Министерство образования и науки Российской Федерации
Новосибирский государственный технический университет
Кафедра прикладной математики

Языки программирования и методы трансляции

Лабораторная работа №2

Факультет	прикладной математики и информатики
Группа	ПМ-01
Студенты	Александров М.Е.
	Жигалов П.С.
Преподаватели	Еланцева И.Л.
	Полетаева И.А.
Вариант	7

Новосибирск
2013

1. Цель работы

Изучить методы лексического анализа. Получить представление о методах обработки лексических ошибок. Научиться проектировать сканер на основе детерминированных конечных автоматов.

2. Задание

Подмножество языка C++ включает:

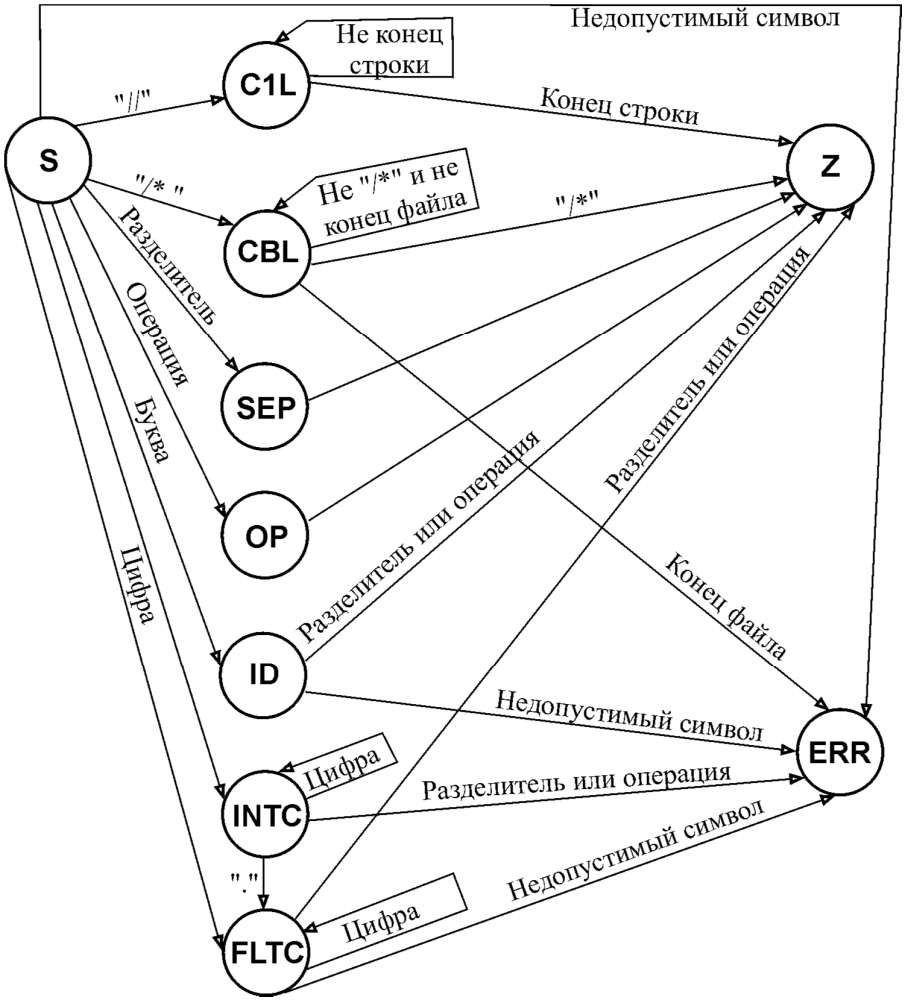
- данные типа **int, float, массивы** из элементов указанных типов;
- инструкции описания переменных;
- операторы присваивания в любой последовательности;
- операции **+, – , *, = =, != , <, >** .

В соответствии с выбранным вариантом задания к лабораторным работам разработать и реализовать лексический анализатор на основе детерминированных конечных автоматов. Исходными данными для сканера является программа на языке C++ и постоянные таблицы, реализованные в лабораторной работе №1. Результатом работы сканера является создание файла токенов, переменных таблиц (таблицы символов и таблицы констант) и файла сообщений об ошибках.

3. Структура входных и выходных данных

Входные данные представляют собой имена файлов: содержащего исходный код, файла токенов и файла ошибок, а также файлы, содержащие допустимые буквы, числа, операции, ключевые слова и разделители. Результатом работы программы являются два файла – файл токенов и файл ошибок.

4. Детерминированный конечный автомат



Описание состояний

S – начальное состояние
C1L – однострочный комментарий
CBL – блочный комментарий
SEP – разделитель
OP – операция
ID – идентификатор или ключевое слово
INTC – целочисленная константа
FLTC – вещественная константа
Z – конечное состояние
ERR – ошибка

5. Алгоритм разбора

1. Считать строку. Если конец файла – перейти к шагу 9.

2. Очистить строку от комментариев.

3. Анализировать первые два символа. Если первый символ – разделитель, перейти к шагу 7; если первый символ или первые два символа – операция, перейти к шагу 6; если первый символ – буква, перейти к шагу 4; если первый символ – цифра, перейти к шагу 5; если строка пуста – к шагу 1; иначе перейти к шагу 8.

4. Выделить идентификатор путем добавления к первому символу всех последующих букв и цифр. Если идентификатор – ключевое слово, сформировать и вывести соответствующий токен, иначе добавить идентификатор в таблицу идентификаторов и вывести соответствующий токен. За строку считать строку после идентификатора и перейти к шагу 3.

5. Выделить константу путем добавления к первому символу всех последующих цифр и/или одной точки. Если после константы нет разделителя или знака операции, перейти к шагу 8; иначе сформировать и вывести соответствующий токен для константы. За строку считать строку после константы и перейти к шагу 3.

6. Выделить одно- или двухсимвольную операцию, сформировать и вывести соответствующий токен. За строку считать строку после операции и перейти к шагу 3.

7. Выделить разделитель, сформировать и вывести соответствующий токен. За строку считать строку после разделителя и перейти к шагу 3.

8. Ошибка, вывести соответствующее сообщение и прекратить разбор.

9. Конец, успешный рабор.

6. Тесты

6.1 Верный исходный код с разнообразным форматированием и стилем комментариев:

Исходный код

```
/**
 * Doxygen-style
 * commentary
 */

int main()
{
    //34d;
    int a = 0; //!< it is a
    float b[2];
    b[  0  ]   =   2.0   ;
    b[1]=1.5;
    a--;
/* multi
string
comment
*/
    ;
/* */ /* */ a += b[1] + b[0] /* this is b1+b0
*/ * 2;
    a++;
/*  // */
/* //
```

Файл токенов

```
3 2 -1
3 3 -1
4 0 -1
4 1 -1
4 5 -1
3 2 -1
5 97 0
4 11 -1
6 48 -1
4 2 -1
3 1 -1
5 98 0
4 3 -1
6 48 -1
4 4 -1
4 2 -1
5 98 0
4 3 -1
6 48 -1
4 4 -1
4 11 -1
6 44 -1
4 2 -1
```

```
//
// */
    return 0;
}
```

Содержимое таблиц

```
ID`s:
97:     [ a     notype  dim=1   init={0} ]
98:     [ b     notype  dim=1   init={0} ]
CONST`s:
44:     [ 2.0   notype  dim=1   init={0} ]
48:     [ 0     notype  dim=1   init={0},
        1.5     notype  dim=1   init={0} ]
49:     [ 1     notype  dim=1   init={0} ]
50:     [ 2     notype  dim=1   init={0} ]
```

```
5 98 0
4 3 -1
6 48 -1
4 4 -1
4 11 -1
6 44 -1
4 2 -1
5 97 0
4 7 -1
4 2 -1
4 2 -1
5 97 0
4 5 -1
5 98 0
4 3 -1
6 49 -1
4 4 -1
4 3 -1
5 98 0
4 3 -1
6 48 -1
4 4 -1
4 1 -1
6 50 -1
4 2 -1
5 97 0
4 4 -1
4 2 -1
3 4 -1
6 48 -1
4 2 -1
4 6 -1
```

6.2 Незакрытый комментарий

Исходный код

```
int main()
{
/*
    return 0;
}
```

Файл ошибок

```
Error: incorrect coment
Error in string 3: /*
```

Файл токенов

```
3 2 -1
3 3 -1
4 0 -1
4 1 -1
4 5 -1
```

Содержимое таблиц

```
ID`s:
CONST`s:
```

6.3 Недопустимые символы в коде

Исходный код

```
int main()
{
    @#@#@#@#
    return 0;
}
```

Файл ошибок

```
Error: can`t determine symbol "@"
Error in string 3:     @#@#@#@#
```

Файл токенов

```
3 2 -1
3 3 -1
4 0 -1
4 1 -1
4 5 -1
```

Содержимое таблиц

```
ID`s:
CONST`s:
```

6.4 Некорректный идентификатор

Исходный код

```
int main()
{
    abc@d = 0;
    return 0;
}
```

Файл ошибок

```
Error: can`t determine symbol "@"
Error in string 3:     abc@d = 0;
```

Файл токенов

```
3 2 -1
3 3 -1
4 0 -1
4 1 -1
4 5 -1
5 94 0
```

Содержимое таблиц

```
ID`s:
94:     [ abc   notype  dim=1   init={0} ]
CONST`s:
```

6.5 Некорректная константа

Исходный код

```
int main()
{
    a = 12.0asbd;
    return 0;
}
```

Файл токенов

```
3 2 -1
3 3 -1
4 0 -1
4 1 -1
4 5 -1
5 97 0
4 11 -1
```

Файл ошибок

```
Error: incorrect constant
Error in string 3:     a = 12.0asbd;
```

Содержимое таблиц

```
ID`s:
97:    [ a    notype  dim=1   init={0} ]
CONST`s:
```

6.6 Две точки в константе

Исходный код

```
int main()
{
    a = 12.0.4;
    return 0;
}
```

Файл токенов

```
3 2 -1
3 3 -1
4 0 -1
4 1 -1
4 5 -1
5 97 0
4 11 -1
```

Файл ошибок

```
Error: incorrect constant
Error in string 3:     a = 12.0.4;
```

Содержимое таблиц

```
ID`s:
97:    [ a    notype  dim=1   init={0} ]
CONST`s:
```

Текст программы

translator.h

```
#ifndef TRANSLATOR_H_INCLUDED
#define TRANSLATOR_H_INCLUDED

#include <iostream>
#include <stdio.h>
#include <sstream>
#include <fstream>
#include <string>
#include "table_const.h"
#include "table_var.h"
#include "lexeme.h"
#include "token.h"

using namespace std;

class translator
{
private:
    // Постоянные таблицы
    table_const<char> letters;      // 0
    table_const<char> numbers;      // 1
    table_const<string> operations; // 2
    table_const<string> keywords;   // 3
    table_const<char> separators;   // 4
    // Переменные таблицы
    table_var identifiers;          // 5
    table_var constants;            // 6
    // Файловые потоки
    ifstream in_source;
    ofstream out_token;
    ofstream out_error;
    // Анализ строки
    bool analyze_lexical_string(string str);
    // Удаление комментариев
    bool analyze_lexical_decomment(string& str, bool is_changed);
    // Счетчики для подробных сообщений об ошибке
    int analyze_lexical_strnum, analyze_lexical_strinc;
    // Удаление пробелов
    static inline void ltrim(string& out_)
    {
        int notwhite = out_.find_first_not_of(" \t\n");
        out_.erase(0, notwhite);
    }
    static inline void rtrim(string& out_)
```

```
    {
        int notwhite = out_.find_last_not_of(" \t\n");
        out_.erase(notwhite + 1);
    }
    static inline void trim(string& out_)
    {
        ltrim(out_);
        rtrim(out_);
    }
public:
    // Конструктор со вводом постоянных таблиц
    translator();
    // Лексический анализ
    bool analyze_lexical(string file_source, string file_tokens, string file_error);
    // Отладочный вывод таблиц
    void debug_print(ostream& stream);
};

#endif // TRANSLATOR_H_INCLUDED
```

translator.cpp

```
#include "translator.h"

// Конструктор со вводом постоянных таблиц
translator::translator()
{
    letters.read_file("files/table_letters.txt");
    numbers.read_file("files/table_numbers.txt");
    operations.read_file("files/table_operations.txt");
    keywords.read_file("files/table_keywords.txt");
    separators.read_file("files/table_separators.txt");
}

// Лексический анализ
bool translator::analyze_lexical(string file_source, string file_tokens, string file_error)
{
    in_source.open(file_source.c_str(), ios::in);
    out_token.open(file_tokens.c_str(), ios::out);
    out_error.open(file_error.c_str(), ios::out);
    bool flag_error = false;
    bool flag_coment = false;
    string str;
    analyze_lexical_strnum = 1;
    while(!in_source.eof() && !flag_error)
    {
        getline(in_source, str);
        if(!in_source.eof())
        {
            analyze_lexical_strinc = 0;
            string strold = str;
            if(!analyze_lexical_decomment(str, true))
            {
                out_error << "Error in string " << analyze_lexical_strnum << ": " << strold << endl;
                cout << "Error in string " << analyze_lexical_strnum << ": " << strold << endl;
                return false;
            }
            analyze_lexical_strnum += analyze_lexical_strinc;
            flag_error = !analyze_lexical_string(str);
            if(flag_error)
            {
                out_error << "Error in string " << analyze_lexical_strnum << ": " << str << endl;
                cout << "Error in string " << analyze_lexical_strnum << ": " << str << endl;
            }
            analyze_lexical_strnum ++;
        }
    }
    in_source.close();
    out_token.close();
    out_error.close();
    return !flag_error;
}

// Очистка от комментариев
bool translator::analyze_lexical_decomment(string& str, bool is_changed)
{
    if(str.size())
    {
        bool change = false;
        size_t index_c = str.find("//"), index_c1 = str.find("/*"), index_c2;
        if (index_c != string::npos && index_c < index_c1)
        {
            str.erase(index_c);
```

```
            change = true;
        }
        index_c1 = str.find("/*");
        index_c2 = str.find("*/");
        if(index_c2 < index_c1)
        {
            out_error << "Error: incorrect coment" << endl;
            cout << "Error: incorrect coment" << endl;
            return false;
        }
        while(index_c1 != string::npos && index_c2 != string::npos)
        {
            string tmpstr = str;
            str.erase(index_c1);
            tmpstr.erase(0, index_c2 + 2);
            str += tmpstr;
            index_c1 = str.find("/*");
            index_c2 = str.find("*/");
            change = true;
        }
        index_c1 = str.find("/*");
        index_c2 = str.find("*/");
        if(index_c1 != string::npos && index_c2 == string::npos)
        {
            str.erase(index_c1);
            string tmpstr;
            if(!in_source.eof())
            {
                getline(in_source, tmpstr);
                analyze_lexical_strinc++;
            }
            else
            {
                out_error << "Error: incorrect coment" << endl;
                cout << "Error: incorrect coment" << endl;
                return false;
            }
            while(tmpstr.find("*/") == string::npos)
            {
                if(!in_source.eof())
                {
                    getline(in_source, tmpstr);
                    analyze_lexical_strinc++;
                }
                else
                {
                    out_error << "Error: incorrect coment" << endl;
                    cout << "Error: incorrect coment" << endl;
                    return false;
                }
            }
            index_c2 = tmpstr.find("*/");
            tmpstr.erase(0, index_c2 + 2);
            str += " " + tmpstr;
            change = true;
        }
        index_c1 = str.find("/*");
        index_c2 = str.find("*/");
        if(index_c1 != string::npos && index_c2 == string::npos ||
                index_c1 == string::npos && index_c2 != string::npos)
        {
            out_error << "Error: incorrect coment" << endl;
            cout << "Error: incorrect coment" << endl;
            return false;
        }
        if(is_changed)
            return analyze_lexical_decomment(str, change);
    }
    return true;
}

// Анализ строки
bool translator::analyze_lexical_string(string str)
{
    trim(str);
    bool flag_error = false;
    if(str.size())
    {
        char sym_1 = str[0], sym_2 = str[1];
        // Проверка первого символа
        string str_1, str_2;
        stringstream str_stream;
        str_stream << sym_1;
        str_1 = str_stream.str();
```

```
        str_stream << sym_2;
        str_2 = str_stream.str();
        int first_sym_type = -1;
        if(letters.contains(sym_1))
            first_sym_type = 0;
        if(numbers.contains(sym_1) || sym_1 == '-')
            first_sym_type = 1;
        if(operations.contains(str_1) || operations.contains(str_2))
            first_sym_type = 2;
        if(separators.contains(sym_1))
            first_sym_type = 3;

        switch(first_sym_type)
        {
        case 0: // Идентификатор
        {
            // Получим полное название идентификатора
            string idname = str;
            int i;
            bool finded = false;
            for(i = 1; i < idname.size() && !finded; i++)
                finded = !(letters.contains(str[i]) || numbers.contains(str[i]));
            if(finded)
            {
                idname.erase(i - 1);
                str.erase(0, i - 1);
            }
            else
                str.erase(0);
            trim(idname);
            trim(str);
            if(keywords.contains(idname)) // Если ключевое слово
            {
                if(keywords.get_num(idname, i))
                    out_token << token(3, i, -1);
            }
            else // Иначе в таблицу идентификаторов
            {
                identifiers.add(idname);
                int table, chain;
                identifiers.get_location(idname, table, chain);
                out_token << token(5, table, chain);
            }
            return analyze_lexical_string(str);
        }
        break;
        case 1:  // Константа
        {
            string constval = str;
            int i;
            bool finded = false;
            for(i = 1; i < constval.size() && !finded; i++)
                finded = !(numbers.contains(str[i]) || str[i] == '.' || str[i] == ' ');
            string str_t1, str_t2;
            stringstream str_stream_t;
            str_stream_t << str[i - 1];
            str_t1 = str_stream_t.str();
            str_stream_t << str[i];
            str_t2 = str_stream_t.str();
            if(!operations.contains(str_t1) && !operations.contains(str_t2) && !separa-
tors.contains(str[i - 1]))
            {
                out_error << "Error: incorrect constant" << endl;
                cout << "Error: incorrect constant" << endl;
                return false;
            }
            if(finded)
            {
                constval.erase(i - 1);
                str.erase(0, i - 1);
            }
            else
                str.erase(0);
            trim(constval);
            trim(str);
            if(constval.find_last_of('.') - constval.find_first_of('.') != 0)
            {
                out_error << "Error: incorrect constant" << endl;
                cout << "Error: incorrect constant" << endl;
                return false;
            }
            else
            {
                constants.add(constval);
```

```
                int table, chain;
                identifiers.get_location(constval, table, chain);
                out_token << token(6, table, chain);
            }
            return analyze_lexical_string(str);
        }
        break;
        case 2: // Операция
        {
            int table;
            if(operations.contains(str_2)) // Двухсимвольная
            {
                operations.get_num(str_2, table);
                out_token << token(4, table, -1);
                str.erase(0, 2);
                trim(str);
                return analyze_lexical_string(str);
            }
            if(operations.contains(str_1)) // Односимвольная
            {
                operations.get_num(str_1, table);
                out_token << token(4, table, -1);
                str.erase(0, 1);
                trim(str);
                return analyze_lexical_string(str);
            }
        }
        break;
        case 3: // Разделитель
        {
            int table;
            separators.get_num((const char)str[0], table);
            out_token << token(4, table, -1);
            str.erase(0, 1);
            trim(str);
            return analyze_lexical_string(str);
        }
        break;
        default: // Непонятно что
        {
            out_error << "Error: can`t determine symbol \"" << str_1 << "\"" << endl;
            cout << "Error: can`t determine symbol \"" << str_1 << "\"" << endl;
            return false;
        }
        break;
        }
    }
    return !flag_error;
}

// Отладочный вывод таблиц
void translator::debug_print(ostream& stream)
{
    stream << "ID`s:" << endl;
    identifiers.debug_print(stream);
    stream << "CONST`s:" << endl;
    constants.debug_print(stream);
}
```

token.h

```
#ifndef TOKEN_H_INCLUDED
#define TOKEN_H_INCLUDED
#include <iostream>
#include <fstream>

using namespace std;

// Класс токенов
class token
{
public:
    int table; // Номер таблицы
    int place; // Положение в таблице
    int chain; // Положение в цепочке
    // Конструкторы
    token();
    token(int table_, int place_, int chain_);
    // Ввод-вывод токенов
    friend istream& operator >> (istream& istream_, token& token_);
    friend ostream& operator << (ostream& ostream_, const token& token_);
};
```

```
#endif // TOKEN_H_INCLUDED
```

token.cpp

```
#include "token.h"

token::token() { }

token::token(int table_, int place_, int chain_)
{
    table = table_;
    place = place_;
    chain = chain_;
}

istream& operator >> (istream& istream_, token::token& token_)
{
    istream_ >> token_.table >> token_.place >> token_.chain;
    return istream_;
}

ostream& operator << (ostream& ostream_, const token::token& token_)
{
    ostream_ << token_.table << " " << token_.place << " " << token_.chain << endl;
    return ostream_;
}
```

main.cpp

```
#include <iostream>
#include <stdio.h>
#include "translator.h"

using namespace std;

int main()
{
    translator a;
    a.analyze_lexical("files/source.txt","files/tokens.txt","files/errors.txt");
    a.debug_print(cout);
    return 0;
}
```