

# СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc184921538)

[1 ПОСТАНОВКА ЗАДАЧИ 5](#_Toc184921539)

[2 ПОРЯДОК ВЫПОЛНЕНИЯ 7](#_Toc184921540)

[3 ГРАММАТИКА МОДЕЛЬНОГО ЯЗЫКА 8](#_Toc184921541)

[4 РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА 10](#_Toc184921542)

[5 РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА 12](#_Toc184921543)

[6 СЕМАНТИЧЕСКИЙ АНАЛИЗ 13](#_Toc184921544)

[7 ТЕСТИРОВАНИЕ ПРОГРАММЫ 14](#_Toc184921545)

[ЗАКЛЮЧЕНИЕ 16](#_Toc184921546)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 17](#_Toc184921547)

[ПРИЛОЖЕНИЯ 18](#_Toc184921548)

# ВВЕДЕНИЕ

Несмотря на более чем полувековую историю вычислительной техники, рождение теории формальных языков ведет отсчет с 1957 года. В этот год американский ученый Джон Бэкус разработал первый компилятор языка Фортран. Он применил теорию формальных языков, во многом опирающуюся на работы известного ученого-лингвиста Н. Хомского – автора классификации формальных языков. Хомский в основном занимался изучением естественных языков, Бекус применил его теорию для разработки языка программирования. Это дало толчок к разработке сотен языков программирования.

Несмотря на наличие большого количества алгоритмов, позволяющих автоматизировать процесс написания транслятора для формального языка, создание нового языка требует творческого подхода. В основном это относится к синтаксису языка, который, с одной стороны, должен быть удобен в прикладном программировании, а с другой, должен укладываться в область контекстно-свободных языков, для которых существуют развитые методы анализа.

Основы теории формальных языков и практические методы разработки распознавателей формальных языков составляют неотъемлемую часть образования современного инженера-программиста.

**Целью** данной курсовой работы является:

* освоение основных методов разработки распознавателей формальных языков на примере модельного языка программирования;
* приобретение практических навыков написания транслятора языка программирования;

закрепление практических навыков самостоятельного решения инженерных задач, умения пользоваться справочной литературой и технической документацией.

**1 ПОСТАНОВКА ЗАДАЧИ**

Разработать распознаватель модельного языка программирования согласно заданной формальной грамматике.

Распознаватель представляет собой специальный алгоритм, позволяющий вынести решение и принадлежности цепочки символов некоторому языку.

Распознаватель можно схематично представить в виде совокупности входной ленты, читающей головки, которая указывает на очередной символ на ленте, устройства управления (УУ) и дополнительной памяти (стек).

Конфигурацией распознавателя является:

* состояние УУ;
* содержимое входной ленты;
* положение читающей головки;
* содержимое дополнительной памяти (стека).

Трансляция исходного текста программы происходит в несколько этапов. Основными этапами являются следующие:

* лексический анализ;
* синтаксический анализ;
* семантический анализ;
* генерация целевого кода.

Лексический анализ является наиболее простой фазой и выполняется с помощью *регулярной* грамматики. Регулярным грамматикам соответствуют конечные автоматы, следовательно, разработка и написание программы лексического анализатора эквивалентна разработке конечного автомата и его диаграммы состояний (ДС).

Синтаксический анализатор строится на базе *контекстно-свободных* (КС) грамматик. Задача синтаксического анализатора – провести разбор текста программы и сопоставить его с формальным описание языка.

Семантический анализ позволяет учесть особенности языка программирования, которые не могут быть описаны правилами КС- грамматики. К таким особенностям относятся:

* обработка описаний;
* анализ выражений;
* проверка правильности операторов.

Обработки описаний позволяет убедиться в том, что каждая переменная в программе описана и только один раз.

Анализ выражений заключается в том, чтобы проверить описаны ли переменные, участвующие в выражении, и соответствуют ли типы операндов друг другу и типу операции.

Этапы синтаксического и семантического анализа обычно можно объединить.

# 2 ПОРЯДОК ВЫПОЛНЕНИЯ

1. В соответствии с номером варианта составить описание модельного языка программирования в виде правил вывода формальной грамматики;
2. Составить таблицу лексем и нарисовать диаграмму состояний для распознавания и формирования лексем языка;
3. Разработать процедуру лексического анализа исходного текста программы на языке высокого уровня;
4. Разработать процедуру синтаксического анализа исходного текста методом рекурсивного спуска на языке высокого уровня;
5. Построить программный продукт, читающий текст программы, написанной на модельном языке, в виде консольного приложения;
6. Протестировать работу программного продукта с помощи серии тестов, демонстрирующих все основные особенности модельного языка программирования, включая возможные лексические и синтаксические ошибки.

# 3 ГРАММАТИКА МОДЕЛЬНОГО ЯЗЫКА

Согласно индивидуальному варианту задания на курсовую работу грамматика языка включает следующие синтаксические конструкции:

<операции\_группы\_отношения> ::= != | == | < | <= | > | >=

<операции\_группы\_сложения> ::= + | - | ||

<операции\_группы\_умножения> ::= \* | / | &&

<унарная\_операция> ::= !

<программа> ::= «{» {/ (<описание> | <оператор>) ; /} «}»

<описание> ::= **dim** <идентификатор> {, <идентификатор> } <тип>

<тип> ::= **integer** | **real** | **boolean**

<оператор> ::= <составной> | <присваивания> | <условный> | <фиксированного\_цикла> | <условного\_цикла> | <ввода> | <вывода>

<составной> ::= **begin** <оператор> { ; <оператор> } **end**

<присваивания> ::= <идентификатор> := <выражение>

<условный> ::= **if** «(»<выражение> «)» <оператор> [**else** <оператор>]

<фиксированного\_цикла> ::= **for** <присваивания> **to** <выражение> [**step** <выражение>] <оператор> **next**

<условного\_цикла> ::= **while** «(»<выражение> «)» <оператор>

<ввода> ::= **readln** идентификатор {, <идентификатор> }

<вывода> ::= **writeln** <выражение> {, <выражение> }

<выражение> ::= <операнд> {<операции\_группы\_отношения>

<операнд>}

<операнд> ::= <слагаемое> {<операции\_группы\_сложения>

<слагаемое>}

<слагаемое> ::= <множитель> {<операции\_группы\_умножения>

<множитель>}

<множитель> ::= <идентификатор> | <число> |

<логическая\_константа> | <унарная\_операция> <множитель> |

**(**<выражение>**)**

<логическая\_константа> ::= ***true***| ***false***

<идентификатор> ::= <буква>{<буква> | <цифра>}

<число> ::= <цифра>{<цифра>}

<буква> ::= 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 | 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

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Здесь для записи правил грамматики используется форма Бэкуса-Наура (БНФ). В записи БНФ левая и правая части порождения разделяются символом “::=”, нетерминалы заключены в угловые скобки, а терминалы – просто символы, используемые в языке. Жирным выделены терминалы, представляющие собой ключевые слова языка.

**4 РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА**

Лексический анализатор – подпрограмма, которая принимает на вход исходный текст программы и выдает последовательность *лексем* – минимальных элементов программы, несущих смысловую нагрузку.

В модельном языке программирования выделяют следующие типы лексем:

* ключевые слова;
* ограничители;
* числа;
* идентификаторы.

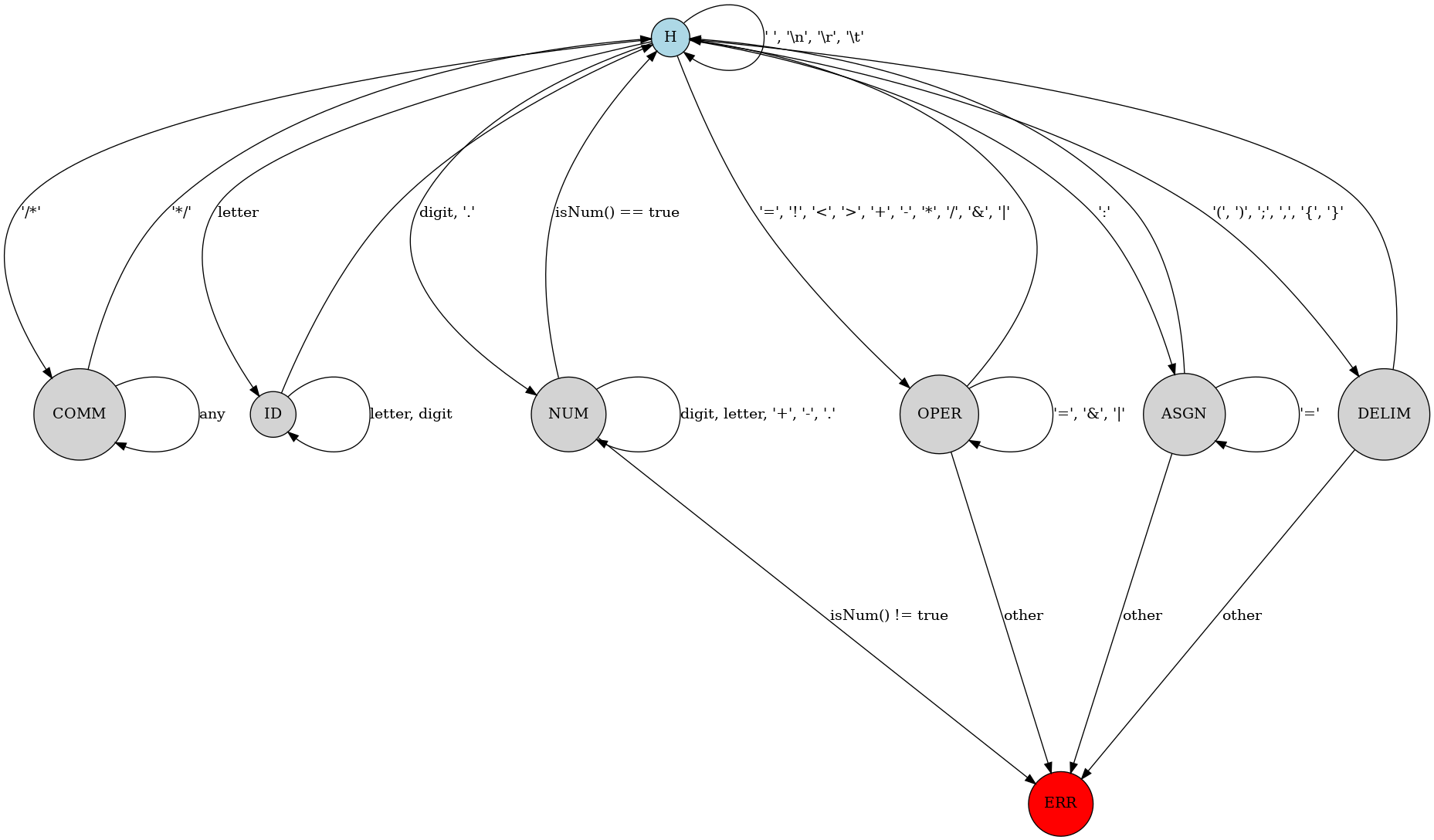
При разработке лексического анализатора, ключевые слова и ограничителя известны заранее, идентификаторы и числовые константы – вычисляются в момент разбора исходного текста.

Для каждого типа лексем предусмотрена отдельная таблица. Таким образом, внутреннее представление лексемы – пара чисел (*n*, *k*), где *n* – номер таблицы лексем, *k* – номер лексемы в таблице.

Кроме того, в исходном коде программы кроме ключевых слов, идентификаторов и числовых констант может находиться произвольное число пробельных символов («пробел», «табуляция», «перенос строки», «возврат каретки») и комментариев, заключенных в фигурные скобки.

Лексический анализ текста проводится по регулярной грамматике. Известно, что регулярная грамматика эквивалентна конченому автомату, следовательно, для написания лексического анализатора необходимо построить диаграмму состояний, соответствующего конечного автомата (рис. 1).

Исходные код лексического анализатора приведен в Приложении А.



**Рисунок 1 – Диаграмма состояний лексического анализатор**

# 5 РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

Будем считать, что лексический и синтаксической анализаторы взаимодействуют следующим образом. Если синтаксическому анализатору для анализа требуется очередная лексема, он запрашивает ее у лексического анализатора. Таким образом, разбор исходного текста программы идет под управлением подпрограммы синтаксического анализатора (*parser*).

Разработку синтаксического анализатора проведем с помощью метода *рекурсивного спуска* (РС). В основе метода лежит тот факт, что каждому нетерминалу ставится в соответствие рекурсивная функция. Для того, чтобы в явном виде представить множество рекурсивных функций, перепишем грамматические правила следующим образом:

P → «{» D1 | S «}» ⊥

D1 → dim D

D → I {, I} integer | real | boolean

B → begin S {; S} end

S → I := E | if (E) S [else S] | while (E) S | for I := E to E [step E] S next | readln I {, I} | writeln I {, I}

E → E1{[ != | = = | < | <= | > | >= ] E1}

E1 → T{[ + | − | || ] T}

T → F{[ \* | / | && ] F}

F → I | N | L | != F | (E)

𝐿 → *true | false*

𝐼 → 𝐶 { 𝐶 | 𝑅 }

𝑁 → 𝑅 { 𝑁 | 𝑅 }

𝐶 → 𝑎 | 𝑏 | … | 𝑧 | 𝐴 | 𝐵 | … | 𝑍

𝑅 → 0 | 1 | … | 9

Здесь правила для нетерминалов *L*, *I*, *N*, *C* и *R* описаны на этапе лексического разбора. Следовательно, остается описать функции для нетерминалов *P*, *D1*, *D*, *B*, *S*, *E*, *E1*, *T*, *F*.

Исходный код синтаксического анализатора приведен в Приложении Б.

# 6 СЕМАНТИЧЕСКИЙ АНАЛИЗ

Некоторые особенности модельного языка не могут быть описаны контекстно-свободной грамматикой. К таким правилам относятся:

* + любой идентификатор, используемый в теле программы должен быть описан;
  + повторное описание одного и того же идентификатора не разрешается;
  + в операторе присваивания типы идентификаторов должны совпадать;
  + в условном операторе и операторе цикла в качестве условия допустимы только логические выражения;
  + операнды операций отношения должны быть целочисленными.

Указанные особенности языка разбираются на этапе *семантического анализа*. Удобно процедуры семантического анализа совместить с процедурами синтаксического анализа. На практике это означает, что в рекурсивные функции встраиваются дополнительные контекстно-зависимые проверки. Например, на этапе лексического анализа в структуру *Lex* заносятся данные обо всех лексемах- идентификаторах, которые встречаются в тексте программы. На этапе синтаксического анализа в структуру Identifier заносятся данные о типе идентификатора (поле *type*) и его имени (поле name).

Описания функций семантических проверок приведены в листинге в Приложении Б.

# 7 ТЕСТИРОВАНИЕ ПРОГРАММЫ

В качестве программного продукта разработано консольное приложение *parser.exe*. Приложение принимает на вход исходный текст программы на модельном языке и выдает в качестве результата сообщение о синтаксической и семантической корректности написанной программы. В случае обнаружения ошибки программа выдает сообщение об ошибке с некорректной лексемой. Рассмотрим примеры.

1. Исходный код программы приведен в листинге 1.

*Листинг 1 – Тестовая программа*

{

dim a, b integer

a := 10

b := a + 5

if (a < b)

writeln a

else

writeln b

}

writeln b

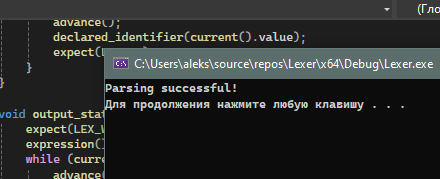
}

while x < y do x as x + 1

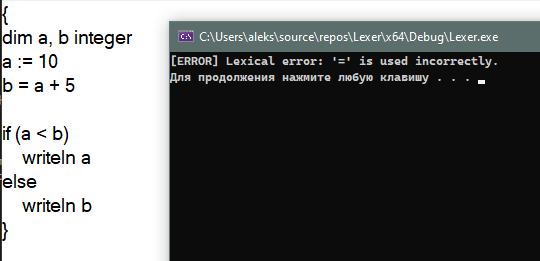
read(x, y)

write(x, y)

end

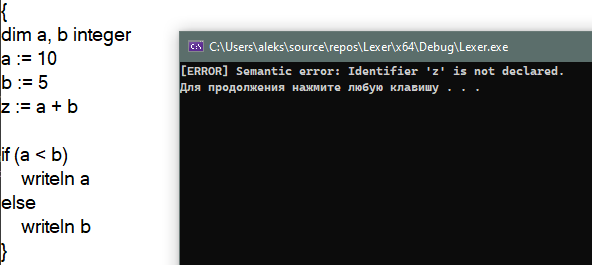
Данная программа синтаксически корректна, поэтому анализатор выдает следующее сообщение (рис. 2).

**Рисунок 2 – Пример синтаксически корректной программы**

1. Исходный код программы, содержащий синтаксическую ошибку, приведен на рис. 3 совместно с сообщением об ошибке.

**Рисунок 3 – Пример программы, содержащей ошибку**

Здесь ошибка допущена в строке 3: неправильное использование оператора присвоения (:=). В сообщении об ошибке указана ошибочная лексема.

1. Исходный текст программы, содержащей семантическую проверку, приведен на рис. 4 вместе с сообщением об ошибке. Здесь к необъявленной переменной (z) был использован оператор присваивания. (as).

**Рисунок 4 – Пример программы, содержащей семантическую ошибку**

# ЗАКЛЮЧЕНИЕ

В работе представлены результаты разработки анализатора языка программирования. Грамматика языка задана с помощью правил вывода и описана в форме Бэкуса-Наура (БНФ). Согласно грамматике, в языке присутствуют лексемы следующих базовых типов: числовые константы, переменные, разделители и ключевые слова.

Разработан лексический анализатор, позволяющий разделить последовательность символов исходного текста программы на последовательность лексем. Лексический анализатор реализован на языке среднего уровня C++ в виде класса *Lexer*.

Разбором исходного текста программы занимается синтаксический анализатор, который реализован в виде класса *Parser* на языке C++. Анализатор распознает входной язык по методу рекурсивного спуска. Для применимости необходимо было преобразовать грамматику, в частности, специальным образом обрабатывать встречающиеся итеративные синтаксически конструкции (нетерминалы *D*, *D1*, *B*, *E1* и *T*).

В код рекурсивных функций включены проверки дополнительных семантических условий, в частности, проверка на повторное объявление одной и той же переменной.

Тестирование программного продукта показало, что синтаксически и семантически корректно написанная программа успешно распознается анализатором, а программа, содержащая ошибки, отвергается.

В ходе работы изучены основные принципы построения интеллектуальных систем на основе теории автоматов и формальных грамматик, приобретены навыки лексического, синтаксического и семантического анализа предложений языков программирования.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Свердлов С. З. Языки программирования и методы трансляции: учебное пособие. – Санкт-Петербург: Лань, 2019.
2. Малявко А. А. Формальные языки и компиляторы: учебное пособие для вузов. – М.: Юрайт, 2020.
3. Миронов С. В. Формальные языки и грамматики: учебное пособие для студентов факультета компьютерных наук и информационных технологий. – Саратов: СГУ, 2019.
4. Унгер А.Ю. Основы теории трансляции: учебник. – М.: МИРЭА – Российский технологический университет, 2022.
5. Антик М. И., Казанцева Л. В. Теория формальных языков в проектировании трансляторов: учебное пособие. – М.: МИРЭА, 2020.
6. Ахо А. В., Лам М. С., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструментарий. – М.: Вильямс, 2008.
7. Ишакова Е.Н. Теория языков программирования и методов трансляции: учебное пособие. – Оренбург: ИПК ГОУ ОГУ, 2007.

# ПРИЛОЖЕНИЯ

Приложение А – Класс лексического анализатора

Приложение Б – Класс синтаксического анализатора

## **Приложение А**

Класс лексического анализатора

*Листинг А.1 – класс Lexer*

class Lexer {

private:

ifstream file;

char current\_char;

unordered\_map<string, lex\_type> keywords = {

{"dim", LEX\_DIM}, {"integer", LEX\_INTEGER}, {"real", LEX\_REAL}, {"boolean", LEX\_BOOLEAN},

{"if", LEX\_IF}, {"else", LEX\_ELSE}, {"for", LEX\_FOR}, {"to", LEX\_TO}, {"step", LEX\_STEP},

{"next", LEX\_NEXT}, {"while", LEX\_WHILE}, {"readln", LEX\_READLN}, {"writeln", LEX\_WRITELN},

{"true", LEX\_TRUE}, {"false", LEX\_FALSE}, {"begin", LEX\_BEGIN}, {"end", LEX\_END}

};

void advance() { current\_char = file.get(); }

void skip\_whitespace() { while (isspace(current\_char)) advance(); }

void skip\_comment() {

advance(); advance();

while (current\_char != EOF) {

if (current\_char == '\*' && file.peek() == '/') {

advance(); advance();

return;

}

advance();

}

}

Lex parse\_identifier() {

string result;

while (isalnum(current\_char)) {

result += current\_char;

advance();

}

return keywords.count(result) ? Lex(keywords[result], result) : Lex(LEX\_ID, result);

}

Lex parse\_number() {

string result;

while (isalnum(current\_char) || current\_char == '.' || current\_char == '+' || current\_char == '-') {

result += current\_char;

advance();

}

if (isNum(result)) {

if (result.find(".") != -1) {

return Lex(LEX\_RNUM, result);

}

else {

return Lex(LEX\_NUM, result);

}

}

else {

throw runtime\_error("Lexical error: Invalid number '" + result + "'.");

}

}

Lex parse\_operator() {

string result(1, current\_char);

advance();

if (result == ":") {

if (current\_char == '=') {

result += current\_char;

advance();

return Lex(LEX\_ASSIGN, result);

}

else {

throw runtime\_error("Lexical error: ':' is used incorrectly.");

}

}

if (result == "=") {

if (current\_char == '=') {

result += current\_char;

advance();

return Lex(LEX\_EQ, result);

}

else {

throw runtime\_error("Lexical error: '=' is used incorrectly.");

}

}

if (result == "!") {

if (current\_char == '=') {

result += current\_char;

advance();

return Lex(LEX\_NEQ, result);

}

return Lex(LEX\_NOT, result);

}

if (result == "<") {

if (current\_char == '=') {

result += current\_char;

advance();

return Lex(LEX\_LEQ, result);

}

return Lex(LEX\_LT, result);

}

if (result == ">") {

if (current\_char == '=') {

result += current\_char;

advance();

return Lex(LEX\_GEQ, result);

}

return Lex(LEX\_GT, result);

}

if (result == "|") {

if (current\_char == '|') {

result += current\_char;

advance();

return Lex(LEX\_OR, result);

}

}

if (result == "&") {

if (current\_char == '&') {

result += current\_char;

advance();

return Lex(LEX\_AND, result);

}

}

if (result == "+") return Lex(LEX\_PLUS, result);

if (result == "-") return Lex(LEX\_MINUS, result);

if (result == "\*") return Lex(LEX\_TIMES, result);

if (result == "/") return Lex(LEX\_DIV, result);

throw runtime\_error("Lexical error: Unknown operator: '" + result + "'.");

}

Lex parse\_delimiter() {

string result(1, current\_char);

advance();

if (result == "(" || result == ")" || result == ";" || result == "{" || result == "}" || result == ",") {

if (result == "(") return Lex(LEX\_LPAREN, result);

if (result == ")") return Lex(LEX\_RPAREN, result);

if (result == ";") return Lex(LEX\_SEMICOLON, result);

if (result == ",") return Lex(LEX\_COMMA, result);

if (result == "{") return Lex(LEX\_LBRACE, result);

if (result == "}") return Lex(LEX\_RBRACE, result);

}

else {

throw runtime\_error("Lexical error: Unknown symbol: '" + result + "'.");

}

}

public:

explicit Lexer(const string& filename) {

file.open(filename);

if (!file.is\_open()) {

throw runtime\_error("File not found or cannot be opened.");

}

advance();

}

vector<Lex> get\_lex\_table() {

vector<Lex> lex\_table;

Lex token;

while ((token = get\_next\_lex()).type != LEX\_FIN) {

lex\_table.push\_back(token);

}

return lex\_table;

}

Lex get\_next\_lex() {

while (current\_char != EOF) {

if (current\_char == '/' && file.peek() == '\*') {

skip\_comment();

continue;

}

if (isspace(current\_char)) {

skip\_whitespace();

continue;

}

if (isalpha(current\_char)) return parse\_identifier();

if (isdigit(current\_char) || current\_char == '.') return parse\_number();

if (current\_char == '=' || current\_char == '!' || current\_char == '<' || current\_char == '>' ||

current\_char == '+' || current\_char == '-' || current\_char == '\*' || current\_char == '/' ||

current\_char == '&' || current\_char == '|' || current\_char == ':') {

return parse\_operator();

}

return parse\_delimiter();

}

return Lex(LEX\_FIN, "");

}

};

**Приложение Б**

Класс синтаксического анализатора

*Листинг Б.1 – класс Parser*

class Parser {

private:

vector<Lex> lex\_table;

unordered\_map<string, lex\_type> symbol\_table; // Таблица идентификаторов

size\_t pos;

Lex current() {

if (pos < lex\_table.size()) {

return lex\_table[pos];

}

return Lex(LEX\_FIN, "");

}

void advance() {

if (pos < lex\_table.size()) {

pos++;

}

}

void expect(lex\_type type) {

if (current().type != type) {

throw runtime\_error("Syntax error: Expected " + string(lex\_type\_description(type)) + ", but found " + lex\_type\_description(current().type) + ".");

}

advance();

}

void declared\_identifier(const string& identifier) {

if (symbol\_table.find(identifier) == symbol\_table.end()) {

throw runtime\_error("Semantic error: Identifier '" + identifier + "' is not declared.");

}

}

void validate\_identifier(const string& identifier) {

if (identifier.empty()) {

throw runtime\_error("Semantic error: Identifier cannot be empty.");

}

if (!isalpha(identifier[0])) {

throw runtime\_error("Semantic error: Identifier must start with a letter.");

}

for (char c : identifier) {

if (!isalnum(c)) {

throw runtime\_error("Semantic error: Identifier can only contain alphanumeric characters.");

}

}

}

void declare\_identifier(const string& identifier, lex\_type type) {

if (symbol\_table.find(identifier) != symbol\_table.end()) {

throw runtime\_error("Semantic error: Identifier '" + identifier + "' is already declared.");

}

symbol\_table[identifier] = type;

}

lex\_type get\_identifier\_type(const string& identifier) {

auto it = symbol\_table.find(identifier);

if (it == symbol\_table.end()) {

throw runtime\_error("Semantic error: Identifier '" + identifier + "' is not declared.");

}

return it->second;

}

void check\_boolean\_expression() {

lex\_type expr\_type = expression();

if (expr\_type != LEX\_BOOLEAN) {

throw runtime\_error("Semantic error: Condition must be a boolean expression.");

}

}

void check\_relation\_operands(lex\_type left, lex\_type right) {

if (left != LEX\_INTEGER || right != LEX\_INTEGER) {

throw runtime\_error("Semantic error: Relational operators require integer operands.");

}

}

public:

Parser(const vector<Lex>& lex\_table) : lex\_table(lex\_table), pos(0) {}

void parse() {

program();

if (current().type != LEX\_FIN) {

throw runtime\_error("Syntax error: Unexpected token after program end.");

}

}

void program() {

expect(LEX\_LBRACE);

while (current().type != LEX\_RBRACE) {

if (current().type == LEX\_DIM) {

description();

}

else {

statement();

}

}

expect(LEX\_RBRACE);

}

void description() {

vector<string> temp;

expect(LEX\_DIM);

temp.push\_back(current().value);

expect(LEX\_ID);

while (current().type == LEX\_COMMA) {

advance();

validate\_identifier(current().value);

temp.push\_back(current().value);

expect(LEX\_ID);

}

if (current().type != LEX\_INTEGER && current().type != LEX\_REAL && current().type != LEX\_BOOLEAN) {

throw runtime\_error("Syntax error: Expected type (integer, real, boolean).\n");

}

lex\_type type = current().type;

advance();

for (auto identifier : temp) {

declare\_identifier(identifier, type);

}

}

void statement() {

if (current().type == LEX\_ID) {

assignment();

}

else if (current().type == LEX\_IF) {

conditional();

}

else if (current().type == LEX\_FOR) {

for\_loop();

}

else if (current().type == LEX\_WHILE) {

while\_loop();

}

else if (current().type == LEX\_READLN) {

input\_statement();

}

else if (current().type == LEX\_WRITELN) {

output\_statement();

}

else if (current().type == LEX\_BEGIN) {

compound\_statement();

}

else {

throw runtime\_error("Syntax error: Unexpected token " + string(lex\_type\_description(current().type)));

}

}

void assignment() {

string identifier = current().value;

declared\_identifier(identifier);

expect(LEX\_ID);

expect(LEX\_ASSIGN);

lex\_type left\_type = get\_identifier\_type(identifier);

lex\_type right\_type = expression();

if (left\_type != right\_type) {

throw runtime\_error("Semantic error: Type mismatch in assignment to '" + identifier + "'.");

}

}

void conditional() {

expect(LEX\_IF);

expect(LEX\_LPAREN);

check\_boolean\_expression();

expect(LEX\_RPAREN);

statement();

if (current().type == LEX\_ELSE) {

advance();

statement();

}

}

void for\_loop() {

expect(LEX\_FOR);

assignment();

expect(LEX\_TO);

check\_boolean\_expression();

if (current().type == LEX\_STEP) {

advance();

lex\_type expr\_type = expression();

if (expr\_type != LEX\_INTEGER) {

throw runtime\_error("Semantic error: Loop step must be an integer.");

}

}

statement();

expect(LEX\_NEXT);

}

void while\_loop() {

expect(LEX\_WHILE);

expect(LEX\_LPAREN);

check\_boolean\_expression();

expect(LEX\_RPAREN);

statement();

}

lex\_type expression() {

lex\_type result = operand();

while (current().type == LEX\_EQ || current().type == LEX\_NEQ ||

current().type == LEX\_LT || current().type == LEX\_LEQ ||

current().type == LEX\_GT || current().type == LEX\_GEQ) {

lex\_type left = result;

advance();

lex\_type right = operand();

check\_relation\_operands(left, right);

result = LEX\_BOOLEAN;

}

return result;

}

lex\_type operand() {

lex\_type result = term();

while (current().type == LEX\_PLUS || current().type == LEX\_MINUS || current().type == LEX\_OR) {

advance();

lex\_type temp\_result = term();

if (temp\_result != result) {

result = LEX\_NULL;

}

}

return result;

}

lex\_type term() {

lex\_type result = factor();

while (current().type == LEX\_TIMES || current().type == LEX\_DIV || current().type == LEX\_AND) {

advance();

lex\_type temp\_result = factor();

if (temp\_result != result) {

result = LEX\_NULL;

}

}

return result;

}

lex\_type factor() {

if (current().type == LEX\_ID) {

string identifier = current().value;

declared\_identifier(identifier);

advance();

return get\_identifier\_type(identifier);

}

else if (current().type == LEX\_NUM) {

advance();

return LEX\_INTEGER;

}

else if (current().type == LEX\_RNUM) {

advance();

return LEX\_REAL;

}

else if (current().type == LEX\_LPAREN) {

advance();

lex\_type result = expression();

expect(LEX\_RPAREN);

return result;

}

else if (current().type == LEX\_TRUE || current().type == LEX\_FALSE) {

advance();

return LEX\_BOOLEAN;

}

else if (current().type == LEX\_NOT) {

advance();

lex\_type result = factor();

if (result != LEX\_BOOLEAN) {

throw runtime\_error("Semantic error: NOT operator requires a boolean operand.");

}

return LEX\_BOOLEAN;

}

else {

throw runtime\_error("Syntax error: Unexpected token " + string(lex\_type\_description(current().type)));

}

}

void compound\_statement() {

expect(LEX\_BEGIN);

statement();

while (current().type == LEX\_SEMICOLON) {

advance();

statement();

}

expect(LEX\_END);

}

void input\_statement() {

expect(LEX\_READLN);

declared\_identifier(current().value);

expect(LEX\_ID);

while (current().type == LEX\_COMMA) {

advance();

declared\_identifier(current().value);

expect(LEX\_ID);

}

}

void output\_statement() {

expect(LEX\_WRITELN);

expression();

while (current().type == LEX\_COMMA) {

advance();

expression();

}

}

};