НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування та спеціалізованих комп'ютерних систем

Розрахунково-графічна робота

з дисципліни

«Основи проектування трансляторів»

Тема **«РОЗРОБКА СИНТАКСИЧНОГО АНАЛІЗАТОРА»**

Виконав: студент III курсу

ФПМ групи КВ-84

Байдаус М.В.

Перевірив:

Київ – 2021

Варіант 3

Варіант 3

```
1.    <signal-program> --> <program>
2.    <program> --> PROGRAM <procedure-identifier> ;
              <block>.
3.    <block> --> <declarations> BEGIN <statements-
              list> END
4.    <declarations> --> <constant-declarations>
5.    <constant-declarations> --> CONST <constant-
              declarations-list> |
              <empty>
6.    <constant-declarations-list> --> <constant-
              declaration> <constant-declarations-
              list> |
              <empty>
7.    <constant-declaration> --> <constant-
              identifier> = <constant>;
8.    <statements-list> --> <statement> <statements-
              list> |
              <empty>
9.    <statement> --> <variable-identifier> :=
              <constant> ;
10.   <constant> --> <unsigned-integer>
11.   <constant> --> - <unsigned-integer>
12.   <constant-identifier> --> <identifier>
13.   <variable-identifier> --> <identifier>
14.   <procedure-identifier> --> <identifier>
15.   <identifier> --> <letter><string>
16.   <string> --> <letter><string> |
              <digit><string> |
              <empty>
17.   <unsigned-integer> --> <digit><digits-string>
18.   <digits-string> --> <digit><digits-string> |
              <empty>
19.   <digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
              9
20.   <letter> --> A | B | C | D | ... | Z
```

Було використано алгоритм рекурсивного спуску

Код програми

main.cpp

```
//
//  main.cpp
//  lab1OPT
//
//  Created by Michael on 19.05.2021.
//  Copyright © 2021 Michael. All rights reserved.
//

#include <iostream>
#include "lexer.hpp"
#include "parser.hpp"

std::stringstream   m_Errors;
```

```
std::ofstream       out;
//output
std::vector<Token>   m_tokens;


int main(int argc, char** argv) {
   if (argc !=2 ){
        std::cout << "Invalid arguments!\n";
        return 1;
    }
    TokenAnalyzer Analyzer(argv[1]);
    Analyzer.parseFile();
    Analyzer.printLog();

    Parser parser(argv[1], Analyzer.getTokens(), Analyzer.getConstantsTable(), Analyzer.ge
tUserIdentifiersTable());
    parser.doParsing();
    parser.printLog();
    return 0;
}
```

common.hpp

```
//
//  common.h
//  lab1OPT
//
//  Created by Michael on 22.05.2021.
//  Copyright © 2021 Michael. All rights reserved.
//

#pragma once
#include <fstream>
#include <deque>
#include <sstream>
#include <vector>


struct Token {
    int code;
    int line = 1;
    int column = 1;

    std::string name;

    void clear() {
        name.clear();
    }
};


struct terminal_t{
    terminal_t(const std::string& name) : name(name) {}
    void addSubterm(const terminal_t& );
```

```
    std::string name;
    std::vector<terminal_t> subTerminals;
};

    extern std::stringstream   m_Errors;

    extern std::ofstream       out;
    //output
    extern std::vector<Token>   m_tokens;
```

lexer.hpp

```
//
//  lexer.hpp
//  lab1OPT
//
//  Created by Michael on 19.05.2021.
//  Copyright © 2021 Michael. All rights reserved.
//


#pragma once

#include <iostream>
#include <fstream>
#include <string>
#include <deque>
#include <array>
#include <sstream>
#include <vector>
#include <iomanip>

#include "common.hpp"



#define CONST_START_VALUE       501
#define USERIDENT_START_VALUE   1001
#define ROZDILN_START_VALUE     301

#define WHITESPACE  0
#define DIGIT       1
#define LETTER      2
#define DM1         3 // ; . =
#define DM2         4 // -
#define DM3         5 // :
#define COM         6 // (
#define ERR         7 // rest symbols


class TokenAnalyzer {
public:
    TokenAnalyzer(std::string path);
```

```
    void parseFile();
    void printLog();
    std::vector<Token>   getTokens() const;
    std::deque<Token>   getConstantsTable() const;
    std::deque<Token>   getUserIdentifiersTable() const;
private:
    char readChar();

    void whitespace();
    void digit(bool toClear = true);
    void letter();
    void dm1();
    void dm2();
    void dm3();
    void com();
    void err();

    bool isWhitespace(char ch) const;
    bool isDigit(char ch) const;
    bool isLetter(char ch) const;
    bool isDM1(char ch) const;
    bool isDM2(char ch) const;
    bool isDM3(char ch) const;
    bool isCom(char ch) const;

    size_t exists(const std::deque<Token>& table, const Token& token) const;

private:
    std::array<unsigned char, 128> m_Attributes;
    std::string path;
    //array of read tokens
    std::deque<Token> m_Tokens;

    //array of 'rozdilnik(:=)'
    std::deque<Token>   m_Rozdilnik;
    //array of reserved keywords
const std::deque<Token> m_Reserved = {
    Token{401, 0, 0, "PROGRAM"},
    Token{402, 0, 0, "BEGIN"},
    Token{403, 0, 0, "END"},
    Token{404, 0, 0, "CONST"}
};




//array of user identifiers
    std::deque<Token>   m_UserIdent;
//array of constants
    std::deque<Token>   m_Constants;
```

```
    std::ifstream       m_InputFile;

    Token               m_CurrentToken;
    size_t              m_CurrentX = 1;
    char                m_CurrentChar;

};
```

lexer.cpp

```
//
//  lexer.cpp
//  lab1OPT
//
//  Created by Michael on 19.05.2021.
//  Copyright © 2021 Michael. All rights reserved.
//

#include "lexer.hpp"
#include "out.hpp"

TokenAnalyzer::TokenAnalyzer(std::string path)
    :path(path)
 {

    m_InputFile.open(path + "input.sig");
    if (!m_InputFile.is_open()){
        std::cout<< "Error while opening file " << path + "input.sig\n";
        exit(1);
    }

    for (size_t i = 0; i < 128; i++){
        if (((i >= 8) && (i <= 13)) || i == 32) {
            m_Attributes[i] = WHITESPACE;
        } else if ((i >= '0') && (i <= '9')) {
            m_Attributes[i] = DIGIT;
        } else if ((i >= 'A') && (i <= 'Z')) {
            m_Attributes[i] = LETTER;
        } else if ((i == ';') || (i == '.') || (i == '=')) {
            m_Attributes[i] = DM1;
        } else if (i == '-') {
            m_Attributes[i] = DM2;
        } else if (i == ':') {
            m_Attributes[i] = DM3;
        } else if (i == '(') {
            m_Attributes[i] = COM;
        } else {
            m_Attributes[i] = ERR;
        }
    }
}


void TokenAnalyzer::parseFile() {
```

```
     m_CurrentChar = readChar();
     while (!m_InputFile.eof()) {
          switch (m_Attributes[m_CurrentChar]) {
          case WHITESPACE:
               whitespace();
               break;
          case DIGIT:
               digit();
               break;
          case LETTER:
               letter();
               break;
          case DM1:
               dm1();
               break;
          case DM2:
               dm2();
               break;
          case DM3:
               dm3();
               break;
          case COM:
               com();
               break;
          case ERR:
               err();
               break;
          }
     }

}

char TokenAnalyzer::readChar() {
    char tmp = m_InputFile.get();
    //++m_CurrentX;
    return tmp;
}

void TokenAnalyzer::whitespace() {
    /*m_CurrentChar = readChar();
    ++m_CurrentX;*/

    while (m_Attributes[m_CurrentChar] == WHITESPACE && !m_InputFile.eof()) {
        if (m_CurrentChar == '\n') {
            m_CurrentToken.line++;
            m_CurrentX = 0;
        }
        m_CurrentChar = readChar();
        ++m_CurrentX;
    }
}

void TokenAnalyzer::digit(bool toClear) {
     if (toClear) {
```

```
        m_CurrentToken.clear();
        m_CurrentToken.column = m_CurrentX;
    }

    while (isDigit(m_CurrentChar)) {
        m_CurrentToken.name += m_CurrentChar;
        m_CurrentChar = readChar();
        ++m_CurrentX;

    }


    if((m_CurrentToken.code = exists(m_Constants, m_CurrentToken)) == 0) {
        //if constant doesn't exist
        if (m_Constants.size() == 0) {
            m_CurrentToken.code = CONST_START_VALUE;
        }
        else {
            m_CurrentToken.code = CONST_START_VALUE + m_Constants.size();
        }
         m_Constants.push_back(m_CurrentToken);
    }

    m_tokens.push_back(m_CurrentToken);
}

void TokenAnalyzer::letter() {
    m_CurrentToken.clear();
    m_CurrentToken.column = m_CurrentX;
    while ((isLetter(m_CurrentChar)) || (isDigit(m_CurrentChar))){
        m_CurrentToken.name += m_CurrentChar;
        ++m_CurrentX;
        m_CurrentChar = readChar();
    }

    if((m_CurrentToken.code = exists(m_Reserved, m_CurrentToken)) == 0){


        if((m_CurrentToken.code = exists(m_UserIdent, m_CurrentToken)) == 0) {
               //if constant doesn't exist
               if (m_UserIdent.size() == 0) {
                   m_CurrentToken.code = USERIDENT_START_VALUE;
               }
               else {
                   m_CurrentToken.code = USERIDENT_START_VALUE + m_UserIdent.size();
               }
               m_UserIdent.push_back(m_CurrentToken);
          }

        }
    m_tokens.push_back(m_CurrentToken);
}


void TokenAnalyzer::dm1() {
```

```
    m_CurrentToken.clear();
    m_CurrentToken.column = m_CurrentX;
    m_CurrentToken.name = m_CurrentChar;
    m_CurrentToken.code = m_CurrentChar;
    m_tokens.push_back(m_CurrentToken);
    ++m_CurrentX;
    m_CurrentChar = readChar();
}

void TokenAnalyzer::dm2() {
    m_CurrentToken.clear();
    m_CurrentToken.column = m_CurrentX;
    m_CurrentToken.name += m_CurrentChar;
    m_CurrentChar = readChar();
    if (isDigit(m_CurrentChar)) {
        digit(false);
    }
    else {
        m_Errors << "Lexer: Error (line " << m_CurrentToken.line << ", column " << m_Curre
ntX
        << "): Expected digit but '"
        << m_CurrentChar << "' detected\n";
        return;
    }
}

void TokenAnalyzer::dm3() {
    m_CurrentToken.clear();
    m_CurrentToken.column = m_CurrentX;
    m_CurrentToken.name += m_CurrentChar;
    m_CurrentChar = readChar();
    m_CurrentToken.name += m_CurrentChar;
    if(m_CurrentChar != '='){
        m_Errors << "Lexer: Error (line " << m_CurrentToken.line << ", column " << m_Curre
ntX
        << "): Expected '=' but '"
        << m_CurrentChar << "' found\n";
        return;
    }
    else {

        m_CurrentToken.code = ROZDILN_START_VALUE ;

        m_tokens.push_back(m_CurrentToken);
        m_CurrentChar = readChar();
        ++m_CurrentX;

    }

}

void TokenAnalyzer::com() {
     size_t col = m_CurrentX;
     size_t row = m_CurrentToken.line;
    m_CurrentToken.column = m_CurrentX;
```

```
    m_CurrentChar = readChar();
    bool isCOM = false;
    if(m_CurrentChar != '*'){
        m_Errors << "Lexer: Error (line " << m_CurrentToken.line << ", column " << m_Curre
ntX
        << "): Illegal symbol '(' detected\n";
        return;
    }
    //begin of commentar

    while(!m_InputFile.eof()){
        m_CurrentChar = readChar();
        ++m_CurrentX;
        if (m_CurrentChar == '\n'){
            m_CurrentToken.line++;
            m_CurrentX = 0;
            continue;
        }
        if (m_CurrentChar == '*') {
            isCOM = true;
            continue;
        }
        if(m_CurrentChar == ')' && isCOM) {
            m_CurrentChar = readChar();
            ++m_CurrentX;
            break;
        }
        if(m_InputFile.eof()){
            m_Errors << "Lexer: Error (line " << row << ", column " << col
            << "): unclosed commentar\n";
            break;
        }
        isCOM = false;
    }
}

void TokenAnalyzer::err() {
    std::string err ("Illegal symbol '");
    err += m_CurrentChar;
    err += "' detected";
    m_Errors << "Lexer: Error (line " << m_CurrentToken.line << ", column " << m_CurrentX
              << "): "<< err << std::endl;
    m_CurrentChar = readChar();
    ++m_CurrentX;
}


bool TokenAnalyzer::isWhitespace(char ch) const {
    return m_Attributes[ch] == WHITESPACE;
}

bool TokenAnalyzer::isDigit(char ch) const {
    return m_Attributes[ch] == DIGIT;
}
```

```cpp
bool TokenAnalyzer::isLetter(char ch) const {
    return m_Attributes[ch] == LETTER;
}

bool TokenAnalyzer::isDM1(char ch) const {
    return m_Attributes[ch] == DM1;
}

bool TokenAnalyzer::isDM2(char ch) const {
    return m_Attributes[ch] == DM2;
}

bool TokenAnalyzer::isDM3(char ch) const {
    return m_Attributes[ch] == DM3;
}

bool TokenAnalyzer::isCom(char ch) const {
    return m_Attributes[ch] == COM;
}


size_t TokenAnalyzer::exists(const std::deque<Token>& table, const Token& token) const {
    for (const Token& i : table) {
        if (i.name == token.name) {
            return i.code;
        }
    }
    return 0;
}


void TokenAnalyzer::printLog() {
    out.open(path + "generated.txt");
    if (!out.is_open()){
        out.close();
        throw("cannot open output file\n");
    }
    ShowTabTokens(m_tokens);
    OutputString("\nError table:\n");
    ShowTabError(m_Errors);
    OutputString("\nUser Identificator Table: \nName    Code\n");
    ShowTabIdent(m_UserIdent);
    OutputString("\nConstants Table: \nName    Code\n");
    ShowTabConst(m_Constants);
    CloseFile();
}



std::deque<Token>  TokenAnalyzer::getUserIdentifiersTable() const {
    return m_UserIdent;
}

std::deque<Token>  TokenAnalyzer::getConstantsTable() const {
    return m_Constants;
```

```
}

std::vector<Token>  TokenAnalyzer::getTokens() const {
    return ::getTokens();
}
```

out.hpp

```
//
//  out.hpp
//  lab1OPT
//
//  Created by Michael on 21.05.2021.
//  Copyright © 2021 Michael. All rights reserved.
//

#pragma once
#include <iomanip>
#include <fstream>
#include <deque>
#include <iterator>
#include <string>
#include "lexer.hpp"
#include "common.hpp"


void ShowTabIdent(std::deque<Token>& m_UserIdent);
void ShowTabConst(std::deque<Token>& m_Constants);

void ShowTabTokens(std::vector<Token>& m_tokens);
void ShowTabError(std::stringstream& m_Errors);

void OutputString(const std::string&str);
void CloseFile();



bool outPutIsOpen();
void addError(std::string error);
void outputTree(const terminal_t& tree, const std::string& point);
std::vector<Token>   getTokens();
```

out.cpp

```
//
//  out.cpp
//  lab1OPT
//
//  Created by Michael on 21.05.2021.
//  Copyright © 2021 Michael. All rights reserved.
//

#include "out.hpp"
```

```cpp
#include "lexer.hpp"
#include <iomanip>


void ShowTabConst(std::deque<Token>& m_Constants){
    if(m_Constants.empty()){

        out << "Table of Constants is empty\n";
        return;
    }
    std::deque<Token>::iterator it;
    for (it = m_Constants.begin(); it != m_Constants.end(); ++it){

        out << it->name << "---" << it->code << std::endl;
    }
}


void ShowTabIdent(std::deque<Token>& m_UserIdent){
    if(m_UserIdent.empty()){

        out << "Table of Identifiers is empty\n";
        return;
    }
    std::deque<Token>::iterator it;
    for (it = m_UserIdent.begin(); it != m_UserIdent.end(); it++){

        out << it->name << "---" << it->code << std::endl;
    }
}


void ShowTabError(std::stringstream& m_Errors) {

   out << m_Errors.str()<< std::endl;
}

void ShowTabTokens(std::vector<Token>& m_tokens) {
    if (m_tokens.empty()) {

        out << "No tokens\n";
        return;
    }
    std::vector<Token>::const_iterator it;
    for (it = m_tokens.begin(); it != m_tokens.end(); it++) {

         out << std::left << std::setw(6) << it->line
             << std::left << std::setw(8) << it->column
             << std::left << std::setw(8) << it->code
             << std::left << std::setw(6) << it->name << '\n';
    }
}

void OutputString(const std::string&str) {
    if (!out.is_open()) {
```

```
        return;
    }

    out << str;
}

void CloseFile() {
    out.close();
}




bool outPutIsOpen() {
    return out.is_open();
}


void addError(std::string error) {
    m_Errors << error << std::endl;
}


std::vector<Token> getTokens() {
    return m_tokens;
}

void outputTree(const terminal_t& tree, const std::string& point )  {
    out << point << tree.name << std::endl;
    //std::cout << point << tree.name << std::endl;
    std::string tmp = point;
    for (const terminal_t& node : tree.subTerminals) {
        outputTree(node, tmp + "..");
    }
}
```

parser.hpp

```
#ifndef PARSER_H
#define PARSER_H

#include <vector>
#include <map>
#include <string>
#include <deque>

#include "common.hpp"
#include "out.hpp"


class Parser{
public:
    Parser(const std::string& path, const std::vector<Token>& fileTokens, std::deque<Token
> ConstantsTable,
```

```
                std::deque<Token> UserIdentifiersTable) ;
    void doParsing();
    void printLog();
private:

    // functions to handle non-terminals
    bool signal_program();
    bool program(terminal_t&);
    bool block(terminal_t&);
    bool declarations(terminal_t&);
    bool const_declarations_list(terminal_t&);
    bool const_declarations(terminal_t&);
    bool const_declaration(terminal_t&);
    bool statement_list(terminal_t&);
    bool statement(terminal_t&);
    bool constant(terminal_t&);
    bool variable_identifier(terminal_t&);;
    bool constant_identifier(terminal_t&);
    bool procedure_identifier(terminal_t&);
    bool identifier(terminal_t&);
    bool unsigned_integer(terminal_t&);
    bool empty(terminal_t&);

    size_t isUserIdentifier(const std::string& word) const;
    size_t isUnsignedInteger(const std::string& word) const;
    size_t exists(const std::deque<Token>& table, const Token& token) const;
        //true is reading successful
    bool readNextToken();
    void printEOFError(const std::string& expected);
    void printExpectedError(const std::string& expected);

private:

    std::vector<Token>            m_FileTokens;
    std::vector<Token>::iterator              m_CurrentToken;
    std::deque<Token>   m_Constants;
    std::deque<Token>   m_UserIdent;
    terminal_t                    m_SyntaxTree;

    //bool                          m_ErrorFound;
    bool                          m_IsEmpty;

    std::string                   path;
};

#endif // PARSER_H
```

parser.cpp

```
#include "parser.hpp"

#include <cstring>
#include <string>
#include <deque>
```

```cpp
void terminal_t::addSubterm(const terminal_t& newSubterm) {
    subTerminals.push_back(newSubterm);
}

Parser::Parser(const std::string& path, const std::vector<Token>& fileTokens,
               std::deque<Token> ConstantsTable,
               std::deque<Token> UserIdentifiersTable)
    :m_FileTokens(fileTokens), m_Constants(ConstantsTable), m_UserIdent(UserIdentifiersTab
le),
    m_SyntaxTree("<signal-program>"), path(path)
{
    m_CurrentToken = m_FileTokens.begin();
}

void Parser::doParsing() {
    if (m_CurrentToken == m_FileTokens.end()) {
        return;
    }

    signal_program();

}

void Parser::printLog() {
    //m_Logger.openOutput();
    out.open(path + "generated.txt");
    if (!out.is_open()) {
        throw ("Parser: Logger cannot open output file");
        return;
    }
    ShowTabTokens(m_tokens);
    OutputString("\n\n");

    outputTree(m_SyntaxTree, "");

    OutputString("\nError table:\n");
    ShowTabError(m_Errors);
    OutputString("\nUser Identificator Table: \nName    Code\n");
    ShowTabIdent(m_UserIdent);
    OutputString("\nConstants Table: \nName    Code\n");
    ShowTabConst(m_Constants);
    CloseFile();

}

bool Parser::signal_program() {
    m_SyntaxTree.addSubterm(terminal_t("<program>"));
    if (!program(m_SyntaxTree.subTerminals.back())) {
            return false;
    }
    return true;

}

bool Parser::program(terminal_t& terminal) {
```

```
    if (m_CurrentToken->name == "PROGRAM") {
        terminal.addSubterm(terminal_t(std::to_string(401) + " PROGRAM"));
        terminal.addSubterm(terminal_t("<procedure-identifier>"));

        if (!procedure_identifier(terminal.subTerminals.back())) {
            return false;
        }
        if (!readNextToken()) {
            printEOFError("';'");
            return false;
        }
        if (m_CurrentToken->name == ";") {
            terminal.addSubterm(terminal_t(std::to_string((int)';') + " ;"));
        } else {
            printExpectedError("';'");
            return false;
        }
        terminal.addSubterm(terminal_t("<block>"));

        if (!block(terminal.subTerminals.back())) {
            return false;
        }
        if (!readNextToken()) {
            printEOFError("'.'");
            return false;
        }
        if (m_CurrentToken->name == ".") {
            terminal.addSubterm(terminal_t(std::to_string((int)'.') + " ."));
        } else {
            printExpectedError("'.'");
            return false;
        }
    }
    else {
        printExpectedError("'PROGRAM'");
        return false;
    }
    return true;
}

bool Parser::block(terminal_t& terminal) {
    terminal.addSubterm(terminal_t("<declarations>"));

    if (!declarations(terminal.subTerminals.back())) {
        return false;
    }

    if (m_CurrentToken->name == "BEGIN") {
        terminal.addSubterm(terminal_t(std::to_string(402) + " BEGIN"));
    } else {
        printExpectedError("'BEGIN'");
        return false;
    }
```

```
    terminal.addSubterm(terminal_t("<statement_list>"));

    if (!statement_list(terminal.subTerminals.back())) {
        return false;
    }
    if (m_CurrentToken->name == "END") {
        terminal.addSubterm(terminal_t(std::to_string(403) + " END"));
    } else {
        printExpectedError("'END'");
        return false;
    }
    return true;
}

bool Parser::declarations(terminal_t& terminal) {
    terminal.addSubterm(terminal_t("<constant-declarations>"));

    if (!const_declarations(terminal.subTerminals.back())) {
        return false;
    }
    return true;
}

bool Parser::const_declarations(terminal_t& terminal) {

    if (!readNextToken()) {
        printEOFError("'CONST'");
        return false;
    }
    if (m_CurrentToken->name == "CONST") {
        terminal.addSubterm(terminal_t(std::to_string(404) + " CONST"));
        terminal.addSubterm(terminal_t("<constant-declarations-list>"));

        if (!const_declarations_list(terminal.subTerminals.back())) {
            return false;
        }
    } else {
        terminal.addSubterm(terminal_t("<empty>"));
    }
    return true;
}

bool Parser::const_declarations_list(terminal_t& terminal) {
    if (!readNextToken()) {
        printEOFError("<identifier>");
        return false;
    }
    if (isUserIdentifier(m_CurrentToken->name) != 0) {
        m_CurrentToken--;
        terminal.addSubterm(terminal_t("<constant-declaration>"));

        if (!const_declaration(terminal.subTerminals.back())) {
            return false;
        }
        terminal.addSubterm(terminal_t("<constant-declarations-list>"));
```

```
        if (!const_declarations_list(terminal.subTerminals.back())) {
            return false;
        }
    } else {
        terminal.addSubterm(terminal_t("<empty>"));
    }
    return true;
}

bool Parser::const_declaration(terminal_t& terminal) {
    terminal.addSubterm(terminal_t("<constant-identifier>"));

    if (!constant_identifier(terminal.subTerminals.back())) {
        //printExpectedError("'<identifier>'");
        return false;
    }
    if (!readNextToken()) {
        printEOFError("'='");
        return false;
    }
    if (m_CurrentToken->name == "=") {
        terminal.addSubterm(terminal_t(std::to_string((int)'=') + " ="));
    } else {
        printExpectedError("'='");
        return false;
    }
    terminal.addSubterm(terminal_t("<constant>"));

    if (!constant(terminal.subTerminals.back())) {
        return false;
    }
    if (!readNextToken()) {
        printEOFError("';'");
        return false;
    }
    if (m_CurrentToken->name == ";") {
        terminal.addSubterm(terminal_t(std::to_string((int)';') + " ;"));
    } else {
        printExpectedError("';'");
        return false;
    }
    return true;
}

bool Parser::statement_list(terminal_t& terminal) {
    if (!readNextToken()) {
        terminal.addSubterm(terminal_t("<empty>"));
        printEOFError("END");
        return false;
    }
    if (isUserIdentifier(m_CurrentToken->name) != 0 ) {
        terminal.addSubterm(terminal_t("<statement>"));
        m_CurrentToken--;
```

```
        if (!statement(terminal.subTerminals.back())) {
            return false;
        }
        terminal.addSubterm(terminal_t("<statement_list>"));

        if (!statement_list(terminal.subTerminals.back())) {
            return false;
        }

    } else {
        terminal.addSubterm(terminal_t("<empty>"));
    }
    return true;
}

bool Parser::statement(terminal_t& terminal) {
    terminal.addSubterm(terminal_t("<variable-identifier>"));

    if (!variable_identifier(terminal.subTerminals.back())) {
        return false;
    }
    if (!readNextToken()) {
        printEOFError("':='");
        return false;
    }
    terminal.addSubterm(terminal_t(std::to_string(301) + " :="));
    if (m_CurrentToken->name != ":=") {
        printExpectedError("':='");
        return false;
    }


    terminal.addSubterm(terminal_t("<constant>"));

    if (!constant(terminal.subTerminals.back())) {
        return false;
    }
    if (!readNextToken()) {
        printEOFError("';'");
        return false;
    }
    if (m_CurrentToken->name == ";") {
        terminal.addSubterm(terminal_t(std::to_string((int)';') + " ;"));
    } else {
        printExpectedError("';'");
        return false;
    }
    return true;

}



bool Parser::constant(terminal_t& terminal) {
    terminal.addSubterm(terminal_t("<unsigned-integer>"));
```

```
    if (!unsigned_integer(terminal.subTerminals.back())) {
        return false;
    }
    return true;
}

bool Parser::variable_identifier(terminal_t& terminal) {
    terminal.addSubterm(terminal_t("<identifier>"));

    if (!identifier(terminal.subTerminals.back())) {
        return false;
    }
    return true;
}

bool Parser::constant_identifier(terminal_t& terminal) {
    terminal.addSubterm(terminal_t("<identifier>"));

    if (!identifier(terminal.subTerminals.back())) {
        return false;
    }
    return true;
}

bool Parser::procedure_identifier(terminal_t& terminal) {
    terminal.addSubterm(terminal_t("<identifier>"));

    if (!identifier(terminal.subTerminals.back())) {
        return false;
    }
    return true;

}

bool Parser::identifier(terminal_t& terminal) {
    if (!readNextToken()) {
        printEOFError("<identifier>");
        return false;
    }
    size_t id = isUserIdentifier(m_CurrentToken->name);
    if (id) {
        terminal.addSubterm(terminal_t(std::to_string(id) + " " + m_CurrentToken->name));
        return true;
    }
    else {
         printExpectedError("<identifier>");
         return false;
    }
    return true;
}

bool Parser::unsigned_integer(terminal_t& terminal) {
    if (!readNextToken()) {
        printEOFError("<unsigned-integer>");
```

```
        return false;
    }
    size_t id = isUnsignedInteger(m_CurrentToken->name);
    if (id) {
        terminal.addSubterm(terminal_t(std::to_string(id) + " " + m_CurrentToken->name));
        return true;
    }
    else {
         printExpectedError("<unsigned-integer>");
         return false;
    }
    return true;
}

bool Parser::empty(terminal_t& terminal) {
    return true;

}

size_t Parser::isUserIdentifier(const std::string& word) const {
    for (auto& i : m_UserIdent) {
        if (i.name == word) {
            return i.code;
        }
    }
    return 0;
}

size_t Parser::isUnsignedInteger(const std::string& value) const {
        for (auto& i : m_Constants) {
        if (i.name == value) {
            return i.code;
        }
    }
    return 0;
}

bool Parser::readNextToken() {
    m_CurrentToken++;
    return m_CurrentToken != m_FileTokens.end();
}

void Parser::printEOFError(const std::string& expected) {

    auto tmp = m_CurrentToken - 1;
    m_Errors << "Parser: Error (line" << tmp->line << ", column "  << tmp->column + tmp-
>name.length() <<  "): Expected " << expected << " but 'EOF' found\n";

}

void Parser::printExpectedError(const std::string& expected) {

    m_Errors << "Parser: Error (line" << m_CurrentToken-
>line << ", column " << m_CurrentToken-
```

```
>column <<  "): Expected " << expected  <<  "' but '" << m_CurrentToken-
>name << "' found\n";


}
```

generator.cpp

```
#include "generator.hpp"

#include <string>

Generator::Generator(const std::string& path, const terminal_t& tree, const std::vector<To
ken>& tokens)
    :m_InputTree(tree), path(path)
{
}

void Generator::generate_code() {
    if (m_InputTree.name.empty()) {
        return;
    }
    // set tree's head to <program>
    terminal_t head = m_InputTree.subTerminals.front();
    m_Generated.push_back("\nCODE SEGMENT");
    m_Generated.push_back("ASSUME CS:CODE, DS:DATA");
    m_Generated.push_back(get_identifier(head.subTerminals[1]) + ":");
    m_ProgramName = get_identifier(head.subTerminals[1]);
    // set tree's head to <block>
    head = head.subTerminals[3];

    m_Generated.push_front(declarations(head.subTerminals.front()));

    m_Generated.push_back("push ebp");
    m_Generated.push_back("mov ebp, esp");

    // set head to 1st <statement-list>
    head = head.subTerminals[2];

    m_Generated.push_back(statement_list(head));

    m_Generated.push_back("pop ebp");
    m_Generated.push_back("ret");
    m_Generated.push_back("CODE ENDS");



    OutputString("\n");
    for (const std::string& str : m_Generated) {
        OutputString(str + "\n");
        m_Code << str << "\n" ;
    }

}
```

```
std::string Generator::declarations(terminal_t term) {

    std::string result = "DATA SEGMENT\n";

    // set to <constant-declarations-list>
    term = term.subTerminals.front().subTerminals.back();

    while (term.subTerminals.back().name != "<empty>") {
        auto newDeclaration = get_const_declaration(term.subTerminals.front());
        auto tmp = get_variable_terminal(term.subTerminals.front());
        if (!exists(m_DeclaredConstants, newDeclaration)) {
            if (newDeclaration.first != m_ProgramName)
            {
                m_DeclaredConstants.push_back(newDeclaration);
            }
            else {
                m_Errors << "Code generator: Error (line" << tmp.row << ", column " << tmp
.column <<  "): " << std::string(newDeclaration.first + " already exists") << "\n";
            }
        } else {
            m_Errors << "Code generator: Error (line" << tmp.row << ", column " << tmp.col
umn <<  "): " << std::string(newDeclaration.first + " already exists") << "\n";
        }
        // set to next <constant-declarations-list>
        term = term.subTerminals[1];
    }

    for (const std::pair<std::string, int>& constant : m_DeclaredConstants) {
        result += constant.first + " DWORD " + std::to_string(constant.second) + '\n';
    }

    result += "DATA ENDS";
    return result;
}

std::string Generator::statement_list(terminal_t term) {
    std::string result;
    if (term.subTerminals.front().name == "<empty>") {
        result += "nop\n";
        return result;
    }
    result += statement(term.subTerminals.front());
    result += statement_list(term.subTerminals.back());
    return result;
}

std::string Generator::statement(terminal_t term) {
    std::string result;


    auto newStatement = get_const_declaration(term);
    auto tmp = get_variable_terminal(term);
```

```
    if (newStatement.first == m_ProgramName) {
        m_Errors << "Code generator: Error (line" << tmp.row <<", column "  << tmp.column
<< "): " << std::string(newStatement.first + " is Program Name") << "\n";
        return result;
    }
    if ((exists(m_DeclaredConstants, newStatement))) {
        m_Errors << "Code generator: Error (line"  << tmp.row << ", column " << tmp.column
 << "): " << std::string(newStatement.first + " is constant") << "\n";
        return result;
    }
    result = "mov " + newStatement.first + "," + std::to_string(newStatement.second) + '\n
';

    return result;
}


std::string Generator::get_identifier(terminal_t term) {
    std::string identifier;
    term = term.subTerminals.front();
    if (term.name == "<identifier>") {
        term = term.subTerminals.front();
    }
    return get_last_word(term.name);
}

int Generator::get_constant(terminal_t term) {
    return std::atoi(get_last_word(term.subTerminals.front().subTerminals.front().name).c_
str());
}

std::pair<std::string, int> Generator::get_const_declaration(terminal_t term) {
    std::string identifier = get_identifier(term.subTerminals.front());
    int value = get_constant(term.subTerminals[2]);
    return std::make_pair(identifier, value);
}
//new
std::pair<std::string, int> Generator::get_statement(terminal_t term) {
    std::string identifier = get_identifier(term.subTerminals.front());
    int value = get_constant(term.subTerminals[2]);
    return std::make_pair(identifier, value);
}

std::string Generator::get_last_word( std::string str) {
    while( !str.empty() && std::isspace( str.back() ) ) str.pop_back() ;
    const auto pos = str.find_last_of( " \t\n" ) ;
    return pos == std::string::npos ? str : str.substr(pos+1) ;
}

bool Generator::exists(const std::deque<std::pair<std::string, int>>& array, const std::pa
ir<std::string,int>& value) {
    for (std::deque<std::pair<std::string, int>>::const_iterator it = array.begin(); it !=
 array.end(); ++it) {
        if (it->first == value.first) {
```

```
            return true;
        }
    }
    return false;
}

std::string Generator::add_instruction(terminal_t term) {
    return get_last_word(term.subTerminals.front().name);
}

terminal_t Generator::get_variable_terminal(terminal_t term) {
    return term.subTerminals.front().subTerminals.front().subTerminals.front();
}
```

generator.hpp

```
#include "common.hpp"
#include "out.hpp"
#include "parser.hpp"

#include <sstream>
#include <utility>
#include <deque>
#include <vector>
#include <iterator>

class Generator {
public:
    Generator(const std::string& path, const terminal_t& tree, const std::vector<Token>& t
okens);

    void generate_code();

private:
    std::string                 declarations(terminal_t term);
    std::string                 statement_list(terminal_t term);
    std::string                 statement(terminal_t term);
    std::string                 get_identifier(terminal_t term);
    int                         get_constant(terminal_t term);
    std::pair<std::string, int> get_const_declaration(terminal_t term);
    std::pair<std::string, int> get_statement(terminal_t term);
    std::string                 get_last_word(const std::string str);
    bool                        exists(const std::deque<std::pair<std::string, int>>& arra
y, const std::pair<std::string,int>& value);

    std::string                 add_instruction(terminal_t term);
    terminal_t                  get_variable_terminal(terminal_t);



private:

    std::deque<std::pair<std::string, int>>         m_DeclaredConstants;
    std::deque<std::pair<std::string, int>>         m_Statement;
    std::string                                     m_ProgramName;
```

```
    terminal_t                                  m_InputTree;
    std::deque<std::string>                     m_Generated;
    std::string                                 path;

    std::vector<std::string>                     m_CaseLabels;
};
```

Tests
Test01
input.sig

```
PROGRAM QWR;
CONST VAR1 = 1234;
VAR2 = 1234; VAR3 = 12345;
BEGIN
VAR5 := 123;
VAR6 := -123;
END.
```

generated.txt

```
1    1      401    PROGRAM
1    9      1001    QWR
1    12      59     ;
2    1      404    CONST
2    7      1002    VAR1
2    12      61     =
2    14      501     1234
2    18      59     ;
3    1      1003    VAR2
3    6      61     =
3    8      501    1234
3    12      59     ;
3    14      1004    VAR3
3    19      61     =
3    21      502     12345
3    26      59     ;
4    1      402    BEGIN
5    1      1005    VAR5
5    6      301    :=
5    8      503    123
5    11      59     ;
6    1      1006    VAR6
6    6      301    :=
6    8      504    -123
6    11      59     ;
7    1      403    END
7    4      46     .
```

<signal-program>
..<program>

....401 PROGRAM
....<procedure-identifier>
......<identifier>
........1001 QWR
....59 ;
....<block>
......<declarations>
........<constant-declarations>
..........404 CONST
..........<constant-declarations-list>
............<constant-declaration>
..............<constant-identifier>
................<identifier>
..................1002 VAR1
..............61 =
..............<constant>
................<unsigned-integer>
..................501 1234
..............59 ;
............<constant-declarations-list>
..............<constant-declaration>
................<constant-identifier>
..................<identifier>
....................1003 VAR2
................61 =
................<constant>
..................<unsigned-integer>
....................501 1234
................59 ;
..............<constant-declarations-list>
................<constant-declaration>
..................<constant-identifier>
....................<identifier>
......................1004 VAR3
..................61 =
..................<constant>
....................<unsigned-integer>
......................502 12345
..................59 ;
................<constant-declarations-list>
..................<empty>
......402 BEGIN
......<statement_list>
........<statement>
..........<variable-identifier>
............<identifier>
..............1005 VAR5
..........301 :=

.....<constant>
.....<unsigned-integer>
.....503 123
.....59 ;
.....<statement_list>
.....<statement>
.....<variable-identifier>
.....<identifier>
.....1006 VAR6
.....301 :=
.....<constant>
.....<unsigned-integer>
.....504 -123
.....59 ;
.....<statement_list>
.....<empty>
.....403 END
....46 .

Error table:

User Identificator Table:
Name Code
QWR---1001
VAR1---1002
VAR2---1003
VAR3---1004
VAR5---1005
VAR6---1006

Constants Table:
Name Code
1234---501
12345---502
123---503
-123---504

DATA SEGMENT
VAR1 DWORD 1234
VAR2 DWORD 1234
VAR3 DWORD 12345
DATA ENDS

CODE SEGMENT
ASSUME CS:CODE, DS:DATA
QWR:

```
push ebp
mov ebp, esp
mov VAR5,123
mov VAR6,-123
nop

pop ebp
ret
CODE ENDS
```

Test02

input.sig

```
PROGRAM QWR;
CONST VAR1 = 1234;
VAR2 = 1234; VAR3 = 12345;
VAR3 = 1234567;
BEGIN
VAR5 := 123;
VAR6 := -123;
VAR5 := -123;
VAR2 := 1234;
QWR := 234;
END.
```

generated.txt

```
1    1     401    PROGRAM
1    9     1001    QWR
1    12     59     ;
2    1     404    CONST
2    7     1002    VAR1
2    12     61     =
2    14     501     1234
2    18     59     ;
3    1     1003    VAR2
3    6     61     =
3    8     501    1234
3    12     59     ;
3    14     1004     VAR3
3    19     61     =
3    21     502     12345
3    26     59     ;
4    1     1004    VAR3
4    6     61     =
4    8     503    1234567
4    15     59     ;
5    1     402    BEGIN
6    1     1005    VAR5
```

```
6    6     301    :=
6    8     504    123
6    11     59    ;
7    1     1006   VAR6
7    6     301    :=
7    8     505    -123
7    11     59    ;
8    1     1005   VAR5
8    6     301    :=
8    8     505    -123
8    11     59    ;
9    1     1003   VAR2
9    6     301    :=
9    8     501    1234
9    12     59    ;
10   1     1001    QWR
10   5     301    :=
10   7     506    234
10   10     59    ;
11   1     403    END
11   4     46     .
```

```
<signal-program>
..<program>
....401 PROGRAM
....<procedure-identifier>
......<identifier>
........1001 QWR
....59 ;
....<block>
......<declarations>
........<constant-declarations>
..........404 CONST
..........<constant-declarations-list>
............<constant-declaration>
..............<constant-identifier>
................<identifier>
..................1002 VAR1
..............61 =
..............<constant>
................<unsigned-integer>
..................501 1234
..............59 ;
............<constant-declarations-list>
..............<constant-declaration>
................<constant-identifier>
..................<identifier>
```

.....1003 VAR2
.....61 =
.....<constant>
.....<unsigned-integer>
.....501 1234
.....59 ;
.....<constant-declarations-list>
.....<constant-declaration>
.....<constant-identifier>
.....<identifier>
.....1004 VAR3
.....61 =
.....<constant>
.....<unsigned-integer>
.....502 12345
.....59 ;
.....<constant-declarations-list>
.....<constant-declaration>
.....<constant-identifier>
.....<identifier>
.....1004 VAR3
.....61 =
.....<constant>
.....<unsigned-integer>
.....503 1234567
.....59 ;
.....<constant-declarations-list>
.....<empty>
.....402 BEGIN
.....<statement_list>
.....<statement>
.....<variable-identifier>
.....<identifier>
.....1005 VAR5
.....301 :=
.....<constant>
.....<unsigned-integer>
.....504 123
.....59 ;
.....<statement_list>
.....<statement>
.....<variable-identifier>
.....<identifier>
.....1006 VAR6
.....301 :=
.....<constant>
.....<unsigned-integer>
.....505 -123

.....59 ;
.....<statement_list>
.....<statement>
.....<variable-identifier>
.....<identifier>
.....1005 VAR5
.....301 :=
.....<constant>
.....<unsigned-integer>
.....505 -123
.....59 ;
.....<statement_list>
.....<statement>
.....<variable-identifier>
.....<identifier>
.....1003 VAR2
.....301 :=
.....<constant>
.....<unsigned-integer>
.....501 1234
.....59 ;
.....<statement_list>
.....<statement>
.....<variable-identifier>
.....<identifier>
.....1001 QWR
.....301 :=
.....<constant>
.....<unsigned-integer>
.....506 234
.....59 ;
.....<statement_list>
.....<empty>
.....403 END
....46 .

Error table:
Code generator: Error (line4, column 1): VAR3 already exists
Code generator: Error (line9, column 1): VAR2 is constant
Code generator: Error (line10, column 1): QWR is Program Name

User Identificator Table:
Name Code
QWR---1001
VAR1---1002
VAR2---1003
VAR3---1004

VAR5---1005
VAR6---1006

Constants Table:
Name Code
1234---501
12345---502
1234567---503
123---504
-123---505
234---506

```
DATA SEGMENT
VAR1 DWORD 1234
VAR2 DWORD 1234
VAR3 DWORD 12345
DATA ENDS

CODE SEGMENT
ASSUME CS:CODE, DS:DATA
QWR:
push ebp
mov ebp, esp
mov VAR5,123
mov VAR6,-123
mov VAR5,-123
nop

pop ebp
ret
CODE ENDS
```