|  |  |  |
| --- | --- | --- |
|  | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | |
|  | | |
| Кафедра теоретической и прикладной информатики | | |
| Лабораторная работа № 4 | | |
| по дисциплине «Языки программирования и методы трансляции» | | |
| **Разработка и реализация блока генерации кода** | | |
|  | | |
|  | Бригада | Голубь Андрей |
|  | Буданцев Дмитрий |
| Группа ПМ-13 |  |
| Вариант 8 |  |
|  |  |
|  |  |
| Преподаватель | Дворецкая Виктория Константиновна |
|  |  |
| Новосибирск,2024 | | |

1. **Цель работы**

Изучить методы генерации кода с учётом различных промежуточных форм представления программы. Изучить методы управления памятью и особенности их использования на этапе генерации кода. Научиться проектировать генератор кода.

1. ***Задачи***

Подмножество языка С++ включает:

* данные типа **int, float, struct** из элементов указанных типов**;**
* инструкции описания переменных;
* операторы присваивания в любой последовательности;
* операции +, – , <, >, побитовые операции <<, >>, &, | .

В соответствии с выбранным вариантом реализовать генератор кода. Исходными данными являются:

* синтаксическое дерево или постфиксная запись, построенные в лабораторной работе №3.
* таблицы лексем

Результатом выполнения лабораторной работы является программа на языке Ассемблер, разработанная на основе знаний и практических навыков, полученых при изучении курса “Языки программирования и методы трансляции (часть 1)”.

1. ***Входные и выходные данные***

Постоянные таблицы заполняются с помощью текстовых файлов, хранящие ключевые слова, допустимые символы, числа, операции, разделители. На файлы не накладываются дополнительные ограничения, кроме аппаратных ограничений компьютера.

Переменные таблицы заполняются в ходе работы программы, используя в качестве ключа идентификатор элемента или значение константы. Есть возможность изменять данные в процессе работы.

На количество элементов в таблицах накладываются только аппаратные ограничения.

Выходными файлами являются файлы: файл программы на языке Ассемблера, файл ошибок.

1. ***Выбор промежуточной формы хранения данных***

|  |  |  |
| --- | --- | --- |
| ***Выражение на С++*** | ***Выражение на ASM*** | ***Комментарий*** |
| struct People {  int age = 100 - 32 - 50;  float val = 98.23 - 32;  } | People struct  age dd 18  val real4 66.03  People ends | Объявление структуры |
| int age; | age dd ? | Объявление переменных |
| float val; | val real4 ? |
| age | fild age | При участии идентификатора в арифметическом выражении целого типа |
| val | fld val | При участии идентификатора в арифметическом выражении вещественного типа |
| age = val | fld val  fistp age | Присваивание при age целого типа val вещественного |
| val = age | fild age  fstp val |
| age + val | fild age  fld val  fadd | Сложение при age целого типа, val вещественного типа |
| age + 1.4 | fild age  fld constname1  fadd |
| Вычитание аналогично | | |
| age >> 4 | mov ecx, constname2  shr age, cl | Сдвиг вправо на 4, при age целое |
| age << 4 | mov ecx, constname2  shl age, cl | Сдвиг влево на 4, при age целое |
| Аналогично для выражения с операцией сдвига вправо/влево, где оба операнда целые | | |
| age | num | mov eax, num  or age, eax | Побитовый оператор “или” при age и num целым |
| age & num | mov eax, num  and age, eax | Побитовый оператор “и” при age и num целым |
| age > val | fild age  fld val  fcomip st(0), st(1)  ja greater  jb less  greater:  fild age  jmp mark\_block\_end  less:  fld val  mark\_block\_end: | Сравнение переменных age целого и val вещественного |
| val < age | fld val  fild age  fcomip st(0), st(1)  ja greater  jb less  greater:  fild age  jmp mark\_block\_end  less:  fld val  mark\_block\_end: |
| 0.1 | constname1 dd 0.1 | Объявление константы |

***5. Тесты***

|  |  |  |
| --- | --- | --- |
| **Наименование** | **Программа** | **Вывод программы** |
| Программа, удовлетворяющая правилам грамматики | struct People {  int age = 100 - 32 - 50;  float val = 98.23 - 32 >> 43;  }  // Hello  void main()  {  float a;  int b = 56 + 42;  a = b - 432;  } | age 100 32 - 50 - = ; val 98.23 32 - 43 >> = ; b 56 42 + = ; a b 432 - = ;  Ошибок нет |
| Программа, удовлетворяющая правилам грамматики | void main()  {  int a = 0;  int b;  int c;  int d;  c = 1 + 45 + a;  a = 2 >> 4 + 3 - 1;  d = 23 - 2;  } | a 0 = ; c 1 45 + a + = ; a 2 4 3 + 1 - >> = ; d 23 2 - = ;  Ошибок нет |
| Использована неинициализированная переменная | void main() {  // -\_-  temp = 321 - 984 << 2;  int j;  float temp;  } | Syntax Error! Unknown name temp. |
| Использование недопустимого терминала | void main() {  int temp = float + 32;  } | Syntax Error: Using an unknown terminal "float"  Must be one of these: const var - ( |
| Использование не сбалансированного количества скобок | void main() {  int temp = ( 15 + 32 + 23 ) );  } | Syntax Error: Using an unknown terminal ")"  Must be one of these: + - > < >> << | & ; |

**7. Программа**

**Translator.h**

#pragma once

#include "ConstTable.h"

#include "VariableHashTable.h"

#include "Token.h"

#include "Error.h"

#define KEYWORDFILENAME "input\_files/Types.txt"

#define SEPARATORFILENAME "input\_files/Separator.txt"

#define NUMFILENAME "input\_files/Numbers.txt"

#define OPERATIONFILENAME "input\_files/Operation.txt"

#define LETTERSFILENAME "input\_files/Letters.txt"

#define PARSERFILENAME "input\_files/parse\_table.txt"

#define NAMEFILEERRORS "out\_files/error.txt"

#define NAMEFILETOKEN "out\_files/token.txt"

#define NAMEFILEIDENTIFICATORS "out\_files/identificators.txt"

#define NAMEFILECONSTANS "out\_files/constans.txt"

#define NAMEFILECONSTTABLE "out\_files/consttables.txt"

#define NAMEFILEPOSTFIX "out\_files/postfix.txt"

#ifndef TRANSLATOR

class Translator {

private:

    ConstantTable<string> operation; // 0

    ConstantTable<char> numbers; // 1

    ConstantTable<string> keyword; // 2

    ConstantTable<char> separators; // 3

    ConstantTable<char> letters; // 4

    VariableHashTable identificators; // 5

    VariableHashTable constans; // 6

    queue<Errors> QErrors;

    queue<Token> QToken;

    int num\_line = 0;

    struct table\_parser\_elem {

        vector<string> terminal;

        int jump = 0;

        bool accept = false;

        bool stack\_ = false;

        bool return\_ = false;

        bool err = false;

    };

    vector<table\_parser\_elem> parser\_table;

    vector<string> postfix\_buffer;

    void ltrim(string& str) {

        str.erase(0, str.find\_first\_not\_of("\t\n\v\f\r "));

    }

    void rtrim(string& str) {

        str.erase(str.find\_last\_not\_of("\t\n\v\f\r ") + 1);

    }

    void CheckingComments(string& line, bool& continuecheck, bool& nextcheck) {

        size\_t ind3 = line.find("\*/");

        if (continuecheck) {

            if (ind3 != -1) {

                line.erase(0, ind3 + 2);

                ind3 = line.find("\*/");

            }

            else {

                line = "";

                continuecheck = true;

                return;

            }

        }

        size\_t ind1 = line.find("//"); size\_t ind2 = line.find("/\*");

        while (ind1 != -1 || ind2 != -1 || ind3 != -1) {

            if (ind1 != -1 && (ind1 < ind2 || ind2 == -1) && (ind1 < ind3 || ind3 == -1)) {

                line.erase(ind1);

                rtrim(line);

                continuecheck = false;

                return;

            }

            else if (ind2 != -1) {

                if (ind3 == -1) {

                    line.erase(ind2);

                    rtrim(line);

                    nextcheck = true;

                    continuecheck = false;

                    return;

                }

                else if (ind2 < ind3) {

                    line.erase(ind2, ind3 - ind2 + 2);

                    ind1 = line.find("//"); ind2 = line.find("/\*"); ind3 = line.find("\*/");

                    continue;

                }

                else {

                    QErrors.push(Errors(num\_line, ERR\_BLOCKCOMMENT));

                    line = "";

                    continuecheck = false;

                    return;

                }

            }

            else {

                QErrors.push(Errors(num\_line, ERR\_BLOCKCOMMENT));

                line = "";

                continuecheck = false;

                return;

            }

        }

        ltrim(line);

        rtrim(line);

        continuecheck = false;

        return;

    }

    bool is\_number(const std::string& s)

    {

        if (s == "-") return false;

        return !s.empty() && (s.find\_first\_not\_of("-0123456789") == s.npos);

    }

    bool fill\_parser\_table() {

        ifstream ParserFile(PARSERFILENAME, ios::in);

        if (!ParserFile.is\_open()) return false;

        string temp;

        int num;

        getline(ParserFile, temp);

        while (!ParserFile.eof()) {

            table\_parser\_elem el;

            ParserFile >> temp;

            while (!is\_number(temp)) {

                el.terminal.push\_back(temp);

                ParserFile >> temp;

            }

            el.jump = stoi(temp) - 1;

            ParserFile >> temp;

            el.accept = stoi(temp) == 1;

            ParserFile >> temp;

            el.stack\_ = stoi(temp) == 1;

            ParserFile >> temp;

            el.return\_ = stoi(temp) == 1;

            ParserFile >> temp;

            el.err = stoi(temp) == 1;

            parser\_table.push\_back(el);

        }

        ParserFile.close();

    }

    string buffer\_to\_string(queue<char> buffer) {

        stringstream ss;

        while (!buffer.empty()) {

            ss << buffer.front();

            buffer.pop();

        }

        return ss.str();

    }

    void clear\_buffer(queue<char>& buffer) {

        while (!buffer.empty()) buffer.pop();

    }

    Code\_Errors LexicalLineScanWithoutComments(string& line) {

        ltrim(line);

        queue<char> buffer;

        /\*

        0 - не выбрано

        1 - число

        2 - ключевое слово

        3 - операция

        4 - идентификатор

        \*/

        /\*

        0 - не выбрано

        1 - число

        2 - ключевое слово

        3 - операция

        \*/

        int temp\_mode = 0, num\_dot = 0;

        string temp\_str;

        char temp\_char;

        for (char ch : line) {

            if (ch == ' ' || ch == ';') {

                switch (temp\_mode)

                {

                case 0:

                    if (ch == ';')

                        QToken.push(Token(3, separators.getIndex(ch)));

                    continue;

                case 1:

                    temp\_str = buffer\_to\_string(buffer);

                    if (!constans.Contains(temp\_str)) {

                        if (!constans.Add(temp\_str, "const")) return ERR\_FAILED\_ADD\_TABLE;

                    }

                    QToken.push(Token(6, constans.getIndex(temp\_str)));

                    if (ch == ';') QToken.push(Token(3, separators.getIndex(ch)));

                    temp\_mode = 0;

                    num\_dot = 0;

                    clear\_buffer(buffer);

                    continue;

                case 2:

                    temp\_str = buffer\_to\_string(buffer);

                    if (keyword.Contains(temp\_str)) {

                        keyword.Add(temp\_str);

                        QToken.push(Token(2, keyword.getIndex(temp\_str)));

                        if (ch == ';') QToken.push(Token(3, separators.getIndex(ch)));

                        temp\_mode = 0;

                        clear\_buffer(buffer);

                        continue;

                    }

                    else {

                        if (!identificators.Contains(temp\_str)) {

                                if (!identificators.Add(temp\_str)) return ERR\_FAILED\_ADD\_TABLE;

                        }

                        QToken.push(Token(5, identificators.getIndex(temp\_str)));

                        if (ch == ';') QToken.push(Token(3, separators.getIndex(ch)));

                        temp\_mode = 0;

                        clear\_buffer(buffer);

                        continue;

                    }

                case 3:

                    if (ch == ' ') {

                        temp\_str = buffer\_to\_string(buffer);

                        if (operation.Contains(temp\_str)) {

                            operation.Add(temp\_str);

                            QToken.push(Token(0, operation.getIndex(temp\_str)));

                            clear\_buffer(buffer);

                            temp\_mode = 0;

                            continue;

                        }

                    }

                    return ERR\_INVALID\_OPERATION;

                default:

                    break;

                }

            }

            else if (separators.Contains(ch)) {

                if (temp\_mode == 2) {

                    temp\_str = buffer\_to\_string(buffer);

                    if (keyword.Contains(temp\_str)) {

                        keyword.Add(temp\_str);

                        QToken.push(Token(2, keyword.getIndex(temp\_str)));

                        QToken.push(Token(3, separators.getIndex(ch)));

                        temp\_mode = 0;

                        clear\_buffer(buffer);

                        continue;

                    }

                    else {

                        if (!identificators.Contains(temp\_str)) {

                            if (!identificators.Add(temp\_str)) return ERR\_FAILED\_ADD\_TABLE;

                        }

                        QToken.push(Token(5, identificators.getIndex(temp\_str)));

                        QToken.push(Token(3, separators.getIndex(ch)));

                        temp\_mode = 0;

                        clear\_buffer(buffer);

                        continue;

                    }

                }

                separators.Add(ch);

                QToken.push(Token(3, separators.getIndex(ch)));

                continue;

            }

            else if (numbers.Contains(ch)) {

                switch (temp\_mode)

                {

                case 0:

                    buffer.push(ch);

                    temp\_mode = 1;

                    continue;

                case 1:

                    if (buffer.size() == 1) {

                        temp\_char = buffer.front();

                        if (temp\_char != '0') {

                            numbers.Add(ch);

                            buffer.push(ch);

                            continue;

                        }

                        else {

                            return ERR\_INVALID\_INT\_NUM;

                        }

                    }

                    else {

                        numbers.Add(ch);

                        buffer.push(ch);

                        continue;

                    }

                case 2:

                    numbers.Add(ch);

                    buffer.push(ch);

                    continue;

                case 3:

                    return ERR\_INVALID\_SYMBOL;

                default:

                    break;

                }

            }

            else if (letters.Contains(ch)) {

                letters.Add(ch);

                switch (temp\_mode)

                {

                case 0:

                    buffer.push(ch);

                    temp\_mode = 2;

                    continue;

                case 1:

                    return ERR\_INVALID\_INT\_NUM;

                case 2:

                    buffer.push(ch);

                    continue;

                case 3:

                    return ERR\_INVALID\_OPERATION;

                default:

                    break;

                }

            }

            else {

                if (temp\_mode == 0) {

                    if (operation.Contains(to\_string(ch))) {

                        operation.Add(to\_string(ch));

                        QToken.push(Token(0, operation.getIndex(to\_string(ch))));

                        continue;

                    }

                    temp\_mode = 3;

                    buffer.push(ch);

                    continue;

                }

                else {

                    if (temp\_mode != 3) {

                        if (ch == '.' && temp\_mode == 1) {

                            if (num\_dot == 0) {

                                buffer.push(ch);

                                num\_dot++;

                                continue;

                            }

                            else {

                                return ERR\_INVALID\_FLOAT\_NUM;

                            }

                        }

                        return ERR\_INVALID\_NAME;

                    }

                    else {

                        buffer.push(ch);

                        temp\_str = buffer\_to\_string(buffer);

                        if (operation.Contains(temp\_str)) {

                            operation.Add(temp\_str);

                            QToken.push(Token(0, operation.getIndex(temp\_str)));

                            clear\_buffer(buffer);

                            temp\_mode = 0;

                            continue;

                        }

                        else {

                            return ERR\_INVALID\_OPERATION;

                        }

                    }

                }

            }

        }

        return NO\_ERROR;

    }

    string get\_token\_info(Token& val) {

        switch (val.NUM\_TABLE)

        {

        case 0:

            return operation.getElembyIndex(val.INDEX).first;

        case 1:

            return numbers.getElembyIndex(val.INDEX).first + string();

        case 2:

            return keyword.getElembyIndex(val.INDEX).first;

        case 3:

            return separators.getElembyIndex(val.INDEX).first + string();

        case 4:

            return letters.getElembyIndex(val.INDEX).first + string();

        case 5:

            return identificators.getElemIndex(val.INDEX).name;

        case 6:

            return constans.getElemIndex(val.INDEX).name;

        default:

            return "";

        }

    }

    Errors SentenceAnalysis() {

        Token fir\_token, sec\_token;

        stack<int> parser\_stack;

        parser\_stack.push(0);

        int row = 0;

        string type;

        vector<Token> infix\_expr;

        bool postfix = false;

        bool have\_type = false;

        queue<Token> CQToken = QToken;

        string name\_token;

        bool flag\_err = false;

        fir\_token = CQToken.front(); CQToken.pop();

        bool end\_flag = CQToken.empty();

        sec\_token = CQToken.front(); CQToken.pop();

        while (!flag\_err) {

            name\_token = get\_token\_info(fir\_token);

            if (fir\_token.NUM\_TABLE == 5 && name\_token != "main") name\_token = "var";

            if (fir\_token.NUM\_TABLE == 6) name\_token = "const";

            bool is\_terminal = false;

            for (int i = 0; i < parser\_table[row].terminal.size() && !is\_terminal; i++) {

                if (parser\_table[row].terminal[i] == name\_token) is\_terminal = true;

            }

            if (is\_terminal) {

                if (end\_flag && parser\_table[row].accept) break;

                if (parser\_table[row].stack\_)

                    parser\_stack.push(row + 1);

                if (parser\_table[row].accept) {

                    if ((name\_token == "var" || name\_token == "const") && get\_token\_info(sec\_token) == "=")

                        postfix = true;

                    if (!have\_type && name\_token == "var" && identificators.getElemIndex(fir\_token.INDEX).type == "") {

                        return Errors(ERR\_UNKNOWN\_VARIABLE, parser\_table[row].terminal, get\_token\_info(fir\_token));

                    }

                    if (postfix)

                        infix\_expr.push\_back(fir\_token);

                    if (name\_token == ";" || name\_token == ",") {

                        if (!make\_postfix(infix\_expr))

                            flag\_err = true;

                        infix\_expr.clear();

                        postfix = false;

                    }

                    if (name\_token == ";")

                        have\_type = false;

                    if (name\_token == "int" || name\_token == "float" || name\_token == "struct") {

                        have\_type = true;

                        type = name\_token;

                    }

                    if (name\_token == "var" && have\_type && (row == 54 || row == 31 || row == 18))

                        identificators.SetType(get\_token\_info(fir\_token), type);

                    end\_flag = CQToken.empty();

                    fir\_token = sec\_token;

                    if (!end\_flag) {

                        sec\_token = CQToken.front(); CQToken.pop();

                    }

                }

                if (parser\_table[row].return\_) {

                    if (!parser\_stack.empty()) {

                        row = parser\_stack.top();

                        parser\_stack.pop();

                    }

                    else {

                        return Errors(ERR\_PARSESTACK);

                    }

                }

                else {

                    row = parser\_table[row].jump;

                }

            }

            else {

                if (parser\_table[row].err) {

                    return Errors(ERR\_UNKNOWN\_TERMINAL, parser\_table[row].terminal, get\_token\_info(fir\_token));

                }

                else {

                    row++;

                }

            }

        }

        if (!flag\_err && parser\_stack.size() > 1) {

            return Errors(ERR\_PARSESTACK\_NO\_EMPTY, parser\_stack);

        }

        return Errors(NO\_ERROR);

    }

    bool make\_postfix(vector<Token> v) {

        stack<string> stack\_temp;

        bool error\_flag = false;

        int index = 0;

        while (index < v.size() && !error\_flag) {

            int i;

            for (i = index; i < v.size() && !error\_flag && get\_token\_info(v[i]) != ";" && get\_token\_info(v[i]) != ","; i++) {

                string token\_text = get\_token\_info(v[i]);

                if (v[i].NUM\_TABLE == 5 || v[i].NUM\_TABLE == 6) {

                    postfix\_buffer.push\_back(token\_text);

                }

                else if (token\_text == "(") {

                    stack\_temp.push(token\_text);

                }

                else if (token\_text == ")")

                {

                    while (!stack\_temp.empty() && stack\_temp.top() != "(")

                    {

                        string tmpstr = stack\_temp.top();

                        postfix\_buffer.push\_back(tmpstr);

                        stack\_temp.pop();

                    }

                    if (stack\_temp.empty())

                    {

                        // Syntax Error: Unexpected ")"

                        QErrors.push(Errors(ERR\_INVALID\_SYMBOL));

                        error\_flag = true;

                    }

                    else

                    {

                        stack\_temp.pop();

                    }

                }

                else if (v[i].NUM\_TABLE == 0)

                {

                    while (!stack\_temp.empty() && priority\_le(token\_text, stack\_temp.top()))

                    {

                        string tmpstr = stack\_temp.top();

                        postfix\_buffer.push\_back(tmpstr);

                        stack\_temp.pop();

                    }

                    stack\_temp.push(token\_text);

                }

            }

            if (error\_flag)

            {

                postfix\_buffer.clear();

                return false;

            }

            else

            {

                while (!stack\_temp.empty() &&

                    stack\_temp.top() != "(" && stack\_temp.top() != ")")

                {

                    string tmpstr = stack\_temp.top();

                    postfix\_buffer.push\_back(tmpstr);

                    stack\_temp.pop();

                }

                if (!stack\_temp.empty())

                {

                    //Syntax Error: Brackets balance error!

                    QErrors.push(Errors(ERR\_BRACKET));

                    error\_flag = true;

                }

            }

            if (error\_flag)

            {

                postfix\_buffer.clear();

                return false;

            }

            index = i + 1;

            postfix\_buffer.push\_back(";");

        }

        return true;

    }

    // Печать постфиксной записи в файл и на экран

    void postfix\_print(const string& file\_tree)

    {

        ofstream out(file\_tree);

        for (int i = 0; i < postfix\_buffer.size(); i++)

        {

            out << postfix\_buffer[i] << " ";

        }

        out.close();

    }

    bool priority\_le(string what, string with\_what)

    {

        int pw = 0, pww = 0;

        if (what == "=" || what == "+=" || what == "-=") pw = 10;

        else if (what == "|") pw = 20;

        else if (what == "&") pw = 30;

        else if (what == "<" || what == ">") pw = 40;

        else if (what == "<<" || what == ">>") pw = 50;

        else if (what == "+" || what == "-") pw = 60;

        else pw = 70;

        if (with\_what == "=" || with\_what == "+=" || with\_what == "-=") pww = 10;

        else if (with\_what == "|") pww = 20;

        else if (with\_what == "&") pww = 30;

        else if (with\_what == "<" || with\_what == ">") pww = 40;

        else if (with\_what == "<<" || with\_what == ">>") pww = 50;

        else if (with\_what == "+" || with\_what == "-") pww = 60;

        else pww = 70;

        if (pw <= pww) return true;

        return false;

    };

public:

    Translator() {

        if (!operation.ReadKeysFile(OPERATIONFILENAME)) QErrors.push(Errors(0, FATALERR\_FAILLEDREADFILE));

        if (!numbers.ReadKeysFile(NUMFILENAME)) QErrors.push(Errors(0, FATALERR\_FAILLEDREADFILE));

        if (!keyword.ReadKeysFile(KEYWORDFILENAME)) QErrors.push(Errors(0, FATALERR\_FAILLEDREADFILE));

        if (!separators.ReadKeysFile(SEPARATORFILENAME)) QErrors.push(Errors(0, FATALERR\_FAILLEDREADFILE));

        if (!letters.ReadKeysFile(LETTERSFILENAME)) QErrors.push(Errors(0, FATALERR\_FAILLEDREADFILE));

        if (!fill\_parser\_table()) QErrors.push(Errors(FATALERR\_FAILLEDREADFILE));

    }

    void PrintToken() {

        cout << "TABLE" << "\t" << "NUM\_TABLE" << endl;

        queue<Token> copy\_token = QToken;

        Token temp\_token;

        while (!copy\_token.empty()) {

            temp\_token = copy\_token.front();

            copy\_token.pop();

            cout << temp\_token.NUM\_TABLE << "\t" << temp\_token.INDEX << endl;

        }

    }

    void PrintErrors() {

        cout << "NUM LINE" << "\t" << "CODE ERROR" << endl;

        queue<Errors> copy\_Error = QErrors;

        Errors temp\_error;

        while (!copy\_Error.empty()) {

            temp\_error = copy\_Error.front();

            copy\_Error.pop();

            cout << temp\_error.NUM\_LINE << "\t" << temp\_error.CODE << endl;

        }

    }

    void ScanningFile(const string& NameFile) {

        if (operation.empty() || numbers.empty() || keyword.empty() || letters.empty() || separators.empty() || parser\_table.empty()) return;

        ifstream ScanFile(NameFile, ios::in);

        if (!ScanFile.is\_open()) {

            QErrors.push(Errors(0, FATALERR\_FAILLEDREADSCANFILE));

            return;

        }

        string line;

        bool flag\_err = false;

        bool check = false, nextcheck = false;

        Code\_Errors code;

        while (getline(ScanFile, line)) {

            if (nextcheck) {

                check = true;

                nextcheck = false;

            }

            CheckingComments(line, check, nextcheck);

            if (check == true || line.empty()) {

                num\_line++;

                continue;

            }

            if ((code = LexicalLineScanWithoutComments(line)) != NO\_ERROR) {

                QErrors.push(Errors(num\_line, code));

                flag\_err = true;

                break;

            }

            num\_line++;

        }

        ScanFile.close();

        if (!flag\_err) {

            Errors err = SentenceAnalysis();

            if (err.CODE == NO\_ERROR) {

                postfix\_print(NAMEFILEPOSTFIX);

            }

            else {

                QErrors.push(err);

            }

        }

        if (check || nextcheck) QErrors.push(Errors(num\_line, ERR\_BLOCKCOMMENT));

        if (!outFileToken()) {

            cerr << NAMEFILETOKEN << " file not opened!";

        }

        if (!outFileErrors()) {

            cerr << NAMEFILEERRORS << " file not opened!";

            return ;

        }

        if (!outFileIC()) {

            cerr << NAMEFILEIDENTIFICATORS << " file not opened!";

            return ;

        }

        if (!outFileC()) {

            cerr << NAMEFILECONSTANS << " file not opened!";

            return ;

        }

        if (!outConstTable()) {

            cerr << NAMEFILECONSTTABLE << " file not opened!";

            return ;

        }

    }

    bool outFileToken() {

        ofstream FileOut(NAMEFILETOKEN);

        if (!FileOut.is\_open()) {

            cerr << "Error! File not opened!" << endl;

            return false;

        }

        queue<Token> CQtoken = QToken;

        Token t\_token;

        while (!CQtoken.empty()) {

            t\_token = CQtoken.front();

            CQtoken.pop();

            FileOut << t\_token.NUM\_TABLE << "\t" << t\_token.INDEX << endl;

        }

        FileOut.close();

        return true;

    }

    bool outFileErrors() {

        ofstream FileOut(NAMEFILEERRORS);

        if (!FileOut.is\_open()) {

            cerr << "Error! File not opened!" << endl;

            return false;

        }

        queue<Errors> CQErrors = QErrors;

        Errors t\_error;

        while (!CQErrors.empty()) {

            t\_error = CQErrors.front();

            CQErrors.pop();

            FileOut << t\_error.info() << endl;

        }

        FileOut.close();

        return true;

    }

    bool outFileIC() {

        ofstream FileOut(NAMEFILEIDENTIFICATORS);

        return identificators.outFile(FileOut);

        FileOut.close();

    }

    bool outFileC() {

        ofstream FileOut(NAMEFILECONSTANS);

        return constans.outFile(FileOut);

        FileOut.close();

    }

    bool outConstTable() {

        ofstream FileOut(NAMEFILECONSTTABLE);

        if (!FileOut.is\_open()) {

            cerr << "Error! File not opened!" << endl;

            return false;

        }

        letters.outfile(FileOut);

        keyword.outfile(FileOut);

        operation.outfile(FileOut);

        separators.outfile(FileOut);

        numbers.outfile(FileOut);

        FileOut.close();

        return true;

    }

};

#endif // !TRANSLATOR