    _vs << "Token table\n";
    _vs << "Type      Code      Row       Col\n";
    for (const auto& pair : _tokenTable) {
        // string str = this->token_map[pair.code];
        _vs << Utilities::getLeftString(pair.type, 10);
        _vs << Utilities::getLeftString(pair.code, 10);
        _vs << Utilities::getLeftString(pair.row, 10);
        _vs << Utilities::getLeftString(pair.col, 10) << '\n';
    }
}

void InformationTables::outputOneCharacterSeparatorTable() {
```

```
    _vs << "Onecharacter separator table:\n";
    for (const auto& pair : _oneCharacterSeparatorTable) {
        _vs << pair.first << " -> " << pair.second << '\n';
    }
}

void InformationTables::outputMultiCharecterSeparatorTable() {
    _vs << "Multicharacter separator table:\n";
    for (const auto& pair : _multiCharecterSeparatorTable) {
        _vs << pair.first << " -> " << pair.second << '\n';
    }
}

void InformationTables::outputKeywordTable() {
    _vs << "Keyword table:\n";
    for (const auto& pair : _keywordTable) {
        _vs << pair.first << " -> " << pair.second << '\n';
    }
}

void InformationTables::outputConstantTable() {
    _vs << "Constant table:\n";
    for (const auto& pair : _constantTable) {
        _vs << pair.first << " -> " << pair.second << '\n';
    }
}

void InformationTables::outputIdentifierTable() {
    _vs << "Identifier table:\n";
    for (const auto& pair : _identifierTable) {
        _vs << pair.first << " -> " << pair.second << '\n';
    }
}

void InformationTables::outputAssemblyInsertionTable() {
    _vs << "Assembly insertion\n";
    for (const auto& pair : _asssemblyInsertionTable) {
        _vs << pair.first << " -> " << pair.second << '\n';
    }
}

void InformationTables::addToken(Token token) {
    _tokenTable.push_back(token);
}


void InformationTables::prosessToken(short tokenType, string tokenName, int row,
int col) {
    int code;
    switch (tokenType) {
        case NUMBER:
            code = _constantTable[tokenName];
            addToken(Token({NUMBER, code, row, col}));
            break;
        case KEYWORD:
            code = _keywordTable[tokenName];
            addToken(Token({KEYWORD, code, row, col}));
            break;
        case IDENTIFIER:
            code = _identifierTable[tokenName];
            addToken(Token({IDENTIFIER, code, row, col}));
```

```
            break;
        case SEPARATOR:
            code = _oneCharacterSeparatorTable[tokenName];
            addToken(Token({SEPARATOR, code, row, col}));
            break;
        case ASSEMBLY_INSERTION:
            string trimedTokenName = Utilities::trim(tokenName);

            addAssemblyInsertion(trimedTokenName);

            code = _asssemblyInsertionTable[trimedTokenName];
            addToken(Token({ASSEMBLY_INSERTION, code, row, col}));
            break;
    }
}

void InformationTables::addConstant(string constantValue) {
    if (!_constantTable.count(constantValue)) {
        _constantTable[constantValue] = _nextNumberInConstantTable++;
    }
}

void InformationTables::addIdentifier(string identifierValue) {
    if (!_identifierTable.count(identifierValue)) {
        _identifierTable[identifierValue] = _nextNumberInIdentifierTable++;
    }
}

void InformationTables::addAssemblyInsertion(string assemblyInsertion) {
    string trimedTokenName = Utilities::trim(assemblyInsertion);

    if (!_asssemblyInsertionTable.count(trimedTokenName)) {
        _asssemblyInsertionTable[assemblyInsertion] =
_nextNumberInAssemblyInsertionTable++;
    }
}

bool InformationTables::isKeyword(string token) {
    auto it = _keywordTable.find(token);

    if (it != _keywordTable.end()) {
        return true;
    } else {
        return false;
    }
}


Token InformationTables::getNextToken() {
    if (_tokenTable.size() - 1 >= (size_t)_currentTokenIndex) {
        return _tokenTable[_currentTokenIndex++];
    } else {
        return Token({END_OF_FILE, 0, 0, 0});
    }
}

string InformationTables::getTokenValue(int tokenCode) const {
    map<string, int> currentTokenTable;
    if (tokenCode < 300) {
        currentTokenTable = _oneCharacterSeparatorTable;
    } else if (tokenCode < 600) {
```

```
        currentTokenTable = _multiCharecterSeparatorTable;
    } else if (tokenCode < 1000) {
        currentTokenTable = _keywordTable;
    } else if (tokenCode < 1000000) {
        currentTokenTable = _constantTable;
    } else if (tokenCode < 2000000) {
        currentTokenTable = _identifierTable;
    } else {
        currentTokenTable = _asssemblyInsertionTable;
    }

    for (const auto& pair : currentTokenTable) {
        if (pair.second == tokenCode) {
            return pair.first;
        }
    }

    return "Unknown token";
}


// InformationTables.hpp
#include <string>
#include <map>
#include <vector>
#include <iomanip>
#include <iostream>
#include "../ViewStream.hpp"
#include "../Utilities/Utilities.hpp"
#include "Global.hpp"

#ifndef INFORMATAIONTABLES_HPP
#define INFORMATAIONTABLES_HPP

using namespace std;

typedef struct {
    short type;
    int code;
    int row;
    int col;
} Token;


class InformationTables {
    public:
    InformationTables(ViewStream& vs);
    void initializeTables();

    void outputOneCharacterSeparatorTable();
    void outputMultiCharecterSeparatorTable();
    void outputKeywordTable();
    void outputConstantTable();
    void outputIdentifierTable();
    void outputTokenTable();
    void outputAssemblyInsertionTable();
    void outputAllTables();

    bool isKeyword(string token);

    void addConstant(string constantValue);
```

```
    void addIdentifier(string identifierValue);
    void addAssemblyInsertion(string assemblyInsertion);

    void prosessToken(short tokenType, string tokenName, int row, int col);

    Token getNextToken();

    string getTokenValue(int tokenCode) const;

    ViewStream& getViewStream();

    private:
    map<string, int> _oneCharacterSeparatorTable;
    map<string, int> _multiCharecterSeparatorTable;
    map<string, int> _keywordTable;
    map<string, int> _constantTable;
    map<string, int> _identifierTable;
    map<string, int> _asssemblyInsertionTable;

    int _nextNumberInOneCharacterSeperatorTable;
    int _nextNumberInMultiCharacterSeperatorTable;
    int _nextNumberInKeywordTable;
    int _nextNumberInConstantTable;
    int _nextNumberInIdentifierTable;
    int _nextNumberInAssemblyInsertionTable;

    ViewStream _vs;

    vector<Token> _tokenTable;
    int _currentTokenIndex;

    void initializeOneCharacterSeparatorTable();
    void initializeMultiCharecterSeparatorTable();
    void initializeKeywordTable();

    void addToken(Token token);
};

#endif


// Tree.cpp
#include "Tree.hpp"

Tree::~Tree() {}

Tree::Tree(InformationTables& informationTables) :
    _parent(nullptr),
    _informationTables(informationTables),
    _vs(_informationTables.getViewStream()) {}

void Tree::setParent(Tree* parent) {
    _parent = parent;
}

Tree* Tree::getParent() const {
    return _parent;
}

bool Tree::isNode() const {
    return false;
```

```
}

void Tree::add(Tree*) {}

void Tree::remove(Tree*) {}

void Tree::clearMemory() {}

// Node class implementation

Node::Node(InformationTables& informationTables, string nodeName) :
Tree(informationTables), _nodeName(move(nodeName)) {}

bool Node::isNode() const {
    return true;
}

void Node::add(Tree* child) {
    if (!child) {
        return;
    }
    _children.push_back(child);
    child->setParent(this);
}

void Node::remove(Tree* child) {
    if (!child) {
        return;
    }

    _children.remove(child);
    child->setParent(nullptr);
}

void Node::clearMemory() {
    for (auto child : _children) {
        child->clearMemory();
        delete child;
    }
    _children.clear();
}

void Node::print(int indent) const {
    for (int i = 0; i < indent; ++i) {
        _vs << " ";
    }
    _vs << _nodeName << '\n';
    for (const auto& child : _children) {
        child->print(indent + 2);
    }
}

list<Tree*> Node::getChildren() {
    return _children;
}

string Node::getName() const {
    return _nodeName;
}

// Leaf class implementation
```

```
Leaf::Leaf(InformationTables& informationTables, Token token) :
Tree(informationTables), _token(token) {}

void Leaf::print(int indent) const {
    for (int i = 0; i < indent; ++i) {
        _vs << " ";
    }
    _vs << _informationTables.getTokenValue(_token.code) << '\n';
}

string Leaf::getName() const {
    return _informationTables.getTokenValue(_token.code);
}

Token Leaf::getToken() const {
    return _token;
}


// Tree.hpp
#ifndef TREE_HPP
#define TREE_HPP

using namespace std;

#include <list>
#include <string>
#include "InformationTables.hpp"

class Tree {
    protected:
    Tree* _parent;
    InformationTables& _informationTables;
    ViewStream& _vs;

    public:
    virtual ~Tree();
    Tree(InformationTables& informationTables);

    void setParent(Tree* parent);
    Tree *getParent() const;

    virtual bool isNode() const;

    virtual void add(Tree* child);
    virtual void remove(Tree* child);

    virtual void clearMemory();

    virtual void print(int indent = 0) const = 0;

    virtual string getName() const = 0;
};

class Node : public Tree {
    public:
    Node(InformationTables& informationTables, string nodeName);
    bool isNode() const override;
    void add(Tree* child) override;
    void remove(Tree* child) override;
```

```
    void clearMemory() override;

    void print(int indent = 0) const override;

    virtual string getName() const override;

    list<Tree*> getChildren();

    protected:
    list<Tree*> _children;
    string _nodeName;
};

class Leaf : public Tree {
    private:
    Token _token;

    public:
    Leaf(InformationTables& InformationTables, Token token);

    virtual string getName() const override;

    Token getToken() const;

    void print(int indent = 0) const override;
};

#endif


// CodeGenerator.cpp
#include "CodeGenerator.hpp"

CodeGenerator::CodeGenerator(InformationTables& informationTables, ErrorManager&
errorManager, string filename) : _file(fstream(filename, std::ios::out)),
_it(informationTables), _em(errorManager) {}

CodeGenerator::~CodeGenerator() {
    _file.close();
}

void CodeGenerator::run(Tree* tree) {
    try {
        _tree = tree;
        if (tree->isNode()) {
            auto programChildren = dynamic_cast<Node*>(tree)->getChildren();
            for (Tree* child : programChildren) {
                // cout << child->getName() << endl;
                if (child->getName() == "<PROCEDURE_IDENTIFIER>") {
                    auto ch = dynamic_cast<Node*>(child)->getChildren();
                    _file << ch.front()->getName() << ":\n";
                    _file << setw(4) << ' ' << left << setw(10) << "push" <<
"rbp" << endl;
                    _file << setw(4) << ' ' << left << setw(10) << "mov" <<
"rbp, rsp" << endl;
                } else if (child->getName() == "<PARAMETERS_LIST>") {
                    auto ch = dynamic_cast<Node*>(child)->getChildren();

                    int i = 0;
                    for (Tree* param : ch) {
                        // auto paramLeaf = dynamic_cast<Leaf*>(param);
```

```
                        // Token token = paramLeaf->getToken();
                        _file << setw(4) << ' ' << left << setw(10) << "mov" <<
"DWORD PTR [rbp-" << 20 + 4*i << "], " << PARAM_REG_MAP[i] << endl;
                        i++;
                    }
                } else if (child->getName() == "<BLOCK>") {
                    block(child);
                }
            }
            _file << setw(4) << ' ' << left << setw(10) << "pop" << "rbp" <<
endl;
            _file << setw(4) << ' ' << left << setw(10) << "ret" << endl;
        }

        for (auto label : _labels) {
            if (!label.second) {
                stringstream ss;
                ss << "Semantic error: Unused declared label: " <<  label.first;
                throw runtime_error(ss.str());
            }
        }

    }
    catch (const runtime_error& e) {
        _em.addProgramError(e.what());
        // cout << "[]" << endl;
    }
}

void CodeGenerator::block(Tree* tree) {
    if (tree->isNode()) {
        auto blockChildren = dynamic_cast<Node*>(tree)->getChildren();
        for (Tree* child : blockChildren) {
            if (child->getName() == "<LABEL_DECLARATIONS>") {
                auto labels = dynamic_cast<Node*>(child)->getChildren();

                for (auto label : labels) {
                    auto labelLeaf = dynamic_cast<Leaf*>(label);
                    string labelStr =  _it.getTokenValue(labelLeaf-
>getToken().code);
                    _labels[stoi(labelStr)] = false;
                }
            } else if (child->getName() == "<STATEMENTS_LIST>") {
                auto statements = dynamic_cast<Node*>(child)->getChildren();

                for (auto state : statements) {
                    if (state->getName() == "<STATEMENT>") {
                        statement(state);
                    }
                }
            }
        }
    }
}


void CodeGenerator::statement(Tree* tree) {
    auto statementChildren = dynamic_cast<Node*>(tree)->getChildren();

    if (statementChildren.front()->getName() == "RETURN") {
```

```
        _file << setw(4) << ' ' << left << setw(10) << "pop" << "rbp" << endl;
        _file << setw(4) << ' ' << left << setw(10) << "ret" << endl;
    } else if (statementChildren.front()->getName() == "<ASSEMBLY_INSERTION>") {
        auto assemblyInsertionNode =
dynamic_cast<Node*>(statementChildren.front());

        printAssemblyInsertion(assemblyInsertionNode->getChildren().front());
    } else if (statementChildren.front()->getName() == "<STATEMENT_WITH_LABEL>")
{
        auto statementList = dynamic_cast<Node*>(statementChildren.front())-
>getChildren();

        int i = 0;
        for (auto labelStatement : statementList) {
            if (i == 0) {
                Leaf* labelSatementLeaf = dynamic_cast<Leaf*>(labelStatement);

                int labelValue = stoi(_it.getTokenValue(labelSatementLeaf-
>getToken().code));

                if (_labels.count(labelValue) != 0) {
                    _labels[labelValue] = true;
                    _file << setw(2) << ' ' << labelValue << ':' << endl;
                } else {
                    stringstream ss;
                    ss << "Semantic error: Used undeclared label: " <<
labelValue;
                    throw runtime_error(ss.str());
                }
            } else if (i == 1) {
                statement(labelStatement);
            }
            i++;
        }
    } else if (statementChildren.front()->getName() == "GOTO") {
        if (statementChildren.size() >= 2) {
            auto it = statementChildren.begin();
            ++it;
            auto gotoNumberLeaf = dynamic_cast<Leaf*>(*it);
            _file << setw(4) << ' ' << left << setw(10) << "jump" <<
_it.getTokenValue(gotoNumberLeaf->getToken().code) << endl;
        }
    }
}


void CodeGenerator::printAssemblyInsertion(Tree* tree) {
    Leaf* leaf = dynamic_cast<Leaf*>(tree);
    istringstream ss(_it.getTokenValue(leaf->getToken().code));
    string line;

    while (getline(ss, line)) {
        transform(line.begin(), line.end(), line.begin(),
                  [](unsigned char c) { return std::tolower(c); });
        _file << setw(4) << ' ' << left << setw(10) << Utilities::trim(line) <<
endl;
    }
}


// CodeGenerator.hpp
```

```
#ifndef CODEGENERATOR_HPP
#define CODEGENERATOR_HPP

#include <string>
#include <fstream>
#include <iomanip>
#include <map>
#include <sstream>
#include <algorithm>
#include <cctype>

#include "../DataManagement/Tree.hpp"
#include "../DataManagement/Global.hpp"
#include "Utilities.hpp"
#include "ErrorManager.hpp"

class CodeGenerator {
    public:
    CodeGenerator(InformationTables& informationTables, ErrorManager&
errorManager, string filename);
    ~CodeGenerator();
    void run(Tree* tree);

    private:
    fstream _file;
    Tree* _tree;
    map<int, bool> _labels;
    InformationTables& _it;
    ErrorManager& _em;

    void block(Tree* tree);
    void statement(Tree* tree);

    void printAssemblyInsertion(Tree* tree);
};

#endif


// ErrorManager.cpp
#include "ErrorManager.hpp"

ErrorManager::ErrorManager(ViewStream& vs) : _vs(vs) {}

void ErrorManager::addProgramError(string errorMessage) {
    this->_programErrorMessages.push_back(errorMessage);
}

void ErrorManager::addCompilingError(string errorMessage) {
    this->_compilingErrorMessages.push_back(errorMessage);
}

void ErrorManager::output() {
    if (_compilingErrorMessages.size() > 0 || _programErrorMessages.size() > 0)
{
        _vs << "There are " << to_string(_compilingErrorMessages.size() +
_programErrorMessages.size()) << " errors in the program:\n";
        for (auto& err : _programErrorMessages) {
            _vs << err << '\n';
        }
        for (auto& err : _compilingErrorMessages) {
```

```
            _vs << err << '\n';
        }
    } else {
        _vs << "There are no errors in a program.\n";
    }
}


// ErrorManager.hpp
#include <string>
#include <vector>
#include "../ViewStream.hpp"

#ifndef ERRORMANAGER_HPP
#define ERRORMANAGER_HPP

using namespace std;

class ErrorManager
{
    public:
    ErrorManager(ViewStream& vs);
    void addProgramError(string errorMessage);
    void addCompilingError(string errorMessage);

    void output();

    private:
    ViewStream _vs;
    vector<string> _programErrorMessages;
    vector<string> _compilingErrorMessages;
};

#endif


// Parser.cpp
#include "Parser.hpp"

Parser::Parser(InformationTables& informationTables, ErrorManager& em, string
filepath) : _informationTables(informationTables),
_vs(_informationTables.getViewStream()), _em(em), _filepath(filepath),
_currentToken(_informationTables.getNextToken()), _root(nullptr) {}

Tree* Parser::getRoot() const {
    return _root;
}

void Parser::eat(short type, string message = "undefined") {
    if (_currentToken.code != type || _currentToken.type == END_OF_FILE) {
        stringstream ss;
        if (_currentToken.type == END_OF_FILE) {
            ss << Utilities::getErrorMessage(_filepath, -1, -1, "In", message)
<< " part of syntax was end of file without next token:" <<
_informationTables.getTokenValue(_currentToken.code);
            throw runtime_error(ss.str());
        }
        ss << Utilities::getErrorMessage(_filepath, _currentToken.row,
_currentToken.col, "In", message) << " part of syntax was used prohibited token
" <<  _informationTables.getTokenValue(_currentToken.code) << " or nessasary
token was missed";
```

```
        throw runtime_error(ss.str());
    }
    _currentToken = _informationTables.getNextToken();
}

void Parser::eatType(short type, string message = "undefined") {
    if (_currentToken.type != type) {
        stringstream ss;
        ss << Utilities::getErrorMessage(_filepath, _currentToken.row,
_currentToken.col, "In", message) << " part of syntax was used prohibited token
" <<  _informationTables.getTokenValue(_currentToken.code) << " or nessasary
token was missed";
        throw runtime_error(ss.str());
    }
    _currentToken = _informationTables.getNextToken();
}

void Parser::parse() {
    try {
        Tree* tree = program();
        // Token token = _informationTables.getNextToken();
        // Tree * root = new Node(_informationTables, "root");
        // Tree* leaf1  = new Leaf(_informationTables, token);
        // root->add(leaf1);
        // Tree* leaf2  = new Leaf(_informationTables,
_informationTables.getNextToken());
        // root->add(leaf2);
        // Tree* leaf3  = new Leaf(_informationTables,
_informationTables.getNextToken());
        // root->add(leaf3);
        // root->print();

        // root->clearMemory();
        // delete root;

        _root = tree;
    }
    catch (const exception& e) {
        _em.addProgramError(e.what());
        if (_root) {
            _root->clearMemory();
            delete _root;
            _root = nullptr;
        }
    }
    catch (...) {
        _em.addProgramError("Unknown error occurred in parser.");
        if (_root) {
            _root->clearMemory();
            delete _root;
            _root = nullptr;
        }
    }

}

Tree* Parser::program() {
    Tree* root = new Node(_informationTables, "<PROGRAM>");

    eat(PROCEDURE, "PROGRAM");
```

```
    root->add(procedureIndetifier());

    if (_currentToken.code == LEFT_PARENTHESIS) {
        eat(LEFT_PARENTHESIS, "PARAMETER_LIST");
        root->add(parametersList());
        eat(RIGHT_PARENTHESIS, "PARAMETER_LIST");
    }

    eat(SEMICOLON, "PROGRAM");

    root->add(block());

    eat(SEMICOLON, "PROGRAM");

    return root;
}

Tree* Parser::procedureIndetifier() {
    Tree* node = new Node(_informationTables, "<PROCEDURE_IDENTIFIER>");
    node->add(new Leaf(_informationTables, _currentToken));
    eatType(IDENTIFIER);
    return node;
}

Tree* Parser::block() {
    Tree* node = new Node(_informationTables, "<BLOCK>");

    if (_currentToken.code == LABEL) {
        node->add(labelDeclarations());
    }

    eat(BEGIN, "BLOCK");

    node->add(statementsList());

    eat(END, "BLOCK");

    return node;
}

Tree* Parser::statementsList() {
    Tree* node = new Node(_informationTables, "<STATEMENTS_LIST>");
    while (_currentToken.code != END) {
        node->add(statement());
    }

    return node;
}

Tree* Parser::statement() {
    Tree* node = new Node(_informationTables, "<STATEMENT>");

    switch (_currentToken.code) {
        case GOTO:
            {
                Tree* gotoNode = new Node(_informationTables, "GOTO");
                node->add(gotoNode);
                eat(GOTO, "GOTO");
                node->add(new Leaf(_informationTables, _currentToken));
                eatType(NUMBER, "GOTO");
                eat(SEMICOLON, "GOTO");
```

```
                break;
            }

        case RETURN:
            node->add(new Leaf(_informationTables, _currentToken));
            eat(RETURN, "RETURN");
            eat(SEMICOLON, "RETURN");
            break;

        default:
            {
                if (_currentToken.type == NUMBER) {
                    Tree* statementNode = new Node(_informationTables,
"<STATEMENT_WITH_LABEL>");
                    node->add(statementNode);

                    statementNode->add(new Leaf(_informationTables,
_currentToken));
                    eatType(NUMBER, "STATEMENT_WITH_LABEL");
                    eat(COLON, "STATEMENT_WITH_LABEL");
                    statementNode->add(statement());
                } else if (_currentToken.type == ASSEMBLY_INSERTION) {
                    Tree* assemblyInsertionNode = new Node(_informationTables,
"<ASSEMBLY_INSERTION>");
                    node->add(assemblyInsertionNode);
                    assemblyInsertionNode->add(new Leaf(_informationTables,
_currentToken));
                    eatType(ASSEMBLY_INSERTION, "ASSEMBLY_INSERTION");

                } else {
                    throw runtime_error(Utilities::getErrorMessage(_filepath,
_currentToken.row, _currentToken.col, "In STATEMENT part of program is
unexpected token or nessasary was missed:",
_informationTables.getTokenValue(_currentToken.code)));
                }
            }
    }
    return node;
}

Tree* Parser::parametersList() {
    Tree* node = new Node(_informationTables, "<PARAMETERS_LIST>");
    node->add(new Leaf(_informationTables, _currentToken));
    eatType(IDENTIFIER, "PARAMETERS_LIST");

    while (_currentToken.code == COMMA) {
        eat(COMMA, "PARAMETERS_LIST");
        node->add(new Leaf(_informationTables, _currentToken));
        eatType(IDENTIFIER, "PARAMETERS_LIST");
    }

    return node;
}

Tree* Parser::labelDeclarations() {
    Tree* node = new Node(_informationTables, "<LABEL_DECLARATIONS>");
    eat(LABEL, "LABEL_DECLARATIONS");

    node->add(new Leaf(_informationTables, _currentToken));
    eatType(NUMBER, "LABEL_DECLARATIONS");
```

```
    while (_currentToken.code == COMMA) {
        eat(COMMA, "LABEL_DECLARATIONS");
        node->add(new Leaf(_informationTables, _currentToken));
        eatType(NUMBER, "LABEL_DECLARATIONS");
    }

    eat(SEMICOLON, "LABEL_DECLARATIONS");

    return node;
}


// Parser.hpp
#ifndef PARSER_HPP
#define PARSER_HPP

#include "../DataManagement/InformationTables.hpp"
#include "../DataManagement/Tree.hpp"
#include "../Utilities/ErrorManager.hpp"
#include "../Utilities/Utilities.hpp"
#include "../DataManagement/Global.hpp"

class Parser {
    public:
    enum TokenValue {
        SEMICOLON = 0,
        COMMA = 1,
        LEFT_PARENTHESIS = 2,
        RIGHT_PARENTHESIS = 3,
        COLON = 4,
        COMMENT_START = 301,
        COMMENT_END = 302,
        ASSEMBLY_INSERTION_START = 303,
        ASSEMBLY_INSERTION_END = 304,
        PROCEDURE = 601,
        BEGIN = 602,
        END = 603,
        LABEL = 604,
        GOTO = 605,
        RETURN = 606,
    };

    Parser(InformationTables& informationTables, ErrorManager& errorManager,
string filepath);
    void parse();
    Tree* getRoot() const;

    private:
    InformationTables &_informationTables;
    ViewStream &_vs;
    ErrorManager &_em;
    string _filepath;
    Token _currentToken;
    Tree* _root;

    void eat(short type, string message);
    void eatType(short type, string message);

    Tree* program();
    Tree* procedureIndetifier();
    Tree* block();
```

```
    Tree* statementsList();
    Tree* statement();
    Tree* parametersList();
    Tree* labelDeclarations();
};

#endif


// Utilities.cpp
#include "Utilities.hpp"

string Utilities::getLeftString(int value, int width) {
    return getLeftString(to_string(value), width);
}


string Utilities::getLeftString(string str, int width) {
    ostringstream ss;
    ss << left << setw(width) << str;
    return ss.str();
}

string Utilities::getErrorMessage(string filename, int row, int col, string
errorMessage, string problemPart) {
    ostringstream ss;
    if (row != -1) {
        ss << filename << ':' << row << ':' << col << ": error: " <<
errorMessage << ' ' << problemPart;
    } else {
        ss << filename << ": error: " << errorMessage << ' ' << problemPart;
    }
    return ss.str();
}

string Utilities::trim(const string& str) {
    if (str.empty()) return str;

    size_t start = str.find_first_not_of(" \t\n\r");
    if (start == string::npos) return "";

    size_t end = str.find_last_not_of(" \t\n\r");

    return str.substr(start, end - start + 1);
}


// Utilities.hpp
#include <string>
#include <sstream>
#include <iostream>
#include <iomanip>
#include <algorithm>
#include "../DataManagement/Global.hpp"

#ifndef UTILITIES_HPP
#define UTILITIES_HPP

using namespace std;

class Utilities {
```

```
    public:
    static string getLeftString(int value, int width);
    static string getLeftString(string value, int width);
    static string getErrorMessage(string filename, int row, int col, string
errorMessage, string problemPart);
    static string trim(const string& str);
};

#endif
```