МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Направление 02.03.01 Математика и компьютерные науки

Отчет по дисциплине «Математическая логика и теория автоматов»

Лабораторная работа №1

«Лексический анализатор»

Вариант 19

Обучающийся: _____ Шклярова Ксения Алексеевна

Группа: 5130201/20102

Руководитель: _____ Востров Алексей Владимирович

«_____»_____20___г.

Санкт-Петербург, 2025

Содержание

Введение

В отчете представлено описание выполнения лабораторной работы №1 по дисциплине «Математическая логика». Суть лабораторной работы заключалась в реализации лексического анализатора, который выполняет лексический анализ входного текста в соответствии с вариантом и порождает таблицу лексем с указанием их типов и значений. Программа должна выдавать сообщения о наличие во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа.

Варианту 19 соответствуют следующие правила для анализа: входной язык содержит операторы цикла while ... do и do ... while, разделенные символом | (вертикальная полоса). Операторы условия содержат идентификаторы, знаки сравнения $<$, $>$, $=$, вещественные числа, знак присваивания ($:=$). Вещественные числа обязательно содержат знак и не могут начинаться с точки. Длина идентификатора и строковых констант ограничена 16 символами, только латиница. Допускается наличие комментариев неограниченной длины, форма комментариев выбирается самостоятельно.

1 Математическое описание

1.1 Структура транслятора

В соответствии с идеей синтаксически-ориентированной трансляции, процесс трансляции в информатике связывается с двумя основными этапами:

• На первом этапе блок, который можно назвать распознавателем, строит структуру входной цепочки на основе порождающей грамматики входного языка

• На втором этапе построенная структура используется для генерации выхода, выражающего смысл входной цепочки

Во многих трансляторах языков программирования процессы распознавания и генерации разделены не так явно. Но во всех случаях метод синтаксически-ориентированной трансляции основан на том, что строится целиком или по частям структура входной цепочки.

Лексемы

В реальных трансляторах ЯП первой фазой является так называемый лексический анализ входной программы - предварительная обработка входного текста с выделением в нем структурно значимых единиц – лексем.

Лексемы – минимальные единицы языка, которые имеют смысл.

Назначение лексического анализа

Значение лексемы, определяющее подстроку символов входной цепочки, соответствующих распознанному классу лексемы. В зависимости от класса, значение лексемы может быть преобразовано во внутреннее представление уже на этапе лексического анализа.

Класс лексемы, определяющий общее название для категории элементов, обладающих общими свойствами (идентификатор, целое число, строка символов...).

Лексический анализатор обрабатывает входную цепочку, а на его вход подаются символы, сгруппированные по категориям. Поэтому перед лексическим анализом осуществляется дополнительная обработка, сопоставляющая с каждым символом его класс, что позволяет сканеру манипулировать единым понятием для целой группы символов.

Устройство, осуществляющее сопоставление класса с каждым отдельным символом, называется транслитератором. Наиболее типичными классами символов являются:

• буква - класс, с которым сопоставляется множество букв (необязательно одного алфавита);

• цифра - множество символов, относящихся к цифрам, чаще всего от 0 до 9;

• разделитель - пробел, перевод строки, возврат каретки перевод формата;

• игнорируемый - может встречаться во входном потоке, но игнорируется и поэтому просто отфильтровывается из него (например, невидимый код звукового сигнала);

• запрещенный - символы, который не относятся к алфавиту языка, но встречается во входной цепочке;

• прочие - символы, не вошедшие ни в одну из определенных категорий.

Пара, класс символа и его значение, поступают на вход сканера, который выбирает для анализа самый подходящий элемент.

Задача лексического анализа – представить исходную программу как последовательность лексем (лексических единиц).

Лексемы задаются автоматными языками. Лексический анализ выполняется с помощью алгоритма, реализующего функционирование конечного автомата, с добавлением необходимых семантических операций.

Удобной формой представления динамики (поведения) конечного автомата являются синтаксические диаграммы.

1.2 Синтаксические диаграммы

Синтаксическая диаграмма – это направленный граф с одним входом и одним выходом. Вершины графа помечены символами, которые могут встретиться в цепочке языка. Последовательность символов, встреченных на любом пути от входа к выходу синтаксической диаграммы, является цепочкой языка, задаваемого этой синтаксической диаграммой.

Синтаксическая диаграмма – удобный способ задания языка. Ее можно рассматривать, как порождающий формализм языка. Если выбор пути в каждом разветвлении однозначно определяется очередным входным символом, то это – распознающий формализм.

Реализация синтаксической диаграммы выполняется аналогично реализации конечного автомата. Пример построения синтаксической диаграммы из конечного автомата для трансляции языка целых чисел показан на рис. 1.

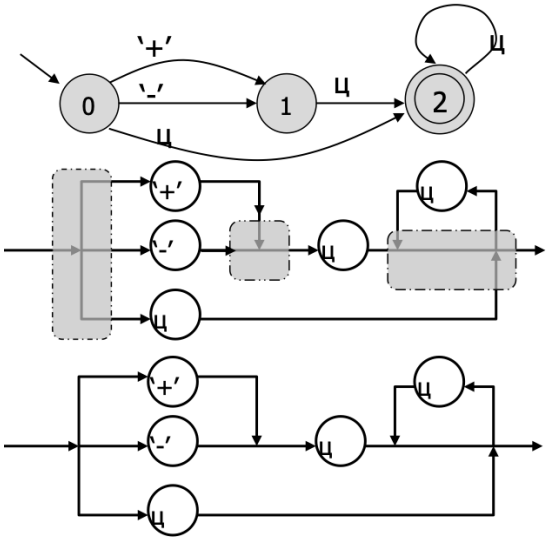

Рис. 1. Процесс построения синтаксической диаграммы

Семантические действия могут встраиваться в синтаксическую диаграмму в любом месте.

Синтаксическая диаграмма с встроенными семантиками полностью определяет алгоритм трансляции.Пример построения синтаксической диаграммы со встроенными семантическими действиями показан на рис.2.

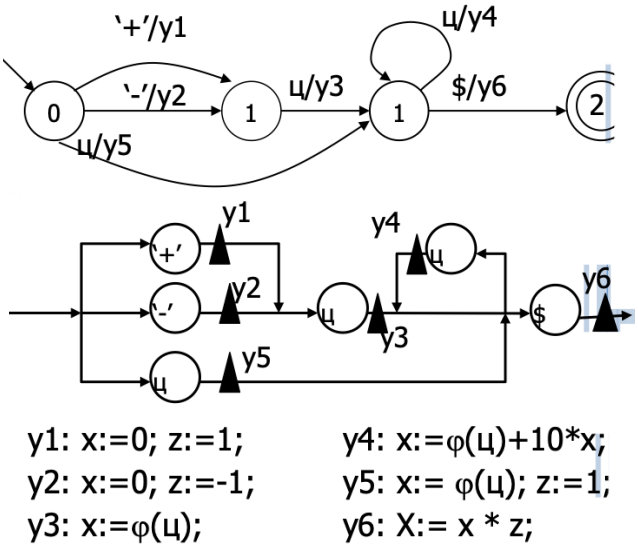

Рис. 2. Построение синтаксической диаграммы с встроенными семантическими действиями

Обработка синтаксических ошибок на синтаксической диаграмме не показывается, чтобы не загромождать структуру. Но в реализации она обязательно присутствует

1.3 Построенная синтаксическая диаграмма

Лексемы, которые обрабатывает реализованный лексический анализатор:

1. Операторы цикла do и while
2. Идентификатор (максимальная длина - 16 символов, может содержать буквы латинского алфавита и цифры, но начинается только с буквы)
3. Строковая константа (максимальная длина - 16 символов, может содержать буквы латинского алфавита и цифры, но начинается только с буквы)
4. Знаки сравнения >, <, =
5. Знак присваивания :=
6. Вещественная константа (начинается со знака + или -, не может начинаться с точки)
7. Разделитель |

На рис.3 представлена диаграмма работы лексического анализатора в данной работе.

Семантические действия:

y1: символ в буфер

y2: выдать лексему «while»

y3: выдать лексему «do»

y4: выдать индентификатор и его номер

y5: выдать строковую константу

y6: выдать лексему >=

y7: выдать лексему >

y8: выдать лексему <=

y9: выдать лексему <

y10: выдать лексему =

y11: выдать лексему :=

y12: выдать вещественную константу

y13: выдать лексему |

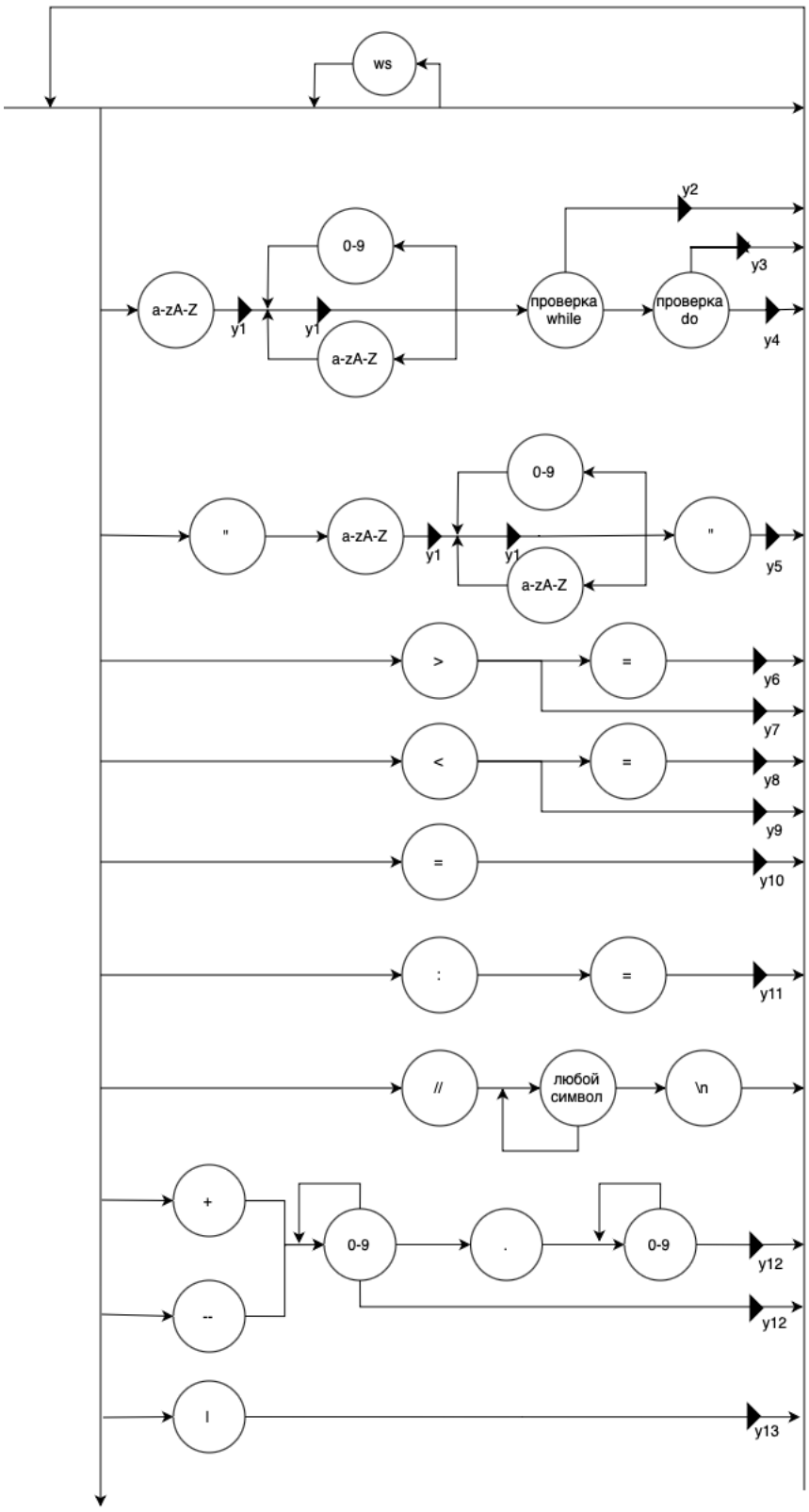

Рис. 3. Синтаксическая диаграмма

2 Особенности реализации

2.1 Структуры данных

В реализации лексического анализатора были использованы следующие структуры данных:

1. Строки (str). Используется для хранения входного текста, который необходимо проанализировать. В процессе анализа строка text постепенно уменьшается, так как из нее "извлекаются"лексемы.

2. Список (list). Используется для хранения информации о найденной лексеме в списке lexeme.

3. Двумерный список для хранения всех лексем в списке table и для хранения ошибок в списке error.

4. Целое число (int). Счетчик, используемый для присвоения уникальных числовых значений идентификаторам в словаре self.dict_iter.

5. Словарь (dict). Используется для хранения соответствия между идентификаторами и их уникальными числовыми значениями. Ключами словаря являются идентификаторы (строки), а значениями - целые числа.

2.2 Реализация класса-анализатора

Реализация лексического анализатора заключается в классе LexicalAnalysis. Рассмотрим методы данного класса.

Конструктор класса __init__(self) инициализирует атрибуты экземпляра. На вход получает ссылку на экзмепляр класса self. Результатом является создание пустых списков self.table для хранения таблицы лексем, self.errors для хранения списка ошибок, пустые словари self.identifier_table для хранения таблиц идентификаторов.

Метод remove_comments(self, text: str) -> str удаляет комментарии из текста. Комментарии начинаются с "//"и продолжаются до конца строки. На вход данного метода передаются ссылка на экземпляр класса self и входной текст text. На выходе получаем строку - текст без комментариев. Для удаления комментариев используется регулярное выражение re.sub(r'//[^]*') для поиска «//» и замены текста после на пустой символ.

Метод check_text(self, text) проверяет, не начинается ли входной текст с символа разделителя ". На вход данного метода подается ссылка на экзмепляр класса self и входной текст text. В случае, если текст начинается с то возвращается сообщение об ошибке в список self.error. В противном случае: None и исходный text. Реализация данного метода заключается в следующем: сначала удаляем пробелы в начале строки с помощью text.lstrip(). Дале проверяем первый символ

строк, не является ли он '|'. Если да, добавляет сообщение об ошибке в список self.error с помощью self.error.append(...) .

Метод check_word(self, word) проверяет, соответствует ли входное слово шаблону, допустимому для идентификатора. На вход данного метода подается ссылка на экземпляр класса self и строка, представляющая слово для проверки - word. На выходе возвращается булево значение: True или False. Для реализации данного метода спользуется функция re.fullmatch из модуля re для проверки, соответствует ли строка word регулярному выражению $[a-zA-Z]+[0-9]*[a-zA-Z]*^*_|$. Это регулярное выражение означает:

1. $[a-zA-Z]+$: Одна или более букв (латинского алфавита, в любом регистре).
2. $[0-9]*$: Ноль или более цифр.
3. $[a-zA-Z]*$: Ноль или более букв (латинского алфавита, в любом регистре).
4. $\backslash s*$: Ноль или более пробельных символов.

Если соответствие найдено, функция возвращает True; в противном случае — False.

Метод find_lexem(self, text) пытается найти следующую лексему в входном тексте. На вход данного метода подается ссылка на экземпляр класса self и входной текст text. На выходе данный метод возвращает кортеж, содержащий: список list представляющий найденную лексему, либо None, если лексема не найдена, а так же строку, представляющую оставшийся текст после обработки найденной лексемы. Реализация данного метода заключается в следующем: последовательно проверяется наличие следующих лексем: <, >, =, :=, |, идентификаторы, строковые константы, вещественные константы, операторы цикла while и do. Если найдена лексема, метод возвращает ее описание и оставшийся текст. Если ни одна лексема не найдена, возвращается None и первый символ входного текста, записанный в ошибку. Обработка идентификаторов включает в себя проверку их длины и допустимости символов при помощи регулярных выражений. Были использованы следующие регулярные выражения:

1. text.startswith('while '), text.startswith('do '), которые проверяют, что начало входной строки соответствует оператору цикла.
2. re.match(r"$[+-]\backslash d+\backslash.\backslash d+|[+-]\backslash d+\ text$), проверяет на вещественную константу, которая должна начинаться со знака, не может начинаться с точки.
3. re.match(r'"$[A-Za-z0-9]+$ı$, $text$) и метод check_word(self, $text[1:-2]$), регулярное выражение проверяет чтобы считанное значение начиналось, заканчивалось «»> и содержало только латинские буквы и цифры, а метод check_word проверяет строковую константу, содержащуюся в ковычках.

4. text.startswith('<'), text.startswith('>'), text.startswith('='), text.startswith(':='), text.startswith('|') проверяют, начинается ли входной текст знаком сравнения,присваивания, разделения.

5. re.match($r"[a-zA-Z]+[\da-zA-Z]*|\d*[a-zA-Z]+text$) и метод check_word(self, text), регулярное выражение используется, чтобы считать идентификатор, который может сожержать буквы латинского алфавита и цифры, а метод check_word проверяет, чтобы данный идентификатор содержал только буквы латинского алфавита, цифры и не начинался с цифры.

Метод check_text_final(self, text) является основным методом для обработки всего входного текста. На вход данного метода подается ссылка на экземпляр класса self и исходный текст text. На выходе метод возвращает список ошибок, возникших во время анализа, и список найденных лексем (таблицу лексем), он инициализирует пустые списки для ошибок self.error и таблицы лексем self.table. Затем с помощью метода remove_comments он удаляет комментарии из текста, а затем в цикле while вызывает метод find_lexem до тех пор, пока весь текст не будет обработан. Найденные лексемы добавляются в self.table. В конце возвращаются списки ошибок и таблицы лексем.

Объявление классов и реализацию данных методо см. в листинге 1.

Листинг 1. Реализация аналитического анализатора

```
class LexicalAnalysis:
    def __init__(self):
        self.table = []
        self.error = []
        self.dict_iter = {}
        self.dict_const = {}

    def remove_comments(self, text):
        text = re.sub(r'//[^\n]*', '', text)
        return text


    def check_text(self, text):
        text = text.lstrip()
        if text.startswith('|'):
            return self.error.append(['Разделитель не может находиться в начале', '']),
                text[1:]
        else:
            return None, text

    def check_word(self, word):
        word = re.fullmatch(r"[a-zA-Z]+[0-9]*[a-zA-Z]*\s*", word)
```

```
22          if word:
23              return True
24          else:
25              return False
26
27      i = 1
28      j = 1
29      def find_lexem(self, text):
30
31          text = text.lstrip()
32
33          if not text:
34              return None, text
35
36          if text.startswith('while '):
37              return ['Оператор цикла        ', 'while', 'X1'], text[5:]
38
39          if text.startswith('do '):
40              return ['Оператор цикла        ', 'do', 'X2'], text[2:]
41
42          real_number = re.match(r"[+-]\d+\.\d+|[+-]\d+", text)
43          if real_number:
44              lexema = real_number.group(0)
45              return ['Вещественная константа', lexema, lexema], text[len(lexema):]
46
47          real_number1 = re.match(r"[+-]\.+\d+", text)
48          if real_number1:
49              lexema = real_number1.group(0)
50              self.error.append(['Вещественная константа не может начинаться с точки',
                    lexema]), text[len(lexema):]
51              return None, text[len(lexema):]
52
53          real_number2 = re.match(r"\d+\.*\d+", text)
54          if real_number2:
55              lexema = real_number2.group(0)
56              self.error.append(['Вещественная константа должна содержать спереди знак',
                    lexema]), text[len(lexema):]
57              return None, text[len(lexema):]
58
59          str_const = re.match(r'"[A-Za-z0-9]+"', text)
60          if str_const:
61              lexema = str_const.group(0)
62              if (self.check_word(lexema[1:-2]) == True):
63                  if (len(lexema) > 16):
```

```
64                  self.error.append(["Длина строковой константы больше 16 символов. С
                        троковая константа:", lexema])
65                  return None, text[len(lexema):]
66              else:
67                  if lexema not in self.dict_const:
68                      self.dict_const[lexema] = self.j
69                      self.j += 1
70                  return ['Строковая константа   ', lexema, lexema], text[len(lexema)
                        :]
71          else:
72              self.error.append(["Строковая константа содержит недопустимые символы.
                    Строковая константа:", lexema])
73              return None, text[len(lexema):]
74
75      if text.startswith('<'):
76          return ['Знак сравнения        ', '<', ''], text[1:]
77      if text.startswith('>'):
78          return ['Знак сравнения        ', '>', ''], text[1:]
79      if text.startswith('='):
80          return ['Знак сравнения        ', '=', ''], text[1:]
81      if text.startswith(':='):
82          return ['Знак присваивания     ', ':=', ''], text[2:]
83      if text.startswith('|'):
84          return ['Разделитель           ', '|', ''], text[1:]
85
86
87      identifier = re.match(r"[a-zA-Z]+[\da-zA-Z]*|\d*[a-zA-Z]+", text)
88      if identifier:
89          lexema = identifier.group(0)
90          if (self.check_word(lexema) == True):
91              if len(lexema) > 16:
92                  self.error.append(["Длина идентификатора больше 16 символов. Иденти
                        фикатор:", lexema])
93                  return None, text[len(lexema):]
94              else:
95                  if lexema not in self.dict_iter:
96                      self.dict_iter[lexema] = self.i
97                      self.i += 1
98                  x = self.dict_iter[lexema]
99                  return ['Идентификатор         ', lexema, lexema + ': ' + str(x)],
                        text[len(lexema):]
100         else:
101             self.error.append(['Идентификатор содержит недопустимые символы. Иденти
                    фикатор:', lexema])
```

```
102                 return None, text[len(lexema):]
103
104         self.error.append(["Неопознанная лексема", text[0]])
105         return None, text[1:]
106
107     def check_text_final(self, text):
108         self.table = []
109         self.error =[]
110         text = self.remove_comments(text)
111         while text:
112             lexeme, text = self.find_lexem(text)
113             if lexeme:
114                 self.table.append(lexeme)
115         return self.error, self.table
```

3 Результаты работы программы

На рис. 4 показан результат выполнения лексического анализатора для первого примера кода, который представлен в листинге 2. В данном коде содержится только 1 недопустимый символ «;».

Листинг 2. Первый пример

```
1 // Пример 1: Простой цикл while
2 x := +1.0
3 while x < +5.0
4   x := +1.0;
```

```
--- Пример 1 ---
Таблица лексем:
------------------------------------------------------------
Тип                    | Значение       | Номер идентификатора
------------------------------------------------------------
Идентификатор          | x              | x: 1
Знак присваивания      | :=             |
Вещественная константа | +1.0           | +1.0
Оператор цикла         | while          | X1
Идентификатор          | x              | x: 1
Знак сравнения         | <              |
Вещественная константа | +5.0           | +5.0
Идентификатор          | x              | x: 1
Знак присваивания      | :=             |
Вещественная константа | +1.0           | +1.0
--------------------------------------
Ошибки
--------------------------------------
Неопознанная лексема ;
```

Рис. 4. Результат анализа первого примера кода

На рис. 5 показан результат выполнения лексического анализатора для первого примера кода, который представлен в листинге 3. В данном коде содержатся недопустимые символы «(», «)», вещественная константа, которая начинается с точки, и идентификаторы, которые начинаются с цифры.

Листинг 3. Второй пример

```
1 // Пример 2
2 y := +1.0
3 do1 y ::= y +.0
4     1while
5  (1y < +10.0)
```

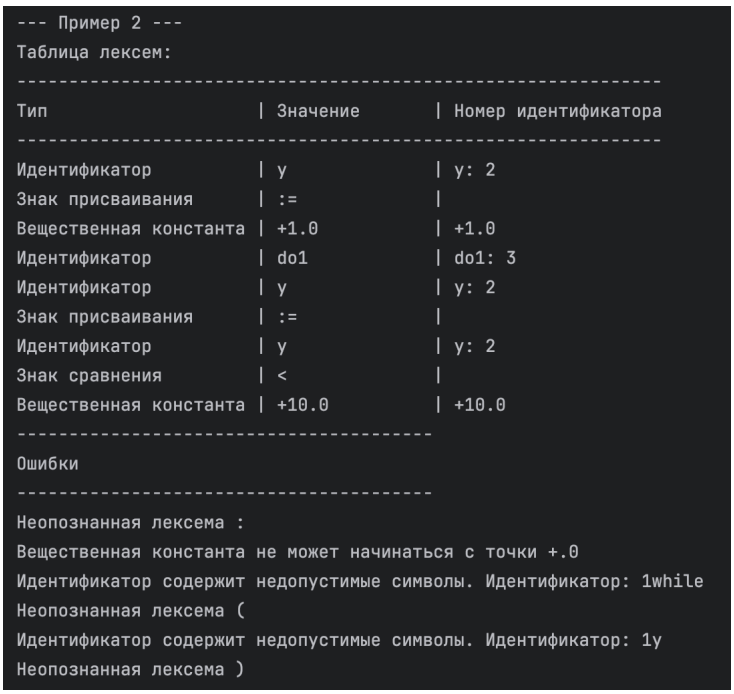

Рис. 5. Результат анализа второго примера кода

На рис. 6 показан результат выполнения лексического анализатора для первого примера кода, который представлен в листинге 4. В данном коде содержатся недопустимые символы «(», «)» и строковая константа, длина которой превышает 16 символлов.

Листинг 4. Третий пример

```
1 // Пример 3 ла ла ла бим бам бом
2 z := +1.5
3 "ijohuk"
4 while (z <= +10.0) do  // скобки лишние
5     z := z +1.0
6     "TooLongStr_)" // Ошибка: строковая константа слишком длинная
```

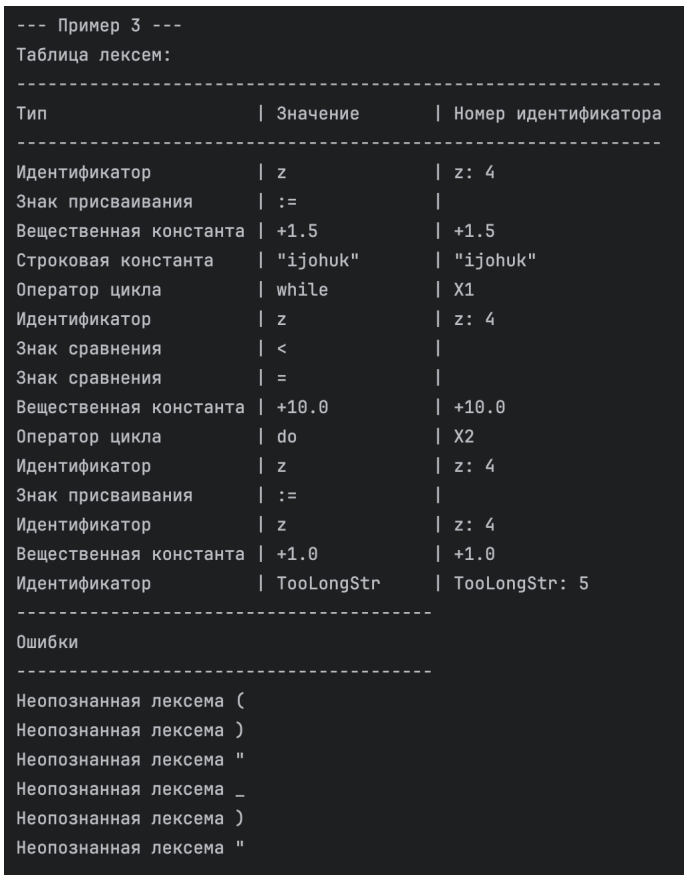

Рис. 6. Результат анализа третьего примера кода

На рис. 7 показан результат выполнения лексического анализатора для первого примера кода, который представлен в листинге 5. В данном коде содержится строковая константа, которая содержит пробел, что недопустимо.

Листинг 5. Четвертый пример

```
1 "jshdf djs"
2 a := +0.0
3 a > -1.0
4     b := +1.0
5     c := -1.0
```

```
--- Пример 4 ---
Таблица лексем:
------------------------------------------------------------
Тип                    | Значение       | Номер идентификатора
------------------------------------------------------------
Идентификатор          | jshdf          | jshdf: 6
Идентификатор          | djs            | djs: 7
Идентификатор          | a              | a: 8
Знак присваивания      | :=             |
Вещественная константа | +0.0           | +0.0
Идентификатор          | a              | a: 8
Знак сравнения         | >              |
Вещественная константа | -1.0           | -1.0
Идентификатор          | b              | b: 9
Знак присваивания      | :=             |
Вещественная константа | +1.0           | +1.0
Идентификатор          | c              | c: 10
Знак присваивания      | :=             |
Вещественная константа | -1.0           | -1.0
--------------------------------------
Ошибки
--------------------------------------
Неопознанная лексема "
Неопознанная лексема "
```

Рис. 7. Результат анализа четвертого примера кода

На рис. 8 показан результат выполнения лексического анализатора для первого примера кода, который представлен в листинге 6. В данном коде содержится идентификатор, который содержит «_», поэтому он разбивается на 2 части, а нижнее подчеркивание считается неопознанной лексемой, идентификатор, который начинается с цифры, и вещественное число, начинающееся с точки.

Листинг 6. Пятый пример

```
1 x1 := x_hfhf < +1.0
2 while 1x <= +10.0
3 |
4 while1 y < +20.0
5  do y := +.1
```

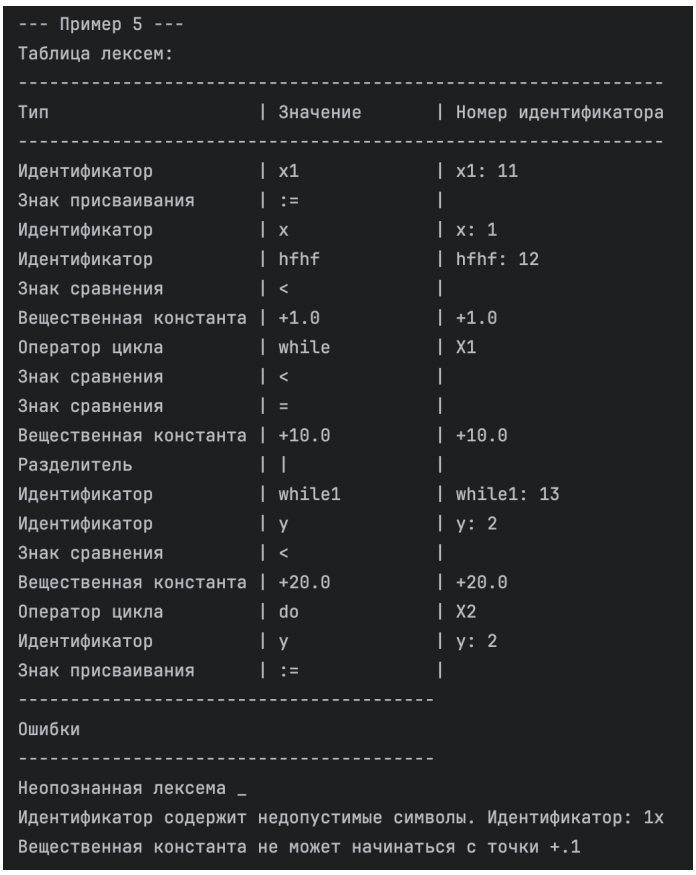

Рис. 8. Результат анализа пятого примера кода

Заключение

В результате выполнения лабораторной работы был реализован лексический анализатор для языка, который поддерживает знаки сравнения >, <, =, знак присваивания :=, разделитель |, операторы цикла do и while, вещественные цифры, которые содержат знак и не начинаютя с точки, идентификатор и строковые константы, длина которых не длинее 16 символов. Данный язык является автоматным, что подтверждается построенной синтаксической диаграммой, отражающей его грамматику.

Реализованная программа включает следующие ключевые функции: удаление комментариев с помощью регулярных выражений, посимвольный анализ входного текста и распознавание лексем в соответствии с заданными правилами, формирование таблицы лексем с указанием их типов, значений и номеров идентификаторов, обнаружение и вывод ошибок.

Анализатор был протестирован на нескольких примерах, включая как корректные конструкции, так и ошибочные. В ходе тестирования подтверждена его способность: корректно выделять лексемы и классифицировать их, обнаруживать ошибки и формировать информативные сообщения о них.

Из плюсов программы можно выделить простую логику проверки соответствия шаблону, фиксацию определенных ошибок.

Недостатки: избыточная проверка пробелов, ограниченная обработка строк(например, не учтены escape-символы), нет обработки многозначных операторов(например, >=, <=).

Масштабируемость: можно добавить больше ключевых слов, например for, if. Можно добавить обработку чисел других счистем счисления. Реализовать обработку многострочных комментариев.

Работа выполнена на языке программирования Python в среде разработки PyCharm версии 2023.3.4.

Список литературы

[1] Востров, А. В. Математическая логика

URL:https://tema.spbstu.ru/compiler (Дата обращения: 23.03.2025).

[2] Сети, Р.; Ахо, А. Компиляторы: принципы, технологии и инструменты / Р. Сети, А. Ахо. - М.: Издательство «Наука», 2006. - С. 104.