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 1 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>::= + | - | or

<операции_группы_умножения>::= * | / | and

<унарная_операция>::= not

<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> { (: | перевод строки) <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> } «)»

{ … }

<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

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.

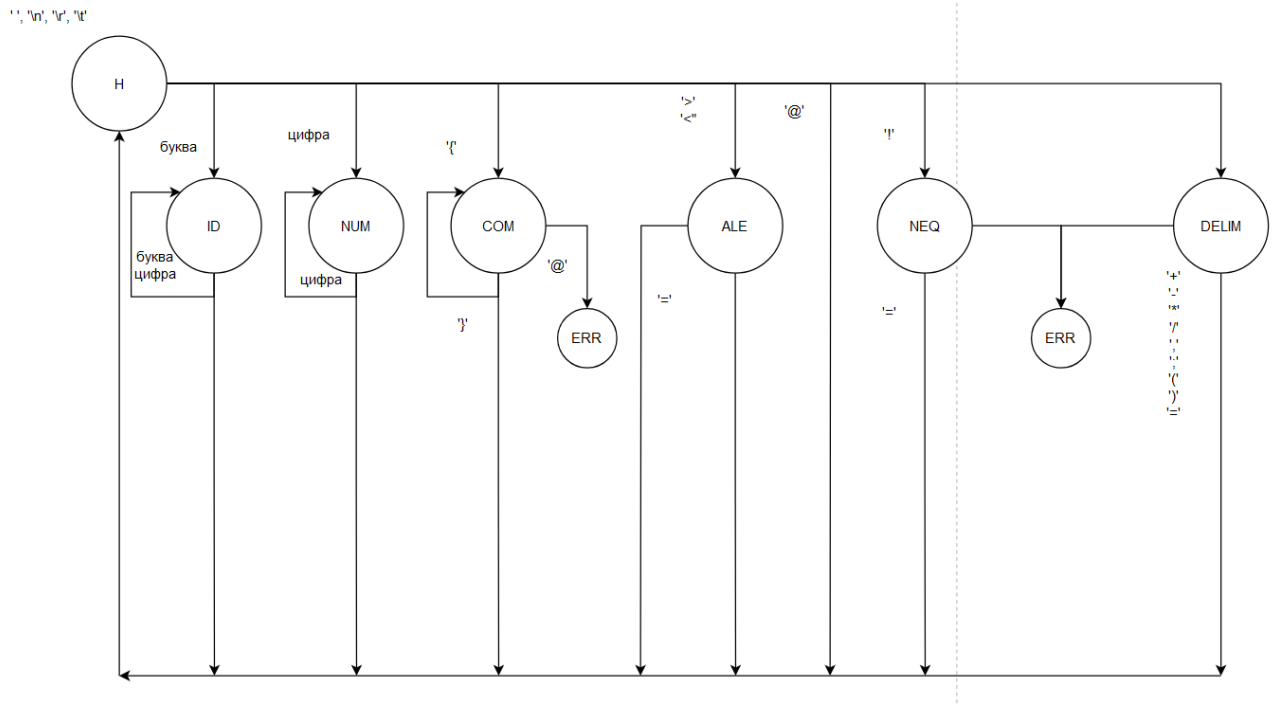

Рисунок 1 - Диаграмма состояний лексического анализатора

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:

$P \rightarrow \boldsymbol{program}$

$D1 \rightarrow \boldsymbol{var}\ D\ \{, D\}$

$D \rightarrow I\ \{, I\}: [\boldsymbol{int}|\boldsymbol{bool}]$

$B \rightarrow \boldsymbol{begin}\ S\ \{; S\}\ \boldsymbol{end}$

$S \rightarrow I \coloneqq E|\ \boldsymbol{if}\ E\ \boldsymbol{then}\ S\ \boldsymbol{else}\ S \mid \boldsymbol{while}\ E\ \boldsymbol{do}\ S \mid B \mid \boldsymbol{read}(I) \mid \boldsymbol{write}(E)$

$E \rightarrow E1\{[= \mid > \mid < \mid >= \mid <=]E1\}$

$E1 \rightarrow T\{[+ \mid - \mid \boldsymbol{or}\ ]\ T\}$

$T \rightarrow F\{[* \mid / \mid \boldsymbol{and}\ ]\ F\}$

$F \rightarrow I \mid N \mid L|\ \boldsymbol{not}\ F \mid (E)$

$L \rightarrow \boldsymbol{true} \mid \boldsymbol{false}$

$I \rightarrow C \mid IC \mid IR$

$N \rightarrow R \mid NR$

$C \rightarrow a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z$

$R \rightarrow 0 \mid 1 \mid \ldots \mid 9$

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 TID заносятся данные обо всех лексемах-идентификаторах, которые встречаются в тексте программы. На этапе синтаксического анализа в ту же таблицу заносятся данные о типе идентификатора (поле type) и о наличии для него описания (поле declared).

С учетом сказанного, правила вывода для нетерминала D (раздел описаний) принимают вид:

D→stack.reset() I stack.push(c_val) {, I stack.push(c_val)} : [int dec(LEX_INT)| bool dec(LEX_BOOL)]

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 Б.

Исходный код семантического анализатора приведен в Приложении В.

7 ТЕСТИРОВАНИЕ

В качестве программного продукта разработано консольное приложение lexpars.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. Исходный код программы приведен в листинге 1.

Листинг А.1 – main.py

```
program
var x, y : %;
begin
  x := 5;  { Присваиваем x значение 5 }
  y := 10; { Присваиваем y значение 10 }

  if x < y then [
    write (x);
    write (y);
  ]
  else [
    write (y);
    write (x);
  ]
end.
```

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

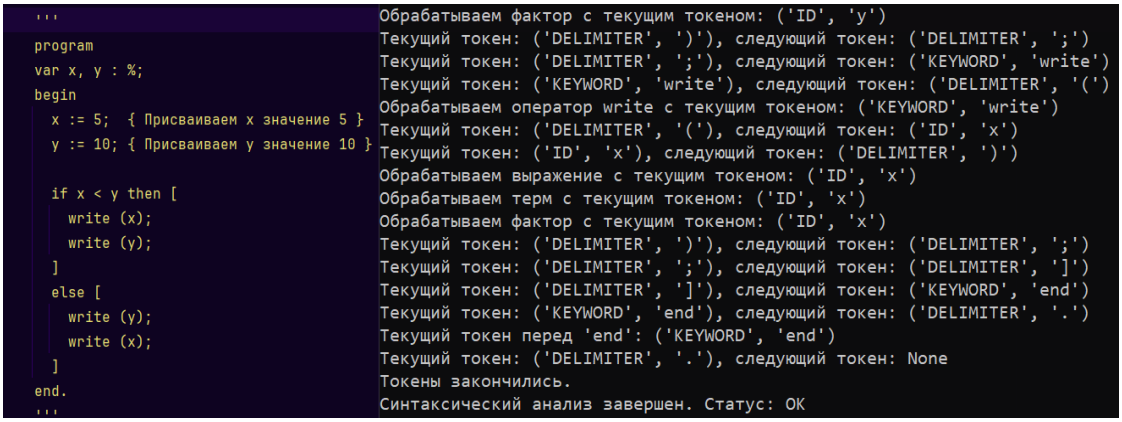

Рисунок 2 - Пример синтаксически корректной программы

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

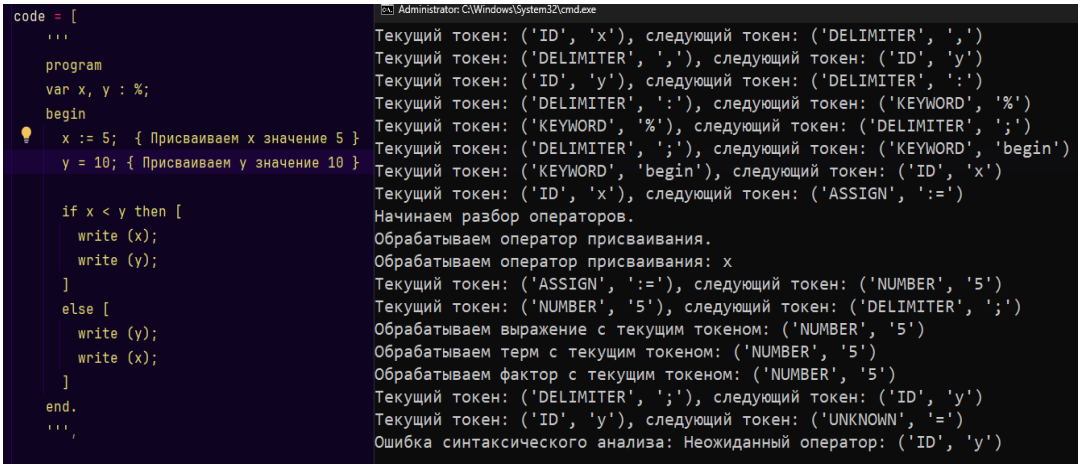

Рисунок 3 - Пример программы, содержащей ошибку

Здесь ошибка допущена в строке 5: неправильное использование оператора сравнения (=).

3. Исходный текст программы, содержащей семантическую проверку, приведен на рис. 4 вместе с сообщением об ошибке. Здесь переменная d не объявлена.

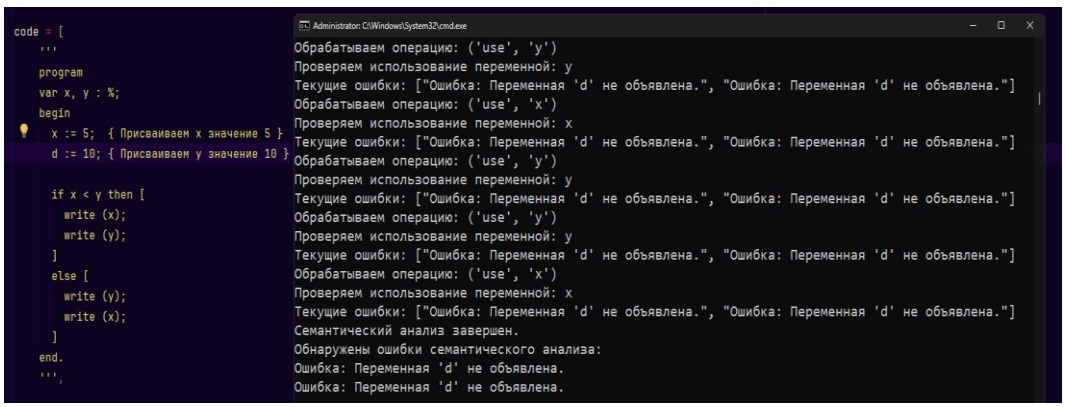

Рисунок 4 - Пример программы, содержащей семантическую ошибку

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 уровня Python в виде класса LexicalAnalyzer.

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

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

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

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

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Свердлов С. З. Языки программирования и методы трансляции: учебное пособие. – Санкт-Петербург: Лань, 2019.

2. Малявко А. А. Формальные языки и компиляторы: учебное пособие для вузов. – М.: Юрайт, 2020.

3. Миронов С. В. Формальные языки и грамматики: учебное пособие для студентов факультета компьютерных наук и информационных технологий. – Саратов: СГУ, 2019.

4. Унгер А.Ю. Основы теории трансляции: учебник. – М.: МИРЭА – Российский технологический университет, 2022.

5. Антик М. И., Казанцева Л. В. Теория формальных языков в проектировании трансляторов: учебное пособие. – М.: МИРЭА, 2020.

6. Ахо А. В., Лам М. С., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструментарий. – М.: Вильямс, 2008.

7. Ишакова Е.Н. Теория языков программирования и методов трансляции: учебное пособие. – Оренбург: ИПК ГОУ ОГУ, 2007.

ПРИЛОЖЕНИЯ

Приложение А – Класс лексического анализатора

Приложение Б – Класс синтаксического анализатора

Приложение В – Класс семантического анализатора

ПРИЛОЖЕНИЕ А

Класс лексического анализатора

Листинг А.1 – lexer.py

```
class LexicalAnalyzer:
    class State(Enum):
        ID = "ID"  # Идентификаторы
        NUM = "NUM"  # Числа
        COM = "COM"  # Комментарии
        ALE = "ALE"  # Операции отношения
        NEQ = "NEQ"  # Неравенство
        DELIM = "DELIM"  # Разделители
        STR = "STR"  # Строковые литералы

    # Ключевые слова
    TW = [
        "program", "var", "begin", "end", "if", "else",
"while", "for", "to", "then", "next", "as",
        "readln", "write", "true", "false", "%", "!", "$",
"end_else", "real", "integer", "boolean"
    ]

    # Разделители и операторы
    TD = [
        "[", "]", "{", "}", "(", ")", ",", ":", ";", ":=",
".", "+", "-", "*", "/", "and", "/", "not",
        "!=", "==", "<", "<=", ">", ">="
    ]

    def __init__(self, input_text):
        self.text = input_text
        self.pos = 0
        self.current_char = self.text[self.pos] if self.text
else None
        self.tokens = []
        self.before_begin = True

    def advance(self):
        self.pos += 1
        self.current_char = self.text[self.pos] if self.pos <
len(self.text) else None

    def add_token(self, type_, value):
        self.tokens.append((type_, value))
```

Продолжение листинга А.1

```
def clear_whitespace(self):
        while self.current_char and self.current_char in '
\n\r\t':
            self.advance()

    def parse_identifier_or_keyword(self):
        start = self.pos
        while self.current_char and
(self.current_char.isalnum() or self.current_char == '_'):
            self.advance()
        text = self.text[start:self.pos]
        if text in self.TW:
            self.add_token('KEYWORD', text, )
            if text == 'begin':
                self.before_begin = False
        else:
            self.add_token('ID', text, )

    def parse_number(self):
        start = self.pos
        base_detected = False
        if self.current_char == '0':
            self.advance()
            if self.current_char in 'Bb':
                base_detected = True
                self.advance()
                while self.current_char and self.current_char
in '01':
                    self.advance()
            elif self.current_char in 'Oo':
                base_detected = True
                self.advance()
                while self.current_char and self.current_char
in '01234567':
                    self.advance()
            elif self.current_char in 'Xx':
                base_detected = True
                self.advance()
                while self.current_char and
(self.current_char.isdigit() or self.current_char.upper() in
'ABCDEF'):
                    self.advance()
        if not base_detected:
            while self.current_char and
self.current_char.isdigit():
                self.advance()
```

Продолжение листинга А.1

```
if self.current_char == '.':
                self.advance()
                while self.current_char and
self.current_char.isdigit():
                    self.advance()
            if self.current_char and self.current_char.upper()
== 'E':
                self.advance()
                if self.current_char in '+-':
                    self.advance()
                if self.current_char and
self.current_char.isdigit():
                    while self.current_char and
self.current_char.isdigit():
                        self.advance()
                else:
                    self.add_token('ERROR',
self.text[start:self.pos], )
                    return
        if self.current_char in 'bohBOH':
            suffix = self.current_char.lower()
            self.advance()
            self.add_token('NUMBER', self.text[start:self.pos]
+ suffix, )
        else:
            text = self.text[start:self.pos]
            self.add_token('NUMBER', text, )

    def parse_string(self):
        self.advance()
        start = self.pos
        while self.current_char and self.current_char != "'":
            self.advance()
        text = self.text[start:self.pos]
        self.add_token('STRING', f"'{text}'", )
        self.advance()

    def parse_comment(self):
        self.advance()
        while self.current_char and self.current_char != '}':
            self.advance()
        self.advance()

    def parse_delimiter_or_operator(self):
        start = self.pos
        self.advance()
        if self.current_char and (self.text[start:self.pos +
1] in self.TD):
            self.advance()
```

Продолжение листинга А.1

```
text = self.text[start:self.pos]
        if text in ["==", "!=", "<", "<=", ">", ">="]:
            self.add_token('REL_OP', text, )
        elif text in ["+", "-", "||"]:
            self.add_token('ADD_OP', text, )
        elif text in ["*", "/", "&&"]:
            self.add_token('MUL_OP', text, )
        elif text == ":=":
            self.add_token('ASSIGN', text, )
        elif text in self.TD:
            self.add_token('DELIMITER', text, )
        else:
            self.add_token('UNKNOWN', text, )

    def tokenize(self):
        while self.current_char:
            self.clear_whitespace()
            if not self.current_char:
                break
            if self.current_char.isalpha():
                self.parse_identifier_or_keyword()
            elif self.current_char.isdigit():
                self.parse_number()
            elif self.current_char == "'":
                self.parse_string()
            elif self.current_char == '{':
                self.parse_comment()
            elif self.current_char == '!':
                if self.text[self.pos:self.pos + 2] == "!=":
                    self.add_token('REL_OP', '!=', )
                    self.advance()
                    self.advance()
                else:
                    if self.before_begin:
                        self.add_token('KEYWORD', '!', )
                    else:
                        self.add_token('DELIMITER', '!', )
                    self.advance()
            elif self.current_char in "%$":
                self.add_token('KEYWORD', self.current_char, )
                self.advance()
            elif self.current_char in self.TD:
                self.parse_delimiter_or_operator()
            else:
                self.add_token('UNKNOWN', self.current_char, )
                self.advance()
        return self.tokens
```

ПРИЛОЖЕНИЕ Б

Класс синтаксического анализатора

Листинг Б.1 – parserr.py

```
class SyntaxAnalyzer:
    def __init__(self, tokens):
        self.tokens = deque(tokens)
        self.current_token = None
        self.next_token()

    def next_token(self):
        if self.tokens:
            self.current_token = self.tokens.popleft()
            next_token = self.tokens[0] if self.tokens else
None
            print(f"Текущий токен: {self.current_token},
следующий токен: {next_token}")
            if self.current_token[0]=="NUMBER" and
next_token[0]=="ID":
                raise SyntaxError(f"Неожиданный оператор:
{self.current_token[1]}{next_token[1]}")

        else:
            self.current_token = None
            print("Токены закончились.")

    def parse(self):
        print("Начинаем синтаксический анализ...")
        self.program()
        if self.current_token is not None:
            raise SyntaxError(f"Неожиданный токен:
{self.current_token}")
        return "OK"

    def program(self):
        print(f"Начинаем разбор программы, текущий токен:
{self.current_token}")
        if self.current_token[0] == 'KEYWORD' and
self.current_token[1] == 'program':
            self.next_token()
            self.block()
        else:
            raise SyntaxError("Ожидалось 'program'.")
```

Продолжение листинга Б.1

```
def block(self):
    print(f"Проверка токена в блоке: {self.current_token}")
    if self.current_token[0] == 'KEYWORD' and
self.current_token[1] == 'var':
        self.variable_declarations()
    found_begin = False
    while self.current_token is not None:
        if self.current_token[0] == 'KEYWORD' and
self.current_token[1] == 'begin':
            found_begin = True
            self.next_token()
            break
        self.next_token()
    if not found_begin:
        raise SyntaxError("Ожидалось 'begin' после объявления
переменных.")
    self.statements()
    print(f"Текущий токен перед 'end': {self.current_token}")
    if self.current_token[0] == 'KEYWORD' and
self.current_token[1] == 'end':
        self.next_token()
        if self.current_token[0] == 'DELIMITER' and
self.current_token[1] == '.':
            self.next_token()
        else:
            raise SyntaxError("Ожидалась точка '.' после
'end'.")
    else:
        raise SyntaxError("Ожидалось 'end' после блока.")

def variable_declarations(self):
    print(f"Начинаем разбор объявлений переменных.")
    while self.current_token[0] == 'ID':
        print(f"Обрабатываем переменную:
{self.current_token[1]}")
        self.next_token()
        if self.current_token[0] == 'DELIMITER' and
self.current_token[1] == ',':
            self.next_token()
        elif self.current_token[0] == 'DELIMITER' and
self.current_token[1] == ';':
            self.next_token()
            break
        else:
            raise SyntaxError("Ожидалось ',' или ';' после
объявления переменной.")
```

Продолжение листинга Б.1

```
def statements(self):
    print(f"Начинаем разбор операторов.")
    while self.current_token is not None and
(self.current_token[0] != 'KEYWORD' or self.current_token[1]
!= 'end'):
        if self.current_token[0] == 'ID' and self.tokens and
self.tokens[0][1] == ':=':
            print("Обрабатываем оператор присваивания.")
            self.assignment_statement()
        elif self.current_token[0] == 'KEYWORD' and
self.current_token[1] == 'if':
            print("Обрабатываем условие 'if'.")
            self.if_statement()
        elif self.current_token[0] == 'KEYWORD' and
self.current_token[1] == 'while':
            print("Обрабатываем условие 'while'.")
            self.while_statement()
        else:
            raise SyntaxError(f"Неожиданный оператор:
{self.current_token}")

def assignment_statement(self):
    print(f"Обрабатываем оператор присваивания:
{self.current_token[1]}")
    var_name = self.current_token[1]
    self.next_token()
    if self.current_token[0] == 'ASSIGN' and
self.current_token[1] == ':=':
        self.next_token()
        self.expression()
        if self.current_token[0] == 'DELIMITER' and
self.current_token[1] == ';':
            self.next_token()
        else:
            raise SyntaxError("Ожидался символ ';' после
присваивания.")
    else:
        raise SyntaxError("Ожидалось ':=' в операторе
присваивания.")
```

Продолжение листинга Б.1

```
def expression(self):
    print(f"Обрабатываем выражение с текущим токеном:
{self.current_token}")
    self.term()
    while self.current_token is not None and
self.current_token[0] in ['ADD_OP', 'SUB_OP']:
        print(f"Обрабатываем операцию:
{self.current_token[1]}")
        self.next_token()
        self.term()

def term(self):
    print(f"Обрабатываем терм с текущим токеном:
{self.current_token}")
    self.factor()
    while self.current_token is not None and
self.current_token[0] in ['MUL_OP', 'DIV_OP']:
        print(f"Обрабатываем операцию:
{self.current_token[1]}")
        self.next_token()
        self.factor()

def factor(self):
    print(f"Обрабатываем фактор с текущим токеном:
{self.current_token}")
    if self.current_token[0] == 'ID' or self.current_token[0]
== 'NUMBER':
        self.next_token()
    else:
        raise SyntaxError(f"Неожиданный токен в факторе:
{self.current_token}")

def write_statement(self):
    print(f"Обрабатываем оператор write с текущим токеном:
{self.current_token}")
    self.next_token()
    if self.current_token[0] == 'DELIMITER' and
self.current_token[1] == '(':
        self.next_token()
        self.expression()
        if self.current_token[0] == 'DELIMITER' and
self.current_token[1] == ')':
            self.next_token()
        else:
            raise SyntaxError("Ожидалась закрывающая скобка
')' после аргумента write.")
    else:
        raise SyntaxError("Ожидалась открывающая скобка '('
после write.")
```

Продолжение листинга Б.1

```
def if_statement(self):
    print(f"Обрабатываем оператор 'if' с текущим токеном:
{self.current_token}")
    self.next_token()
    self.expression()
    if self.current_token[0] == 'REL_OP' and
self.current_token[1] in ['>', '<', '=', '>=', '<=']:
        self.next_token()
        self.next_token()
        if self.current_token[0] == 'KEYWORD' and
self.current_token[1] == 'then':
            self.next_token()
            if self.current_token[0] == 'DELIMITER' and
self.current_token[1] == '[':
                self.next_token()
                while self.current_token[0] != 'DELIMITER' or
self.current_token[1] != ']':
                    if self.current_token[0] == 'KEYWORD' and
self.current_token[1] == 'write':
                        self.write_statement()
                    elif self.current_token[0] == 'DELIMITER'
and self.current_token[1] == ';':
                        self.next_token()
                    else:
                        self.statements()
                if self.current_token[0] == 'DELIMITER' and
self.current_token[1] == ']':
                    self.next_token()
                else:
                    raise SyntaxError("Ожидался закрывающий
']' после блока операторов.")
            else:
                raise SyntaxError("Ожидался блок операторов
после 'then'.")
            if self.current_token[0] == 'KEYWORD' and
self.current_token[1] == 'else':
                self.next_token()
                if self.current_token[0] == 'DELIMITER' and
self.current_token[1] == '[':
                    self.next_token()
                    while self.current_token[0] != 'DELIMITER'
or self.current_token[1] != ']':
                        if self.current_token[0] == 'KEYWORD'
and self.current_token[1] == 'write':
                            self.write_statement()
                        elif self.current_token[0] ==
'DELIMITER' and self.current_token[1] == ';':
                            self.next_token()
```

Продолжение листинга Б.1

```
else:
                                self.statements()
                        if self.current_token[0] == 'DELIMITER'
and self.current_token[1] == ']':
                            self.next_token()
                        else:
                            raise SyntaxError("Ожидался
закрывающий ']' после блока операторов в 'else'.")
                    else:
                        self.statements()
            else:
                raise SyntaxError("Ожидалось 'then' после условия
'if'.")
        else:
            raise SyntaxError("Ожидался оператор сравнения после
условия 'if'.")

def while_statement(self):
    print(f"Обрабатываем оператор 'while' с текущим токеном:
{self.current_token}")
    self.next_token()
    self.expression()
    if self.current_token[0] == 'KEYWORD' and
self.current_token[1] == 'do':
        self.next_token()
        self.statements()
    else:
        raise SyntaxError("Ожидалось 'do' после условия
'while'.")
```

ПРИЛОЖЕНИЕ В

Класс синтаксического анализатора

Листинг В.1 – semantic.py

```
class SemanticAnalyzer:
    def __init__(self, symbol_table):
        self.symbol_table = symbol_table
        self.errors = []

    def analyze(self, operations):
        print("Начало семантического анализа...")
        for operation in operations:
            print(f"Обрабатываем операцию: {operation}")
            if operation[0] == 'assign':
                variable = operation[1]
                print(f"Проверяем переменную для
присваивания: {variable}")
                if variable not in self.symbol_table:
                    self.errors.append(f"Ошибка: Переменная
'{variable}' не объявлена.")
            elif operation[0] == 'use':
                variable = operation[1]
                print(f"Проверяем использование переменной:
{variable}")
                if variable not in self.symbol_table:
                    self.errors.append(f"Ошибка: Переменная
'{variable}' не объявлена.")
            else:
                self.errors.append(f"Ошибка: Неизвестная
операция '{operation[0]}'.")
            print(f"Текущие ошибки: {self.errors}")
        print("Семантический анализ завершен.")
        return self.errors

    def get_errors(self):
        return self.errors
```

Продолжение листинга В.1

```
def generate_symbol_table_and_operations(tokens):
    global global_type
    symbol_table = {}
    operations = []
    current_type = None
    in_var_section = False
    var_seen = False
    begin_seen = False

    for j, token in enumerate(tokens):
        print(f"Обрабатываем токен: {token}")
        if token[0] == 'KEYWORD':
            if token[1] in ['%', '!', '$']:
                current_type = token[1]
                global_type=token[1]
                print(current_type)

    for i, token in enumerate(tokens):
        print(f"Обрабатываем токен: {token}")
        if token[0] == 'KEYWORD':
            if token[1] == 'var':
                var_seen = True
                in_var_section = True
            elif token[1] in ['integer', 'real', 'boolean']
and var_seen and not begin_seen:
                current_type = token[1]
            elif token[1] == 'begin':
                begin_seen = True
                in_var_section = False
                current_type = None
        elif token[0] == 'ID' and var_seen and not
begin_seen:
            if token[1] not in symbol_table:
                symbol_table[token[1]] = {'type':
current_type, 'scope': 'global'}
                print(global_type)
                print(f"Переменная '{token[1]}' добавлена в
таблицу символов с типом '{current_type}'")
        elif token[0] == 'ID' and not in_var_section:
            if token[1] not in symbol_table:
                operations.append(('use', token[1]))
                print(f"Переменная '{token[1]}' используется,
но не найдена в таблице символов.")
            else:
                operations.append(('use', token[1]))
                print(f"Переменная '{token[1]}' используется
и найдена в таблице символов.")
```

Продолжение листинга В.1

```
elif token[0] == 'ASSIGN':
        if i > 0 and tokens[i - 1][0] == 'ID':
            var_name = tokens[i - 1][1]
            if var_name not in symbol_table:
                operations.append(('assign', var_name))
                print(f"Переменная '{var_name}' используется
для присваивания, но не найдена в таблице символов.")
            else:
                operations.append(('assign', var_name))
                print(f"Переменная '{var_name}' используется
для присваивания и найдена в таблице символов.")

for pencil, token in enumerate(tokens):
    if token[0]=="NUMBER":
        for i in token[1]:
            print(global_type)
            if i not in ["0", "1", "2", "3", "4", "5", "6",
"7", "8", "9", ".", "true", "false"]:
                print(i)
                raise SyntaxError(f"Неожиданное число:
{token[1]}")
            if i=="." and global_type!="!":
                raise SyntaxError(f"Неправильный тип:
{global_type}")

print(f"Таблица символов: {symbol_table}")
return symbol_table, operations
```