СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Невозможно недооценивать важность языков программирования в современном мире, когда всем правит техника и технологии необходимо иметь удобный инструмент для того, чтобы уметь их контролировать и управлять ими. 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 сообщать команды бездушной машине и которые будут однозначно восприняты ею. Для такого и существуют различные распознаватели языков (компиляторы).

Осознавая важность этого перед нами ставится цель – разработать распознаватель модельного языка программирования, включающий 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:

a) РБНФ;

b) диаграмм Вирта;

c) формальных грамматик.

2) Написать пять содержательных примеров программ, раскрывающих особенности конструкций модельного языка программирования, отразив в этих примерах все его функциональные возможности.

3) Составить таблицы лексем и диаграмму состояний с действиями для распознавания и формирования лексем языка.

4) По диаграмме с действиями написать функцию сканирования текста входной программы на модельном языке.

5) Разработать программное средство, реализующее лексический анализ текста программы на входном языке.

6) Реализовать синтаксический анализатор текста программы на модельном языке методом рекурсивного спуска.

7) Построить цепочку вывода и дерево разбора простейшей программы на модельном языке из начального символа грамматики.

8) Дополнить синтаксический анализатор процедурами проверки семантической правильности программы на модельном языке в соответствии с контекстными условиями вашего варианта.

9) Распечатать пример таблиц идентификаторов и двуместных операций.

10) Показать динамику изменения содержимого стека при семантическом анализе программы на примере одного синтаксически правильного выражения.

11) Составить набор контрольных примеров, демонстрирующих все возможные типы лексических, синтаксических и семантических ошибок в программах на модельном языке.

2. ФОРМАЛЬНАЯ МОДЕЛЬ ЗАДАЧИ

2.1. РБНФ

В листинге 2.1 представлено описание модели языка с помощью расширенной формы Бэкуса-Наура.

Листинг 2.1 – Описание модели языка с помощью 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>
```

2.2. Диаграмма Вирта

На Рисунках 2.1 и 2.2 представлено описание модели языка с помощью диаграмм Вирта.

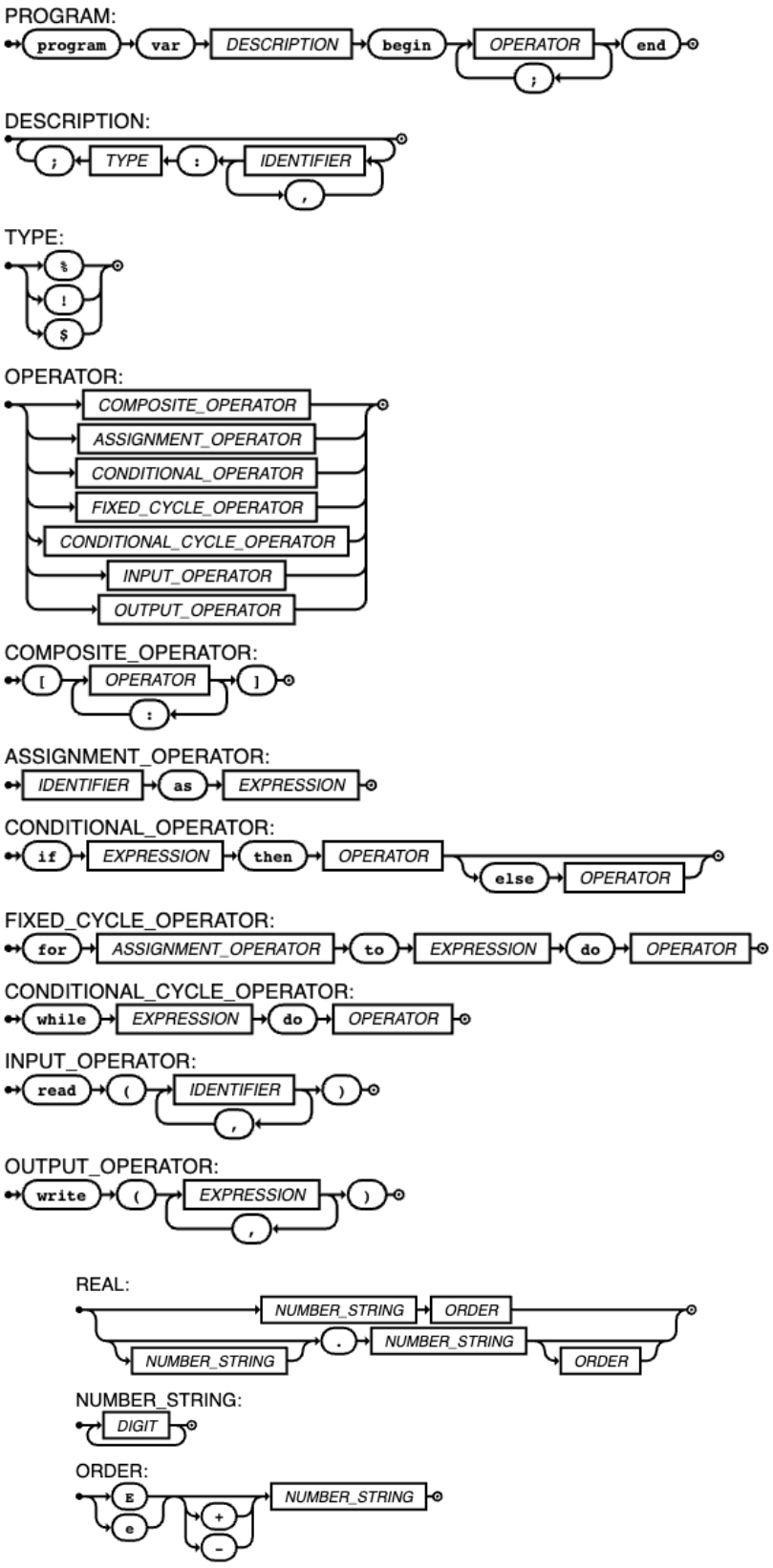

Рисунок 2.1 – Описание языка с помощью диаграмм Вирта

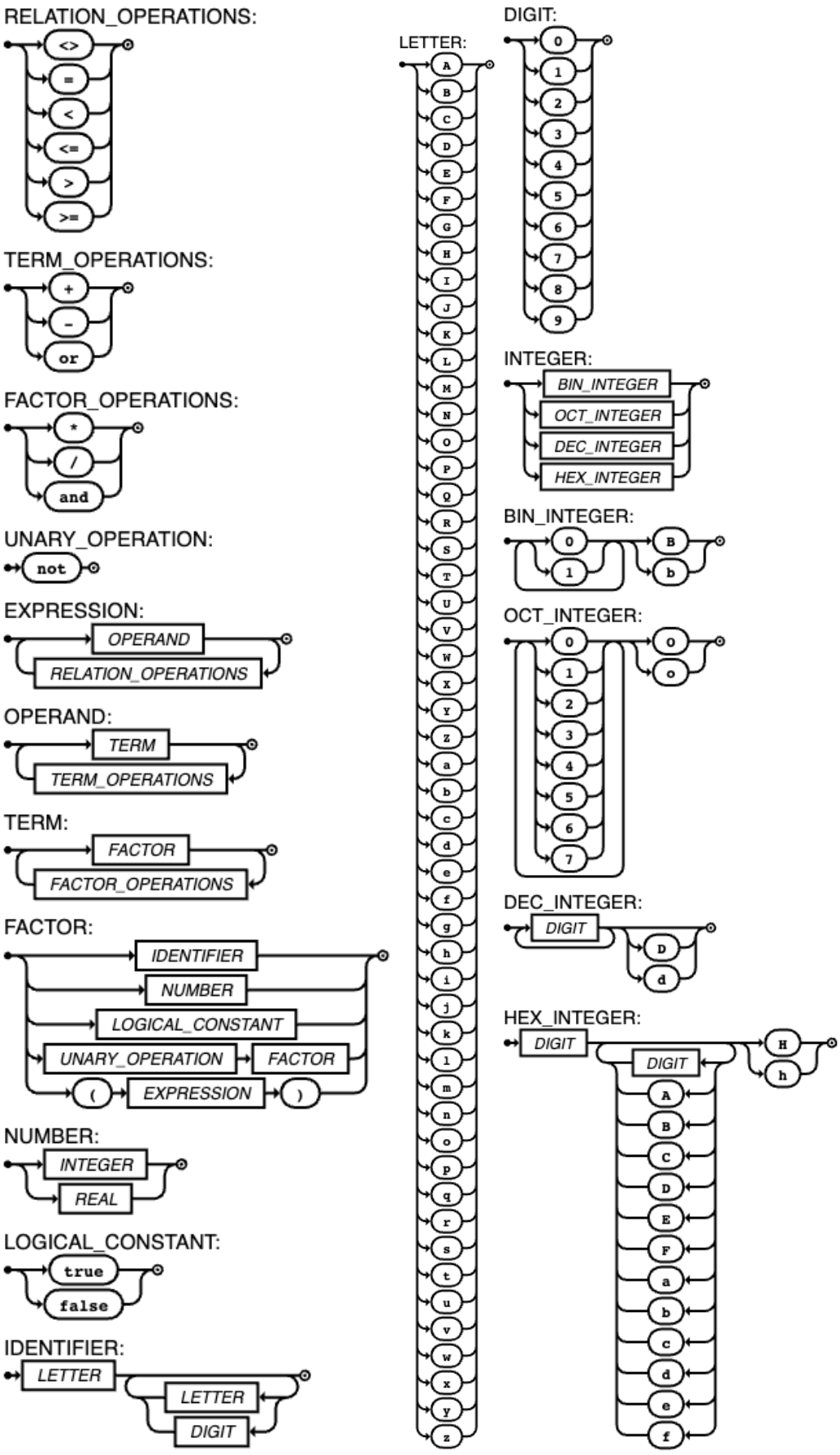

Рисунок 2.2 – Описание языка с помощью диаграмм Вирта

2.3. Формальные грамматики

В Листинге 2.2 представлено описание модели языка с помощью формальных грамматик.

Листинг 2.2 – Описание модели языка с помощью формальных грамматик

```
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
```

Листинг 2.2 (Продолжение)

```
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
```

3. СПЕЦИФИКАЦИЯ ОСНОВНЫХ ФУНКЦИЙ

3.1. Лексический анализатор

Задача лексического анализа - выделить лексемы и преобразовать их к виду, удобному для последующей обработки. ЛА использует регулярные грамматики.

Была составлена диаграмма состояний с действиями для модельного языка на Рисунке 3.1:

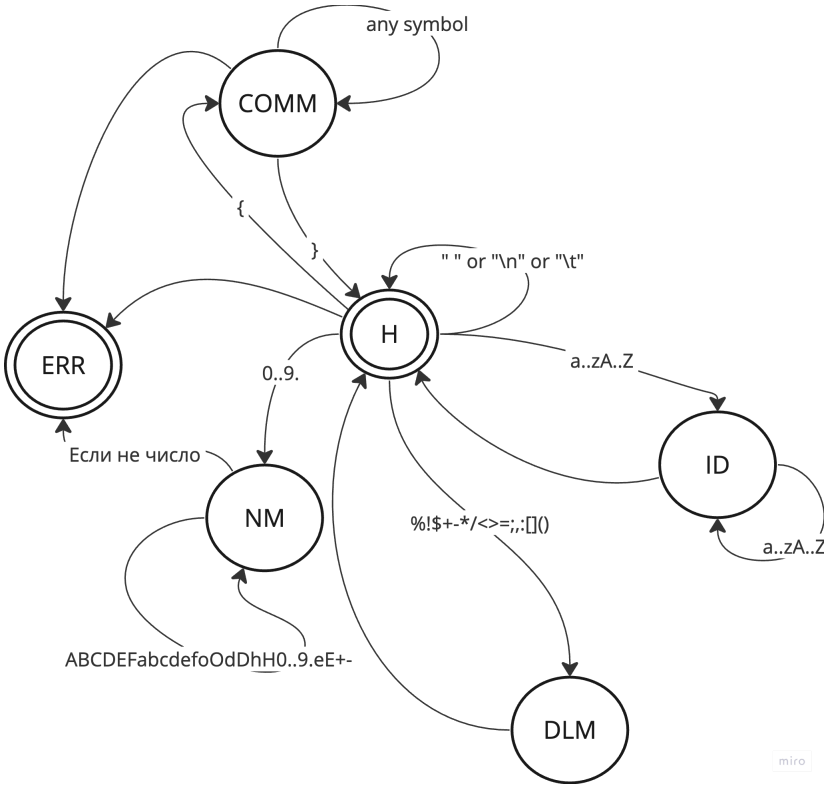

Рисунок 3.1 – Упрощенная диаграмма состояний для языка

В Листинге А.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 удобства и оптимизации разработки анализатора.

В Листинге А.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 А.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

3.2. Синтаксический анализатор

Для решения задачи синтаксического анализа программы используется метод рекурсивного спуска. В его основе лежит левосторонний разбор строки языка. Исходной сентенциальной формой является начальный символ грамматики, а целевой – заданная строка языка. На каждом шаге разбора правило грамматики применяется к самому левому нетерминалу сентенции. Данный процесс соответствует *построению дерева разбора* цепочки сверху вниз (от корня к листьям).

В Листинге А.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 операторов

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

3.3. 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 А.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

3.4. Основная программа

В основной программе, приведенной в Листинге А.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.

4. 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             DELIM ;                   KWORD as
KWORD var                 KWORD begin               NUM10 100
IDENT int1                IDENT int1                DELIM ;
DELIM ,                   KWORD as                  KWORD read
IDENT int2                NUM10 22                  DELIM (
DELIM ,                   DELIM ;                   IDENT int2
IDENT int3                KWORD if                  DELIM ,
DELIM :                   IDENT int1                IDENT int3
TYPE %                    OPER <                    DELIM )
DELIM ;                   NUM10 50                  DELIM ;
IDENT float1              KWORD then                KWORD write
DELIM ,                   IDENT int1                DELIM (
IDENT float2              KWORD as                  IDENT int2
DELIM ,                   NUM10 50                  DELIM ,
IDENT float3              KWORD else                IDENT int3
DELIM :                   IDENT int1                ARITH +
TYPE !                    KWORD as                  NUM10 4
DELIM ;                   NUM10 100                 ARITH *
IDENT bool1               DELIM ;                   NUM10 10
DELIM ,                   KWORD if                  OPER >
IDENT bool2               IDENT int1                NUM10 20
DELIM ,                   OPER =                    DELIM )
IDENT bool3               NUM10 50                  KWORD end
DELIM :                   KWORD then
TYPE $                    IDENT int1



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 to                IDENT float2
KWORD var               IDENT float1            OPER <=
IDENT int1              OPER <                  NUM10 100
DELIM ,                 NUM10 50                KWORD do
IDENT int2              KWORD do                DELIM [
DELIM ,                 DELIM [                 IDENT float2
IDENT int3              IDENT float1            KWORD as
DELIM :                 KWORD as                IDENT float2
TYPE %                  IDENT float1            ARITH *
DELIM ;                 ARITH +                 REAL 1.3
IDENT float1            REAL 10.3               DELIM :
DELIM ,                 DELIM :                 IDENT bool1
IDENT float2            KWORD write             KWORD as
DELIM ,                 DELIM (                 IDENT float2
IDENT float3            NUM10 1                 OPER >
DELIM :                 DELIM ,                 NUM10 4
TYPE !                  IDENT float1            DELIM :
DELIM ;                 DELIM )                 KWORD if
IDENT bool1             DELIM :                 IDENT bool1
DELIM ,                 IDENT float1            KWORD then
IDENT bool2             KWORD as                KWORD write
DELIM ,                 IDENT float1            DELIM (
IDENT bool3             ARITH +                 NUM10 1
DELIM :                 REAL 2.1e+1             DELIM )
TYPE $                  DELIM ]                 KWORD else
DELIM ;                 DELIM ;                 KWORD write
KWORD begin             IDENT float2            DELIM (
KWORD for               KWORD as                NUM10 0
IDENT float1            REAL 1.1                DELIM )
KWORD as                DELIM ;                 DELIM ]
REAL 22.34              KWORD while             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 соответственно.

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'
```

В Листингах 4.9, 4.11 представлены примеры программ, 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: '('
```

Примеры, в которых произойдет ошибка на этапе семантического анализа, представлены 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

5. Ахо, А.В. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Д. Ульман; перевод с англ. И.В. Красикова и др. - М.: Вильямс, 2001. - 767 с.: ил.; 24 см. - Библиогр.: с. 742-763. - Предм. указ.: 764-767. - 5000 экз. - ISBN 5-8459- 0189-8 (в пер.).

6. Власенко, А.В. Теория языков программирования и методы трансляции: учеб. пособие / А.В. Власенко, В.И. Ключко; М-во образования и науки РФ, ГОУ ВПО «Кубан. гос. технол. ун-т». - Краснодар: Изд-во КубГТУ, 2004. - 119 с.: ил.; 21 см. - Библиогр.: с. 118. - 75 экз. - ISBN 5-8333-0176-9.

7. Гавриков, М.М. Основы конструирования компиляторов: учеб. пособие / М.М. Гавриков, А.Н. Иванченко, Д.В. Гринченков; М-во общ. и проф. образования РФ, Новочеркас. гос. техн. ун-т. – Новочеркасск: НГТУ, 1997. - 80 с.: ил.; 20 см. - Библиогр.: с. 79. – 75 экз. - ISBN 5-88998-059-9.

8. Гордеев, А.В. Системное программное обеспечение: учеб. для вузов / А. Ю. Молчанов. - 3-е изд. - СПб.: Питер, 2010. - 398 с.: ил. - (Учебник для вузов). - Указ. лит.: с. 387-390. - Алф. указ.: с. 391-397. - ISBN 978-5-49807-153-4.

Приложение А

Код программы

Листинг А.1 – Вспомогательные 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 А.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 А.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 А.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 А.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 А.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 А.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 А.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 А.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 А.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()
```