

Министерство науки и высшего образования Российской Федерации

Новосибирский государственный технический университет

Кафедра ТПИ

ЯЗЫКИ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Лабораторная работа № 3

Разработка и реализация блока синтаксического анализа

Факультет: ПМИ

Преподаватели:

Еланцева И.Л.,

Петров Р. В.

Группа: ПМ-81

Студенты: Ефремов А. А.,
Ртищева К. С.

Бригада: 1

Вариант: 1

Новосибирск
2021

1. Цель работы

Изучить табличные методы синтаксического анализа. Получить представление о методах диагностики и исправления синтаксических ошибок. Научиться проектировать синтаксический анализатор на основе табличных методов.

2. Условие задачи

Подмножество языка C++ включает:

- данные типа `int`;
- инструкции описания переменных;
- операторы присваивания, `if`, `if-else` любой вложенности и в любой последовательности;
- операции `+`, `-`, `*`, `==`, `!=`, `<`, `/`.

В соответствии с выбранным вариантом задания к лабораторным работам реализовать синтаксический анализатор с использованием одного из табличных методов (LL-, LR-метод, метод предшествования).

Этапы проектирования синтаксического анализатора:

1. Сконструировать КС-грамматику в соответствии с вариантом задания.
2. В случае несоответствия построенной грамматики требованиям выбранного табличного метода разбора следует провести эквивалентные преобразования грамматики либо выбрать другой метод разбора.
3. Построить таблицу разбора и запрограммировать драйвер, реализующий работу с этой таблицей.

Исходные данные – файл токенов, таблицы лексем.

Результатом работы синтаксического анализатора является:

- синтаксическое дерево или постфиксная запись;
- файл сообщений об ошибках. В лабораторной работе необходимо реализовать возможности табличного метода по диагностике и исправлению синтаксических ошибок в исходной программе.

3. Вид, структура входных и выходных данных

Входные данные:

Файл токенов `"tokens.txt"`; файл `"postfix.txt"` для вывода результата; файл `"parsingTable"`, в котором содержится таблица разбора.

Токен имеет вид: `(tableNum, index)`, `tableNum` – номер таблицы, `index` – номер лексемы в таблице.

`tableNum = 10` – таблица ключевых слов;

`tableNum = 20` – таблица операторов;

`tableNum = 30` – таблица переменных;

`tableNum = 40` – таблица констант.

5. Грамматика входного языка

Начальный символ		
S -> int main () { BODY EXIT }		
Операторы		
Арифметические		Логические
1	OP1 -> +	1 OP2 -> ==
2	OP1 -> -	2 OP2 -> !=
3	OP1 -> *	3 OP2 -> <
4	OP1 -> /	
Переменные и константы		
VAR -> VAR1		Имя переменной
VAR -> VAR2		Значение константы
VAR1 -> v_name		VAR2 -> const
Выражение		
1	EXPR1 -> VAR EXPR2	Переменная и константа и дальнейшая операция
2	EXPR1 -> (EXPR1) EXPR2	Выражение в скобках
1	EXPR2 -> OP1 EXPR1	Бинарный оператор и выражение
2	EXPR2 -> emps	Конец выражения
Тело программы		
1	BODY -> ASSIGNMENT BODY	Операции присваивания
2	BODY -> INT DEC1; BODY	Операции объявления
3	BODY -> IF BODY	Условный оператор
4	BODY -> emps	Пустая строка
Объявление		
DEC1 -> v_name DEC2 DEC3		Имя переменной
1	DEC2 -> = EXPR	Результат выражения
2	DEC2 -> emps	Ничего
1	DEC3 -> , DEC1	Еще одна переменная
2	DEC3 -> emps	Ничего
Присваивание		
ASSIGNMENT -> v_name = EXPR ;		Имя, тип присваивания, выражение
Условный оператор		
IF -> if (CONDITION) { BODY } ELSE		
Условное выражение		
CONDITION -> EXPR1 OP2 EXPR1		
Иначе		
1	ELSE -> else { BODY }	Действие
2	ELSE -> emps	Бездействие
Выход из программы		
1	EXIT -> return 0 ;	
2	EXIT -> emps	

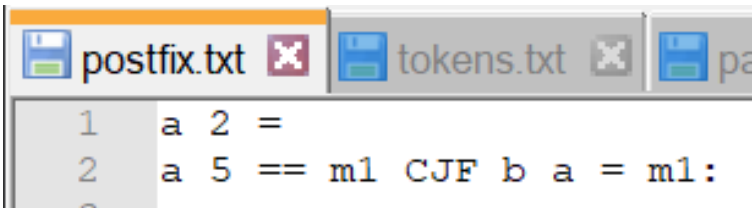
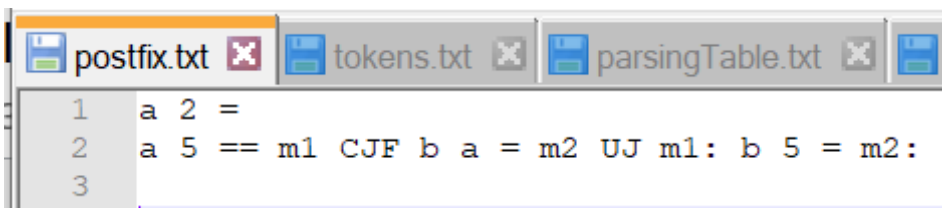
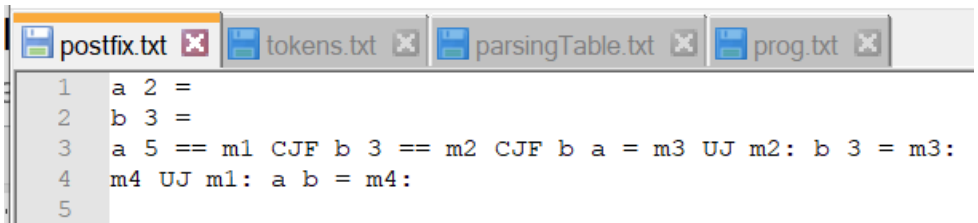
6. Таблица разбора

№	Terminals	jump	accept	stack	return	error
1	int	2	0	0	0	1
2	int	3	1	0	0	1
3	main	4	1	0	0	1
4	(5	1	0	0	1
5)	6	1	0	0	1
6	{	7	1	0	0	1
7	v_name int if return }	10	0	1	0	1
8	return }	89	0	1	0	1
9	}	0	1	0	1	1
10	v_name	14	0	0	0	0
11	int	45	0	0	0	0
12	if	62	0	0	0	0
13	return }	88	0	0	0	1
14	v_name	16	0	1	0	1
15	v_name int if return }	10	0	0	0	1
16	v_name	17	1	0	0	1
17	=	18	1	0	0	1
18	v_name const (20	0	1	0	1
19	;	0	1	0	1	1
20	v_name const	22	0	0	0	0
21	(41	0	0	0	1
22	v_name const	24	0	1	0	1
23	+ - * / ; ,) == != <	28	0	0	0	1
24	v_name	26	0	0	0	0
25	const	27	0	0	0	1
26	v_name	0	1	0	1	1
27	const	0	1	0	1	1
28	+ - * /	30	0	0	0	0
29	; ,) == != <	40	0	0	0	1
30	+ - * /	32	0	1	0	1
31	v_name const (20	0	0	0	1
32	+	36	0	0	0	0
33	-	37	0	0	0	0
34	*	38	0	0	0	0
35	/	39	0	0	0	1
36	+	0	1	0	1	1
37	-	0	1	0	1	1
38	*	0	1	0	1	1
39	/	0	1	0	1	1
40	; ,) == != <	0	0	0	1	1
41	(42	1	0	0	1
42	v_name const (20	0	1	0	1
43)	44	1	0	0	1
44	+ - * / ; , != <) ==	28	0	0	0	1
45	int	46	1	0	0	1
46	v_name	49	0	1	0	1
47	;	48	1	0	0	1

48	v_name int if return }	10	0	0	0	1
49	v_name	50	1	0	0	1
50	= , ;	52	0	1	0	1
51	, ;	57	0	0	0	1
52	=	54	0	0	0	0
53	, ;	56	0	0	0	1
54	=	55	1	0	0	1
55	v_name const (20	0	0	0	1
56	, ;	0	0	0	1	1
57	,	59	0	0	0	0
58	;	61	0	0	0	1
59	,	60	1	0	0	1
60	v_name	49	0	0	0	1
61	;	0	0	0	1	1
62	if	64	0	1	0	1
63	v_name int if return }	10	0	0	0	1
64	if	65	1	0	0	1
65	(66	1	0	0	1
66	v_name const (72	0	1	0	1
67)	68	1	0	0	1
68	{	69	1	0	0	1
69	v_name int if return }	10	0	1	0	1
70	}	71	1	0	0	1
71	else return }	81	0	0	0	1
72	v_name const (20	0	1	0	1
73	== != <	75	0	1	0	1
74	v_name const (20	0	0	0	1
75	==	78	0	0	0	0
76	!=	79	0	0	0	0
77	<	80	0	0	0	1
78	==	0	1	0	1	1
79	!=	0	1	0	1	1
80	<	0	1	0	1	1
81	else	83	0	0	0	0
82	return }	87	0	0	0	1
83	else	84	1	0	0	1
84	{	85	1	0	0	1
85	v_name int if return }	10	0	1	0	1
86	}	0	1	0	1	1
87	return }	0	0	0	1	1
88	return }	0	0	0	1	1
89	return	91	0	0	0	0
90	}	94	0	0	0	1
91	return	92	1	0	0	1
92	0	93	1	0	0	1
93	;	0	1	0	1	1
94	}	0	0	0	1	1

7. Тестовые примеры

№	Входные данные	Выходные данные	Назначение
1	<pre>int main() { a = 2; return 0; }</pre>	<p>Выбрать Консоль отладки Microsoft Visual Studio</p> <p>Error: Unknown identifier 'a'!</p>	Необъявленная переменная
2	<pre>int main() { int a; int a = 2; return 0; }</pre>	<p>Консоль отладки Microsoft Visual Studio</p> <p>Error: redefining the type of a variable 'a'!</p>	Повторное объявление переменной
3	<pre>int main() { int c; int a = c + 2; return 0; }</pre>	<p>Консоль отладки Microsoft Visual Studio</p> <p>Error: Value of the variable 'c' is not set!</p>	Значение переменной не установлено
4	<pre>int main() { return; }</pre>	<p>Консоль отладки Microsoft Visual Studio</p> <p>Error: Unexpected symbol! Possible symbols: '0'</p>	Неожиданный символ
5	<pre>int main() { return 0; }</pre>	<p>Консоль отладки Microsoft Visual Studio</p> <p>Error: Incorrect end of the program! Expected '}'</p>	Некорректное завершение программы
6	<pre>int main() { int a = 2 / (1 * (3 + 4)) - 5; return 0; }</pre>	<p>postfix.txt tokens.txt pa</p> <p>1 a 2 1 3 4 + * / = 5 -</p> <p>2</p>	Проверка на формирование ОПЗ арифметического выражения

7	<pre> int main() { int a = 2, b; if (a == 5) { b = a; } return 0; } </pre>		Проверка на формирование ОПЗ if без else
8	<pre> int main() { int a = 2, b; if (a == 5) { b = a; } else { b = 5; } return 0; } </pre>		Проверка на формирование ОПЗ if с else
9	<pre> int main() { int a = 2, b = 3; if (a == 5) { if (b == 3) { b = a; } else { b = 3; } } else { a = b; } return 0; } </pre>		Проверка на формирование ОПЗ вложенного if

8. Тексты программ

Файл "VarTableRow.h"

```
#pragma once
#include <string>

using namespace std;

class VarTableRow
{
public:
    bool value;
    string name;
    bool is_set;

    VarTableRow() {}

    VarTableRow(const bool& t_value, const string& t_name, const bool t_is_set) :
        value(t_value), name(t_name), is_set(t_is_set) {}

    bool operator == (VarTableRow lhs)
    {
        return value == lhs.value && name == lhs.name && is_set == lhs.is_set;
    }
};
```

Файл "VarTable.h"

```
#pragma once
#include <vector>
#include <string>
#include <iomanip>
#include "VarTableRow.h"

using namespace std;

class VarTable
{
public:
    vector<VarTableRow> table;
```

```

// Создание пустой таблицы
VarTable()
{
    table = vector<VarTableRow>(0);
}

// Функция поиска номера строки таблицы по идентификатору,
// возвращает -1 в случае отсутствия строки с таким идентификатором в таблице
int GetRowIndex(const VarTableRow& t_row)
{
    for(size_t i = 0; i < table.size(); i++)
        if(table[i] == t_row)
            return i;

    return -1;
}

// Функция добавления строки в таблицу, если такого вхождения нет,
// возвращает номер строки
int AddRow(const VarTableRow& t_row)
{
    int index = GetRowIndex(t_row);
    if(index == -1)
    {
        table.push_back(t_row);
        return table.size() - 1;
    }
    else
        return index;
}

// Функция, возвращающая
VarTableRow GetRow(const int& t_index)
{
    if(t_index < table.size())
        return table[t_index];
    else
        printf_s("Error!");
}

void Output(const string& OUT_FILE)

```

```

{
    ofstream fout(OUT_FILE);
    fout << "i   value   name is set" << endl;
    for (size_t i = 0; i < table.size(); i++)
    {
        fout << i << setw(5) << table[i].value;
        fout << setw(8) << table[i].name;
        fout << setw(5) << table[i].is_set;
        fout << endl;
    }
    fout.close();
}

// Получение значений атрибутов
bool  GetValue(const int& t_index) { return table[t_index].value; }
string GetName(const int& t_index) { return table[t_index].name; }
bool  GetIsSet(const int& t_index) { return table[t_index].is_set; }

// Установление значений атрибутов
void SetValue(const int& t_index, const bool& t_value) { table[t_index].value = t_value; }
void SetName(const int& t_index, const string& t_name) { table[t_index].name = t_name; }
void SetIsSet(const int& t_index, const bool t_is_set) { table[t_index].is_set = t_is_set; }
};

```

Файл "ConstTableRow.h"

```

#pragma once
#include <string>

using namespace std;

class ConstTableRow
{
public:
    string name;
    ConstTableRow() {};

    ConstTableRow(const string& t_name) :
        name(t_name) {};

    bool operator == (const ConstTableRow& lhs)

```

```

    {
        return name == lhs.name;
    }
};

```

Файл "ConstTable.h"

```

#pragma once
#include <vector>
#include <string>
#include <iomanip>
#include "ConstTableRow.h"

using namespace std;

class ConstTable
{
public:
    vector<ConstTableRow> table;

    // Создание пустой таблицы
    ConstTable()
    {
        table = vector<ConstTableRow>(0);
    }

    // Создание таблицы с ключевыми словами
    void FillKeyWords()
    {
        const int k = 5;
        table.resize(k);
        string key_words[k] = {"if", "else", "main", "return", "int"};
        for(size_t i = 0; i < k; i++)
            table[i] = ConstTableRow(key_words[i]);
    }

    // Создание таблицы с операторами
    void FillOperators()
    {
        const int k = 14;
        table.resize(k);
        string operators[k] = { "=", "+", "-", "*", "/", "==", "!=", "<", "(", ")", "{", "}", ",", ";"};
    }

```

```

    for(size_t i = 0; i < k; i++)
        table[i] = ConstTableRow(operators[i]);
}

// Создание таблицы со всеми символами алфавита языка
void FillAlphabet()
{
    const int k = 15;
    table.resize(k + 26 + 26 + 10);
    string operators[k] = { "=", "+", "-", "*", "/", "=", "!", "<", "(", ")", "{", "}", ",", ";", "_" };
    for(size_t i = 0; i < k; i++)
        table[i] = ConstTableRow(operators[i]);

    for(int i = 0; i < 26; i++)
        table[i + k] = ConstTableRow(string(1, (char)('a' + i)));

    for(int i = 0; i < 26; i++)
        table[i + k + 26] = ConstTableRow(string(1, (char)('A' + i)));

    for (int i = 0; i < 10; i++)
        table[i + k + 26 + 26] = ConstTableRow(string(1, (char)('0' + i)));
}

// Создание таблицы со всеми символами алфавита языка с которых
// могут начинаться идентификаторы
void FillIdentName()
{
    table.resize(1 + 26 + 26);

    for (int i = 0; i < 26; i++)
        table[i] = ConstTableRow(string(1, (char)('a' + i)));

    for (int i = 0; i < 26; i++)
        table[i + 26] = ConstTableRow(string(1, (char)('A' + i)));

    table[52] = ConstTableRow(string(1, (char)('_')));
}

// Создание таблицы со всеми цифрами алфавита языка
void FillNumbers()
{
    table.resize(10);

```

```

    for(int i = 0; i < 10; i++)
        table[i] = ConstTableRow(string(1, (char)('0' + i)));
}

// Функция поиска номера строки таблицы по идентификатору,
// возвращает -1 в случае отсутствия строки с таким идентификатором в таблице
int GetRowIndex(const ConstTableRow& t_row)
{
    for(int i = 0; i < table.size(); i++)
        if(table[i] == t_row)
            return i;
    return -1;
}

string GetRow(const int& index) { return table[index].name; }

void Output(const string& OUT_FILE)
{
    ofstream fout(OUT_FILE);
    fout << "i    name" << endl;
    for (size_t i = 0; i < table.size(); i++)
    {
        fout << setw(2) << i;
        fout << setw(10) << table[i].name;
        fout << endl;
    }
    fout.close();
}
};

```

Файл "LexicalAnalyzer.h"

```
#pragma once
#include <iostream>
#include <string>
#include <fstream>
#include "VarTable.h"
#include "ConstTable.h"

enum class WordType
{
    Blank,
    Operator,
    Word,
    Constant
};

enum class SymbolType
{
    Separator,
    Operator,
    Letter,
    Number,
    Error
};

class LexicalAnalyzer
{
public:
    ConstTable alphabet, key_words, operators, numbers, ident_name;
    VarTable var_table, const_table;

    LexicalAnalyzer()
    {
        alphabet.FillAlphabet();
        key_words.FillKeyWords();
        operators.FillOperators();
        numbers.FillNumbers();
        ident_name.FillIdentName();
    }

    // Определение типа символа и получение его индекса
```

```

// в соответствующей таблице
SymbolType GetSymbolType(const string& s, int& place)
{
    // Разделитель
    if(s == " " || s == "\n" || s == "\t")
        return SymbolType::Separator;

    // Ошибка
    place = alphabet.GetRowIndex(ConstTableRow(s));
    if(place == -1)
        return SymbolType::Error;

    // Оператор
    place = operators.GetRowIndex(ConstTableRow(s));
    if(place != -1)
        return SymbolType::Operator;

    // Символ с которого может начинаться имя переменной
    place = ident_name.GetRowIndex(ConstTableRow(s));
    if(place != -1)
        return SymbolType::Letter;

    // Цифра
    place = numbers.GetRowIndex(ConstTableRow(s));
    if(place != -1)
        return SymbolType::Number;
}

// Печать всех таблиц
void PrintAllTables(const string& directory)
{
    alphabet.Output(directory + "/alphabet.txt");
    key_words.Output(directory + "/keyWords.txt");
    operators.Output(directory + "/operators.txt");
    numbers.Output(directory + "/numbers.txt");
    ident_name.Output(directory + "/ident_name.txt");
    const_table.Output(directory + "/const.txt");
    var_table.Output(directory + "/var.txt");
}

void MakeTokens(const string& in_filename, const string& out_filename)
{

```



```

ifstream fin(in_filename);
ofstream fout(out_filename);

int symbol_n = 0, line_n = 1;
char c;
string word = "", symbol;

// Тип предыдущего слова
WordType word_type = WordType::Blank;

// Тип символа
SymbolType symbol_type;

// Место символа в соответствующей таблице
int place = 0;

// Место предыдущего символа в соответствующей таблице
int prev_place = 0;

// Если комментирование оператором */
bool is_op_comment = false;
string prev_symbol;

// Если комментирование оператором //
bool is_line_comment = false;

while(fin.get(c))
{
    symbol = c;
    symbol_n++;
    symbol_type = GetSymbolType(symbol, place);

    if(symbol_type == SymbolType::Error)
    {
        cout << "Error at line " << line_n << " pos " << symbol_n;
        cout << ": Invalid symbol! ";
        exit(2);
    }

    if(is_op_comment)
    {
        string temp_s = prev_symbol + symbol;

```

```

    if(temp_s == "*/")
        is_op_comment = false;
    else
        prev_symbol = symbol;
}
else if(is_line_comment)
{
    if(symbol == "\n")
        is_line_comment = false;
}
else
    switch(word_type)
    {
        // Слово не задано
        case WordType::Blank:
        {
            switch(symbol_type)
            {
                case SymbolType::Separator:
                {
                    if(symbol == "\n")
                    {
                        symbol_n = 0;
                        line_n++;
                        fout << endl;
                    }
                    word_type = WordType::Blank;
                    break;
                }
                case SymbolType::Operator:
                {
                    prev_place = place;
                    word = symbol;
                    word_type = WordType::Operator;
                    break;
                }
                case SymbolType::Letter:
                {
                    word = symbol;
                    word_type = WordType::Word;
                    break;
                }
            }
        }
    }
}

```

```

        case SymbolType::Number:
        {
            word = symbol;
            word_type = WordType::Constant;
            break;
        }
    }
    break;
}
// Слово - оператор
case WordType::Operator:
{
    switch(symbol_type)
    {
        // Символ - разделитель
        case SymbolType::Separator:
        {
            fout << "(20," << place << ")";
            word_type = WordType::Blank;
            word = "";

            if(symbol == "\n")
            {
                symbol_n = 0;
                fout << endl;
                line_n++;
            }
            break;
        }
        // Символ - оператор
        case SymbolType::Operator:
        {
            string temp_op = word + symbol;

            if(temp_op == "/*")
            {
                word_type = WordType::Blank;
                word = "";
                is_op_comment = true;
                break;
            }
            if(temp_op == "//")

```

```

{
    word_type = WordType::Blank;
    word = "";
    is_line_comment = true;
    break;
}

int temp_place = operators.GetRowIndex(ConstTableRow(temp_op));

// Если оператор - "==" или "!="
if(temp_place != -1)
{
    fout << "(20," << temp_place << ")";
    word_type = WordType::Blank;
    word = "";
}
else if(temp_op == "()")
{
    fout << "(20," << prev_place << ")";
    word_type = WordType::Operator;
    word = symbol;
    prev_place = place;
}
else
{
    cout << "Error at line " << line_n << " pos " << symbol_n;
    cout << ": Invalid operator! ";
    exit(2);
    break;
}
break;
}
// Символ - буква
case SymbolType::Letter:
{
    fout << "(20," << prev_place << ")";
    word_type = WordType::Word;
    word = symbol;
    prev_place = place;
    break;
}
}

```

```

// Символ - цифра
case SymbolType::Number:
{
    fout << "(20," << prev_place << ")";
    word_type = WordType::Constant;
    word = symbol;
    prev_place = place;
    break;
}
}
break;
}
// Слово - слово
case WordType::Word:
{
    switch(symbol_type)
    {
        // Символ - разделитель
        case SymbolType::Separator:
        {
            int kw_place = key_words.GetRowIndex(ConstTableRow(word));

            // Если слово - ключевое слово
            if(kw_place != -1)
                fout << "(10," << kw_place << ")";
            else
                fout << "(30," << var_table.AddRow(VarTableRow(0, word, false)) << ")";

            word_type = WordType::Blank;
            word = "";

            if(symbol == "\n")
            {
                symbol_n = 0;
                line_n++;
                fout << endl;
            }
            break;
        }
    }
    // Символ - оператор
    case SymbolType::Operator:
    {

```

```

int kw_place = key_words.GetRowIndex(ConstTableRow(word));

// Если слово - ключевое слово
if(kw_place != -1)
    fout << "(10," << kw_place << ")";
else
    fout << "(30," << var_table.AddRow(VarTableRow(0, word, false)) << ")";
word_type = WordType::Operator;
prev_place = place;
word = symbol;
break;
}
// Символ - буква
case SymbolType::Letter:
{
    word += symbol;
    break;
}
// Символ - цифра
case SymbolType::Number:
{
    word += symbol;
    break;
}
}
break;
}
// Слово - константа
case WordType::Constant:
{
    switch(symbol_type)
    {
        // Символ - разделитель
        case SymbolType::Separator:
        {
            fout << "(40," << const_table.AddRow(VarTableRow(0, word, false)) << ")";
            word_type = WordType::Blank;
            word = "";

            if(symbol == "\n")
            {
                symbol_n = 0;
            }
        }
    }
}

```

```

        line_n++;
        fout << endl;
    }
    break;
}
// Символ - оператор
case SymbolType::Operator:
{
    fout << "(40," << const_table.AddRow(VarTableRow(0, word, false)) << ")";
    word_type = WordType::Operator;
    word = symbol;
    prev_place = place;
    break;
}
// Символ - буква
case SymbolType::Letter:
{
    cout << "Error at line " << line_n << " pos " << symbol_n;
    cout << ": Invalid constant (identifier)! ";
    exit(2);
    break;
}
// Символ - цифра
case SymbolType::Number:
{
    word += symbol;
    break;
}
}
break;
}
}
}

if(is_op_comment)
{
    cout << "Unclosed comment!";
    exit(2);
}

fout.close();
fin.close();

```

```
    }  
};
```

Файл "SyntaxAnalyzer.cpp"

```
#pragma once  
#include "LexicalAnalyzer.h"  
#include <sstream>  
#include <stack>  
#include <queue>  
#include <map>  
  
using namespace std;  
  
class SyntaxAnalyzer  
{  
public:  
    SyntaxAnalyzer()  
    {  
  
    };  
  
    struct token  
    {  
        int tableNum, index;  
        char buf;  
  
        string readToken(istream& ftoken, LexicalAnalyzer la)  
        {  
            ftoken >> buf >> tableNum;  
            ftoken >> buf >> index >> buf;  
            switch (tableNum)  
            {  
            case 10:  
                return la.key_words.GetRow(index);  
            case 20:  
                return la.operators.GetRow(index);  
            case 30:  
                return "v_name";  
            case 40:  

```



```

        if (la.const_table.GetName(index) == "0") return "0";
        return "const";
    default:
        break;
    }
};

struct parsingTableRow
{
    vector<string> terminals;
    int jump, accept, stack, retn, error;

    parsingTableRow(vector<string> _terminals, int _jump, int _accept, int _stack, int _retn, int _error)
    {
        terminals = _terminals;
        jump = _jump;
        accept = _accept;
        stack = _stack;
        retn = _retn;
        error = _error;
    }

    bool isExist(const string& terminal)
    {
        for (size_t i = 0; i < terminals.size(); i++)
            if (terminals[i] == terminal)
                return true;
        return false;
    }
};

vector<parsingTableRow> parsingTable;
map<string, int> priority = { {"+", 2}, {"-", 2}, {"*", 3}, {"=", 0}, {"==", 1}, {"!=", 1 }, {"<", 1}, {"/", 3}, {"", 0} };

void readParseTable(const string& ParseTableFile)
{
    ifstream fin(ParseTableFile);
    string line, temp;

    while (getline(fin, line))
    {

```

```

        vector<string> str;
        vector<string> terminals;
        stringstream ss(line);

        while (ss >> temp)
            str.push_back(temp);

        int i = 0;
        for (i; i < str.size() - 5; i++)
            terminals.push_back(str[i]);

        parsingTableRow row(terminals, stoi(str[i]), stoi(str[i+1]), stoi(str[i+2]), stoi(str[i+3]), stoi(str[i+4]));
        parsingTable.push_back(row);
    }
    fin.close();
}

void postfix(ofstream& postfixFile, vector<token>& infix, LexicalAnalyzer& la)
{
    stack<string> op;
    string tmpStr, sCur;
    token tknCur;
    queue<string> postfix;
    for (int i = 0; i < infix.size(); i++)
    {
        tknCur = infix[i];
        if (tknCur.tableNum == 30)
            postfix.push(la.var_table.GetName(tknCur.index));
        else
            if (tknCur.tableNum == 40)
                postfix.push(la.const_table.GetName(tknCur.index));
            else
            {
                sCur = la.operators.GetRow(tknCur.index);
                if (sCur == "(")
                    op.push(sCur);
                else
                {
                    if (sCur == ")")
                    {
                        while (op.top() != "(")
                        {

```

```

        postfix.push(op.top());
        op.pop();
    }
    op.pop();
}
else
{
    if (op.empty() || op.top() == "(")
        op.push(sCur);
    else
    {
        int p_in = priority[sCur];
        int p_top = priority[op.top()];
        if (p_in > p_top) { op.push(sCur); }
        else
        {
            while ((op.top() != "(" || priority[op.top()] >= p_in))
            {
                postfix.push(op.top());
                op.pop();
                if (op.empty()) break;
            }
            op.push(sCur);
        }
    }
}
}
}
}

while (!op.empty())
{
    postfix.push(op.top());
    op.pop();
}

while (!postfix.empty())
{
    if (postfix.front() != ";")
        postfixFile << postfix.front() << " ";
    postfix.pop();
}

```

```

}

bool LL1(const string& tokenFile, const string& postfixFile, LexicalAnalyzer& la)
{
    ifstream ftoken(tokenFile);
    ofstream fpostfix(postfixFile);
    token tknCur, tknNext;
    stack<int> states, m1, m2;
    int currState = 0, index = 0, if_count = 0;
    string sCur, sNext;
    vector<token> infix;
    bool OPZ = false;

    if (ftoken.peek() == EOF) { return true; }

    sCur = tknCur.readToken(ftoken, la);

    do
    {
        if (parsingTable[currState].isExist(sCur))
        {
            if (parsingTable[currState].accept)
            {
                if (currState == 15 || currState == 48) //объявление или идентификатор в присваивании
                {
                    sCur = la.var_table.GetName(tknCur.index);
                    if (currState == 15) //идентификатор в присваивании
                    {
                        if (!la.var_table.GetIsSet(tknCur.index)) //если не задан тип переменной
                        {
                            cout << "Error: Unknown identifier '" << sCur << "'!";
                            return false;
                        }
                    }
                    else //объявление
                    {
                        if (la.var_table.GetIsSet(tknCur.index)) //если тип переменной уже задан
                        {
                            cout << "Error: redescrbing the type of a variable '" << sCur << "'!";
                            return false;
                        }
                        else { la.var_table.SetIsSet(tknCur.index, 1); }
                    }
                }
            }
        }
    }

```

```

    }
    if (ftoken.peek() != EOF)
    {
        sNext = tknNext.readToken(ftoken, la);

        if (sNext == "=") //идентификатор слева от =
        {
            la.var_table.SetValue(tknCur.index, 1);
            infix.push_back(tknCur);
            OPZ = true;
        }

        sCur = sNext;
        tknCur = tknNext;
        sNext = "";
    }
}
else
{
    if (currState == 25) // идентификатор в выражении
    {
        sCur = la.var_table.GetName(tknCur.index);
        if (!la.var_table.GetIsSet(tknCur.index)) //если не задан тип идентификатора
        {
            cout << "Error: Unknown identifier '" << sCur << "'!";
            return false;
        }
        if (!la.var_table.GetValue(tknCur.index)) //если не задано значение идентификатора
        {
            cout << "Error: Value of the variable '" << sCur << "' is not set!";
            return false;
        }
    }

    if (currState == 58) //объявление нескольких переменных
        if (OPZ)
        {
            postfix(fpostfix, infix, la);
            fpostfix << endl;
            OPZ = false;
            infix.clear();
        }
    }
}

```

```

//if
if (currState == 66) // ) в условии
{
    if (OPZ)
    {
        postfix(fpostfix, infix, la);
        OPZ = false;
        infix.clear();
        index++;
        m1.push(index);
        fpostfix << "m" << index << " C J F ";
    }
}

if (currState == 69) // } в if
{
    if (OPZ)
    {
        postfix(fpostfix, infix, la);
        OPZ = false;
        infix.clear();
    }
}

if (currState == 85) // } в else
{
    if (OPZ)
    {
        index++;
        m2.push(index);
        fpostfix << "m" << index << " U J ";
        fpostfix << "m" << m1.top() << ": ";
        m1.pop();

        postfix(fpostfix, infix, la);
        OPZ = false;
        infix.clear();
        fpostfix << "m" << m2.top() << ": " << endl;
        m2.pop();
        if_count--;
    }
}

```

```

    }
}

if (OPZ)      { infix.push_back(tknCur); }

if (ftoken.peek() != EOF)
{
    sNext = tknNext.readToken(ftoken, 1a);
    if (sNext == "==" || sNext == "!=" || sNext == "<") //идентификатор слева от == != <
    {
        infix.push_back(tknCur);
        OPZ = true;
        if_count++;
    }

    sCur = sNext;
    tknCur = tknNext;
    sNext = "";
}
}

if (currState == 81) // нет else
{
    fpostfix << "m" << m1.top() << ": " << endl;
    m1.pop();
    if_count--;
}

if (parsingTable[currState].stack)
    states.push(currState + 1);

if (parsingTable[currState].jump > 0)
    currState = parsingTable[currState].jump - 1;
else
{
    if (parsingTable[currState].retrn)
    {
        if (!states.empty())
        {
            currState = states.top();
            states.pop();
        }
    }
}

```

```

        if (currState == 18 || currState == 46) //;
        {
            if (OPZ && if_count == 0)
            {
                postfix(fpostfix, infix, la);
                fpostfix << endl;
                OPZ = false;
                infix.clear();
            }
        }
    }
else
{
    if (currState != 8)
    {
        cout << "Syntax error: Stack is empty!";
        return false;
    }
}
}

else //если символа нет в столбце terminal
{
    if (parsingTable[currState].error)
    {
        cout << "Error: Unexpected symbol! Possible symbols: ";
        for (size_t i = 0; i < parsingTable[currState].terminals.size(); i++)
            cout << " " << parsingTable[currState].terminals[i] << " ";
        return false;
    }
    else
        currState++;
}
} while (ftoken.peek() != EOF);

if (currState == 8 && la.operators.GetRow(tknCur.index) == "{")
    cout << "Success!";
else
{
    cout << "Error: Incorrect end of the program! Expected '}'";
    return false;
}
}

```



```

        fpostfix.close();
        ftoken.close();
        return true;
    }
};

```

Файл "main.cpp"

```

#include <iostream>
#include "SyntaxAnalyzer.h"

using namespace std;

int main()
{
    LexicalAnalyzer la = LexicalAnalyzer();

    la.MakeTokens("prog.txt", "tokens.txt");
    la.PrintAllTables("tables");

    SyntaxAnalyzer sa = SyntaxAnalyzer();
    sa.readParseTable("parsingTable.txt");
    sa.LL1("tokens.txt", "postfix.txt", la);

    la.PrintAllTables("tables");
}

```