**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования**

**«Московский государственный технический университет имени Н.Э. Баумана»**

**Кафедра «Теоретическая информатика и компьютерные технологии»**

**ЛАБОРАТОРНАЯ РАБОТА № 5, 8**

по дисциплине:

«Теория формальных языков»

**Выполнил студент группы ИУ9-51Б:**

Варениця Михайло

**Проверил преподаватель:**

Магазов С.С

**Москва**

**2020**

**Постановка задачи.**

Цель — научиться разрабатывать программные реализации лексического анализатора.

**Порядок выполнения лабораторной работы**

1. Выбрать язык программирования и определить в нём фрагмент L, для которого будет реализовываться лексический анализатор.

2. Построить для L грамматику G.

3. Определить для L множество лексем и токенов ∑.

4. Переписать G в грамматику токенов G`.

5. Используя результаты лабораторной работы №5 «Лексический анализатор», нормализовать грамматику токенов и привести её к нормальной форме Грейбах.

6. Привести примеры правильного и неправильного выражений в грамматике токенов (недлинных) и построить вручную их вывод «сверху вниз» и «снизу вверх».

Построить лексический анализатор, который будет выполнять следующие функции:

* нормализацию исходной программы;
* трансляцию программы в промежуточный код;
* заполнение таблиц лексем (можно предложить альтернативный способ хранения информации, отсутствующей в промежуточном коде);
* поиск ошибок на уровне лексера с указанием их места в исходной программе.

**Реализация.**

**Часть 1.**

Выбранный язык программирования для лексического анализа: **JAVA**.

Протестируем программу на фрагменте L:

void method1() {  
  
 int \_\_\_aaa = 0, bbb = 14312, ccc = 4114, ddd, eee; // поддерживание \_\_\_ в наименованиях сущностей  
 signed int c = 2, c1 = 234+(2+9)+\_\_\_aaa;  
 unsigned int sd;  
 bool werty = true;  
 bool werty2 = false;  
 char a;  
 unsigned char \_3546789\_\_\_\_\_\_\_\_\_\_\_;  
 short c;  
 long d;  
 float e = 0.0, f = 0.;  
 double g = 6567.77869;  
 double g1 = 6567.77869;  
 double g2 = 6567.77869;  
  
 g2 = g1;  
  
 /\*  
 \* многострочный  
 \* комментарий  
 \*/  
  
 c = d \* f / 2; /\*tiouhoihohi\*/  
  
 if (i < 3) {  
 c = d \* f / 2; /\*tiouhoihohi\*/  
 }else if (i > 234) { // пустое тело  
  
 }  
  
 for (i = 0; i < 6754; i = i + 1) {  
 if (i < 3) {  
 c = d \* f / 2; /\*tiouhoihohi\*/  
 }else if (i > 234) { // пустое тело  
  
 } else if (i <= 234) { // <=  
  
 int asdfasdf = 3425234512;  
  
 while (i < 23543) {  
 i = i + 1;  
 }  
  
 } else if (i == adsad) { // ==  
 g = g + 3;  
 } else if (i != 42352354) { // !=  
 g = g + i + 1;  
 } else {  
 return 1234;  
 }  
 for (i = 0; i < 6754; i = i + 1) {  
 c = d \* f / 2; /\*tiouhoihohi\*/  
 }  
 }  
  
 while (i < 23543) {  
 while (i < 23543) {  
 i = i + 1;  
 while (i < 23543) {  
 i = i + 1;  
 for (i = 0; i < 6754; i = i + 1) {  
  
 }  
 }  
 }  
 i = i + 1;  
 }  
  
 do {  
  
 do {  
  
 do {  
 i = i - 1;  
 } while (i > 0);  
  
 i = i - 1;  
 } while (i > 0);  
  
 i = i - 1;  
 } while (i > 0);  
  
  
 return tmp;  
}

**Часть 2.**

Грамматика G для выбранного фрагмента L имеет вид:

G = < N, Σ, P, S >, где

N = {

S, H, I, I1, L, L1, T, T1, N, N1, M, A, O, D, Q, B, F } – множество нетерминальных (служебных) символов.

Σ = { bool, float, double, signed, unsigned, short, int, long, char, void, +, -, \*, /, =, !, >, <, ;, (, ), {, }, true, false, if, else, while, for, do, return, ,,

a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

} – алфавит символов,

P – множество правил переходов (вывода),

S – стартовый (начальный) символ.

**Множество P правил вывода описывается следующими правилами:**

*разбор методов (функций):*

S 🡪 T I ( ) { M } H

H 🡪 S | ε

*Представление типов данных:*

T 🡪 T1 short | T1 int | T1 long | T1 char | bool | float | double | void -

T1 🡪 signed | unsigned | ε

*Представление названий функций, переменных (идентификаторы):*

I 🡪 L | L I1

I1 🡪 L I1 | N I1 | ε

*Буквы латинского алфавита:*

L 🡪 a L1 | b L1 | c L1 | d L1 | e L1 | f L1 | g L1 | h L1 | i L1 | j L1 | k L1 | l L1 | m L1 | n L1 | o L1 | p L1 | q L1 | r L1 | s L1 | t L1 | u L1 | v L1 | w L1 | x L1 | y L1 | z L1

L1 🡪 L | ε *- нужно, чтобы не допускались пустые имена.*

*Цифры:*

N 🡪 0 N1 | 1 N1 | 2 N1 | 3 N1 | 4 N1 | 5 N1 | 6 N1 | 7 N1 | 8 N1 | 9 N1

N1 🡪 N | ε *- нужно, чтобы не допускались пустые числа.*

*Разбор кода, который может находиться в фигурных скобках {}:*

M 🡪 T A ; M |

A ; M |

for ( A; B; A ) { M } M |

while ( B ) { M } M |

do { M } while ( B ) M |

if ( B ) { M } F M |

return O ; M |

; M | ε

*Объявления переменных и операции присваивания:*

A 🡪 I, A |

I = O, A |

I = O |

I

*Арифметические операции над числами или переменными:*

O 🡪 (O) Q O |

(O) |

D QO |

D

D 🡪 I | N | true | false

Q 🡪 + | - | \* | /

*Операторы сравнения:*

B 🡪 O == O |

O != O |

O > O |

O < O |

O >= O |

O <= O

*Ветка else для if:*

F 🡪 else { M } |

else if ( B ) { M } F | ε

**Часть 3.**

**Множество лексем ∑1** = {

bool, float, double, short, signed short, unsigned short, int, signed int, unsigned int, long, signed long, unsigned long, char, signed char, unsigned char, void, +, -, \*, /, =, ==, !=, >, <, >=, <=, ;, (, ), {, }, true, false, if, else, while, for, do, return, ,,

a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

}

**Множество токенов ∑2** = {

<T>, <+>, <->, <\*>, </>, <=>, <==>, <!=>, <>>, <<>, <>=>, <<=>, <;>, <(>, <)>, <{>, <}>,

<if>, <else>, <while>, <for>, <do>, <return>, <,>, <ID, n>, <D, n>

}, где <ID, n> - идентификатор, <D, n> - число

**Сопоставим токены лексемам:**

1. bool, float, double, short, signed short, unsigned short, int, signed int, unsigned int, long, signed long, unsigned long, char, signed char, unsigned char, void – лексемы принадлежат токену <T>
2. + - лексема принадлежит токену <+>
3. - - лексема принадлежит токену <->
4. \* - лексема принадлежит токену <\*>
5. / - лексема принадлежит токену </>
6. = - лексема принадлежит токену <=>
7. == - лексема принадлежит токену <==>
8. != - лексема принадлежит токену <!=>
9. > - лексема принадлежит токену <>>
10. < - лексема принадлежит токену <<>
11. >= - лексема принадлежит токену <>=>
12. <= - лексема принадлежит токену <<=>
13. ; - лексема принадлежит токену <;>
14. ( - лексема принадлежит токену <(>
15. ) - лексема принадлежит токену <)>
16. { - лексема принадлежит токену <{>
17. } - лексема принадлежит токену <}>
18. if - лексема принадлежит токену <if>
19. else - лексема принадлежит токену <else>
20. while - лексема принадлежит токену <while>
21. for - лексема принадлежит токену <for>
22. do - лексема принадлежит токену <do>
23. return – лексема принадлежит токену <return>
24. , - лексема принадлежит токену <,>
25. a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

* Лексемы принадлежат токену <ID, n>

1. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

* Лексемы принадлежат токену <D, n>

**Часть 4.**

**Грамматика токенов.**

Грамматика G’ для выбранного фрагмента L имеет вид:

G’ = < N, Σ, P, S >, где

N = { S, H, I, N, M, A, O, D, Q, B, F } – множество нетерминальных (служебных) символов.

Σ = {

<T>, <+>, <->, <\*>, </>, <=>, <==>, <!=>, <>>, <<>, <>=>, <<=>, <;>, <(>, <)>, <{>, <}>, <if>, <else>, <while>, <for>, <do>, <return>, <,>, <ID, n>, <D, n>

} – алфавит символов (множество токенов),

P – множество правил переходов (вывода),

S – стартовый (начальный) символ.

Построенная грамматика G’ является приведенной, с устраненной левой рекурсией.

**Множество P правил вывода описывается следующими правилами:**

*разбор методов (функций):*

S 🡪 <T> I <(> <)> <{> M <}> H

H 🡪 S | ε

*Представление идентификаторов и чисел:*

I 🡪 <ID, n>

N 🡪 <D, n>

*Разбор кода, который может находиться в фигурных скобках {}:*

M 🡪 <T> A <;> M |

A <;> M |

<for> <(> A <;> B <;> A <)> <{> M <}> M |

<while> <(> B<)> <{> M <}> M |

<do> <{> M <}> <while> <(> B <)> M |

<if> <(> B <)> <{> M <}> F M |

<return> O <;> M |

<;> M | ε

*Объявления переменных и операции присваивания:*

A 🡪 I <,> A |

I <=> O <,> A |

I <=> O |

I

*Арифметические операции над числами или переменными:*

O 🡪 <(> O <)> Q O |

<(> O <)> |

D Q O |

D

D 🡪 I | N

Q 🡪 <+> | <-> | <\*> | </>

*Операторы сравнения:*

B 🡪 O <==> O |

O <!=> O |

O <>> O |

O <<> O |

O <>=> O |

O <<=> O

*Ветка else для if:*

F 🡪 <else> <{> M <}> |

<else> <if> <(> B <)> <{> M <}> F | ε

**Часть 5.**

**Грамматика токенов**

**в нормальной форме Грейбах.**

За исходную грамматику возьмем G’, приведенную выше.

Напомню, что грамматика G’ является приведенной, без левых рекурсий

Зададим на ней линейный порядок нетерминалов:

S < H < M < A < Q < B < O < D < I < N < F

**Пункт 1.**

Применив основную часть алгоритма преобразования к форме Грейбах получил грамматику, **правые части правил которой начинаются с терминалов, а далее идет цепочка как терминалов, так и нетерминалов.**

S – стартовое (начальное) состояние.

S 🡪 <T> I <(> <)> <{> M <}> H

H 🡪 <T> I <(> <)> <{> M <}> H | ε

M 🡪 <T> A <;> M |

<ID, n> <,> A <;> M |

<ID, n> <=> O <,> A <;> M |

<ID, n> <=> O <;> M |

<ID, n> <;> M |

<for> <(> A <;> B <;> A <)> <{> M <}> M |

<while> <(> B<)> <{> M <}> M |

<do> <{> M <}> <while> <(> B <)> M |

<if> <(> B <)> <{> M <}> F M |

<return> O <;> M |

<;> M | ε

A 🡪 <ID, n> <,> A |

<ID, n> <=> O <,> A |

<ID, n> <=> O |

<ID, n>

Q 🡪 <+> | <-> | <\*> | </>

B 🡪 <(> O <)> Q O <==> O |

<(> O <)> <==> O |

<ID, n> Q O <==> O |

<D, n> Q O <==> O |

<ID, n> D <==> O |

<D, n> D <==> O |

<(> O <)> Q O <!=> O |

<(> O <)> <!=> O |

<ID, n> Q O <!=> O |

<D, n> Q O <!=> O |

<ID, n> D <!=> O |

<D, n> D <!=> O |

<(> O <)> Q O <>> O |

<(> O <)> <>> O |

<ID, n> Q O <>> O |

<D, n> Q O <>> O |

<ID, n> D <>> O |

<D, n> D <>> O |

<(> O <)> Q O <<> O |

<(> O <)> <<> O |

<ID, n> Q O <<> O |

<D, n> Q O <<> O |

<ID, n> D <<> O |

<D, n> D <<> O |

<(> O <)> Q O <>=> O |

<(> O <)> <>=> O |

<ID, n> Q O <>=> O |

<D, n> Q O <>=> O |

<ID, n> D <>=> O |

<D, n> D <>=> O |

<(> O <)> Q O <<=> O |

<(> O <)> <<=> O |

<ID, n> Q O <<=> O |

<D, n> Q O <<=> O |

<ID, n> D <<=> O |

<D, n> D <<=> O |

O 🡪 <(> O <)> Q O |

<(> O <)> |

<ID, n> Q O |

<D, n> Q O |

<ID, n> |

<D, n>

D 🡪 <ID, n> |

<D, n>

I 🡪 <ID, n>

N 🡪 <D, n>

F 🡪 <else> <{> M <}> |

<else> <if> <(> B <)> <{> M <}> F | ε

**Пункт 2.**

Теперь усилим грамматику сделав так, чтобы правые части правил начинались с терминала, а далее шла цепочка лишь нетерминалов.

S – стартовое (начальное) состояние.

Y1 🡪 <(>

Y2 🡪 <)>

Y3 🡪 <{>

Y4 🡪 <}>

Y5 🡪 <;>

Y6 🡪 <,>

Y7 🡪 <=>

Y8 🡪 <==>

Y9 🡪 <!=>

Y10 🡪 <>>

Y11 🡪 <<>

Y12 🡪 <>=>

Y13 🡪 <<=>

Y14 🡪 <if>

Y15 🡪 <while>

S 🡪 <T> I Y1 Y2 Y3 M Y4 H

H 🡪 <T> I Y1 Y2 Y3 M Y4 H | ε

M 🡪 <T> A Y5 M |

<ID, n> Y6 A Y5 M |

<ID, n> Y7 O Y6 A Y5 M |

<ID, n> Y7 O Y5 M |

<ID, n> Y5 M |

<for> Y1 A Y5 B Y5 A Y2 Y3 M Y4 M |

<while> Y1 BY2 Y3 M Y4 M |

<do> Y3 M Y4 Y15 Y1 B Y2 M |

<if> Y1 B Y2 Y3 M Y4 F M |

<return> O Y5 M |

<;> M | ε

A 🡪 <ID, n> Y6 A |

<ID, n> Y7 O Y6 A |

<ID, n> Y7 O |

<ID, n>

Q 🡪 <+> | <-> | <\*> | </>

B 🡪 <(> O Y2 Q O Y8 O |

<(> O Y2 Y8 O |

<ID, n> Q O Y8 O |

<D, n> Q O Y8 O |

<ID, n> D Y8 O |

<D, n> D Y8 O |

<(> O Y2 Q O Y9 O |

<(> O Y2 Y9 O |

<ID, n> Q O Y9 O |

<D, n> Q O Y9 O |

<ID, n> D Y9 O |

<D, n> D Y9 O |

<(> O Y2 Q O Y10 O |

<(>O Y2 Y10 O |

<ID, n> Q O Y10 O |

<D, n> Q O Y10 O |

<ID, n> D Y10 O |

<D, n> D Y10 O |

<(> O Y2 Q O Y11 O |

<(> O Y2 Y11 O |

<ID, n> Q O Y11 O |

<D, n> Q O Y11 O |

<ID, n> D Y11 O |

<D, n> D Y11 O |

<(> O Y2 Q O Y12 O |

<(> O Y2 Y12 O |

<ID, n> Q O Y12 O |

<D, n> Q O Y12 O |

<ID, n> D Y12 O |

<D, n> D Y12 O |

<(> O Y2 Q O Y13 O |

<(> O Y2 Y13 O |

<ID, n> Q O Y13 O |

<D, n> Q O Y13 O |

<ID, n> D Y13 O |

<D, n> D Y13 O

O 🡪 <(> O Y2 Q O |

<(> O Y2 |

<ID, n> Q O |

<D, n> Q O |

<ID, n> |

<D, n>

D 🡪 <ID, n> |

<D, n>

I 🡪 <ID, n>

N 🡪 <D, n>

F 🡪 <else> Y3 M Y4 |

<else> Y14 Y1 B Y2 Y3 M Y4 F | ε

Здесь пришлось добавить новые состояния, согласно алгоритму:

Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8, Y9, Y10, Y11, Y12, Y13, Y14, Y15

**Исходный код программы.**

**Язык реализации:** C++

**Файл** main.cpp:

#include <iostream>  
#include <fstream>  
#include <sstream>  
#include <regex>  
#include <set>  
  
#include "token.hpp"  
  
  
using namespace std;  
  
  
const static string inputFilePath = "D:\\testFragment.cpp";  
  
// Определение регулярных выражений  
/\*  
 R - отключает ескейп последовательности. \n? - отступ строки 0 ил и1 раз.  
 //.\* - однострочные комментарии | - или многострочные комментарии (\* нужно экранировать т.к. служебный символ)  
\*/  
const static regex commentsRegEx(R"(\n?//.\*|/\\*[\W\w]\*?\\*/)");  
const static regex spacesRegEx(R"([ \t]\*(\W)[ \t]\*)");  
const static regex tokensRegEx(R"(<=|>=|==|!=|[(){};=+\-\*/<>=!,]|[^ \n\t(){};=+\-\*/<>=!,]+)");  
//const static regex numberRegEx(R"(^[0-9]+$)");  
const static regex numberRegEx(R"(^[0-9]+|[0-9]+.[0-9]\*$)");  
const static regex idRegEx(R"(^[a-zA-Z\_][a-zA-Z0-9\_]\*$)");  
  
// Определение прототипов функций  
  
string normalizeProgramCode(string program);  
  
tuple<vector<token>, set<token>, vector<token>> tokenizeProgram(const string &);  
  
string syntaxAnalyzer(const vector<token> &tokens);  
  
bool recognizeExpectedToken(const vector<token> &tokens, size\_t index, const tokenType &expected\_type);  
  
string getErrorMessage(const vector<token> &, size\_t, string);  
  
pair<string, size\_t> rule\_s(const vector<token> &, size\_t);  
  
pair<string, size\_t> rule\_h(const vector<token> &, size\_t);  
  
pair<string, size\_t> rule\_i(const vector<token> &, size\_t);  
  
pair<string, size\_t> rule\_n(const vector<token> &, size\_t);  
  
pair<string, size\_t> rule\_m(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_m\_1(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_m\_2(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_m\_3(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_m\_4(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_m\_5(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_m\_6(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_m\_7(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_m\_8(const vector<token> &, size\_t);  
  
pair<string, size\_t> rule\_a(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_a\_1(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_a\_2(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_a\_3(const vector<token> &, size\_t);  
  
pair<string, size\_t> rule\_o(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_o\_1(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_o\_2(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_o\_3(const vector<token> &, size\_t);  
  
pair<string, size\_t> rule\_d(const vector<token> &, size\_t);  
  
pair<string, size\_t> rule\_q(const vector<token> &, size\_t);  
  
pair<string, size\_t> rule\_b(const vector<token> &, size\_t);  
  
pair<string, size\_t> rule\_f(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_f\_1(const vector<token> &, size\_t);  
  
pair<string, size\_t> subrule\_f\_2(const vector<token> &, size\_t);  
  
  
// Реализация прототипов функций  
  
string normalizeProgramCode(string program)  
{  
 stringstream result;  
  
 if (regex\_search(program, commentsRegEx))  
 {  
 smatch smatchComments;  
 while (regex\_search(program, smatchComments, commentsRegEx))  
 {  
 result << smatchComments.prefix(); // Все начало строки, до первого сопоставления регулярки  
 program = smatchComments.suffix(); // Оставшаяся часть строки, которую regex\_search еще не проверил  
 }  
 }  
 result << program; // Дописываем остольную часть строки, в которой regex\_search ничего не находит  
  
 program = result.str();  
 result.clear();  
  
 if (regex\_search(program, spacesRegEx))  
 {  
 program = regex\_replace(program, spacesRegEx,  
 "$1"); // $n - ссылка на n-ую скобочную группу. Т.е. здесь меняем на (\W) если она есть  
 }  
  
 return program;  
}  
  
  
tuple<vector<token>, set<token>, vector<token>> tokenizeProgram(const string &program)  
{  
 vector<string> lexemes;  
  
 sregex\_iterator sregex\_iterator(program.begin(), program.end(), tokensRegEx);  
 for (auto it = sregex\_iterator, sregex\_iterator\_end = std::sregex\_iterator(); it != sregex\_iterator\_end; it++)  
 {  
 lexemes.emplace\_back(it->str());  
 }  
  
 map<string, token> tokens\_instances;  
 tokens\_instances.emplace("+", token(tokenType::*plus*, "+"));  
 tokens\_instances.emplace("-", token(tokenType::*minus*, "-"));  
 tokens\_instances.emplace("\*", token(tokenType::*multiply*, "\*"));  
 tokens\_instances.emplace("/", token(tokenType::*divide*, "/"));  
 tokens\_instances.emplace("=", token(tokenType::*assignment*, "="));  
 tokens\_instances.emplace("==", token(tokenType::*equal*, "=="));  
 tokens\_instances.emplace("!=", token(tokenType::*not\_equal*, "!="));  
 tokens\_instances.emplace(">", token(tokenType::*greater*, ">"));  
 tokens\_instances.emplace("<", token(tokenType::*less*, "<"));  
 tokens\_instances.emplace(">=", token(tokenType::*greater\_or\_equal*, ">="));  
 tokens\_instances.emplace("<=", token(tokenType::*less\_or\_equal*, "<="));  
 tokens\_instances.emplace(";", token(tokenType::*semicolon*, ";"));  
 tokens\_instances.emplace("(", token(tokenType::*bracket\_left*, "("));  
 tokens\_instances.emplace(")", token(tokenType::*bracket\_right*, ")"));  
 tokens\_instances.emplace("{", token(tokenType::*bracket\_curly\_left*, "{"));  
 tokens\_instances.emplace("}", token(tokenType::*bracket\_curly\_right*, "}"));  
 tokens\_instances.emplace("if", token(tokenType::*if\_*, "if"));  
 tokens\_instances.emplace("else", token(tokenType::*else\_*, "else"));  
 tokens\_instances.emplace("while", token(tokenType::*while\_*, "while"));  
 tokens\_instances.emplace("for", token(tokenType::*for\_*, "for"));  
 tokens\_instances.emplace("do", token(tokenType::*do\_*, "do"));  
 tokens\_instances.emplace("return", token(tokenType::*return\_*, "return"));  
 tokens\_instances.emplace(",", token(tokenType::*comma*, ","));  
  
 set<string> tokens\_datatypes{"bool",  
 "signed char", "unsigned char", "char",  
 "signed short", "unsigned short", "short",  
 "signed int", "unsigned int", "int",  
 "signed long", "unsigned long", "long",  
 "float",  
 "double",  
 "void"  
 };  
  
 vector<token> tokens;  
 set<token> ids;  
 vector<token> numbers;  
  
 for (auto it = lexemes.begin(); it != lexemes.end(); it++)  
 {  
 string lexeme = \*it;  
 auto tokens\_instances\_it = tokens\_instances.find(lexeme);  
 if (tokens\_instances\_it != tokens\_instances.end())  
 {  
 tokens.emplace\_back(tokens\_instances\_it->second);  
 continue;  
 }  
 if (lexeme == "signed" || lexeme == "unsigned")  
 {  
 it++;  
 lexeme += ' ' + \*it;  
 }  
 if (tokens\_datatypes.find(lexeme) != tokens\_datatypes.end())  
 {  
 tokens.emplace\_back(token(*datatype*, lexeme));  
 continue;  
 }  
 if (regex\_match(lexeme, numberRegEx))  
 {  
 token numberToken = token(*number*, lexeme);  
 tokens.emplace\_back(numberToken);  
 numbers.emplace\_back(numberToken);  
 continue;  
 }  
 if (regex\_match(lexeme, idRegEx))  
 {  
 token idToken = token(*id*, lexeme);  
 tokens.emplace\_back(idToken);  
 ids.emplace(idToken);  
 continue;  
 }  
  
 throw "Can not recognize the lexeme: " + lexeme; // Если лексема не распознается  
 }  
  
 return make\_tuple(tokens, ids, numbers);  
}  
  
  
string syntaxAnalyzer(const vector<token> &tokens)  
{  
 string successMessage = "Program is correct.";  
 pair<string, size\_t> result;  
 result = rule\_s(tokens, 0);  
  
 if (!result.first.empty())  
 {  
 return result.first + " at pointer index = " + to\_string(result.second);  
 }  
 if (result.second < tokens.size())  
 {  
 return "Error: incorrect token at the end " + tokens[result.second].get\_value();  
 }  
  
 return successMessage;  
}  
  
  
bool recognizeExpectedToken(const vector<token> &tokens, size\_t index, const tokenType &expected\_type)  
{  
 return index < tokens.size() && tokens[index].get\_type() == expected\_type;  
}  
  
  
string getErrorMessage(const vector<token> &tokens, size\_t index, string expectedToken)  
{  
 string errorMessage = "Error: " + (index < tokens.size() ? "incorrect token near " + tokens[index].get\_value() +  
 ", expected token " + expectedToken  
 : "wrong end of program");  
 return errorMessage;  
}  
  
  
pair<string, size\_t> rule\_s(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 if (!recognizeExpectedToken(tokens, index, *datatype*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<T>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_i(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<(>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<)>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_curly\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<{>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_m(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_curly\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<}>"), index);  
 }  
 index++;  
 /\*  
 resultOfSubRules = rule\_h(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
 \*/  
  
 return make\_pair("", index);  
}  
  
/\*  
pair<string, size\_t> rule\_h(const vector<token> &tokens, size\_t index)  
{  
 pair<string, size\_t> resultOfSubRules;  
  
 resultOfSubRules = rule\_s(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return make\_pair("", index);  
 }  
 index = resultOfSubRules.second;  
  
 return make\_pair("", index);  
}  
\*/  
  
pair<string, size\_t> rule\_i(const vector<token> &tokens, size\_t index)  
{  
 if (!recognizeExpectedToken(tokens, index, *id*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<ID, n>"), index);  
 }  
 index++;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> rule\_n(const vector<token> &tokens, size\_t index)  
{  
 if (!recognizeExpectedToken(tokens, index, *number*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<D, n>"), index);  
 }  
 index++;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> rule\_m(const vector<token> &tokens, size\_t index)  
{  
 do {  
  
 pair<string, size\_t> resultOfSubRules;  
  
 resultOfSubRules = subrule\_m\_1(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 index = resultOfSubRules.second;  
 continue;  
 }  
  
 resultOfSubRules = subrule\_m\_2(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 index = resultOfSubRules.second;  
 continue;  
 }  
  
 resultOfSubRules = subrule\_m\_3(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 index = resultOfSubRules.second;  
 continue;  
 }  
  
 resultOfSubRules = subrule\_m\_4(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 index = resultOfSubRules.second;  
 continue;  
 }  
  
 resultOfSubRules = subrule\_m\_5(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 index = resultOfSubRules.second;  
 continue;  
 }  
  
 resultOfSubRules = subrule\_m\_6(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 index = resultOfSubRules.second;  
 continue;  
 }  
  
 resultOfSubRules = subrule\_m\_7(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 index = resultOfSubRules.second;  
 continue;  
 }  
  
 resultOfSubRules = subrule\_m\_8(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 index = resultOfSubRules.second;  
 continue;  
 }  
  
 break;  
 } while (true);  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_m\_1(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 if (!recognizeExpectedToken(tokens, index, *datatype*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<T>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_a(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *semicolon*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<;>"), index);  
 }  
 index++;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_m\_2(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 resultOfSubRules = rule\_a(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *semicolon*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<;>"), index);  
 }  
 index++;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_m\_3(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 if (!recognizeExpectedToken(tokens, index, *for\_*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<for>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<(>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_a(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *semicolon*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<;>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_b(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *semicolon*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<;>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_a(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<)>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_curly\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<{>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_m(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_curly\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<}>"), index);  
 }  
 index++;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_m\_4(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 if (!recognizeExpectedToken(tokens, index, *while\_*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<while>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<(>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_b(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<)>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_curly\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<{>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_m(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_curly\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<}>"), index);  
 }  
 index++;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_m\_5(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 if (!recognizeExpectedToken(tokens, index, *do\_*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<do>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_curly\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<{>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_m(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_curly\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<}>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *while\_*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<while>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<(>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_b(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<)>"), index);  
 }  
 index++;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_m\_6(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 if (!recognizeExpectedToken(tokens, index, *if\_*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<if>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<(>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_b(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<)>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_curly\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<{>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_m(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_curly\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<}>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_f(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_m\_7(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 if (!recognizeExpectedToken(tokens, index, *return\_*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<return>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_o(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *semicolon*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<;>"), index);  
 }  
 index++;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_m\_8(const vector<token> &tokens, size\_t index)  
{  
 if (!recognizeExpectedToken(tokens, index, *semicolon*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<;>"), index);  
 }  
 index++;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> rule\_a(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 resultOfSubRules = subrule\_a\_1(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
  
 resultOfSubRules = subrule\_a\_2(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
  
 resultOfSubRules = subrule\_a\_3(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
  
 resultOfSubRules = rule\_i(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 return make\_pair("", index);  
  
}  
  
  
pair<string, size\_t> subrule\_a\_1(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 resultOfSubRules = rule\_i(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *comma*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<,>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_a(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_a\_2(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 resultOfSubRules = rule\_i(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *assignment*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<=>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_o(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *comma*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<,>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_a(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_a\_3(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 resultOfSubRules = rule\_i(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *assignment*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<=>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_o(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> rule\_o(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 resultOfSubRules = subrule\_o\_1(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
  
 resultOfSubRules = subrule\_o\_2(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
  
 resultOfSubRules = subrule\_o\_3(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
  
 resultOfSubRules = rule\_d(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_o\_1(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<(>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_o(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<)>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_q(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 resultOfSubRules = rule\_o(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_o\_2(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<(>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_o(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<)>"), index);  
 }  
 index++;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_o\_3(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 resultOfSubRules = rule\_d(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
  
 resultOfSubRules = rule\_q(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 resultOfSubRules = rule\_o(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> rule\_d(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 resultOfSubRules = rule\_i(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
  
 resultOfSubRules = rule\_n(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> rule\_q(const vector<token> &tokens, size\_t index)  
{  
 if (recognizeExpectedToken(tokens, index, tokenType::*plus*) |  
 recognizeExpectedToken(tokens, index, tokenType::*minus*) |  
 recognizeExpectedToken(tokens, index, *multiply*) | recognizeExpectedToken(tokens, index, *divide*))  
 {  
 return make\_pair("", index + 1);  
 }  
  
 //return make\_pair("Error. " + (index < tokens.size() ? "Incorrect token " + tokens[index].get\_value() + ", expected token <+> or <-> or <\*> or </>" : "wrong end of program"), index);  
 return make\_pair(getErrorMessage(tokens, index, "<+> or <-> or <\*> or </>"), index);  
}  
  
  
pair<string, size\_t> rule\_b(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 resultOfSubRules = rule\_o(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, tokenType::*equal*) && !recognizeExpectedToken(tokens, index, *not\_equal*) &&  
 !recognizeExpectedToken(tokens, index, tokenType::*greater*) &&  
 !recognizeExpectedToken(tokens, index, tokenType::*less*) &&  
 !recognizeExpectedToken(tokens, index, *greater\_or\_equal*) &&  
 !recognizeExpectedToken(tokens, index, *less\_or\_equal*))  
 {  
  
 //return make\_pair("Error. " + (index < tokens.size() ? "Incorrect token " + tokens[index].get\_value() + ", expected token <==> or <!=> or <>> or <<> or <>=> or <<=>" : "wrong end of program"), index);  
 return make\_pair(getErrorMessage(tokens, index, "<==> or <!=> or <>> or <<> or <>=> or <<=>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_o(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> rule\_f(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 resultOfSubRules = subrule\_f\_1(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
  
 resultOfSubRules = subrule\_f\_2(tokens, index);  
 if (resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_f\_1(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 if (!recognizeExpectedToken(tokens, index, *else\_*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<else>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_curly\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<{>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_m(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_curly\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<}>"), index);  
 }  
 index++;  
  
 return make\_pair("", index);  
}  
  
  
pair<string, size\_t> subrule\_f\_2(const vector<token> &tokens, size\_t index)  
{  
  
 pair<string, size\_t> resultOfSubRules;  
  
 if (!recognizeExpectedToken(tokens, index, *else\_*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<else>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *if\_*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<if>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<(>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_b(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<)>"), index);  
 }  
 index++;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_curly\_left*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<{>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_m(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 if (!recognizeExpectedToken(tokens, index, *bracket\_curly\_right*))  
 {  
 return make\_pair(getErrorMessage(tokens, index, "<}>"), index);  
 }  
 index++;  
  
 resultOfSubRules = rule\_f(tokens, index);  
 if (!resultOfSubRules.first.empty())  
 {  
 return resultOfSubRules;  
 }  
 index = resultOfSubRules.second;  
  
 return make\_pair("", index);  
}  
  
  
int main()  
{  
  
 // Открываем файл  
 ifstream input(inputFilePath);  
 if (!input)  
 {  
 cerr << "Can not open the file" << endl;  
 exit(1);  
 }  
  
 // Считываем файл  
 stringstream input\_ss;  
 for (string line; getline(input, line); input\_ss << line << endl);  
 input.close();  
  
 string program = input\_ss.str();  
 cout << "Input program code:" << endl << program << endl;  
 cout << "===========================================================" << endl << endl;  
  
 program = normalizeProgramCode(program);  
 cout << "Stage 1. Normalized program code:" << endl << program << endl;  
 cout << "===========================================================" << endl << endl;  
  
 tuple<vector<token>, set<token>, vector<token>> tokens;  
  
 try  
 {  
 tokens = tokenizeProgram(program);  
 } catch (const string &exception)  
 {  
 cerr << exception << endl;  
 exit(1);  
 }  
  
 cout << "Stage 2. Lexer:" << endl;  
 for (const auto &token : get<0>(tokens))  
 {  
 cout << "<" << token.get\_value() << "> ";  
 }  
 cout << endl;  
 cout << "===========================================================" << endl << endl;  
  
 cout << "Stage 3. Tables:" << endl;  
 cout << "Identifiers:" << endl;  
  
 for (const auto &id : get<1>(tokens)) {  
 cout << id.get\_value() << ", ";  
 }  
 cout << endl;  
 cout << "Numbers:" << endl;  
 for (const auto &number : get<2>(tokens))  
 {  
 cout << number.get\_value() << ", ";  
 }  
 cout << endl;  
 cout << "===========================================================" << endl << endl;  
  
 cout << "Stage 4. Errors checking:" << endl;  
  
 cout << syntaxAnalyzer(get<0>(tokens));  
  
 cout << endl;  
 cout << "===========================================================" << endl << endl;  
  
 return 0;  
}  
  
  
  
  
/\*  
// Доопределение класса token  
  
token::token(tokenType type, std::string value): type(type), value(std::move(value))  
{  
}  
  
bool token::operator< (const token &token1) const  
{  
 return value < token1.get\_value();  
}  
  
[[nodiscard]] const tokenType &token::get\_type() const  
{  
 return this->type;  
}  
  
[[nodiscard]] const std::string &token::get\_value() const  
{  
 return this->value;  
}  
\*/  
  
/\*  
[[nodiscard]] указывает на обязательность использования результата при возврате из функции. Может быть применим как к типу (при объявлении класса или перечисления), так и непосредственно к возвращаемому типу функции. Пример:  
  
[[nodiscard]] int f() { return 42; }  
...  
f(); // сообщение о том, что результат функции не использован  
  
Явное приведение результата к void подавляет действие атрибута  
\*/

**Файл** token.hpp:

#ifndef SYNTAX\_ANALYZER\_TOKEN\_HPP  
#define SYNTAX\_ANALYZER\_TOKEN\_HPP  
  
enum tokenType {  
 *datatype*, // <T>  
 *plus*, // <+>  
 *minus*, // <->  
 *multiply*, // <\*>  
 *divide*, // </>  
 *assignment*, // <=>  
 *equal*, // <==>  
 *not\_equal*, // <!=>  
 *greater*, // <>>  
 *less*, // <<>  
 *greater\_or\_equal*, // <>=>  
 *less\_or\_equal*, // <<=>  
 *semicolon*, // <;>  
 *bracket\_left*, // <(>  
 *bracket\_right*, // <)>  
 *bracket\_curly\_left*, // <{>  
 *bracket\_curly\_right*, // <}>  
 *if\_*, // <if>  
 *else\_*, // <else>  
 *while\_*, // <while>  
 *for\_*, // <for>  
 *do\_*, // <do>  
 *return\_*, // <return>  
 *comma*, // <,>  
 *id*, // <ID, n>  
 *number* // <D, n>  
};  
  
class token {  
  
private:  
 tokenType type;  
 std::string value;  
  
public:  
 token(tokenType type, std::string value): type(type), value(std::move(value))  
 {  
 }  
  
 bool operator< (const token &token1) const  
 {  
 return value < token1.get\_value();  
 }  
  
 [[nodiscard]] const tokenType &get\_type() const  
 {  
 return this->type;  
 }  
  
 [[nodiscard]] const std::string &get\_value() const  
 {  
 return this->value;  
 }  
};  
  
#endif //SYNTAX\_ANALYZER\_TOKEN\_HPP  
  
/\*  
[[nodiscard]] указывает на обязательность использования результата при возврате из функции. Может быть применим как к типу (при объявлении класса или перечисления), так и непосредственно к возвращаемому типу функции. Пример:  
  
[[nodiscard]] int f() { return 42; }  
...  
f(); // сообщение о том, что результат функции не использован  
  
Явное приведение результата к void подавляет действие атрибута  
\*/

**Часть 6.**

**Синтаксический анализ сверху вниз.**

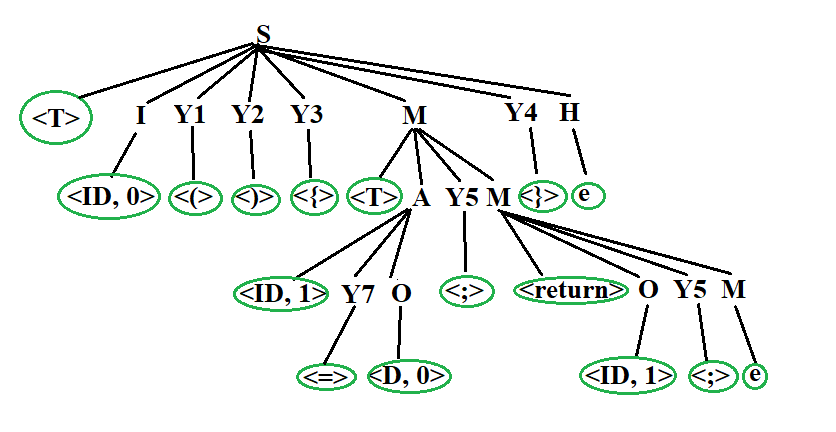
Фрагмент:

*void method1() {  
  
 int aaa = 0;*

*return aaa;  
}*

Вывод осуществляется в грамматике Грейбах.

**Успешный вывод сверху-вниз:**



**Успешный вывод снизу-вверх:**

