**

*Институт транспорта и связи*

Лабораторная работа №2

По дисциплине «Проектирование компиляторов»

Тема: **Программирование синтаксического анализатора**

Студент: Anastasija Mežale

Группа: 4802BD

Рига

2021 г.

Содержание

**Задание** 2

**Теоретическое** **обоснование** 3

**Индивидуальный** **вариант** 4

**Грамматические** **правила** 5

**Алгоритмы** **грамматического** **разбора** 6

**STATE** 6

**VAR** 7

**EXP** 8

**TERM** 9

**Код** **программы** 10

**main**.**cpp** 10

**scanner**.**h** 11

**scanner**.**cpp** 12

**parser**.**h** 19

**parser**.**cpp** 20

**Результаты** **работы** **программы** 23

**Выводы** 26

Задание

1. Для заданного в индивидуальном варианте фрагмента программа на языке программирования PASCAL разработать грамматику для построения грамматического разбора с помощью рекурсивного спуска. Индивидуальные варианты фрагментов – номер индивидуального задания выбирается в соответствии со своим номером по списку групп.
2. В качестве процедуры SCAN использовать лексический анализатор, сконструированный в лабораторной работе №1.
3. Для разработанной в пункте 1 грамматики построить алгоритмы грамматического разбора с помощью рекурсивного спуска для каждого вида нетерминалов.
4. Написать и отладить процедуры рекурсивного спуска.
5. Спроектировать тест для проверки работоспособности синтаксического анализатора.
6. Подготовить отчет о проделанной работе.

Теоретическое обоснование

**Лексема** – это последовательность допустимых символов языка программирования, обрабатываемая компилятором. Лексемами являются идентификаторы, ключевые слова, разделители, литералы (числовые и строковые).

**Лексический анализ** – это процесс аналитического разбора последовательности символов на распознанные лексемы (токены). Лексический анализ используется в компиляторах. Грамматика языка задает набор лексем, которые могут встретиться в ходе лексического анализа. Программа, выполняющая лексический анализ, называется лексическим анализатором или лексическим сканером.

**Синтаксический анализ** – это процесс сопоставления естественного или формального языка с его грамматикой. Результатом синтаксического анализа обычно является синтаксическое дерево.

**Метод рекурсивного спуска** – это алгоритм синтаксического анализа, реализуемый путем взаимного вызова процедур, где каждая процедура соответствует одному из правил заданной грамматики. Суть метода заключается в последовательном поглощении токенов слева-направо.

Индивидуальный вариант

На основании программного фрагмента на языке Pascal, представленном в соответствии с индивидуальным вариантом №11 (рис. 1), необходимо составить грамматические правила и алгоритмы грамматического разбора, после чего разработать сам лексический анализатор.



Рис. 1. Программный фрагмент индивидуального варианта

Грамматические правила

В соответствии с программным фрагментом индивидуального варианта были описаны грамматические правила (рис. 2), необходимые для составления алгоритмов грамматического разбора.

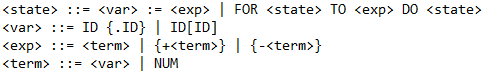


Рис. 2. Грамматические правила программного фрагмента

Алгоритмы грамматического разбора

STATE

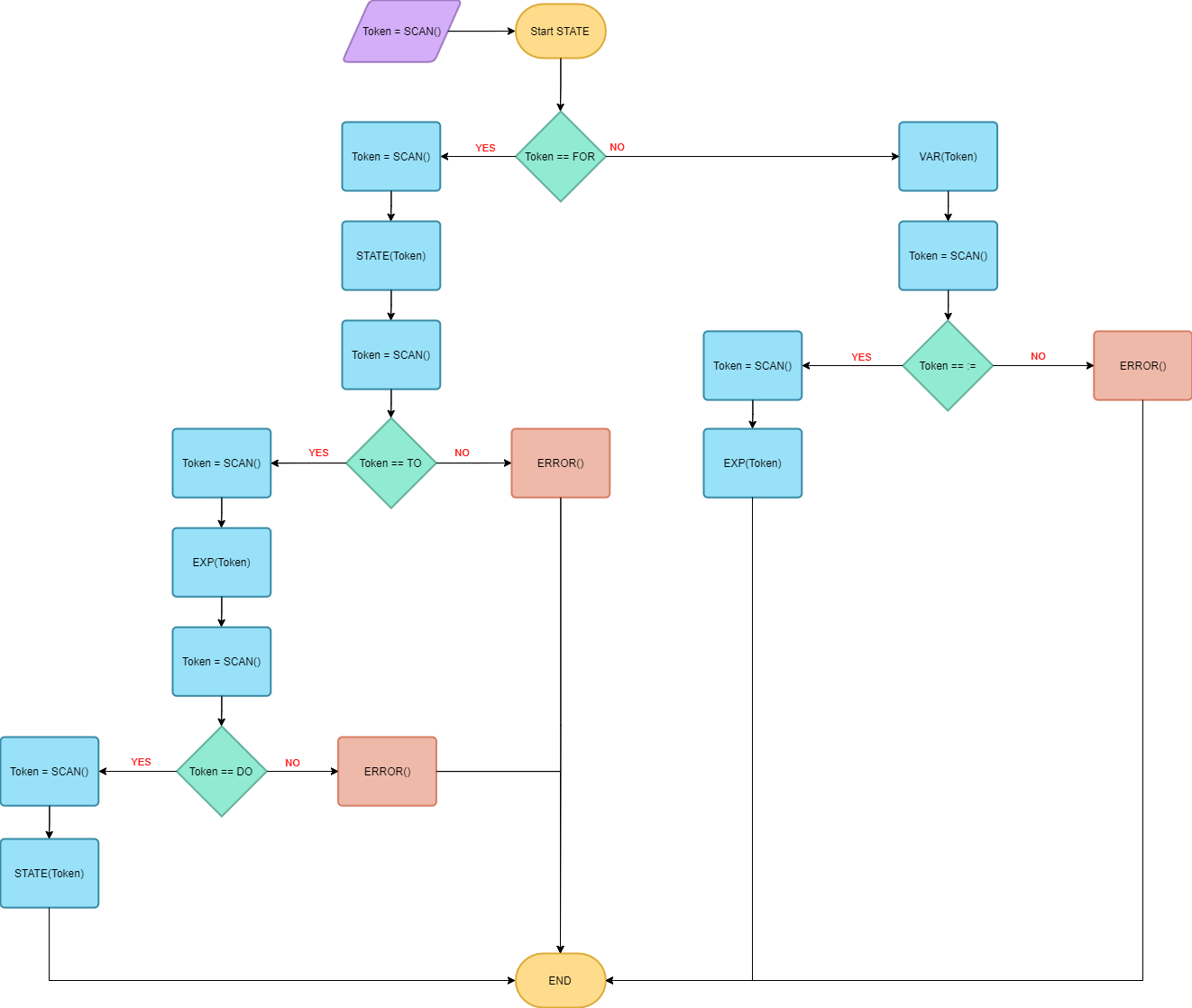


Рис. 3. Алгоритм процедуры STATE

VAR

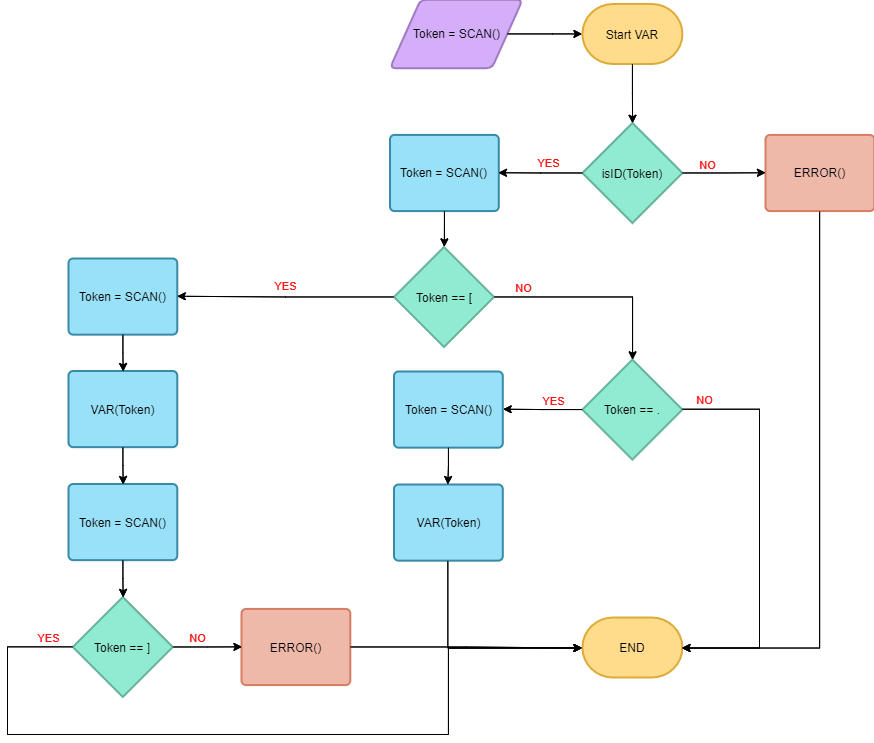


Рис. 4. Алгоритм процедуры VAR

EXP

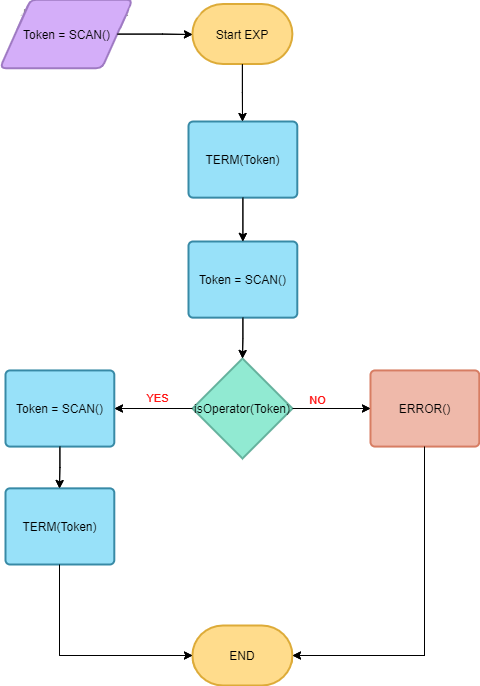


Рис. 5. Алгоритм процедуры EXP

TERM

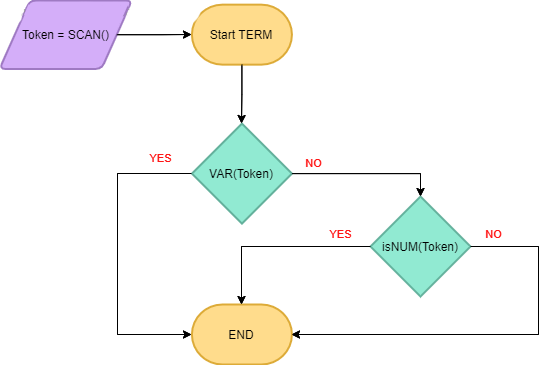


Рис. 6. Алгоритм процедуры TERM

Код программы

main.cpp

#include "scanner.h"

#include "parser.h"

void main()

{

cout << "Lexical scan...\n";

scan("code.txt");

cout << "Parsing... \n";

parser();

system("PAUSE");

}

scanner.h

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

#include <algorithm>

using namespace std;

void toLower(string token);

bool isID(const string &str);

bool isNumericConstant(const string &str);

bool isKeyword(const string &str);

bool isDelimeter(const string &str);

bool isOperator(const string &str);

bool isSpace(const string &str);

void role(const string &token);

void print();

void scan(const string &file);

string getToken(int counter);

scanner.cpp

#include "scanner.h"

const vector<string> keywords{ "for", "to", "do" };

const vector<string> delimeters{ ".", ";", "[", "]", "" };

const vector<string> operators{ ":=", "-", "+" };

const vector<string> numericConstants{ "1" };

vector<string> findedID;

vector<string> findedKeywords;

vector<string> findedDelimeters;

vector<string> findedOperators;

vector<string> findedNumericConstants;

vector<string> scannedTokens;

bool ex = false;

void toLower(string token)

{

for (int i = 0; i < token.size(); i++)

{

token[i] = tolower(token[i]); //регистр не важен

}

}

bool isID(const string &str)

{

bool flagLetter = false;

bool flagDigit = false;

try

{

if (isdigit(str[0]))

{

if (str.size() == 1)

return false;

else

{

flagDigit = true;

ex = true;

throw (str);

}

}

for (int i = 0; i < str.size(); i++)

{

if (ispunct(str[i])) //идентификатор не может содержать знаки пунктуации

{

if (str.size() == 1)

return false;

else

{

ex = true;

throw (str);

}

}

if (isalpha(str[i]))

flagLetter = true;

}

if (!flagLetter)

{

ex = true;

throw (str);

}

}

catch (const string)

{

if (ex)

{

ex = false;

cout << "\nError with '" << str << "'! An identifier cannot contain punctuation marks!" << endl;

}

if (flagLetter)

{

ex = false;

cout << "\nError with '" << str <<"'! An identifier must contain a letter!" << endl;

}

if (flagDigit)

{

ex = false;

cout << "\nError with '" << str << "'! An identifier cannot start with a number!" << endl;

}

return false;

}

return true;

}

bool isNumericConstant(const string &str)

{

bool flag = false; //переменная для переключения между состояниями

for (int i = 0; i < str.size(); i++)

{

if (!(isdigit(str[i]))) //число должно содержать только цифры

{

flag = true;

return false;

}

}

if (!flag)

for (auto numericConstant : numericConstants)

if (numericConstant == str)

{

flag = true;

return true;

}

try

{

if (!flag)

throw (str);

}

catch (const string)

{

ex = true;

cout << "\nError with '"<< str << "'! Number is not a registered numeric constant!" << endl;

return false;

}

}

bool isKeyword(const string &str)

{

toLower(str);

for (auto keyword : keywords)

if (keyword == str)

return true;

return false;

}

bool isDelimeter(const string &str)

{

bool flag = false;

for (int i = 0; i < str.size(); i++)

{

if (!(ispunct(str[i]))) //разделитель может содержать только знаки пунктуации

{

flag = true;

return false;

}

}

if ((!flag) && (str == ":" || str == "=")) //двоеточие и равно не встречаются в качестве однопозиционных разделителей, но являются частями двухпозиционных

{

flag = true;

return false;

}

if (!flag)

for (auto delimeter : delimeters)

if (delimeter == str)

{

flag = true;

return true;

}

else flag = true;

try

{

if (!flag)

throw (str);

}

catch (const string)

{

ex = true;

cout << "\nError with '" << str << "'! Delimeter is not a registered delimeter!" << endl;

return false;

}

return false;

}

bool isOperator(const string &str)

{

bool flag = false;

for (int i = 0; i < str.size(); i++)

{

if (!(ispunct(str[i]))) //оператор может содержать только знаки пунктуации

{

flag = true;

return false;

}

}

if (!flag)

for (auto oper : operators)

if (oper == str)

{

flag = true;

return true;

}

else flag = true;

try

{

if (!flag)

throw (str);

}

catch (const string)

{

ex = true;

cout << "\nError with '" << str << "'! Operator is not a registered operator!" << endl;

return false;

}

return false;

}

bool isSpace(const string &str)

{

if (str == " " || str == "\n")

return true;

return false;

}

void role(const string& str)

{

bool flag = false; //переменная для нахождения совпадений при переборе векторов

string bufferStr = str; //переменная для проверки регистра подаваемой строки

string bufferID; //переменная для проверки регистра хранящихся идентификаторов

if (isKeyword(str))

{

for (auto keyword : findedKeywords)

if (keyword == str)

flag = true;

if (!flag)

findedKeywords.push\_back(str); //если совпадение не было найдено, лексема записывается в вектор и флаг опускается

flag = false;

}

else if (isDelimeter(str))

{

for (auto delimeter : findedDelimeters)

if (delimeter == str)

flag = true;

if (!flag)

findedDelimeters.push\_back(str);

flag = false;

}

else if (isOperator(str))

{

for (auto oper : findedOperators)

if (oper == str)

flag = true;

if (!flag)

findedOperators.push\_back(str);

flag = false;

}

else if (isNumericConstant(str))

{

for (auto numericConstant : findedNumericConstants)

if (numericConstant == str)

flag = true;

if (!flag)

findedNumericConstants.push\_back(str);

flag = false;

}

else if (isID(str))

{

for (int i = 0; i < str.size(); i++)

bufferStr[i] = tolower(str[i]);

for (auto ID : findedID)

{

bufferID = ID;

for (int i = 0; i < ID.size(); i++)

bufferID[i] = tolower(ID[i]);

if (bufferID == bufferStr)

flag = true;

}

if (!flag)

findedID.push\_back(str);

flag = false;

}

else

{

cout << "Invalid token: " << str << endl;

ex = false;

}

}

void print()

{

//вывод таблиц распознанных лексем

cout << "\n==============\nIDENTIFIERS\n==============\n\n";

for (auto ID : findedID)

{

cout << ID << endl;

}

cout << "\n==============\nKEYWORDS\n==============\n\n";

for (auto keyword : findedKeywords)

{

cout << keyword << endl;

}

cout << "\n==============\nDELIMETERS\n==============\n\n";

for (auto delimeter : findedDelimeters)

{

cout << delimeter << endl;

}

cout << "\n==============\nOPERATORS\n==============\n\n";

for (auto oper : findedOperators)

{

cout << oper << endl;

}

cout << "\n==============\nNUMERIC CONSTANTS\n==============\n\n";

for (auto numericConstant : findedNumericConstants)

{

cout << numericConstant << endl;

}

}

void scan(const string &fileName)

{

char ch; //переменная для хранения символов

string buffer; //переменная для хранения последовательностей символов

fstream file(fileName, fstream::in);

if (!file.is\_open())

{

cout << "Error while opening the file\n";

exit(0);

}

else

{

while (file >> noskipws >> ch) //чтение символов из файла с учетом начальных пробельных символов

{

if (isSpace(string(1, ch))) //1 - размер строки, ch - символ для заполнения размера. В буфер добавляется по одному символу

{

if (!buffer.empty()) //если буфер не пуст, проверяет содержимое буфера на совпадение с существующими лексемами, после чего буфер обнуляется

{

role(buffer); //определение типа лексемы

scannedTokens.push\_back(buffer);

buffer = ""; //обнуление буфера

}

continue;

}

if (isDelimeter(string(1, ch)) || isDelimeter(buffer)) //если входящий символ или содержимое буфера является разделителем

{

if (isDelimeter(string(1, ch)) && !isDelimeter(buffer)) //если входящий символ является разделителем, а содержимое буфера - нет, то буфер проверяется на совпадение с лексемами и обнуляется

{

if (!buffer.empty())

{

role(buffer);

scannedTokens.push\_back(buffer);

buffer = "";

}

role(string(1, ch));

scannedTokens.push\_back(string(1, ch));

continue;

}

else if (!isDelimeter(string(1, ch)) && isDelimeter(buffer))

{

role(buffer);

buffer = "";

}

else

{

if (!buffer.empty())

{

role(buffer);

scannedTokens.push\_back(buffer);

buffer = "";

}

else

{

role(string(1, ch));

scannedTokens.push\_back(string(1, ch));

}

}

}

buffer += ch; //формирование буфера, посимвольное добавление

}

if (string(1, ch) != ";")

{

role(buffer);

scannedTokens.push\_back(buffer);

buffer = "";

}

//print(); //вывод таблиц распознанных лексем

cout << "Lexical scan is complete." << endl;

}

file.close();

}

string getToken(int counter)

{

return scannedTokens[counter];

}

parser.h

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

#include <algorithm>

using namespace std;

bool state(string currentToken);

bool var(string token);

bool exp(string token);

bool term(string token);

void parser();

parser.cpp

#include "scanner.h"

#include "parser.h"

int counter = 0;

bool isState = false;

string msg;

bool exc = false;

bool state(string currentToken)

{

toLower(currentToken);

try

{

if (currentToken == "for")

{

isState = true;

counter++;

currentToken = getToken(counter);

state(currentToken);

currentToken = getToken(counter);

toLower(currentToken);

if (currentToken == "to")

{

counter++;

currentToken = getToken(counter);

exp(currentToken);

currentToken = getToken(counter);

toLower(currentToken);

if (currentToken == "do")

{

counter++;

currentToken = getToken(counter);

state(currentToken);

isState = false;

return true;

}

else

{

msg = "Error! Missing 'do'!\n";

throw (msg);

}

}

else

{

msg = "Error! Mssing 'to'!\n";

throw (msg);

}

}

else if (isState)

{

var(currentToken);

currentToken = getToken(counter);

if (currentToken == ":=")

{

counter++;

currentToken = getToken(counter);

exp(currentToken);

//return true;

}

else

{

msg = "Error! Missing ':='!\n";

throw (msg);

}

}

else

{

msg = "Error! Missing 'for'!\n";

throw (msg);

}

}

catch (string)

{

cout << msg;

exc = true;

return false;

}

}

bool var(string token)

{

toLower(token);

try

{

if (isID(token))

{

counter++;

token = getToken(counter);

if (token == "[")

{

counter++;

token = getToken(counter);

var(token);

token = getToken(counter);

if (token == "]")

{

return true;

}

else

{

msg = "Error! Missing ']'!\n";

throw (msg);

}

}

else if (token == ".")

{

counter++;

token = getToken(counter);

var(token);

return true;

}

else return true;

}

else if (isNumericConstant(token))

return false;

else

{

msg = "Error! Missing identifier!\n";

throw (msg);

}

}

catch (string)

{

cout << msg;

exc = true;

return false;

}

}

bool exp(string token)

{

try

{

term(token);

token = getToken(counter);

if (isOperator(token))

{

counter++;

token = getToken(counter);

term(token);

return true;

}

else if (token == "to" || token == "do")

return false;

else

{

msg = "Error! Missing operator!\n";

throw (msg);

}

}

catch (string)

{

cout << msg;

exc = true;

return false;

}

}

bool term(string token)

{

if (var(token))

return true;

else if (isNumericConstant(token))

{

counter++;

return true;

}

else return false;

}

void parser()

{

string currentToken = getToken(counter);

state(currentToken);

if (!exc)

cout << "Parsing is successful.\n";

else cout << "Error with parsing.\n";

}

Результаты работы программы

Представленный выше код программы синтаксического анализатора реализован на языке C++ методом рекурсивного спуска.

Ниже представлена работа программы при отсутствии лексических и синтаксических ошибок (исходный программный фрагмент, представленный в индивидуальном задании):

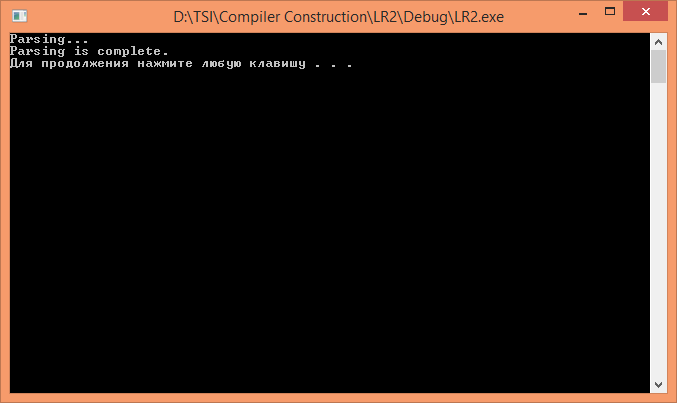


Рис. 7. Результат синтаксического анализа программного фрагмента, заданного в индивидуальном варианте

Если внести изменения в исходный программный код, например, убрать вторую квадратную скобку, программа выдаст исключение (рис. 8 и 9).

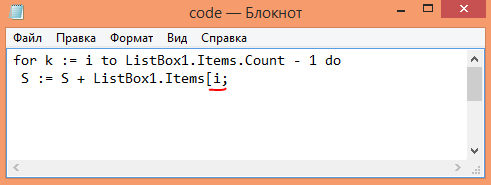


Рис. 8. Измененный программный фрагмент

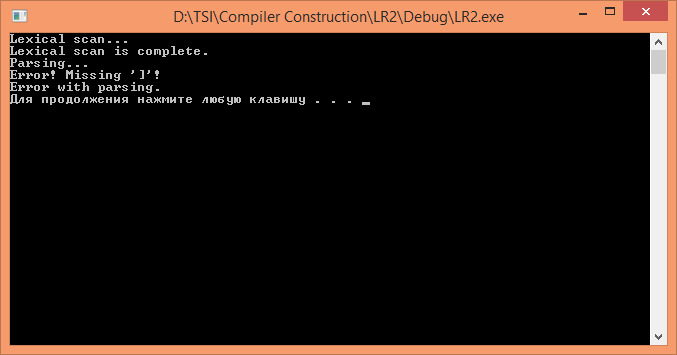


Рис. 9. Обработка ошибки

Также, если заменить ключевое слово ‘for’ на ‘for1’, программа сообщит об ошибке (рис. 10 и 11).

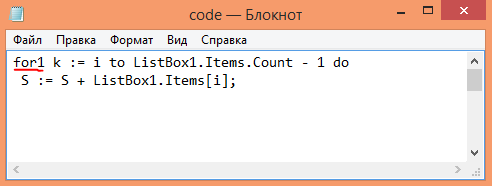


Рис. 10. Измененный программный фрагмент

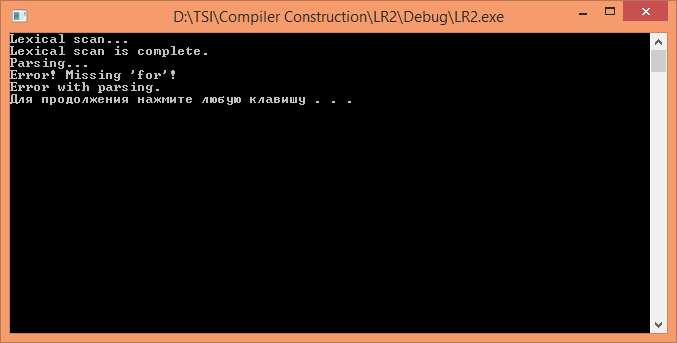


Рис. 11. Обработка синтаксической ошибки

Выводы

В ходе выполнения лабораторной работы были усвоены теоретические и практические знания по работе синтаксических анализаторов, написания правил регулярной грамматики и разработке алгоритмов. Конечный код является реализацией метода рекурсивного спуска, при котором каждому грамматическому правилу соответствует отдельная процедура. Процедуры взаимно вызываются, а считывание токенов происходит путем записи сканированных лексическим анализатором токенов в отдельный строковый вектор. Лексический и синтаксический анализаторы разделены на отдельные .cpp файлы для улучшения структурного содержания кода и упрощения его чтения.

Наибольшую сложность составила реализация процедур exp и var, так как от их работы зависит корректное объединение большинства лексем в определенные правила.

Реализованная программа работает в соответствии с поставленными перед ней требованиями и выполняет поставленную перед ней задачу, проводя синтаксический анализ программного фрагмента на языке Pascal на основании заданных грамматических правил.