СОДЕРЖАНИЕ

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>::= program var <описание> begin <оператор> {;
<оператор>} end.
<описание>::= {<идентификатор> {, <идентификатор> } : <тип>
;}
<тип>::= % | ! | $
<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> }
<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.

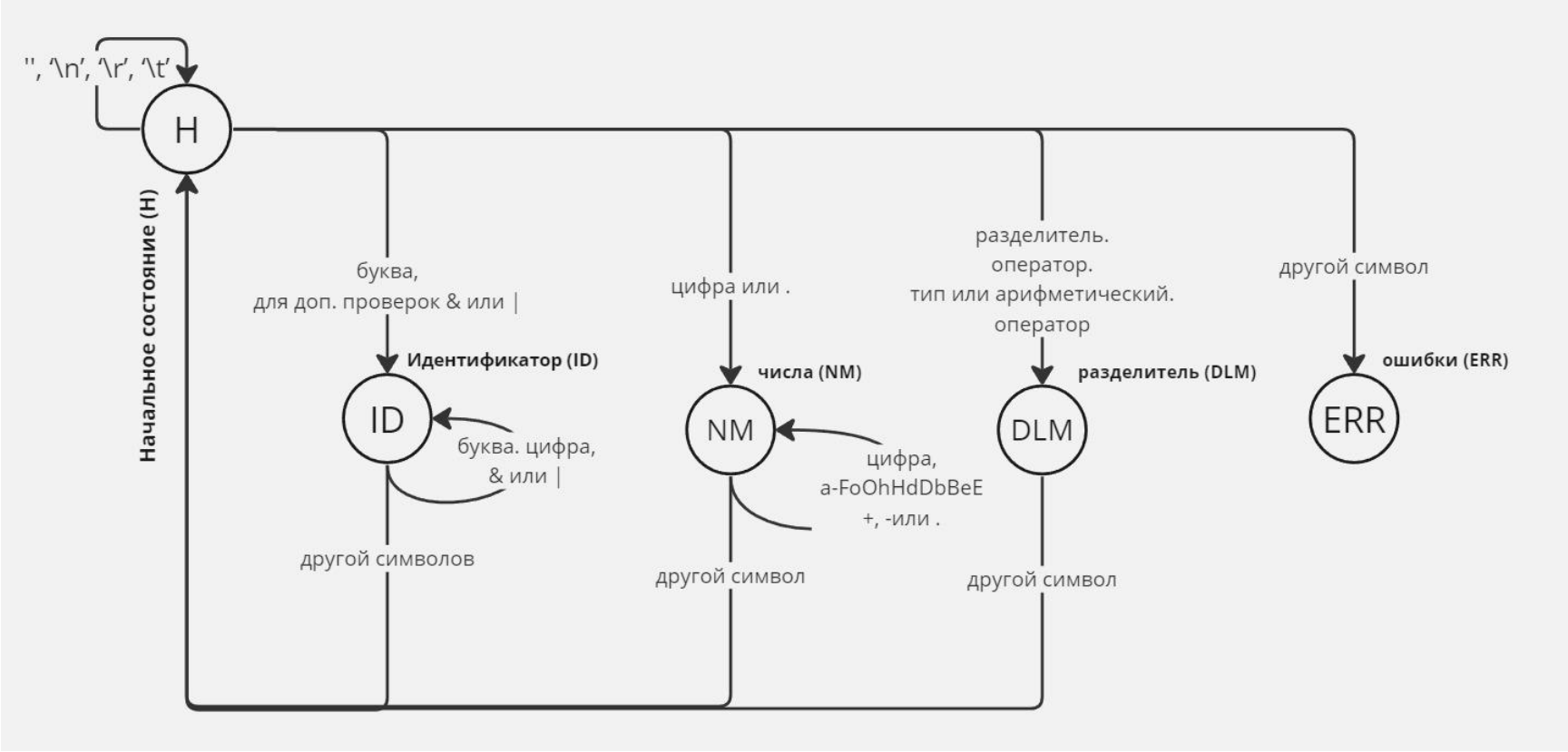

Рисунок 1 – Диаграмма состояний лексического анализатора

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:

$P \to \boldsymbol{program}\, D1; B \perp$
$D1 \to \boldsymbol{var}\, D\, \{, D\}$
$D \to I\, \{, I\}: [\,\% \mid \$ \mid !]$
$B \to \boldsymbol{begin}\, S\, \{; S\}\, \boldsymbol{end}$
$S \to I := E \mid \boldsymbol{if}\ \text{“(“}E\text{”)”}\ S\ \boldsymbol{else}\ S \mid \boldsymbol{while}\ \text{“(“}E\text{”)”}\ S \mid B \mid \boldsymbol{readln}(I) \mid \boldsymbol{writeln}(E)$
$E \to E1\{[:= \mid > \mid < \mid >= \mid <= \mid !=]E1\}$
$E1 \to T\{[+ \mid - \mid || \,]T\}$
$T \to F\{[* \mid / \mid \&\& \,]F\}$
$F \to I \mid N \mid L \mid !F \mid (E)$
$L \to \boldsymbol{true} \mid \boldsymbol{false}$
$I \to C \mid IC \mid IR$
$N \to R \mid NR$
$C \to a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z$
$R \to 0 \mid 1 \mid \ldots \mid 9$

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 таблицу *Table of Identifiers* заносятся данные обо всех лексемах- идентификаторах, которые встречаются в тексте программы. На этапе синтаксического анализа в ту же таблицу заносятся данные о типе идентификатора и о наличии для него описания.

Здесь *stack* – структура данных, в которую запоминаются идентификаторы (номера строк в таблице *TID*), *dec* – функция, задача которой заключается в занесении информации об идентификаторах (поля *type* и *declared*), а также контроль повторного объявления идентификатора.

Описания функций семантических проверок приведены в листинге в Приложении В.

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 номером некорректной лексемы. Рассмотрим примеры.

1. Исходный код программы приведен в листинге 1.

Листинг 1 – Тестовая программа

```
program
var r1, r2, r3: !;
begin
        r1 := 12.2;
        r2 := 9Dh;
        r3 := 2.3e+1;
        while (!(r2) != 100 &&  r1 <= 100)
        begin
                r3 := r3*17;
                r2 := r2 - 10
        end
end@
```

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

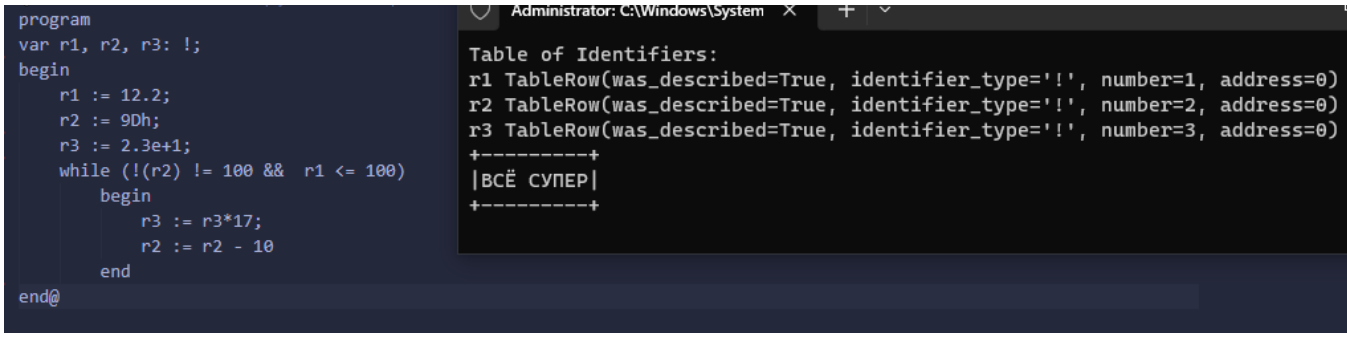

Рисунок 2 – Пример синтаксически корректной программы

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

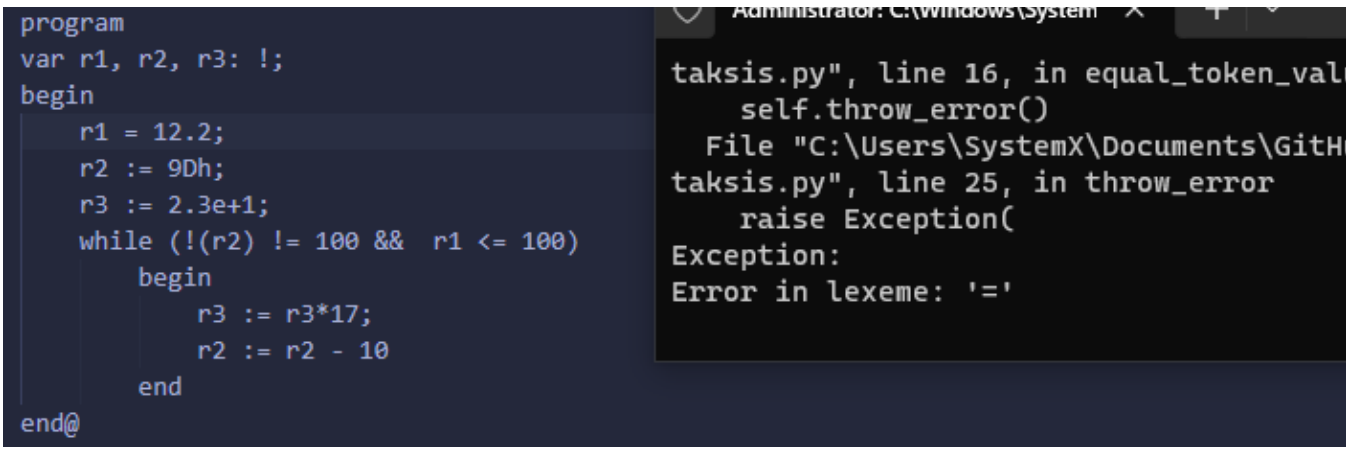

Рисунок 3 – Пример программы, содержащей ошибку

Здесь ошибка допущена в строке 4: неправильное использование оператора сравнения (=). В сообщении об ошибке указана ошибочная лексема.

3. Исходный текст программы, содержащей семантическую проверку, приведен на рис. 4 вместе с сообщением об ошибке. Здесь используется необъявленная переменная (r4).

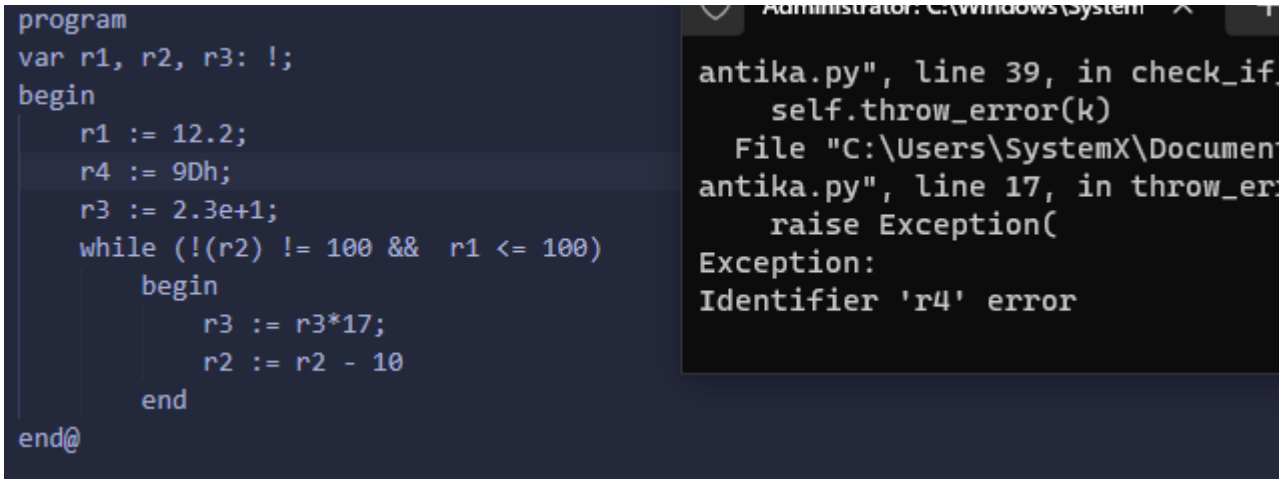

Рисунок 4 – Пример программы, содержащей семантическую ошибку

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 высокого Python в виде класса *Lexer*.

Разбором исходного текста программы занимается синтаксический анализатор, который реализован в виде класса *Sintaksis* на языке Python. Анализатор распознает входной язык по методу рекурсивного спуска. Для применимости необходимо было преобразовать грамматику, в частности, специальным образомобрабатывать 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. https://myfilology.ru/165/yazyki-programmirovaniya-i-ix-ispolzovanie-v-informaczionnyx-sistemax/leksicheskij-sintaksicheskij-semanticheskij-analiz/

Приложения

Приложение А –Код лексического анализатора

Приложение Б – Код синтаксического анализатора

Приложение В – Код семантического анализатор

Приложение А

Код лексического анализатора

Листинг А.1 – Lexer.py

```
import re
from typing import NamedTuple

class LexicalAnalyzer:
    def __init__(self, filename: str, identifiersTable):
        self.identifiersTable = identifiersTable
        self.states = States("H", "COMM", "ID", "ERR", "NM", "DLM")
        self.token_names = Tokens("KWORD", "IDENT", "NUM", "OPER", "DELIM", "NUM2",
"NUM8", "NUM10", "NUM16", "REAL",
                                  "TYPE", "ARITH")
        self.keywords = {"||": 1, "&&": 2, "program": 3, "var": 4, "begin": 5, "end": 6, ":=": 7, "if": 8,
                         "then": 9, "else": 10, "for": 11, "to": 12, "step": 13, "while": 14, "readln": 15, "writeln":
16,
                         "true": 17, "false": 18, "next": 19}
        self.types = {"%", "!", "$"}  # +
        self.arith = {"+", '-', '*', '/'}  # +
        self.operators = {"!=", "==", "<", "<=", ">", ">=", '='}  # +
        self.delimiters = {";", ",", "[", "]", "(", ")", ":"}
        self.fgetc = fgetc_generator(filename)
        self.current = Current(state=self.states.H)
        self.error = Error(filename)
        self.lexeme_table = []

    def analysis(self):
        self.current.state = self.states.H
        self.current.re_assign(*next(self.fgetc))
        while not self.current.eof_state:
            if self.current.state == self.states.H:
                self.h_state_processing()
            elif self.current.state == self.states.ID:
                self.id_state_processing()
            elif self.current.state == self.states.ERR:
                self.err_state_processing()
            elif self.current.state == self.states.NM:
                self.nm_state_processing()
            elif self.current.state == self.states.DLM:
                self.dlm_state_processing()

    def h_state_processing(self):
        while not self.current.eof_state and self.current.symbol in {" ", "\n", "\t"}:
            self.current.re_assign(*next(self.fgetc))
        if self.current.symbol.isalpha() or self.current.symbol == "&" or self.current.symbol == "|":
            self.current.state = self.states.ID
        elif self.current.symbol in set(list("0123456789.")):
            self.current.state = self.states.NM
```

Продолжение листинга А.1

```
        elif self.current.symbol in (self.delimiters | self.operators | self.types | self.arith):
            self.current.state = self.states.DLM
        else:
            self.current.state = self.states.ERR


    def dlm_state_processing(self):
        if self.current.symbol in self.delimiters | self.arith | self.types:
            if self.current.symbol == ":":
                temp_symbol = self.current.symbol
                if not self.current.eof_state:
                    self.current.re_assign(*next(self.fgetc))
                    if temp_symbol + self.current.symbol == ":=":
                        self.add_token(self.token_names.OPER, ":=")
                        if not self.current.eof_state:
                            self.current.re_assign(*next(self.fgetc))
                        self.current.state = self.states.H
                        return
                    else:
                        self.add_token(self.token_names.DELIM, temp_symbol)
                else:
                    self.add_token(self.token_names.DELIM, temp_symbol)
            elif self.current.symbol == "!":
                temp_symbol = self.current.symbol
                if not self.current.eof_state:
                    self.current.re_assign(*next(self.fgetc))
                    if self.current.symbol == "=":
                        self.add_token(self.token_names.OPER, "!=")
                        if not self.current.eof_state:
                            self.current.re_assign(*next(self.fgetc))
                        self.current.state = self.states.H
                        return
                    elif self.current.symbol == "(":
                        # Добавьте обработку, если после "!" идет "("
                        self.add_token(self.token_names.OPER, temp_symbol)
                        self.add_token(self.token_names.DELIM, "(")
                        if not self.current.eof_state:
                            self.current.re_assign(*next(self.fgetc))
                        self.current.state = self.states.H
                        return
                    else:
                        self.add_token(self.token_names.TYPE, temp_symbol)
                else:
                    self.add_token(self.token_names.TYPE, temp_symbol)
            elif self.current.symbol in self.operators:
                temp_symbol = self.current.symbol
                if not self.current.eof_state:
                    self.current.re_assign(*next(self.fgetc))
                    if temp_symbol + self.current.symbol == "==":
```

Продолжение листинга А.1

```
                    self.add_token(self.token_names.OPER, "==")
                    if not self.current.eof_state:
                        self.current.re_assign(*next(self.fgetc))
                    self.current.state = self.states.H
                    return
                else:
                    self.add_token(self.token_names.OPER, temp_symbol)
            else:
                self.add_token(self.token_names.OPER, temp_symbol)
        elif self.current.symbol in self.types:
            self.add_token(self.token_names.TYPE, self.current.symbol)
        else:
            self.add_token(self.token_names.ARITH, self.current.symbol)
        if not self.current.eof_state:
            self.current.re_assign(*next(self.fgetc))
    else:
        temp_symbol = self.current.symbol
        if not self.current.eof_state:
            self.current.re_assign(*next(self.fgetc))
            if temp_symbol + self.current.symbol in self.operators:
                self.add_token(self.token_names.OPER, temp_symbol + self.current.symbol)
                if not self.current.eof_state:
                    self.current.re_assign(*next(self.fgetc))
            else:
                self.add_token(self.token_names.OPER, temp_symbol)
        else:
            self.add_token(self.token_names.OPER, self.current.symbol)
    self.current.state = self.states.H



def err_state_processing(self):
    raise Exception(
        f"\nUnknown: '{self.error.symbol}' in file {self.error.filename} \nline:
{self.current.line_number} and pos: {self.current.pos_number}")

def id_state_processing(self):
    buf = [self.current.symbol]
    if not self.current.eof_state:
        self.current.re_assign(*next(self.fgetc))
    while not self.current.eof_state and (
            self.current.symbol.isalpha() or self.current.symbol.isdigit() or self.current.symbol == "&" or
self.current.symbol == "|"):
        buf.append(self.current.symbol)
        self.current.re_assign(*next(self.fgetc))
    buf = ''.join(buf)
    if self.is_keyword(buf):
        self.add_token(self.token_names.KWORD, buf)
    else:
```

Продолжение листинга А.1

```
            self.add_token(self.token_names.IDENT, buf)
            if buf not in self.keywords:
                self.identifiersTable.put(buf)
        self.current.state = self.states.H


    def nm_state_processing(self):
        buf = []
        buf.append(self.current.symbol)
        if not self.current.eof_state:
            self.current.re_assign(*next(self.fgetc))
        while not self.current.eof_state and (self.current.symbol in
set(list("ABCDEFabcdefoOdDhH0123456789.eE+-"))):
            buf.append(self.current.symbol)
            self.current.re_assign(*next(self.fgetc))

        buf = ''.join(buf)
        is_n, token_num = self.is_num(buf)
        if is_n:
            self.add_token(token_num, buf)
            self.current.state = self.states.H
        else:
            self.error.symbol = buf
            self.current.state = self.states.ERR

    def is_num(self, digit):
        if re.match(r"(^\d+[Ee][+-]?\d+$|^\d*\.\d+([Ee][+-]?\d+)?$)", digit):
            return True, self.token_names.REAL
        elif re.match(r"^[01]+[Bb]$", digit):
            return True, self.token_names.NUM2
        elif re.match(r"^[01234567]+[Oo]$", digit):
            return True, self.token_names.NUM8
        elif re.match(r"^\d+[dD]?$", digit):
            return True, self.token_names.NUM10
        elif re.match(r"^\d[0-9ABCDEFabcdef]*[Hh]$", digit):
            return True, self.token_names.NUM16

        return False, False

    def is_keyword(self, word):
        if word in self.keywords:
            return True
        return False

    def add_token(self, token_name, token_value):
        self.lexeme_table.append(Token(token_name, token_value))
```

Продолжение листинга А.1

```
class Token:
    def __init__(self, token_name, token_value):
        self.token_name = token_name
        self.token_value = token_value
    def __repr__(self):
        return f"{self.token_name} -> {self.token_value}"


class States(NamedTuple):
    H: str
    COMM: str
    ID: str
    ERR: str
    NM: str
    DLM: str


class Tokens(NamedTuple):
    KWORD: str
    IDENT: str
    NUM: str
    OPER: str
    DELIM: str
    NUM2: str
    NUM8: str
    NUM10: str
    NUM16: str
    REAL: str
    TYPE: str
    ARITH: str


class Current:
    def __init__(self, symbol: str = "", eof_state: bool = False, line_number: int = 0, pos_number: int = 0,
                 state: str = ""):
        self.symbol = symbol
        self.eof_state = eof_state
        self.line_number = line_number
        self.pos_number = pos_number
        self.state = state

    def re_assign(self, symbol: str, eof_state: bool, line_number: int, pos_number: int):
        self.symbol = symbol
        self.eof_state = eof_state
        self.line_number = line_number
        self.pos_number = pos_number
```

Окончание листинга А.1

```
class Error:
    def __init__(self, filename: str, symbol: str = "", line: int = 0, pos_in_line: int = 0):
        self.filename = filename
        self.symbol = symbol
        self.line = line
        self.pos_in_line = pos_in_line
  def fgetc_generator(filename: str):
        with open(filename) as fin:
            s = list(fin.read())
            s.append('\n')
            counter_pos, counter_line = 1, 1
            for i in range(len(s)):
                yield s[i], s[i] == "@", counter_line, counter_pos
                if s[i] == "\n":
                    counter_pos = 0
                    counter_line += 1
                else:
                    counter_pos += 1
```

Приложение Б

Код синтаксического анализатора

Листинг Б.1 – Sintaksis.py

```
class SyntacticalAnalyzer:
    def __init__(self, lexeme_table, identifiersTable):
        self.identifiersTable = identifiersTable
        self.lex_get = self.lexeme_generator(lexeme_table)
        self.id_stack = []
        self.current_lex = next(self.lex_get)
        self.relation_operations = {"!=", "==", "<", "<=", ">", ">=", '='}
        self.term_operations = {"+", "-", "||"}
        self.factor_operations = {"*", "/", "&&"}
        self.keywords = {"||": 1, "&&": 2, "program": 3, "var": 4, "begin": 5, "end": 6, ":=": 7, "if": 8,
                         "then": 9, "else": 10, "for": 11, "to": 12, "step": 13, "while": 14, "readln": 15, "writeln":
16,
                         "true": 17, "false": 18, "next": 19}

    def equal_token_value(self, word):
        if self.current_lex.token_value != word:
            self.throw_error()
        self.current_lex = next(self.lex_get)

    def equal_token_name(self, word):
        if self.current_lex.token_name != word:
            self.throw_error()
        self.current_lex = next(self.lex_get)

    def throw_error(self):
        raise Exception(
            f"\nError in lexeme: '{self.current_lex.token_value}'")

    def lexeme_generator(self, lexeme_table):
        for i, token in enumerate(lexeme_table):
            yield token

    def PROGRAMM(self): # <программа>::= program var <описание> begin <оператор> {;
<оператор>} end
        self.equal_token_value("program")
        self.equal_token_value("var")
        self.DESCRIPTION()
        self.equal_token_value("begin")
        self.OPERATOR()

        while self.current_lex.token_value == ";":
            self.current_lex = next(self.lex_get)
            self.OPERATOR()

        if self.current_lex.token_value != "end"
```

Окончание листинга Б.1

```
        self.throw_error()


    def DESCRIPTION(self): # <описание>::= {<идентификатор>{, <идентификатор>} : <тип> ;}
        while self.current_lex.token_value != "begin":
            self.IDENTIFIER(from_description=True)
            while self.current_lex.token_value == ",":
                self.current_lex = next(self.lex_get)
                self.IDENTIFIER(from_description=True)

            # Теперь ожидаем двоеточие и тип, но без точки с запятой
            self.equal_token_value(":")
            self.TYPE(from_description=True)

            # Ожидаем только если не следует begin
            if self.current_lex.token_value != "begin":
                self.equal_token_value(";")


    def IDENTIFIER(self, from_description=False):
        if from_description:
            if self.current_lex.token_name != "IDENT":
                self.throw_error()
            self.id_stack.append(self.current_lex.token_value)
            self.current_lex = next(self.lex_get)
        else:
            self.equal_token_name("IDENT")
    def TYPE(self, from_description=False):
        if from_description:
            if self.current_lex.token_name != "TYPE":
                self.throw_error()
            for item in self.id_stack:
                if item not in self.keywords:
                    self.identifiersTable.put(item, True, self.current_lex.token_value)
            self.id_stack = []
            self.current_lex = next(self.lex_get)
        else:
            self.equal_token_name("TYPE")
    def OPERATOR(
            self):
        if self.current_lex.token_value == "begin":
            self.COMPOSITE_OPERATOR()
        elif self.current_lex.token_value == "if":
            self.CONDITIONAL_OPERATOR()
        elif self.current_lex.token_value == "for":
            self.FIXED_CYCLE_OPERATOR()
```

Окончание листинга Б.1

```
        elif self.current_lex.token_value == "while":

            self.CONDITIONAL_CYCLE_OPERATOR()
        elif self.current_lex.token_value == "readln":
            self.INPUT_OPERATOR()
        elif self.current_lex.token_value == "writeln":
            self.OUTPUT_OPERATOR()
        else:
            self.ASSIGNMENT_OPERATOR()

    def COMPOSITE_OPERATOR(self):
        self.equal_token_value("begin")
        self.OPERATOR()

        while self.current_lex.token_value in {"\n", ";"}:
            self.current_lex = next(self.lex_get)
            self.OPERATOR()

        self.equal_token_value("end")

    def CONDITIONAL_OPERATOR(self):
        self.equal_token_value("if")
        if self.current_lex.token_value == "(":
            self.current_lex = next(self.lex_get)
            self.EXPRESSION()
            if self.current_lex.token_value == ")":
                self.current_lex = next(self.lex_get)
            else:
                self.throw_error()
        else:
            self.throw_error()

        self.OPERATOR()

        if self.current_lex.token_value == "else":
            self.current_lex = next(self.lex_get)
            self.OPERATOR()




    def FIXED_CYCLE_OPERATOR(self):
        self.equal_token_value("for")
        self.ASSIGNMENT_OPERATOR()

        if self.current_lex.token_value == "to":
            self.current_lex = next(self.lex_get)
            end_expression = self.EXPRESSION()
```

Окончание листинга Б.1

```
            step_expression = None
            if self.current_lex.token_value == "step":
                self.current_lex = next(self.lex_get)
                step_expression = self.EXPRESSION()

            self.OPERATOR()

            # Вставка логики для обработки "next"
            if self.current_lex.token_value == "next":
                self.current_lex = next(self.lex_get)
            else:
                self.throw_error()
        else:
            self.throw_error()


    def CONDITIONAL_CYCLE_OPERATOR(self):
        self.equal_token_value("while")
        self.equal_token_value("(")
        self.EXPRESSION()
        self.equal_token_value(")")
        self.OPERATOR()

    def INPUT_OPERATOR(self):
        self.equal_token_value("readln")
        self.IDENTIFIER()
        while self.current_lex.token_value == ",":
            self.current_lex = next(self.lex_get)
            self.IDENTIFIER()

    def OUTPUT_OPERATOR(self):
        self.equal_token_value("writeln")
        self.EXPRESSION()
        while self.current_lex.token_value == ",":
            self.current_lex = next(self.lex_get)
            self.EXPRESSION()

    def ASSIGNMENT_OPERATOR(self):
        self.IDENTIFIER()
        self.equal_token_value(":=")
        self.EXPRESSION()


    def EXPRESSION(self):
        self.OPERAND()
        while self.current_lex.token_value in self.relation_operations:
            self.current_lex = next(self.lex_get)
            self.OPERAND()
```

Окончание листинга Б.1

```
    def OPERAND(self):
        self.TERM()
        while self.current_lex.token_value in self.term_operations:
            self.current_lex = next(self.lex_get)
            self.TERM()

    def TERM(self):
        self.FACTOR()
        while self.current_lex.token_value in self.factor_operations:
            self.current_lex = next(self.lex_get)
            self.FACTOR()

    def FACTOR(self):
        if self.current_lex.token_name in {"IDENT", "NUM", "NUM2", "NUM8", "NUM10", "NUM16",
"REAL"}:
            self.current_lex = next(self.lex_get)
        elif self.current_lex.token_value in {"true", "false"}:
            self.current_lex = next(self.lex_get)
        elif self.current_lex.token_value == "!":
            self.equal_token_value("!")
            self.FACTOR()
        elif self.current_lex.token_value == "(":
            self.current_lex = next(self.lex_get)
            self.EXPRESSION()
            self.equal_token_value(")")
        else:
            self.throw_error()
```

Приложение В

Код семантического анализатора

Листинг Б.1 – Semantika.py

```
from typing import NamedTuple


class TableRow(NamedTuple):
    was_described: bool
    identifier_type: str
    number: int
    address: int


class IdentifiersTable:
    def __init__(self):
        self.table = {}
        self.n = 0

    def throw_error(self, lex):
        raise Exception(
            f"\nIdentifier '{lex}' error")

    def put(self, identifier, was_described=False, identifier_type=None, address=0):
        if identifier not in self.table:
            self.table[identifier] = TableRow(was_described, identifier_type, self.n + 1, address)
            self.n += 1
        elif identifier in self.table and not self.table[identifier].was_described:
            self.table[identifier] = TableRow(was_described, identifier_type, self.table[identifier].number,
address)
        elif identifier in self.table and self.table[identifier].was_described:
            self.throw_error(identifier)


    def __repr__(self):
        res = ["\nTable of Identifiers:"]
        for k, v in self.table.items():
            res.append(f'{k} {v}')
        return "\n".join(res)

    def check_if_all_described(self):
        for k, v in self.table.items():
            if not v.was_described:
                self.throw_error(k)
```