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

Кафедра системного програмування та спеціалізованих комп'ютерних систем

Лабораторна робота № 2

з дисципліни

«Основи проектування трансляторів»

Тема: «РОЗРОБКА ГЕНЕРАТОРА КОДУ»

Виконав: студент IV курсу
групи КВ-84 ФПМ
Іванюк В.І.
Перевірив:

Київ
2021

Мета лабораторної 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 має забезпечувати:
• читання дерева розбору та таблиць, створених синтаксичним аналізатором, який було розроблено в розрахунково-графічній роботі;
• виявлення семантичних помилок;
• генерацію коду та/або побудову внутрішніх таблиць для генерації коду.

3. Входом генератора коду (ГК) мають бути:
• дерево розбору;
• таблиці ідентифікаторів та констант з повною інформацією, необхідною для генерації коду;
• вхідна програма на підмножині мови програмування SIGNAL згідно з варіантом (необхідна для формування лістингу програми).

4. Виходом ГК мають бути:
• асемблерний код згенерований для вхідної програми та/або внутрішні таблиці для генерації коду;
• внутрішні таблиці генератора коду (якщо потрібні).

5. 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 про лексичні, синтаксичні та семантичні помилки.

6. Входом компілятора має бути програма на підмножині мови програмування SIGNAL згідно з варіантом;

7. Виходом компілятора мають бути:
• асемблерний код згенерований для вхідної програми та/або внутрішні таблиці для генерації коду;
• лістинг вхідної програми з повідомленнями про лексичні, синтаксичні та семантичні помилки.

8. Для програмування може бути використана довільна алгоритмічна мова програмування високого рівня. Якщо обрана мова програмування має конструкції або бібліотеки для роботи з регулярними виразами, то використання цих конструкцій та/або бібліотек строго заборонено.

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

Лістинг програми мовою C++

OPT_lab1.cpp

```
#include "LexerGeneration.h"
#include "BinTree.h"
#include "CodeGenerationr.h"

int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Lexer: Invalid number of parameters.");
        return 1;
    }
    else {
        for (int i = 1; i < argc; i++) {
            printf("%s \n", argv[i]);
        }

    }

    FILE* test, * gen;
    char input[30];
    char output[30];
    char inputfile[] = "\\input.sig";
    char outputfile[] = "\\generated.txt";

    // For Visual Studio 2019
    strcpy_s(input, _countof(input), argv[1]);
    strcat_s(input, _countof(input), inputfile);
    strcpy_s(output, _countof(output), argv[1]);
    strcat_s(output, _countof(output), outputfile);
    errno_t err_test, err_gen;
    if ((err_test = fopen_s(&test, input, "r") != 0) || (err_gen = fopen_s(&gen,
output, "w") != 0)) {
        return 1;
    }

    // For g++
    /*strcpy(input, argv[1]);
    strcat(input, inputfile);
    strcpy(output, argv[1]);
    strcat(output, outputfile);
    if (((test = fopen(input, "r")) == NULL) || ((gen = fopen(output, "w")) ==
NULL)) {
        return 1;
    }*/



    else {
        if (!lexer(test, gen)) {
```

```
            parsing(gen);
            codeGeneraion(gen);
        }
        else
            fprintf(gen, "Pasrer cannt work : Lexer found errors\n");
        fclose(test);
        fclose(gen);
    }

    return 0;

}
```

LexerGeneration.h

```
#pragma once
#ifndef LEXERGENERATION_H
#define LEXERGENERATION_H

#include <iostream>
#include <string>

#include <cctype>
#include <algorithm>
#include <vector>
#include <typeinfo>
#include <cstring>
#include <stdio.h>
#include <map>
#include <list>
#include <fstream>
#include <unordered_map>
#include <vector>
#include <sstream>
#include <iostream>

using namespace std;

enum symbolCategories {
    whitespaces,
    digits,
    letters,
    unifier,
    separators,
    errors,
    tests
};

struct Token {
    Token() {};
    Token(int _row, int _column, int _id, string _value) {
```

```
        row = _row;
        column = _column;
        id = _id;
        value = _value;
    }
    int row, column, id;
    string value;
};



vector<Token> getVectorToken();
void printTables(FILE *gen);

/* File operations */
Token* dumpToken(FILE* generated, int row, int column, string token, Token*
tokenStruct, const int count);
void dumpLexError(FILE* generated, int row, int column, string undefinedToken);
void dumpTokError(FILE* generated, int row, int* column, char* err_symb, string
token, int count);

/* Struct operations */
Token* AddToken(int row, int column, int id, string token, Token* tokenStruct,
const  int count);
void showTokens();

/* Lexer operations */
bool lexer(FILE* test, FILE* gen);
int findID(string _token);
int symbolClassifier(char symbol);

#endif
```

LexerGeneration.cpp

```
#include "LexerGeneration.h"
vector<Token> token_vector;

vector<Token> getVectorToken() {
    return token_vector;
}

Token* dumpToken(FILE* generated, int row, int column, string token, Token*
tokenStruct, const int count) {
    tokenStruct = AddToken(row, column, findID(token), token, tokenStruct,
count);
    fprintf(generated, " %4d | %6d | %11d | %s\n", row, column, findID(token),
token.c_str());
    return tokenStruct;
}
```

```
void dumpTokError(FILE* generated, int row, int *column, char *err_symb, string
token, int count) {
    fprintf(generated, " Lexer : Error. Illegam symbol : ");
    for (int i = 0; i < count; i++) {
        fprintf(generated, "'%c'[%d, %d] ", err_symb[i], row, column[i]);
    }
    fprintf(generated, "in %s\n", token.c_str());
}

void dumpLexError(FILE* generated, int row, int column, string token) {
    fprintf(generated, " Lexer : Error. Illegam symbol : '%s'[%d, %d]\n",
token.c_str(), row, column);
}

Token* AddToken(int row, int column, int id, string token, Token* tokenStruct,
const  int count) {
    /*if (count == 0) {
        tokenStruct = new Token[count + 1];
    }
    else {
        Token* tmpToken = new Token[count + 1];
        for (int i = 0; i < count; i++) {
            tmpToken[i] = tokenStruct[i];
        }
        delete[] tokenStruct;

        tokenStruct = tmpToken;
    }
    tokenStruct[count].row = row;
    tokenStruct[count].column = column;
    tokenStruct[count].id = id;
    tokenStruct[count].value = token;*/

    Token tmp(row, column, id, token);
    token_vector.push_back(tmp);

    return tokenStruct;
}

void showTokens() {
    for (vector<Token>::iterator it = token_vector.begin(); it !=
token_vector.end(); it++ ) {
        cout << it->row << " | " << it->column << " | " << it->id << " | " << it-
>value << endl;
    }
}

int symbolClassifier(char symbol) {
    if (symbol == 32 || symbol == 13 || symbol == 10 || symbol == 9 || symbol ==
11 || symbol == 12) {
        return whitespaces;
```

```
    }
    else if (48 <= symbol && symbol <= 57) { //from '0' to '9'
        return digits;
    }
    else if ((65 <= symbol && symbol <= 90) || (97 <= symbol && symbol <= 122)) {
//from 'A' to 'Z' or from 'a' to 'z'
        return letters;
    }
    else if (symbol == 59 || symbol == 58 || symbol == 44 || symbol == 36 ||
symbol == 40 || symbol == 41) {
        return separators;
    }
    /*else if (symbol == 35) {
        return tests;
    }*/
    else if (symbol != -1) {
        return errors;
    }

}

bool lexer(FILE* test, FILE* gen) {
    fprintf(gen, " Line | Column | Ident token | Token\n-------------------------
------------------------\n");
    char symbol = fgetc(test);
    char buff[255], err_symbols[255];
    string lexem;
    int row = 1, column = 1, token_count = 0, buffLen, unifier_col, unifier_row,
err_count, err_column[255];
    bool err_flag = false;
    Token* token_struct = 0;
    bool error_check = false;

    while (symbol != -1) {
        switch (symbolClassifier(symbol)) {
        case whitespaces :
            while (symbolClassifier(symbol) == whitespaces) {
                if (symbol == 9) {
                    column += 6;
                }
                column++;
                if (symbol == 10) {
                    row++;
                    column = 1;
                }
                symbol = fgetc(test);
            }
            break;
        case digits:
            buffLen = 0;
            err_count = 0;
```

```
            while (symbolClassifier(symbol) == digits || symbolClassifier(symbol)
== errors
                || symbolClassifier(symbol) == letters)
            {
                if (symbolClassifier(symbol) == errors ||
symbolClassifier(symbol) == letters) {
                    err_flag = true;
                    err_symbols[err_count] = symbol;
                    err_column[err_count] = column + buffLen;
                    err_count++;
                }
                buff[buffLen] = symbol;
                buffLen++;
                symbol = fgetc(test);
            }
            buff[buffLen] = '\0';
            lexem = string(buff);
            if (err_flag == false) {
                token_struct = dumpToken(gen, row, column, lexem, token_struct,
token_count);
                token_count++;
            }
            else {
                error_check = true;
                dumpTokError(gen, row, err_column, err_symbols, lexem,
err_count);
            }
            column += buffLen;
            err_flag = false;
            break;
        case letters:
            buffLen = 0;
            err_count = 0;
            while (symbolClassifier(symbol) == digits || symbolClassifier(symbol)
== errors
                || symbolClassifier(symbol) == letters)
            {
                if (symbolClassifier(symbol) == errors) {
                    err_flag = true;
                    err_symbols[err_count] = symbol;
                    err_column[err_count] = column + buffLen;
                    err_count++;
                }
                buff[buffLen] = symbol;
                buffLen++;
                symbol = fgetc(test);
            }
            buff[buffLen] = '\0';
            lexem = string(buff);
            if (err_flag == false) {
```

```
                token_struct = dumpToken(gen, row, column, lexem, token_struct,
token_count);
                token_count++;
            }
            else {
                error_check = true;
                dumpTokError(gen, row, err_column, err_symbols, lexem,
err_count);
            }
            column += buffLen;
            err_flag = false;
            break;
        case separators:
            if (symbol == 59) { // ;
                token_struct = dumpToken(gen, row, column, ";", token_struct,
token_count);
                token_count++;
                column++;
                symbol = fgetc(test);
                break;
            }
            else if (symbol == 58) { //:
                token_struct = dumpToken(gen, row, column, ":", token_struct,
token_count);
                token_count++;
                column++;
                symbol = fgetc(test);
                break;
            }
            else if (symbol == 44) { // ,
                token_struct = dumpToken(gen, row, column, ",", token_struct,
token_count);
                token_count++;
                column++;
                symbol = fgetc(test);
                break;
            }
            if (symbol == 40) { // (
                unifier_row = row;
                unifier_col = column;
                symbol = fgetc(test);
                column++;
                if (symbol == 42) { // *
                    while (true) {
                        if (symbol == 10) {
                            row++;
                            column = 0;
                        }
                        if (symbol == -1) {
                            fprintf(gen, " Lexer : Error. Unclosed commet [%d,
%d]\n", unifier_row, unifier_col);
```

```
                            error_check = true;
                            break;
                        }
                        if (symbol == 42) {
                            column++;
                            symbol = fgetc(test);
                            if (symbol == 41) {
                                column++;
                                break;
                            }
                        }
                        else {
                            symbol = fgetc(test);
                            column++;
                        }
                    }
                    symbol = fgetc(test);
                }
                else {
                    token_struct = dumpToken(gen, unifier_row, unifier_col, "(",
token_struct, token_count);
                    token_count++;
                    break;
                }
            }
            else if (symbol == 41) { // )
                token_struct = dumpToken(gen, row, column, ")", token_struct,
token_count);
                token_count++;
                column++;
                symbol = fgetc(test);
                break;
            }
            else if (symbol == 36) { // $
                token_struct = dumpToken(gen, row, column, "$", token_struct,
token_count);
                token_count++;
                column++;
                symbol = fgetc(test);
                break;
            }
            break;
        case errors:
            error_check = true;
            buffLen = 0;
            err_count = 0;
            while (symbolClassifier(symbol) == digits || symbolClassifier(symbol)
== errors
                || symbolClassifier(symbol) == letters)
            {
                if (symbolClassifier(symbol) == errors) {
```

```
                    err_flag = true;
                    err_symbols[err_count] = symbol;
                    err_column[err_count] = column + buffLen;
                    err_count++;
                }
                buff[buffLen] = symbol;
                buffLen++;
                symbol = fgetc(test);
            }
            buff[buffLen] = '\0';
            lexem = string(buff);
            if (buffLen > 1)
                dumpTokError(gen, row, err_column, err_symbols, lexem,
err_count);
            else
                dumpLexError(gen, row, column, lexem);
            column += buffLen;
            err_flag = false;
            break;
        case tests:
            buffLen = 0;
            err_count = 0;
            buff[buffLen] = symbol;
            buffLen++;
            symbol = fgetc(test);
            //#38-0??-???-??-??
            for (buffLen; buffLen < 17; buffLen++) {
                if (buffLen == 1) {
                    if (symbol != 51) {
                        err_flag = true;
                        err_symbols[err_count] = symbol;
                        err_column[err_count] = column + buffLen;
                        err_count++;
                    }
                    buff[buffLen] = symbol;
                    symbol = fgetc(test);
                    continue;
                }
                if (buffLen == 2) {
                    if (symbol != 56) {
                        err_flag = true;
                        err_symbols[err_count] = symbol;
                        err_column[err_count] = column + buffLen;
                        err_count++;
                    }
                    buff[buffLen] = symbol;
                    symbol = fgetc(test);
                    continue;
                }
                if (buffLen == 3 || buffLen == 7 || buffLen == 11 || buffLen ==
14) {
```

```
                    if (symbol != 45) {
                        err_flag = true;
                        err_symbols[err_count] = symbol;
                        err_column[err_count] = column + buffLen;
                        err_count++;
                    }
                    buff[buffLen] = symbol;
                    symbol = fgetc(test);
                    continue;
                }
                if (buffLen == 4 ) {
                    if (symbol != 48) {
                        err_flag = true;
                        err_symbols[err_count] = symbol;
                        err_column[err_count] = column + buffLen;
                        err_count++;
                    }
                    buff[buffLen] = symbol;
                    symbol = fgetc(test);
                    continue;
                }
                if (symbolClassifier(symbol) == digits) {
                    buff[buffLen] = symbol;
                    symbol = fgetc(test);
                    continue;
                }
                else {
                    err_flag = true;
                    err_symbols[err_count] = symbol;
                    err_column[err_count] = column + buffLen;
                    err_count++;
                    buff[buffLen] = symbol;
                    symbol = fgetc(test);
                    continue;
                }
            }
            buff[buffLen] = '\0';
            lexem = string(buff);
            if (err_flag == false) {
                token_struct = dumpToken(gen, row, column, lexem, token_struct,
token_count);
                token_count++;
            }
            else {
                fprintf(gen, "  Lexer : Error. Incorrect number.\n ");
                dumpTokError(gen, row, err_column, err_symbols, lexem,
err_count);
            }
            column += buffLen;
            err_flag = false;
            break;
```

```
        }
    }
    showTokens();
    printTables(gen);
    return error_check;
}
```

LexerTables.cpp

```
#include "LexerGeneration.h"

int ident_count = 1001;
int const_count = 501;
int test_count = 5001;

map <string, int> kwrd = {
  {"PROCEDURE", 401},
  {"BEGIN", 402},
  {"END", 403},
  {"LABEL", 404},
  {"GOTO", 405},
  {"RETURN", 406}
};

map <string, int> sep = {
    {";", 59},
    {",", 44},
    {":", 58},
    {"(", 40},
    {")", 41},
    {"$", 36}
};

unordered_map <string, int> ident;
unordered_map <string, int> _const;
unordered_map <string, int> test;

int findID(string _token) {
    Token token;
    token.value = _token;
    map<string, int>::iterator iter;
    unordered_map<string, int>::iterator iter1;
    if (symbolClassifier(token.value[0]) == letters) {
        if (kwrd.count(token.value) == 1) {
            iter = kwrd.find(token.value);
            token.id = iter->second;
        }
        else if (ident.count(token.value) == 0) {
            ident.insert(make_pair(token.value, ident_count));
            token.id = ident_count;
```

```
            ident_count++;
        }
        else {
            iter1 = ident.find(token.value);
            token.id = iter1->second;
        }
    }
    else if (symbolClassifier(token.value[0]) == digits) {
        if (_const.count(token.value) == 0) {
            _const.insert(pair<string, int>(token.value, const_count));
            token.id = const_count;
            const_count++;
        }
        else {
            iter1 = _const.find(token.value);
            token.id = iter1->second;
        }
    }
    else if (symbolClassifier(token.value[0]) == tests) {
        if (test.count(token.value) == 0) {
            test.insert(pair<string, int>(token.value, test_count));
            token.id = test_count;
            test_count++;
        }
        else {
            iter1 = test.find(token.value);
            token.id = iter1->second;
        }
    }
    else if (sep.count(token.value) == 1) {
        iter = sep.find(token.value);
        token.id = iter->second;
    }

    return token.id;
}

void printTables(FILE* gen) {
    if (ident.size() != 0) {
        fprintf(gen, "\nIdentifier table\n");
        for (auto it : ident) {
            cout << it.first << " " << it.second << endl;
            fprintf(gen, "%s %d\n", it.first.c_str(), it.second);
        }
    }

    if (_const.size() != 0) {
        fprintf(gen, "\nConstant table\n");
        for (auto it : _const) {
            cout << it.first << " " << it.second << endl;
            fprintf(gen, "%s %d\n", it.first.c_str(), it.second);
```

```
        }
    }

}
```

BinTree.h

```
#pragma once
#ifndef BIN_TREE_H
#define BIN_TREE_H

#include "LexerGeneration.h"
#include "CodeGenerationr.h"

struct Nodes {
    Nodes() {};
    Nodes(int _lexem_code, string _lexem_name, Nodes* _parent) {
        lexem_code = _lexem_code;
        lexem_name = _lexem_name;
        parent = _parent;
    }
    int lexem_code;
    string lexem_name;
    Nodes* parent;
    vector<Nodes> childNodes;
};


enum errorCode {
    key_word_not_found,
    delimiter_not_found,
    ident_not_found,
    const_not_found,
    wrong_delimiter,
    wrong_key_word,
    no_equal_rows,
    no_statement,
    label_value_not_found
};

void parsing(FILE* generated);
void createRoot(int _lexem_code, string _lexem_name);
void addChild(int _lexem_code, string _lexem_name);
void gotoChild(string _lexem_name);
void gotoChild(int index);
void setCurrentNode(Nodes* child);
void gotoLastChild();
void gotoBrother(int index);
Nodes* getCurrentNode();
bool gotoParent();
Nodes* getLinkRoot();
```

```
Nodes getRoot();
string getNodeName();

Token getToken();
Token checkKeyToken(Token checkToken, string keyToken);
Token delimiters(Token prev_token, Token current_token, int token_id);

void program(Token token);
Token identifier(Token token);
Token procedureIdentifier(Token prev_token, Token current_token);
Token parametersList(Token prev_token, Token current_token);
Token variableIdentifier(Token prev_token, Token current_token);
Token identifierList(Token prev_token, Token current_token);
Token blok(Token current_token);
Token declaration(Token current_token);
Token labelDeclaration(Token current_token);
Token unsignedInteger(Token prev_token, Token current_token);
Token labelList(Token prev_token, Token current_token);
Token statementList(Token prev_token, Token current_token);
Token statement(Token prev_token, Token current_token);
Token assemblyInsertFileIdentifier(Token prev_token, Token current_token);



void errorOutput(int error_code, Token error_token = Token(), string token = "");
void printTree(FILE* gen, Nodes _tree, int _depth);
void printTree(FILE* gen);
#endif // !BIN_TREE_H
```

BinTree.cpp

```
#include "BinTree.h"
#include "LexerGeneration.h"
#include "LexerGeneration.h"

Nodes root;
Nodes* currentNode = &root;

void createRoot(int _lexem_code, string _lexem_name) {
    root.lexem_code = _lexem_code;
    root.lexem_name = _lexem_name;
    root.parent = NULL;
}

void addChild(int lexem_code, string lexem_name) {
    Nodes tmp(lexem_code, lexem_name, currentNode);
    currentNode->childNodes.push_back(tmp);
}

Nodes* getCurrentNode() {
    return currentNode;
```

```
}

void setCurrentNode(Nodes* newCurrentNode) {
    currentNode = newCurrentNode;
}

void gotoChild(int index) {
    setCurrentNode(&currentNode->childNodes[index]);
}

void gotoChild(string _lexem_name) {
    for (int i = 0; i < (int)currentNode->childNodes.size(); i++) {
        if (currentNode->childNodes[i].lexem_name == _lexem_name) {
            gotoChild(i);
            return;
        }
    }
}

void gotoBrother(int index) {
    gotoParent();
    gotoChild(index);
}

void gotoLastChild() {
    setCurrentNode(&currentNode->childNodes.back());
}

bool gotoParent() {
    if (currentNode == &root) return false;
    currentNode = currentNode->parent;
    return true;
}

void printTree(FILE* gen, Nodes tree, int _depth) {
    if (tree.lexem_code == -1) {
        cout << tree.lexem_name << endl;
        fprintf(gen, "%s\n", tree.lexem_name.c_str());
    }
    else {
        cout << tree.lexem_code << "  " << tree.lexem_name << endl;
        fprintf(gen, "%d %s\n", tree.lexem_code, tree.lexem_name.c_str());
    }

    if (!tree.childNodes.empty()) {
        for (int i = 0; i < (int)tree.childNodes.size(); i++) {
            for (int i = 0; i <= _depth; i++) {
                cout << "..";
                fprintf(gen, "..");
            }
            printTree(gen, tree.childNodes[i], _depth + 1);
```

```
        }
    }
}

void printTree(FILE* gen) {
    cout << endl << "Parse tree" << endl;
    fprintf(gen, "\nParse tree\n");
    printTree(gen, root, 0);
}

string getNodeName() {
    return currentNode->lexem_name;
}

Nodes* getLinkRoot() {
    return &root;
}

Nodes getRoot() {
    return root;
}
```

SyntaxAnalyzer.cpp

```
#include "LexerGeneration.h"
#include "BinTree.h"
#include "CodeGenerationr.h"

vector<Token> vector_lexem;
FILE* gen;

Token getToken() {
    Token tmp = *vector_lexem.begin();
    vector_lexem.erase(vector_lexem.begin());
    return tmp;
}

Token checkKeyToken(Token checkToken, string keyToken) {
    if (checkToken.id == findID(keyToken)) {
        addChild(checkToken.id, checkToken.value);
    }
    else {
        errorOutput(key_word_not_found, checkToken, keyToken);
    }
    if (!vector_lexem.empty()) {
        checkToken = getToken();
    }

    return checkToken;
}
```

```
void errorOutput(int error_code, Token error_token, string token) {
    printTree(gen);
    switch (error_code) {
    case key_word_not_found:
        printf("Parser : Error. Key word \'%s\'[%d, %d] not found.\n",
token.c_str(), error_token.row, error_token.column);
        fprintf(gen, "Parser : Error. Key word \'%s\'[%d, %d] not found.\n",
token.c_str(), error_token.row, error_token.column);
        break;
    case delimiter_not_found:
        printf("Parser : Error. Delimiter \'%s\'[%d, %d] not found.\n",
token.c_str(), error_token.row, error_token.column + error_token.value.size());
        fprintf(gen, "Parser : Error. Delimiter \'%s\'[%d, %d] not found.\n",
token.c_str(), error_token.row, error_token.column + error_token.value.size());
        break;
    case ident_not_found:
        printf("Parser : Error [%d, %d]. Identifier not found.\n",
error_token.row, error_token.column);
        fprintf(gen, "Parser : Error [%d, %d]. Identifier not found.\n",
error_token.row, error_token.column);
        break;
    case const_not_found:
        printf("Parser : Error [%d, %d]. Unsigned integer not found.\n",
error_token.row, error_token.column);
        fprintf(gen, "Parser : Error [%d, %d]. Unsigned integer not found.\n",
error_token.row, error_token.column);
        break;
    case wrong_delimiter:
        printf("Parser : Error [%d, %d]. Wrong delimiter.\n", error_token.row,
error_token.column);
        fprintf(gen, "Parser : Error [%d, %d]. Wrong delimiter.\n",
error_token.row, error_token.column);
        break;
    case wrong_key_word:
        printf("Parser : Error [%d, %d]. Wrong key word.\n", error_token.row,
error_token.column);
        fprintf(gen, "Parser : Error [%d, %d]. Wrong key word.\n",
error_token.row, error_token.column);
        break;
    case no_equal_rows:
        printf("Parser : Error [%d, %d]. Tokens must be on the same line.\n",
error_token.row, error_token.column);
        fprintf(gen, "Parser : Error [%d, %d]. Tokens must be on the same
line.\n", error_token.row, error_token.column);
        break;
    case no_statement:
        printf("Parser : Error [%d, %d]. After the mark should be statement.\n",
error_token.row, error_token.column);
        fprintf(gen, "Parser : Error [%d, %d]. After the mark should be
statement.\n", error_token.row, error_token.column);
        break;
```

```
    case label_value_not_found:
        printf("Parser : Error [%d, %d]. After the mark should be label.\n",
error_token.row, error_token.column + error_token.value.size());
        fprintf(gen, "Parser : Error [%d, %d]. After the mark should be
label.\n", error_token.row, error_token.column + error_token.value.size());
        break;
    }
    exit(error_code);
}

void parsing(FILE* generated) {
    gen = generated;
    vector_lexem = getVectorToken();
    if (vector_lexem.size() == 0) {
        fprintf(generated, " File is empty");
    }
    createRoot(-1, "<signal-program>");
    program(getToken());
    printTree(gen);


}

void program(Token token) {
    addChild(-1, "<program>");
    gotoLastChild();

    Token checkKeyWord = checkKeyToken(token, "PROCEDURE");
    Nodes* currentNode = getCurrentNode();
    Token next_token = procedureIdentifier(token, checkKeyWord);
    setCurrentNode(currentNode);

    next_token = parametersList(checkKeyWord, next_token);
    setCurrentNode(currentNode);
    next_token = delimiters(checkKeyWord, next_token, 59);

    next_token = blok(next_token);

}

Token procedureIdentifier(Token prev_token, Token current_token) {
    if (prev_token.row == current_token.row) {
        addChild(-1, "<procedure-identifier>");
        gotoLastChild();

        return identifier(current_token);
    }
    else {
        errorOutput(no_equal_rows, current_token);
    }
}
```

```
Token identifier(Token token) {
    if (token.id > 1000) {
        addChild(-1, "<identifier>");
        gotoChild("<identifier>");
        addChild(token.id, token.value);
    }
    else {
        errorOutput(ident_not_found, token);
    }
    return getToken();
}

Token delimiters(Token prev_token, Token current_token, int token_id) {
    if (current_token.id > 0 && current_token.id < 255) {
        if (current_token.id == token_id) {
            if (prev_token.row == current_token.row) {
                addChild(current_token.id, current_token.value);
            }
            else {
                errorOutput(no_equal_rows, current_token);
            }
        }
        else {
            errorOutput(wrong_delimiter, current_token);
        }
    }
    else {
        char buff[2];
        buff[0] = (char)token_id;
        buff[1] = '\0';
        string token = string(buff);
        errorOutput(delimiter_not_found, prev_token, token);
    }
    if (!vector_lexem.empty()) {
        current_token = getToken();
    }
    return current_token;
}

Token variableIdentifier(Token prev_token, Token current_token) {
    if (prev_token.row == current_token.row) {
        addChild(-1, "<variable-identifier>");
        gotoLastChild();

        return identifier(current_token);
    }
    else {
        errorOutput(no_equal_rows, current_token);
    }
}
```

```
Token identifierList(Token prev_token, Token current_token) {
    bool isIdentifierList = false;
    addChild(-1, "<identifier-list>");
    gotoLastChild();
    Token next_token;
    Nodes* currentNode = getCurrentNode();
    if (current_token.id != 41) {
        isIdentifierList = true;
        next_token = delimiters(prev_token, current_token, 44);
        current_token = next_token;
        if (next_token.id > 1000) {
            next_token = variableIdentifier(current_token, next_token);
            setCurrentNode(currentNode);
            next_token = identifierList(current_token, next_token);
            return next_token;
        }
        else {
            errorOutput(ident_not_found, next_token);
        }
    }


    if (!isIdentifierList) {
        addChild(-1, "<empty>");
    }

    return current_token;
}

Token parametersList(Token prev_token, Token current_token) {
    bool isParameterList = false;
    addChild(-1, "<parameters-list>");
    gotoLastChild();
    Token next_token;
    Nodes* currentNode = getCurrentNode();
    if (current_token.id != 59) {
        isParameterList = true;
        next_token = delimiters(prev_token, current_token, 40);
        prev_token = next_token;
        next_token = variableIdentifier(current_token, next_token);
        setCurrentNode(currentNode);
        while (next_token.id != 41) {
            next_token = delimiters(prev_token, next_token, 44);
            current_token = next_token;
            next_token = variableIdentifier(current_token, next_token);
            setCurrentNode(currentNode);
        }
        /*current_token = identifierList(prev_token, next_token);
        setCurrentNode(currentNode);*/
        next_token = delimiters(prev_token, next_token, 41);
```

```
        current_token = next_token;
    }

    if (!isParameterList) {
        addChild(-1, "<empty>");
    }

    return current_token;
}


Token blok(Token current_token) {
    addChild(-1, "<block>");
    gotoLastChild();
    Token next_token;
    Nodes* currentNode = getCurrentNode();
    current_token = declaration(current_token);
    setCurrentNode(currentNode);
    next_token = checkKeyToken(current_token, "BEGIN");
    setCurrentNode(currentNode);
    current_token = statementList(current_token, next_token);
    setCurrentNode(currentNode);
    next_token = checkKeyToken(current_token, "END");
    gotoParent();
    next_token = delimiters(current_token, next_token, 59);
    return next_token;
}


Token declaration(Token current_token) {
    addChild(-1, "<declaration>");
    gotoLastChild();
    current_token = labelDeclaration(current_token);
    return current_token;
}

Token labelDeclaration(Token current_token) {
    bool isLabelDeclaration = false;
    addChild(-1, "<label-declaration>");
    gotoLastChild();

    Nodes* currentNode = getCurrentNode();

    if (current_token.id != 402) {
        Token prev_token = current_token;
        isLabelDeclaration = true;
        Token next_token = checkKeyToken(current_token, "LABEL");
        Nodes* currentNode = getCurrentNode();
        current_token = unsignedInteger(current_token, next_token);
```

```
        setCurrentNode(currentNode);
        while (current_token.id != 59) {
            next_token = delimiters(prev_token, current_token, 44);
            current_token = unsignedInteger(current_token, next_token);
            setCurrentNode(currentNode);
        }
        /*next_token = labelList(next_token, current_token);
        setCurrentNode(currentNode);*/
        next_token = delimiters(next_token, current_token, 59);
        current_token = next_token;
    }


    if (!isLabelDeclaration) {
        addChild(-1, "<empty>");
    }

    return current_token;
}

Token unsignedInteger(Token prev_token, Token current_token) {
    if (prev_token.row == current_token.row) {
        if (current_token.id > 500 && current_token.id <= 1000) {
            addChild(-1, "<unsigned-integer>");
            gotoLastChild();
            addChild(current_token.id, current_token.value);
        }
        else {
            errorOutput(const_not_found, current_token);
        }
    }
    else {
        errorOutput(no_equal_rows, current_token);
    }
    return getToken();
}

Token labelList(Token prev_token, Token current_token) {
    bool isLabelList = false;
    addChild(-1, "<label-list>");
    gotoLastChild();
    Token next_token;
    Nodes* currentNode = getCurrentNode();

    if (current_token.id != 59) {

        isLabelList = true;
        next_token = delimiters(prev_token, current_token, 44);
        prev_token = next_token;
        if (next_token.id > 500 && next_token.id < 1000) {
            next_token = unsignedInteger(current_token, next_token);
```

```
        }
        else {
            errorOutput(label_value_not_found, current_token);
        }
        setCurrentNode(currentNode);
        if (prev_token.row != next_token.row) {
            errorOutput(delimiter_not_found, prev_token, ";");
        }
        next_token = labelList(prev_token, next_token);
        return next_token;
    }


    if (!isLabelList) {
        addChild(-1, "<empty>");
    }

    return current_token;
}

Token statement(Token prev_token, Token current_token) {
    addChild(-1, "<statement>");
    gotoLastChild();
    Nodes* currentNode = getCurrentNode();
    Token next_token;
    if (current_token.id > 500 && current_token.id <= 1000) {
        next_token = unsignedInteger(current_token, current_token);
        setCurrentNode(currentNode);
        current_token = delimiters(current_token, next_token, 58);
        //current_token = statement(current_token, current_token);
        setCurrentNode(currentNode);
    }
    else if (current_token.id > 400 && current_token.id <= 500) {
        if (current_token.id == 405) {
            prev_token = current_token;
            current_token = checkKeyToken(current_token, "GOTO");
            current_token = unsignedInteger(prev_token, current_token);
            setCurrentNode(currentNode);
            current_token = delimiters(prev_token, current_token, 59);
        }
        else if (current_token.id == 406) {
            prev_token = current_token;
            current_token = checkKeyToken(current_token, "RETURN");
            current_token = delimiters(prev_token, current_token, 59);
        }
        else {
            errorOutput(wrong_key_word, current_token);
        }
    }
    else if (current_token.id > 0 && current_token.id < 255) {
```

```
        if (current_token.id == 59) {
            current_token = delimiters(prev_token, current_token, 59);
        }
        else if (current_token.id == 40) {
            next_token = delimiters(prev_token, current_token, 40);
            current_token = delimiters(prev_token, next_token, 36);
            next_token = assemblyInsertFileIdentifier(next_token, current_token);
            setCurrentNode(currentNode);
            current_token = delimiters(current_token, next_token, 36);
            next_token = delimiters(next_token, current_token, 41);
            return next_token;
        }
        else {
            errorOutput(wrong_delimiter, current_token);
        }
    }
    else {
        errorOutput(no_statement, current_token);
    }

    return current_token;
}

Token statementList(Token prev_token, Token current_token) {
    bool isStatementList = false;
    addChild(-1, "<statement-list>");
    gotoLastChild();
    Nodes* currentNode = getCurrentNode();
    Token next_token;
    while (current_token.id != 403) {
        isStatementList = true;
        current_token = statement(current_token, current_token);
        setCurrentNode(currentNode);
        //current_token = statementList(current_token, current_token);
    }


    if (!isStatementList) {
        addChild(-1, "<empty>");
    }

    return current_token;
}

Token assemblyInsertFileIdentifier(Token prev_token, Token current_token) {
    addChild(-1, "<assembly-insert-file-identifier>");
    gotoLastChild();
    if (prev_token.row == current_token.row) {
        current_token = identifier(current_token);
    }
    else {
```

```
        errorOutput(no_equal_rows, current_token);
    }


    return current_token;
}
```

CodeGeneration.h

```
#pragma once
#ifndef CODE_GENERATION_H
#define CODE_GENERATION_H

#include "BinTree.h"
#include "LexerGeneration.h"



struct label {
    label(string value, bool wasAdd, bool wasCall) {
        this->value = value;
        this->wasAdd = wasAdd;
        this->wasCall = wasCall;
    }
    string value;
    bool wasAdd;
    bool wasCall;
};

struct param {
    param(string value) {
        this->value = value;
    }
    string value;
};

enum codeGenErrorCode {
    double_labels,
    double_params,
    similar_prog_name,
    not_init_label,
    asm_file_not_found,
    used_but_not_added_label
};


bool codeGeneraion(FILE* generated);

#endif // !CODE_GENERATION_H
```

CodeGeneration.cpp

```
#include "BinTree.h"
#include "LexerGeneration.h"
#include "CodeGenerationr.h"

FILE* gener;
list<string> asm_code;
list<string> asm_data;
list<label*> labels;
list<param*> params;
int param_asm = 20;

void assemFile(Nodes* current_Node);

void printCode() {
    for (string s: asm_code) {
        cout << s << endl;
        fprintf(gener, "%s \n", s.c_str());
    }
}

label* get_label(string label_name) {
    for (label* l: labels) {
        if (l->value == label_name)
            return l;
    }

    return nullptr;
}

void errorCodeOutput(int error_code, string lexem_name = "") {
    printCode();
    switch (error_code) {
    case double_labels :
        cout << "\nCode Generator : Error: different labels cannot have the same
name" << endl;
        fprintf(gener, "\nCode Generator : Error: different labels cannot have
the same name\n");
        break;
    case double_params:
        cout << "\nCode Generator : Error: different parameters cannot have the
same name" << endl;
        fprintf(gener, "\nCode Generator : Error: different parameters cannot
have the same name\n");
        break;
    case similar_prog_name:
        cout << "\nCode Generator : Error :The identifier \"" << lexem_name <<
"\" cannot be similar to the procedure name" << endl;
        fprintf(gener, "\nCode Generator : Error: The identifier \"%s\" cannot be
similar to the procedure name\n", lexem_name.c_str());
        break;
```

```
    case not_init_label:
        cout << "\nCode Generator : Error: Use undeclarated label \"" +
lexem_name + "\"";
        fprintf(gener, "\nCode Generator : Error: Use undeclarated label \"%s\"
\n", lexem_name.c_str());
        break;
    case asm_file_not_found:
        cout << "\nCode Generator : Error: Insert assembler file \"" + lexem_name
+ "\" not found \n";
        fprintf(gener, "\nCode Generator : Error: Insert assembler file \"%s\"
not found \n", lexem_name.c_str());
        break;
    case used_but_not_added_label:
        cout << "\nCode Generator : Error: label \"" + lexem_name + "\" was
called but not added\n";
        fprintf(gener, "\nCode Generator : Error: Error: label \"%s\" was called
but not added\n", lexem_name.c_str());

    }


    exit(error_code);
}


void paramCheck(Nodes* currentNode, string proc_name) {
    for (int i = 0; i < (int)currentNode->childNodes.size(); i++) {
        if (currentNode->childNodes[i].lexem_name == "<variable-identifier>") {
            if (currentNode->childNodes[i].childNodes[0].childNodes[0].lexem_name
!= proc_name) {
                params.push_back(new param(currentNode-
>childNodes[i].childNodes[0].childNodes[0].lexem_name));
                string asm_param = "\tMOV DWORD PTR[rqb-" + to_string(param_asm)
+ "], " + currentNode->childNodes[i].childNodes[0].childNodes[0].lexem_name +
"\n";
                param_asm += 4;
                printf("%s", asm_param.c_str());
            }
            else
                errorCodeOutput(similar_prog_name, currentNode-
>childNodes[i].childNodes[0].childNodes[0].lexem_name);
        }
    }
    for (param* p1 : params) {
        for (param* p2 : params) {
            if (p1 != p2 && p1->value == p2->value)
                errorCodeOutput(double_params, currentNode-
>childNodes[0].lexem_name);
        }
    }
}
```

```
void labelCheck(Nodes* currentNode) {
    for (int i = 0; i < (int)currentNode->childNodes.size(); i++) {
        if (currentNode->childNodes[i].lexem_name == "<unsigned-integer>") {
            labels.push_back(new label(currentNode-
>childNodes[i].childNodes[0].lexem_name, false, false));
        }
    }
    for (label* l1 : labels) {
        for (label* l2 : labels) {
            if (l1 != l2 && l1->value == l2->value) {
                errorCodeOutput(double_labels, currentNode-
>childNodes[0].lexem_name);
            }
        }
    }
}

void checkCorrectLabels() {
    for (label* l : labels) {
        if (l->wasAdd == false && l->wasCall == true) {
            errorCodeOutput(used_but_not_added_label, l->value);
        }
    }
}

void statementsCheck(Nodes* currentNode) {
    if (currentNode->childNodes[0].lexem_name == "<empty>") {
        asm_code.push_back("\tNOP");
        return;
    }
    for (int i = 0; i < (int)currentNode->childNodes.size(); i++) {
        setCurrentNode(currentNode);
        if (currentNode->childNodes[i].childNodes[0].lexem_name == "<unsigned-
integer>") {
            label* l1 = get_label(currentNode-
>childNodes[i].childNodes[0].childNodes[0].lexem_name);
            if (l1 == nullptr) {
                errorCodeOutput(not_init_label, currentNode-
>childNodes[i].childNodes[0].childNodes[0].lexem_name);
            }
            l1->wasAdd = true;
            asm_code.push_back(l1->value + ":\n");
        }
        if (currentNode->childNodes[i].childNodes[0].lexem_name == "GOTO"){
            label* l1 = get_label(currentNode-
>childNodes[i].childNodes[1].childNodes[0].lexem_name);
            if (l1 == nullptr) {
                errorCodeOutput(not_init_label, currentNode-
>childNodes[i].childNodes[1].childNodes[0].lexem_name);
```

```
            }
            l1->wasCall = true;
            asm_code.push_back("\tJMP " + l1->value + "\n");
        }
        if (currentNode->childNodes[i].childNodes[0].lexem_name == "RETURN") {
            asm_code.push_back("\tMOV ax, 4C00h\
                               \n\tINT 21h\n");
        }
        if (currentNode->childNodes[i].childNodes[0].lexem_code == 40) {
            gotoChild(i);
            gotoChild(2);
            assemFile(getCurrentNode());
        }
    }
}


void assemFile(Nodes* currentNode) {
    string asmFileName = currentNode->childNodes[0].childNodes[0].lexem_name;
    ifstream asmFile(asmFileName);

    if (!asmFile) {
        errorCodeOutput(asm_file_not_found, currentNode-
>childNodes[0].childNodes[0].lexem_name);
    }
    stringstream buff;
    buff << asmFile.rdbuf();
    asm_code.push_back(buff.str());
}

void generation(Nodes* tree) {
    gotoChild(0);
    tree = getCurrentNode();
    gotoChild(1); //procedure name
    gotoChild(0);
    gotoChild(0);
    string program_name = getNodeName();
    asm_data.push_back("\ndata  SEGMENT\
                       \ndata  ENDS\n\
                       \ncode  SEGMENT\
                       \n\tASSUME   cs:code, ds:data\n" + program_name + ":\n");
    for (string s : asm_data) {
        cout << s;
        fprintf(gener, "%s", s.c_str());
    }


    setCurrentNode(tree);
    gotoChild(2); //parameter list
    paramCheck(getCurrentNode(), program_name);
```

```
    setCurrentNode(tree);
    gotoChild(4); //blok
    tree = getCurrentNode();
    gotoChild(0);
    gotoChild(0); //label declaration
    labelCheck(getCurrentNode());
    setCurrentNode(tree);
    gotoChild(2); //statements list
    statementsCheck(getCurrentNode());
    checkCorrectLabels();
    printCode();
    asm_data.clear();
    asm_data.push_back("\n\ncode    ENDS\
                \n\tend " + program_name + "\n");
    for (string s : asm_data) {
        cout << s;
        fprintf(gener, "%s", s.c_str());
    }

}


bool codeGeneraion(FILE* generated) {
    gener = generated;
    Nodes* rootNode = getLinkRoot();
    setCurrentNode(rootNode);
    generation(rootNode);

    return 0;
}
```

Результати тестування

Test01:

```
PROCEDURE proc(id1, id2, id3);
LABEL 15, 7, 5, 8;
BEGIN
15: GOTO 7;
7 : RETURN;
8 : ($ asmFile $);
END;
```

```
 Line | Column | Ident token | Token
------------------------------------------------
    1 |      1 |         401 | PROCEDURE
    1 |     11 |        1001 | proc
    1 |     15 |          40 | (
    1 |     16 |        1002 | id1
```

```
    1 |     19 |          44 | ,
    1 |     21 |        1003 | id2
    1 |     24 |          44 | ,
    1 |     26 |        1004 | id3
    1 |     29 |          41 | )
    1 |     30 |          59 | ;
    2 |      1 |         404 | LABEL
    2 |      7 |         501 | 15
    2 |      9 |          44 | ,
    2 |     11 |         502 | 7
    2 |     12 |          44 | ,
    2 |     14 |         503 | 5
    2 |     15 |          44 | ,
    2 |     17 |         504 | 8
    2 |     18 |          59 | ;
    3 |      1 |         402 | BEGIN
    4 |      1 |         501 | 15
    4 |      3 |          58 | :
    4 |      5 |         405 | GOTO
    4 |     10 |         502 | 7
    4 |     11 |          59 | ;
    5 |      1 |         502 | 7
    5 |      3 |          58 | :
    5 |      5 |         406 | RETURN
    5 |     11 |          59 | ;
    6 |      1 |         504 | 8
    6 |      3 |          58 | :
    6 |      5 |          40 | (
    6 |      6 |          36 | $
    6 |      8 |        1005 | asmFile
    6 |     16 |          36 | $
    6 |     17 |          41 | )
    6 |     18 |          59 | ;
    7 |      1 |         403 | END
    7 |      4 |          59 | ;

Identifier table
id3 1004
proc 1001
id1 1002
asmFile 1005
id2 1003

Constant table
15 501
7 502
5 503
8 504

Parse tree
<signal-program>
..<program>
....401 PROCEDURE
```

```
....<procedure-identifier>
......<identifier>
........1001 proc
....<parameters-list>
......40 (
......<variable-identifier>
........<identifier>
..........1002 id1
......44 ,
......<variable-identifier>
........<identifier>
..........1003 id2
......44 ,
......<variable-identifier>
........<identifier>
..........1004 id3
......41 )
....59 ;
....<block>
......<declaration>
........<label-declaration>
..........404 LABEL
..........<unsigned-integer>
............501 15
..........44 ,
..........<unsigned-integer>
............502 7
..........44 ,
..........<unsigned-integer>
............503 5
..........44 ,
..........<unsigned-integer>
............504 8
..........59 ;
......402 BEGIN
......<statement-list>
........<statement>
..........<unsigned-integer>
............501 15
..........58 :
........<statement>
..........405 GOTO
..........<unsigned-integer>
............502 7
..........59 ;
........<statement>
..........<unsigned-integer>
............502 7
..........58 :
........<statement>
..........406 RETURN
..........59 ;
........<statement>
```

```
..........<unsigned-integer>
............504 8
..........58 :
........<statement>
..........40 (
..........36 $
..........<assembly-insert-file-identifier>
............<identifier>
..............1005 asmFile
..........36 $
..........41 )
........<statement>
..........59 ;
......403 END
....59 ;
```

```
data SEGMENT
data ENDS

code SEGMENT
     ASSUME   cs:code, ds:data
proc:
15:
     JMP 7
7:
     MOV ax, 4C00h
     INT 21h
8:
     JMP 15
     MOV ax, esi
     MOV DWORD PTR[rpd-20], ax

code ENDS
     end  proc
```

Test02:

```
PROCEDURE proc;
BEGIN
END;
```

Line	Column	Ident token	Token
1	1	401	PROCEDURE
1	11	1001	proc
1	15	59	;
2	1	402	BEGIN
3	1	403	END
3	4	59	;

```
Identifier table
proc 1001
```

```
Parse tree
<signal-program>
..<program>
....401 PROCEDURE
....<procedure-identifier>
......<identifier>
........1001 proc
....<parameters-list>
......<empty>
....59 ;
....<block>
......<declaration>
........<label-declaration>
..........<empty>
......402 BEGIN
......<statement-list>
........<empty>
......403 END
....59 ;

data SEGMENT
data ENDS

code SEGMENT
     ASSUME   cs:code, ds:data
proc:
     NOP

code ENDS
     end  proc
```

Test03:

```
PROCEDURE proc(id1, id2, id3);
LABEL 15, 17, 18;
BEGIN
GOTO 15;
18 : ($ asmFile $)

END;
```

Line	Column	Ident token	Token
1	1	401	PROCEDURE
1	11	1001	proc
1	15	40	(
1	16	1002	id1
1	19	44	,
1	21	1003	id2
1	24	44	,
1	26	1004	id3
1	29	41	)

```
    1 |     30 |          59 | ;
    2 |      1 |         404 | LABEL
    2 |      7 |         501 | 15
    2 |      9 |          44 | ,
    2 |     11 |         502 | 17
    2 |     13 |          44 | ,
    2 |     15 |         503 | 18
    2 |     17 |          59 | ;
    3 |      1 |         402 | BEGIN
    4 |      1 |         405 | GOTO
    4 |      6 |         501 | 15
    4 |      8 |          59 | ;
    5 |      1 |         503 | 18
    5 |      4 |          58 | :
    5 |      6 |          40 | (
    5 |      7 |          36 | $
    5 |      9 |        1005 | asmFile
    5 |     17 |          36 | $
    5 |     18 |          41 | )
    7 |      1 |         403 | END
    7 |      4 |          59 | ;

Identifier table
id3 1004
proc 1001
id1 1002
asmFile 1005
id2 1003

Constant table
15 501
17 502
18 503

Parse tree
<signal-program>
..<program>
....401 PROCEDURE
....<procedure-identifier>
......<identifier>
........1001 proc
....<parameters-list>
......40 (
......<variable-identifier>
........<identifier>
..........1002 id1
......44 ,
......<variable-identifier>
........<identifier>
..........1003 id2
......44 ,
......<variable-identifier>
........<identifier>
```

```
..........1004 id3
......41 )
....59 ;
....<block>
......<declaration>
........<label-declaration>
..........404 LABEL
..........<unsigned-integer>
............501 15
..........44 ,
..........<unsigned-integer>
............502 17
..........44 ,
..........<unsigned-integer>
............503 18
..........59 ;
......402 BEGIN
......<statement-list>
........<statement>
..........405 GOTO
..........<unsigned-integer>
............501 15
..........59 ;
........<statement>
..........<unsigned-integer>
............503 18
..........58 :
........<statement>
..........40 (
..........36 $
..........<assembly-insert-file-identifier>
............<identifier>
..............1005 asmFile
..........36 $
..........41 )
......403 END
....59 ;

data SEGMENT
data ENDS

code SEGMENT
     ASSUME   cs:code, ds:data
proc:
     JMP 15

18:

     JMP 15
     MOV ax, esi
     MOV DWORD PTR[rpd-20], ax
```

```
Code Generator : Error: Error: label "15" was called but not
added
```

Test04:

```
PROCEDURE proc;
LABEL 1, 2, 4;
BEGIN
1:
GOTO 3;
END;
```

```
 Line | Column | Ident token | Token
-------------------------------------------------
    1 |      1 |         401 | PROCEDURE
    1 |     11 |        1001 | proc
    1 |     15 |          59 | ;
    2 |      1 |         404 | LABEL
    2 |      7 |         501 | 1
    2 |      8 |          44 | ,
    2 |     10 |         502 | 2
    2 |     11 |          44 | ,
    2 |     13 |         503 | 4
    2 |     14 |          59 | ;
    3 |      1 |         402 | BEGIN
    4 |      1 |         501 | 1
    4 |      2 |          58 | :
    5 |      1 |         405 | GOTO
    5 |      6 |         504 | 3
    5 |      7 |          59 | ;
    6 |      1 |         403 | END
    6 |      4 |          59 | ;

Identifier table
proc 1001

Constant table
1 501
2 502
4 503
3 504

Parse tree
<signal-program>
..<program>
....401 PROCEDURE
....<procedure-identifier>
......<identifier>
........1001 proc
....<parameters-list>
......<empty>
....59 ;
....<block>
```

```
......<declaration>
........<label-declaration>
..........404 LABEL
..........<unsigned-integer>
............501 1
..........44 ,
..........<unsigned-integer>
............502 2
..........44 ,
..........<unsigned-integer>
............503 4
..........59 ;
......402 BEGIN
......<statement-list>
........<statement>
..........<unsigned-integer>
............501 1
..........58 :
........<statement>
..........405 GOTO
..........<unsigned-integer>
............504 3
..........59 ;
......403 END
....59 ;

data SEGMENT
data ENDS

code SEGMENT
     ASSUME   cs:code, ds:data
proc:
1:


Code Generator : Error: Use undeclarated label "3"
```

Test05:

```
PROCEDURE proc(id1, id2, id3);
LABEL 1, 2, 3;
BEGIN
3: ($ asmFile $)
RETURN;
END;

Line | Column | Ident token | Token
-------------------------------------------------
    1 |      1 |         401 | PROCEDURE
    1 |     11 |        1001 | proc
    1 |     15 |          40 | (
    1 |     16 |        1002 | id1
    1 |     19 |          44 | ,
```

```
    1 |     21 |        1003 | id2
    1 |     24 |          44 | ,
    1 |     26 |        1004 | id3
    1 |     29 |          41 | )
    1 |     30 |          59 | ;
    2 |      1 |         404 | LABEL
    2 |      7 |         501 | 1
    2 |      8 |          44 | ,
    2 |     10 |         502 | 2
    2 |     11 |          44 | ,
    2 |     13 |         503 | 3
    2 |     14 |          59 | ;
    3 |      1 |         402 | BEGIN
    4 |      1 |         503 | 3
    4 |      2 |          58 | :
    4 |      4 |          40 | (
    4 |      5 |          36 | $
    4 |      7 |        1005 | asmFile
    4 |     15 |          36 | $
    4 |     16 |          41 | )
    5 |      1 |         406 | RETURN
    5 |      7 |          59 | ;
    6 |      1 |         403 | END
    6 |      4 |          59 | ;

Identifier table
id3 1004
proc 1001
id1 1002
asmFile 1005
id2 1003

Constant table
1 501
2 502
3 503

Parse tree
<signal-program>
..<program>
....401 PROCEDURE
....<procedure-identifier>
......<identifier>
........1001 proc
....<parameters-list>
......40 (
......<variable-identifier>
........<identifier>
..........1002 id1
......44 ,
......<variable-identifier>
........<identifier>
..........1003 id2
```

```
......44 ,
......<variable-identifier>
........<identifier>
..........1004 id3
......41 )
....59 ;
....<block>
......<declaration>
........<label-declaration>
..........404 LABEL
..........<unsigned-integer>
............501 1
..........44 ,
..........<unsigned-integer>
............502 2
..........44 ,
..........<unsigned-integer>
............503 3
..........59 ;
......402 BEGIN
......<statement-list>
........<statement>
..........<unsigned-integer>
............503 3
..........58 :
........<statement>
..........40 (
..........36 $
..........<assembly-insert-file-identifier>
............<identifier>
..............1005 asmFile
..........36 $
..........41 )
........<statement>
..........406 RETURN
..........59 ;
......403 END
....59 ;

data SEGMENT
data ENDS

code SEGMENT
     ASSUME   cs:code, ds:data
proc:
3:


Code Generator : Error: Insert assembler file "asmFile" not
found
```

Test06:

```
PROCEDURE proc(id1, id2, id3);
LABEL 1, 1, 3;
BEGIN
3: ($ asmFile $)
RETURN;
END;
```

```
Line | Column | Ident token | Token
-----------------------------------------------
    1 |      1 |         401 | PROCEDURE
    1 |     11 |        1001 | proc
    1 |     15 |          40 | (
    1 |     16 |        1002 | id1
    1 |     19 |          44 | ,
    1 |     21 |        1003 | id2
    1 |     24 |          44 | ,
    1 |     26 |        1004 | id3
    1 |     29 |          41 | )
    1 |     30 |          59 | ;
    2 |      1 |         404 | LABEL
    2 |      7 |         501 | 1
    2 |      8 |          44 | ,
    2 |     10 |         501 | 1
    2 |     11 |          44 | ,
    2 |     13 |         502 | 3
    2 |     14 |          59 | ;
    3 |      1 |         402 | BEGIN
    4 |      1 |         502 | 3
    4 |      2 |          58 | :
    4 |      4 |          40 | (
    4 |      5 |          36 | $
    4 |      7 |        1005 | asmFile
    4 |     15 |          36 | $
    4 |     16 |          41 | )
    5 |      1 |         406 | RETURN
    5 |      7 |          59 | ;
    6 |      1 |         403 | END
    6 |      4 |          59 | ;
```

```
Identifier table
id3 1004
proc 1001
id1 1002
asmFile 1005
id2 1003

Constant table
1 501
3 502

Parse tree
```

```
<signal-program>
..<program>
....401 PROCEDURE
....<procedure-identifier>
......<identifier>
........1001 proc
....<parameters-list>
......40 (
......<variable-identifier>
........<identifier>
..........1002 id1
......44 ,
......<variable-identifier>
........<identifier>
..........1003 id2
......44 ,
......<variable-identifier>
........<identifier>
..........1004 id3
......41 )
....59 ;
....<block>
......<declaration>
........<label-declaration>
..........404 LABEL
..........<unsigned-integer>
............501 1
..........44 ,
..........<unsigned-integer>
............501 1
..........44 ,
..........<unsigned-integer>
............502 3
..........59 ;
......402 BEGIN
......<statement-list>
........<statement>
..........<unsigned-integer>
............502 3
..........58 :
........<statement>
..........40 (
..........36 $
..........<assembly-insert-file-identifier>
............<identifier>
..............1005 asmFile
..........36 $
..........41 )
........<statement>
..........406 RETURN
..........59 ;
......403 END
....59 ;
```

```
data SEGMENT
data ENDS

code SEGMENT
     ASSUME   cs:code, ds:data
proc:

Code Generator : Error: different labels cannot have the same
name
```

Test07:

```
PROCEDURE proc (proc, id2, id3);
LABEL 15, 16, 17;
BEGIN
15 : ($ asmFile $);
16 :
GOTO 16;
RETURN;
END;
```

Line	Column	Ident token	Token
1	1	401	PROCEDURE
1	11	1001	proc
1	16	40	(
1	17	1001	proc
1	21	44	,
1	23	1002	id2
1	26	44	,
1	28	1003	id3
1	31	41	)
1	32	59	;
2	1	404	LABEL
2	7	501	15
2	9	44	,
2	11	502	16
2	13	44	,
2	15	503	17
2	17	59	;
3	1	402	BEGIN
4	1	501	15
4	4	58	:
4	6	40	(
4	7	36	$
4	9	1004	asmFile
4	17	36	$
4	18	41	)
4	19	59	;
5	1	502	16
5	4	58	:

```
    6 |      1 |         405 | GOTO
    6 |      6 |         502 | 16
    6 |      8 |          59 | ;
    7 |      1 |         406 | RETURN
    7 |      7 |          59 | ;
    8 |      1 |         403 | END
    8 |      4 |          59 | ;

Identifier table
id3 1003
proc 1001
asmFile 1004
id2 1002

Constant table
15 501
16 502
17 503

Parse tree
<signal-program>
..<program>
....401 PROCEDURE
....<procedure-identifier>
......<identifier>
........1001 proc
....<parameters-list>
......40 (
......<variable-identifier>
........<identifier>
..........1001 proc
......44 ,
......<variable-identifier>
........<identifier>
..........1002 id2
......44 ,
......<variable-identifier>
........<identifier>
..........1003 id3
......41 )
....59 ;
....<block>
......<declaration>
........<label-declaration>
..........404 LABEL
..........<unsigned-integer>
............501 15
..........44 ,
..........<unsigned-integer>
............502 16
..........44 ,
..........<unsigned-integer>
............503 17
```

```
..........59 ;
......402 BEGIN
......<statement-list>
........<statement>
..........<unsigned-integer>
............501 15
..........58 :
........<statement>
..........40 (
..........36 $
..........<assembly-insert-file-identifier>
............<identifier>
..............1004 asmFile
..........36 $
..........41 )
........<statement>
..........59 ;
........<statement>
..........<unsigned-integer>
............502 16
..........58 :
........<statement>
..........405 GOTO
..........<unsigned-integer>
............502 16
..........59 ;
........<statement>
..........406 RETURN
..........59 ;
......403 END
....59 ;

data SEGMENT
data ENDS

code SEGMENT
     ASSUME   cs:code, ds:data
proc:

Code Generator : Error: The identifier "proc" cannot be similar
to the procedure name
```