СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc153019216)

[1.Постановка задачи 5](#_Toc153019217)

[2.Формальная модель задачи 7](#_Toc153019218)

[2.1.РБНФ 7](#_Toc153019219)

[2.2.Диаграмма Вирта 9](#_Toc153019220)

[2.3.Формальные грамматики 11](#_Toc153019221)

[3.Спецификация основных процедур и функций 13](#_Toc153019222)

[3.1.Лексический анализатор 13](#_Toc153019223)

[3.2.Синтаксический анализатор 15](#_Toc153019224)

[3.3.Семантический анализатор 17](#_Toc153019225)

[3.4.Основная программа 18](#_Toc153019226)

[4.Тестирование 19](#_Toc153019227)

[ЗАКЛЮЧЕНИЕ 25](#_Toc153019228)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 26](#_Toc153019229)

[Приложение A 27](#_Toc153019230)

ВВЕДЕНИЕ

Невозможно недооценивать важность языков программирования в современном мире, когда всем правит техника и технологии необходимо иметь удобный инструмент для управления современными технологиями. Существует множество языков программирования, подходящих для решения конкретной задачи. Так, чтобы разработка программных продуктов была удобной, интерпретируемой, а главное эффективной требуется что-то, что будет помогать разработчику давать команды бездушной машине и которые будут однозначно восприниматься ею. Для такой возможности существуют различные распознаватели формальных языков.

Поэтому осознавая важность перед нами встает цель разработки распознавателя модельного языка программирования, включающего в себя лексический, синтаксический и семантический анализаторы.

1. **Постановка задачи**

Разработать распознаватель модельного языка программирования, выполнив следующие действия.

1. В соответствии с номером варианта составить формальное описание модельного языка программирования с помощью:
2. РБНФ;
3. диаграмм Вирта;
4. формальных грамматик.
5. Написать пять содержательных примеров программ, раскрывающих особенности конструкций модельного языка программирования, отразив в этих примерах все его функциональные возможности.
6. Составить таблицы лексем и диаграмму состояний с действиями для распознавания и формирования лексем языка.
7. По диаграмме с действиями написать функцию сканирования текста входной программы на модельном языке.
8. Разработать программное средство, реализующее лексический анализ текста программы на входном языке.
9. Реализовать синтаксический анализатор текста программы на модельном языке методом рекурсивного спуска.
10. Построить цепочку вывода и дерево разбора простейшей программы на модельном языке из начального символа грамматики.
11. Дополнить синтаксический анализатор процедурами проверки семантической правильности программы на модельном языке в соответствии с контекстными условиями вашего варианта.
12. Распечатать пример таблиц идентификаторов и двуместных операций.
13. Показать динамику изменения содержимого стека при семантическом анализе программы на примере одного синтаксически правильного выражения.
14. Составить набор контрольных примеров, демонстрирующих все возможные типы лексических, синтаксических и семантических ошибок в программах на модельном языке.
15. **Формальная модель задачи**
    1. **РБНФ**

<программа>::= program var <описание> begin <оператор> {; <оператор>} end.

<описание>::= {<идентификатор> {, <идентификатор> } : <тип> ;}

<тип>::= % | ! | $

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

<составной>::= «[» <оператор> { : <оператор> } «]»

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

<условный>::= if <выражение> then <оператор> [ else <оператор>]

<фиксированного\_цикла>::= for <присваивания> to <выражение> do <оператор>

<условного\_цикла>::= while <выражение> do <оператор>

<ввода>::= read «(»<идентификатор> {, <идентификатор> } «)»

<вывода>::= write «(»<выражение> {, <выражение> } «)»

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

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

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

<унарная\_операция>::= not

<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>«)»

<число>::= <целое> | <действительное>

<логическая\_константа>::= true | false

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

<целое>::= <двоичное> | <восьмеричное> | <десятичное> |

<шестнадцатеричное>

<двоичное>::= {/ 0 | 1 /} (B | b)

<восьмеричное>::= {/ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 /} (O | o)

<десятичное>::= {/ <цифра> /} [D | d]

<шестнадцатеричное>::= <цифра> {<цифра> | A | B | C | D | E | F | a | b | c | d | e | f} (H | h)

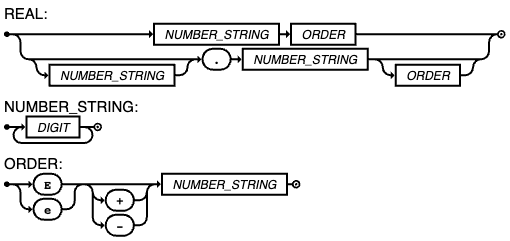
<действительное>::= <числовая\_строка> <порядок> | [<числовая\_строка>] .<числовая\_строка> [порядок]

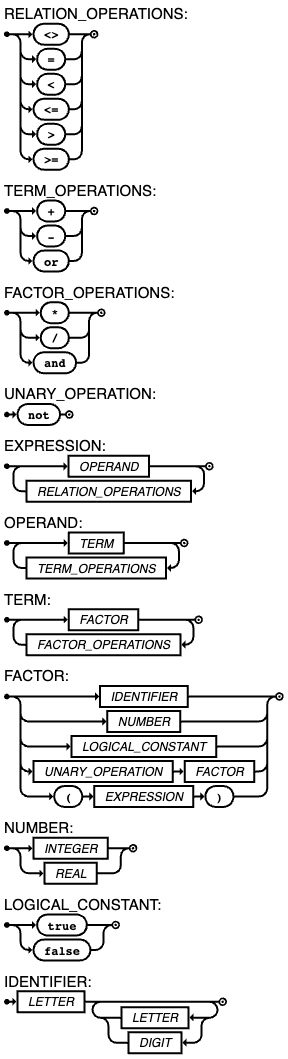
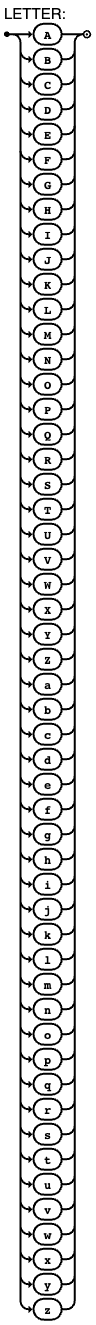
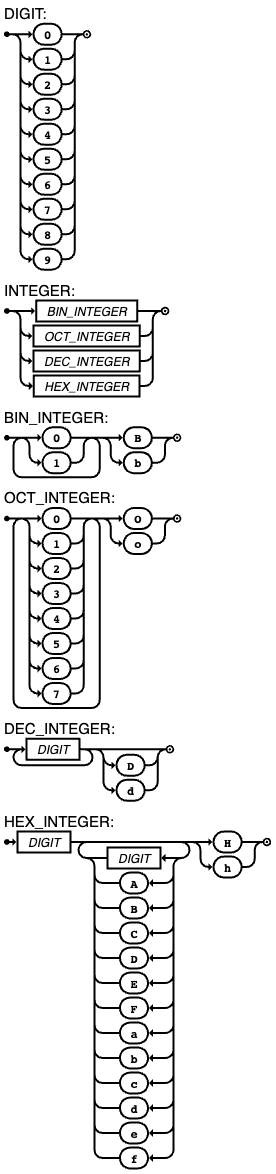
<числовая\_строка>::= {/ <цифра> /}

<порядок>::= ( E | e )[+ | -] <числовая\_строка>

* 1. **Диаграмма Вирта**





* 1. **Формальные грамматики**

PROGRAM → program var DESCRIPTION begin OPERATOR1

OPERATOR1 → OPERATOR ; OPERATOR1| OPERATOR

DESCRIPTION → IDENTIFIER1 : TYPE ; DESCRIPTION | IDENTIFIER1 : TYPE;

IDENTIFIER1 → IDENTIFIER1, IDENTIFIER | IDENTIFIER

TYPE → % | ! | $

OPERATOR → COMPOSITE\_OPERATOR | ASSIGNMENT\_OPERATOR | CONDITIONAL\_OPERATOR | FIXED\_CYCLE\_OPERATOR | CONDITIONAL\_CYCLE\_OPERATOR | INPUT\_OPERATOR | OUTPUT\_OPERATOR

COMPOSITE\_OPERATOR → [COMPOSITE\_OPERATOR1]

COMPOSITE\_OPERATOR1 → OPERATOR | COMPOSITE\_OPERATOR1 : OPERATOR

ASSIGNMENT\_OPERATOR → IDENTIFIER as EXPRESSION

CONDITIONAL\_OPERATOR → if EXPRESSION then OPERATOR | if EXPRESSION then OPERATOR else OPERATOR

FIXED\_CYCLE\_OPERATOR → for ASSIGNMENT\_OPERATOR to EXPRESSION do OPERATOR

CONDITIONAL\_CYCLE\_OPERATOR → while EXPRESSION do OPERATOR

INPUT\_OPERATOR → read (INPUT\_OPERATOR1)

INPUT\_OPERATOR1 → INPUT\_OPERATOR1 , IDENTIFIER | IDENTIFIER

OUTPUT\_OPERATOR → write ( OUTPUT\_OPERATOR1)

OUTPUT\_OPERATOR1 → OUTPUT\_OPERATOR1, EXPRESSION | EXPRESSION

RELATION\_OPERATIONS → <> | = | < | <= | > | >=

TERM\_OPERATIONS → + | - | or

FACTOR\_OPERATIONS → \* | / | and

UNARY\_OPERATION → not

EXPRESSION → OPERAND | OPERAND EXPRESSION1

EXPRESSION1 → RELATION\_OPERATIONS OPERAND | RELATION\_OPERATIONS OPERAND EXPRESSION1

OPERAND → TERM | TERM OPERAND1

OPERAND1 → TERM\_OPERATIONS TERM | OPERAND1 TERM\_OPERATIONS TERM

TERM → FACTOR | FACTOR TERM1

TERM1 → FACTOR\_OPERATIONS FACTOR | TERM1 FACTOR\_OPERATIONS FACTOR

FACTOR → IDENTIFIER | NUMBER | LOGICAL\_CONSTANT | UNARY\_OPERATION FACTOR | ( EXPRESSION )

NUMBER → INTEGER | REAL

LOGICAL\_CONSTANT → true | false

IDENTIFIER → IDENTIFIER LETTER | IDENTIFIER DIGIT | LETTER

LETTER → 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

DIGIT → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

INTEGER → BIN\_INTEGER | OCT\_INTEGER | DEC\_INTEGER | HEX\_INTEGER

BIN\_INTEGER → BIN\_INTEGER1 (B | b)

BIN\_INTEGER1 → 0 | 1 | BIN\_INTEGER1 1 | BIN\_INTEGER1 0

OCT\_INTEGER → OCT\_INTEGER1 O | OCT\_INTEGER1 o

OCT\_INTEGER1 → OCT\_INTEGER1 0 | OCT\_INTEGER1 1 | OCT\_INTEGER1 2 | OCT\_INTEGER1 3 | OCT\_INTEGER1 4 | OCT\_INTEGER1 5 | OCT\_INTEGER1 6 | OCT\_INTEGER1 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

DEC\_INTEGER → DEC\_INTEGER1 | DEC\_INTEGER1 D | DEC\_INTEGER1 d

DEC\_INTEGER1 → DIGIT | DEC\_INTEGER1 DIGIT

HEX\_INTEGER → HEX\_INTEGER1 H | HEX\_INTEGER1 h

HEX\_INTEGER1 → DIGIT | HEX\_INTEGER1 HEX\_INTEGER2

HEX\_INTEGER2 → DIGIT | A | B | C | D | E | F | a | b | c | d | e | f

REAL → NUMBER\_STRING ORDER | NUMBER\_STRING . NUMBER\_STRING ORDER |. NUMBER\_STRING ORDER | NUMBER\_STRING . NUMBER\_STRING | . NUMBER\_STRING

NUMBER\_STRING → DIGIT | NUMBER\_STRING DIGIT

ORDER → ORDER1 NUMBER\_STRING | ORDER1 + NUMBER\_STRING | ORDER1 - NUMBER\_STRING

ORDER1 → E | e

1. **Спецификация основных процедур и функций**
   1. **Лексический анализатор**

В листинге A.1 можно заметить класс *Token*, который используется для удобства представления и вывода лексем с их типом и значением. Также там представлены классы *States* и *Tokens,* которые являются наследниками класса именованных кортежей, созданы для удовства и оптимизации разработки анализатора.

В листинге A.2 отображена реализация метода *fgetc\_generator*, с помощью него происходит создание генератора, считывание из файла и посимвольное извлечение его содержимого. Класс *Error* создан для удобства хранения информации о том, где произошла ошибка. Класс *Current* используется для хранения текущего состояния (символа, состояния, номера строки, позиции в строке, состояния завершения чтения файла)

В листинге A.3 представлена реализация класса *LexicalAnalyzer*, в котором реализованы методы:

*def \_\_init\_\_(self, filename: str, identifiersTable)* – метод инициализирует необходимые поля и начальные состояния

*def analysis(self)* – метод производит лексический анализ, в котором до тех пор пока состояние не станет равным концу файла происходит обработка всевозможных состояний

*def h\_state\_processing(self)* – метод обработки состояния H

*def comm\_state\_processing(self)* – метод обработки состояния COMM (строки комментариев)

*def dlm\_state\_processing(self)* – метод обработки состояния DLM (обработка строк разделителей, арифметических операторов и типов переменных)

*def err\_state\_processing(self)* – метод обработки состояния ERR (ошибки)

*def id\_state\_processing(self)* – метод обработки состояния ID (обработка идентификаторов)

*def nm\_state\_processing(self)* – метод обработки состояния NM (обработка численных строк)

*def is\_digit(self, word)* – вспомогательный метод, который позволяет узнать является ли входная строка word числом и какого типа

*def is\_keyword(self, word)* – вспомогательный метод, который позволяет узнать ключевое слово или нет

*def add\_token(self, token\_name, token\_value)* – вспомогательный метод добавления токенов в таблицу лексем

* 1. **Синтаксический анализатор**

В листинге A.4 представлена реализация класса *SyntacticalAnalyzer*, в котором реализованы методы:

*def \_\_init\_\_(self, lexeme\_table, identifiersTable)* – метод инициализации полей и начальных состояний

*def equal\_token\_value(self, word)* – вспомогательный метод проверки значения текущего токена на равенство с word, если успешно считываем следующую лексему, иначе – исключение

*def equal\_token\_name(self, word)* – вспомогательный метод проверки типа текущего токена на равенство с word, если успешно считываем следующую лексему, иначе – исключение

*def throw\_error(self)* – вспомогательный метод вызова исключения

*def lexeme\_generator(self, lexeme\_table)* – метод генератор последовательно выдающий следующий токен в таблице лексем

*def PROGRAMM(self)* – метод синтаксического разбора всей программы

*def DESCRIPTION(self) –* метод синтаксического разбора описаний

*def IDENTIFIER(self) –* метод синтаксического разбора идентификаторов

*def TYPE(self) –* метод синтаксического разбора типов

*def OPERATOR(self) –* метод синтаксического разбора операторов

*def COMPOSITE\_OPERATOR(self) –* метод синтаксического разбора составных

*def CONDITIONAL\_OPERATOR(self) –* метод синтаксического разбора условных операторов

*def FIXED\_CYCLE\_OPERATOR(self) –* метод синтаксического разбора фиксированного оператора цикла

*def CONDITIONAL\_CYCLE\_OPERATOR(self) –* метод синтаксического разбора условного оператора цикла

*def INPUT\_OPERATOR(self) –* метод синтаксического разбора оператора ввода

*def OUTPUT\_OPERATOR(self) –* метод синтаксического разбора оператора вывода

*def ASSIGNMENT\_OPERATOR(self) –* метод синтаксического разбора оператора присваивания

*def EXPRESSION(self) –* метод синтаксического разбора выражений

*def OPERAND(self) –* метод синтаксического разбора операндов

*def TERM(self) –* метод синтаксического разбора слагаемых

*def FACTOR(self) –* метод синтаксического разбора множителей

* 1. **Семантический анализатор**

На этапе семантического анализа производится *обработка описаний,* для которой на этапе лексического анализа производится добавление всех идентификаторов в таблицу, а на этапе синтаксического анализа идентификаторы, которые были описаны помечаются соответствующим образом, и далее производится проверка на то, что все идентификаторы в таблице описаны и описания не повторяются

В листинге A.5 отображена реализация класса *TableRow*, который является именованным кортежем.

Также в нем представлена реализация класса *IdentifiersTable,* который служит для *обработки описаний.*

В нем реализованы следующие методы:

*def throw\_error(self, lex)* – метод вызова исключений

*def put(self, identifier, was\_described=False, identifier\_type=None, address=0)* – метод добавления иденификатор в таблицу

*def check\_if\_all\_described(self)* – метод проверки на то, что все идентификаторы в таблице были описаны

* 1. **Основная программа**

В основной программе, приведенной в листинге A.6 отображено создание таблицы идентификаторов для выполнения обработки описаний. Используются класс лексического анализа, вызов метода *analysis* этого класса. В случае если программа завершилась без ошибок, то производим синтаксический анализ с помощью вызова метода *PROGRAMM.* И далее производим проверку идентификаторов, что в все они были описаны.

При изменении двух констант – имя файла с программой и выводить ли на экран дополнительную информацию можно протестировать разные программы.

1. **Тестирование**

В листингах 4.1 и 4.2 приведены примеры программ, наиболее раскрывающие особенности используемой грамматики

*Листинг 4.1 – Пример программы на описанном языке*

program var  
{Объявляем переменные}  
int1, int2, int3: %;  
float1, float2, float3: !;  
bool1,bool2, bool3: $;  
  
begin  
{оператор присваивания}  
int1 as 22  
;  
{Условный оператор}  
if int1<50  
 then int1 as 50  
else  
 int1 as 100  
;  
{Условный оператор без else}  
if int1 = 50  
 then int1 as 100  
;  
{Оператор ввода}  
read(int2, int3)  
;  
{Оператор вывода}  
write(int2, int3+4\*10>20)  
  
end@

*Листинг 4.2 – Пример программы на описанном языке*

program var  
{Объявляем переменные}  
int1, int2, int3: %;  
float1, float2, float3: !;  
bool1,bool2, bool3: $;  
  
begin  
{Оператор фиксированного цикла}  
for float1 as 22.34 to float1<50  
do  
 [float1 as float1+10.3 : write(1, float1) : float1 as float1+2.1e+1] {составной оператор}  
;  
float2 as 1.1  
;  
{Оператор условного цикла}  
while float2 <= 100  
do  
 [float2 as float2\*1.3 : bool1 as float2>4 : if bool1 then write(1) else write(0)]  
end@

В листингах 4.3 и 4.4 можно увидеть вывод результата работы нашего программного продукта

*Листинг 4.3 – Пример вывода данных для первого примера программы*

Result of Lexical Analyzer:

KWORD program

KWORD var

IDENT int1

DELIM ,

IDENT int2

DELIM ,

IDENT int3

DELIM :

TYPE %

DELIM ;

IDENT float1

DELIM ,

IDENT float2

DELIM ,

IDENT float3

DELIM :

TYPE !

DELIM ;

IDENT bool1

DELIM ,

IDENT bool2

DELIM ,

IDENT bool3

DELIM :

TYPE $

DELIM ;

KWORD begin

IDENT int1

KWORD as

NUM10 22

DELIM ;

KWORD if

IDENT int1

OPER <

NUM10 50

KWORD then

IDENT int1

KWORD as

NUM10 50

KWORD else

IDENT int1

KWORD as

NUM10 100

DELIM ;

KWORD if

IDENT int1

OPER =

NUM10 50

KWORD then

IDENT int1

KWORD as

NUM10 100

DELIM ;

KWORD read

DELIM (

IDENT int2

DELIM ,

IDENT int3

DELIM )

DELIM ;

KWORD write

DELIM (

IDENT int2

DELIM ,

IDENT int3

ARITH +

NUM10 4

ARITH \*

NUM10 10

OPER >

NUM10 20

DELIM )

KWORD end

Table of Identifiers:

int1 TableRow(was\_described=True, identifier\_type='%', number=1, address=0)

int2 TableRow(was\_described=True, identifier\_type='%', number=2, address=0)

int3 TableRow(was\_described=True, identifier\_type='%', number=3, address=0)

float1 TableRow(was\_described=True, identifier\_type='!', number=4, address=0)

float2 TableRow(was\_described=True, identifier\_type='!', number=5, address=0)

float3 TableRow(was\_described=True, identifier\_type='!', number=6, address=0)

bool1 TableRow(was\_described=True, identifier\_type='$', number=7, address=0)

bool2 TableRow(was\_described=True, identifier\_type='$', number=8, address=0)

bool3 TableRow(was\_described=True, identifier\_type='$', number=9, address=0)

+---------+

| SUCCESS |

+---------+

*Листинг 4.4 – Пример вывода данных для второго примера программы*

Result of Lexical Analyzer:

KWORD program

KWORD var

IDENT int1

DELIM ,

IDENT int2

DELIM ,

IDENT int3

DELIM :

TYPE %

DELIM ;

IDENT float1

DELIM ,

IDENT float2

DELIM ,

IDENT float3

DELIM :

TYPE !

DELIM ;

IDENT bool1

DELIM ,

IDENT bool2

DELIM ,

IDENT bool3

DELIM :

TYPE $

DELIM ;

KWORD begin

KWORD for

IDENT float1

KWORD as

REAL 22.34

KWORD to

IDENT float1

OPER <

NUM10 50

KWORD do

DELIM [

IDENT float1

KWORD as

IDENT float1

ARITH +

REAL 10.3

DELIM :

KWORD write

DELIM (

NUM10 1

DELIM ,

IDENT float1

DELIM )

DELIM :

IDENT float1

KWORD as

IDENT float1

ARITH +

REAL 2.1e+1

DELIM ]

DELIM ;

IDENT float2

KWORD as

REAL 1.1

DELIM ;

KWORD while

IDENT float2

OPER <=

NUM10 100

KWORD do

DELIM [

IDENT float2

KWORD as

IDENT float2

ARITH \*

REAL 1.3

DELIM :

IDENT bool1

KWORD as

IDENT float2

OPER >

NUM10 4

DELIM :

KWORD if

IDENT bool1

KWORD then

KWORD write

DELIM (

NUM10 1

DELIM )

KWORD else

KWORD write

DELIM (

NUM10 0

DELIM )

DELIM ]

KWORD end

Table of Identifiers:

int1 TableRow(was\_described=True, identifier\_type='%', number=1, address=0)

int2 TableRow(was\_described=True, identifier\_type='%', number=2, address=0)

int3 TableRow(was\_described=True, identifier\_type='%', number=3, address=0)

float1 TableRow(was\_described=True, identifier\_type='!', number=4, address=0)

float2 TableRow(was\_described=True, identifier\_type='!', number=5, address=0)

float3 TableRow(was\_described=True, identifier\_type='!', number=6, address=0)

bool1 TableRow(was\_described=True, identifier\_type='$', number=7, address=0)

bool2 TableRow(was\_described=True, identifier\_type='$', number=8, address=0)

bool3 TableRow(was\_described=True, identifier\_type='$', number=9, address=0)

+---------+

| SUCCESS |

+---------+

В листингах 4.5 и 4.7 отображены программы, в которых произойдут ошибки на этапе лексического анализа и листинги 4.6 и 4.8, в которых видно результат работы программы для данных программ соответсвенно

*Листинг 4.5 – Пример программы, завершающейся ошибкой*

program var  
{Объявляем переменные}  
int1, int2, int3: %;  
float1, float2, float3: !;  
bool1,bool2, bool3: $;  
  
2begin  
{оператор присваивания}  
int1 as 22  
;  
{Оператор вывода}  
write(int2, int3+4\*10>20)  
  
end@

*Листинг 4.6 – Результат работы программы*

Exception:

Unknown: '2be' in file first\_program.poullang

line: 7 and pos: 3

*Листинг 4.7 – Пример программы, завершающейся ошибкой*

program var  
{Объявляем переменные}  
int1, int2, int3: %;  
float1, float2, float3: !;  
bool1,bool2, bool3: $;  
  
begin  
float2 as 1.1  
;  
{Оператор условного цикла}  
whale float2 <= 100  
do  
 [float2 as float2\*1.3 : bool1 as float2>4 : if bool1 then write(1) else write(0)]  
end@

*Листинг 4.8 – Результат работы программы*

Exception:

Error in lexeme: 'float2'

Предоставим в листингах 4.9, 4.11 программы, в которых будут происходить ошибки на этапе синтаксического анализа и в листингах 4.10, 4.12 вывод для программ соответственно

*Листинг 4.9 – Пример программы, завершающейся ошибкой*

program var  
{Объявляем переменные}  
int1, int2, int3: %;  
float1, float2, float3: !;  
bool1,bool2, bool3: $;  
  
begin  
{Оператор фиксированного цикла}  
for float1 as 22.34 to float1<50  
do  
 [float1 as float1+10.3 : write(1, float1) : float1 as float1+2.1e+1] {составной оператор}  
;  
float2 as 1.1  
;  
{Оператор условного цикла}  
while do do float2 <= 100  
do  
 [float2 as float2\*1.3 : bool1 as float2>4 : if bool1 then write(1) else write(0)]  
end@

*Листинг 4.10 – Результат работы программы*

Exception:

Error in lexeme: 'do'

*Листинг 4.11 – Пример программы, завершающейся ошибкой*

program var  
{Объявляем переменные}  
int1, int2, int3: %;  
float1, float2, float3: !;  
bool1,bool2, bool3: $;  
  
begin  
{Оператор фиксированного цикла}  
for float1 as 22.34 to float1<50  
do  
 [float1 as float1+10.3 : write(1, float1) : float1 as float1+2.1e+1] {составной оператор}  
;  
float2 as 1.1  
;  
{Оператор условного цикла}  
while float2 <= 100  
do  
 (float2 as float2\*1.3 ; bool1 as float2>4 ; if bool1 then write(1) else write(0))  
end@

*Листинг 4.12 – Результат работы программы*

Exception:

Error in lexeme: '('

И, наконец, можем увидеть примеры, в которых произойдет ошибка на этапе семантического анализа в листингах 4.13 и 4.15, и результаты вывода для данных программ в листингах 4.14, 4.16

*Листинг 4.13 – Пример программы, завершающейся ошибкой*

program var  
{Объявляем переменные}  
bool1,bool2, bool3: $;  
  
begin  
bool1 as true;  
bool2 as false;  
write(bool1, bool2, bool4)  
  
end@

*Листинг 4.14 – Результат работы программы*

Exception:

Identifier 'bool4' error

*Листинг 4.15 – Пример программы, завершающейся ошибкой*

program var  
{Объявляем переменные}  
bool1,bool2, bool3: $;  
bool3: %;  
begin  
bool1 as true;  
bool2 as false;  
write(bool1, bool2, bool3)  
  
end@

*Листинг 4.16 – Результат работы программы*

Exception:

Identifier 'bool3' error

ЗАКЛЮЧЕНИЕ

В процессе выполнения курсовой работы были изучены различные способы описания синтаксиса языков программирования: формальные грамматики, формы Бэкуса-Наура, а также диаграммы Вирта. Была рассмотрена общая схема работы распознавателя, классификация распознавателей. Были изучены методы построения лексического анализатора, синтаксического анализатора и семантического анализатора программы. Разработаны и протестированы на языке программирования Python лексический, синтаксический и семантический анализаторы.

В итоге была успешно выполнена разработка распознавателя модельного языка программирования

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ 19 Единая система программной документации.
2. Методическое пособие студента для выполнения практических заданий, контрольных и курсовых работ по дисциплине «Теория формальных языков» [Электронный ресурс] – URL: <https://online-edu.mirea.ru/mod/resource/view.php?id=498415>
3. Статья «Объясняем бабушке, как написать свой язык программирования» [Электронный ресурс] – URL: <https://habr.com/ru/companies/edison/articles/315068/>
4. Статья «Python RegEx: практическое применение регулярок» [Электронный ресурс] – URL: <https://tproger.ru/translations/regular-expression-python>

Приложение A

**Код программы**

*Листинг A.1 – Вспомогательые классы лексического анализатора*

from typing import NamedTuple  
  
  
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

*Листинг A.2 – Вспомогательные методы для лексического анализатора*

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  
  
  
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

*Листинг A.3 – Код класса лексического анализатора*

import re  
from utils import \*  
  
  
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 = {"or": 1, "and": 2, "not": 3, "program": 4, "var": 5, "begin": 6, "end": 7, "as": 8, "if": 9,  
 "then": 10, "else": 11, "for": 12, "to": 13, "do": 14, "while": 15, "read": 16, "write": 17,  
 "true": 18, "false": 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.COMM:  
 self.comm\_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(): *# переход в состояние идентификаторов* self.current.state = self.states.ID  
 elif self.current.symbol in set(list("0123456789.")): *# переход в состояние чисел* self.current.state = self.states.NM  
 elif self.current.symbol in (self.delimiters | self.operators | self.types | self.arith):  
 self.current.state = self.states.DLM  
 elif self.current.symbol == "{":  
 self.current.state = self.states.COMM  
 else:  
 self.current.state = self.states.ERR  
  
 def comm\_state\_processing(self):  
 while not self.current.eof\_state and self.current.symbol != "}":

*Листинг A.3 (Продолжение)*

self.current.re\_assign(\*next(self.fgetc))  
 if self.current.symbol == "}":  
 self.current.state = self.states.H  
 if not self.current.eof\_state:  
 self.current.re\_assign(\*next(self.fgetc))  
 else:  
 self.error.symbol = self.current.symbol  
 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 in self.delimiters:  
 self.add\_token(self.token\_names.DELIM, self.current.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): *# Completed* 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()): *# ([a-zA-Z]|[0-9])+* 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:  
 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))

*Листинг A.3 (Продолжение)*

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))

*Листинг A.4 – Код класса синтаксического анализатора*

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 = {"+", "-", "or"}  
 self.factor\_operations = {"\*", "/", "and"}  
 self.keywords = {"or": 1, "and": 2, "not": 3, "program": 4, "var": 5, "begin": 6, "end": 7, "as": 8, "if": 9,  
 "then": 10, "else": 11, "for": 12, "to": 13, "do": 14, "while": 15, "read": 16, "write": 17,  
 "true": 18, "false": 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":  
 self.throw\_error()  
  
 *# todo проверить случай, если после end ещё есть какие-то символы* def DESCRIPTION(self):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)  
 self.equal\_token\_value(":")  
  
 self.TYPE(from\_description=True)  
 self.equal\_token\_value(";")  
  
 def IDENTIFIER(self, from\_description=False):

*Листинг A.4 (Продолжение)*

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 == "[":  
 self.COMPOSITE\_OPERATOR()  
 elif self.current\_lex.token\_value == "if":  
 self.CONDITIONAL\_OPERATOR()  
 elif self.current\_lex.token\_value == "for":  
 self.FIXED\_CYCLE\_OPERATOR()  
 elif self.current\_lex.token\_value == "while":  
 self.CONDITIONAL\_CYCLE\_OPERATOR()  
 elif self.current\_lex.token\_value == "read":  
 self.INPUT\_OPERATOR()  
 elif self.current\_lex.token\_value == "write":  
 self.OUTPUT\_OPERATOR()  
 else:  
 self.ASSIGNMENT\_OPERATOR()  
 def COMPOSITE\_OPERATOR(self):  
 self.equal\_token\_value("[")  
 self.OPERATOR()  
 while self.current\_lex.token\_value in {"\n", ":"}:  
 self.current\_lex = next(self.lex\_get)  
 self.OPERATOR()  
 self.equal\_token\_value("]")  
 def CONDITIONAL\_OPERATOR(self):  
 self.equal\_token\_value("if")  
 self.EXPRESSION()  
 self.equal\_token\_value("then")  
 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()  
 self.equal\_token\_value("to")  
 self.EXPRESSION()  
 self.equal\_token\_value("do")  
 self.OPERATOR()  
 def CONDITIONAL\_CYCLE\_OPERATOR(self):  
 self.equal\_token\_value("while")  
 self.EXPRESSION()  
 self.equal\_token\_value("do")

*Листинг A.4 (Продолжение)*

self.OPERATOR()  
  
 def INPUT\_OPERATOR(self):  
 self.equal\_token\_value("read")  
 self.equal\_token\_value("(")  
 self.IDENTIFIER()  
 while self.current\_lex.token\_value == ",":  
 self.current\_lex = next(self.lex\_get)  
 self.IDENTIFIER()  
 self.equal\_token\_value(")")  
  
 def OUTPUT\_OPERATOR(self):  
 self.equal\_token\_value("write")  
 self.equal\_token\_value("(")  
 self.EXPRESSION()  
 while self.current\_lex.token\_value == ",":  
 self.current\_lex = next(self.lex\_get)  
 self.EXPRESSION()  
 self.equal\_token\_value(")")  
 def ASSIGNMENT\_OPERATOR(self):  
 self.IDENTIFIER()  
 self.equal\_token\_value("as")  
 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()  
 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 == "not": self.equal\_token\_value("not")  
 self.FACTOR()  
 else: self.equal\_token\_value("(")  
 self.EXPRESSION()  
 self.equal\_token\_value(")")

*Листинг A.5 – Класс для таблицы идентификаторов*

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)

*Листинг A.6 – Основная программа*

from LexicalAnalizer import LexicalAnalyzer  
from SyntacticalAnalyzer import SyntacticalAnalyzer  
from SemanticalAnalyzer import IdentifiersTable  
  
PRINT\_INFO = True  
PATH\_TO\_PROGRAM = "second\_program.poullang"  
  
  
def main():  
 identifiersTable = IdentifiersTable()  
 lexer = LexicalAnalyzer(PATH\_TO\_PROGRAM, identifiersTable)  
 lexer.analysis()  
 if lexer.current.state != lexer.states.ERR:  
 if PRINT\_INFO:  
 print("Result of Lexical Analyzer:")  
 for i in lexer.lexeme\_table:  
 print(f"{i.token\_name} {i.token\_value}")  
  
  
 syntaxAnalyzer = SyntacticalAnalyzer(lexer.lexeme\_table, identifiersTable)  
 syntaxAnalyzer.PROGRAMM()  
 identifiersTable.check\_if\_all\_described() *# проверка что все Id описаны* if PRINT\_INFO:  
 print(identifiersTable)  
 print("+---------+")  
 print("| SUCCESS |")  
 print("+---------+")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()