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

Лабораторная работа «Лексический анализатор» по дисциплине «Теория алгоритмов»

Обучающийся: _____ Яшнова Дарья Михайловна
группа 5130201/20002

Руководитель: _____ Востров Алексей Владимирович

«___»_____ 2025г

Санкт-Петербург, 2025

Содержание

Введение **3**

1 Математическое описание лексического анализатора **4**
1.1 Структура транслятора . . . 4
1.2 Лексический анализ . . . 4
1.2.1 Назначение лексического анализа . . . 4
1.2.2 Классификация символов . . . 5
1.3 Формальная модель лексического анализатора . . . 5
1.4 Работа лексического анализатора . . . 5
1.5 Синтаксические диаграммы . . . 6
1.6 Обработка ошибок . . . 7

2 Особенности реализации **8**
2.1 Структуры данных . . . 8
2.1.1 Типы токенов . . . 8
2.1.2 Регулярные выражения . . . 8
2.1.3 Управляющие переменные . . . 8
2.1.4 Наборы операторов . . . 8
2.2 Методы лексического анализатора . . . 8
2.2.1 Основные функции . . . 8
2.3 Регулярные выражения в лексическом анализаторе . . . 9
2.3.1 Проверка вещественных чисел . . . 9
2.3.2 Проверка идентификаторов . . . 10
2.3.3 Использование в коде . . . 10
2.4 Код программы . . . 10

3 Результаты работы программы **15**

Заключение **17**

Список литературы **18**

Введение

Лексический анализ — это первый этап обработки текста программы или входных данных, на котором осуществляется разбор последовательности символов и выделение лексем (токенов) — минимальных значимых единиц языка, таких как ключевые слова, идентификаторы, константы, операторы и разделители.

В данной работе рассматривается разработка программы, выполняющей лексический анализ входного текста в соответствии с заданными требованиями. Основные задачи программы:

- Разбиение входного текста на лексемы.
- Классификация лексем по типам (идентификаторы, константы, ключевые слова и т. д.).
- Проверка ограничений на длину идентификаторов и строковых констант (не более 16 символов).
- Обработка комментариев произвольной длины.
- Вывод сообщений об ошибках, которые могут быть обнаружены на этапе лексического анализа (например, недопустимые символы, слишком длинные идентификаторы).

Вариант 21: Входной язык содержит арифметические выражения, разделенные символом | (вертикальная полоса). Арифметические выражения состоят из идентификаторов, вещественные числа в показательной форме, знака присваивания (:=), знаков операций >, <, >=, <=, <> и фигурных скобок.

Результатом работы является программа, способная корректно анализировать входной текст, формировать таблицу лексем с указанием их типов и значений, а также сообщать об ошибках, если они присутствуют.

1 Математическое описание лексического анализатора

1.1 Структура транслятора

Транслятор осуществляет преобразование входного текста L в выходные данные. В соответствии с синтаксически-ориентированной трансляцией процесс включает два этапа:

- **Распознаватель** — строит структуру входной цепочки на основе порождающей грамматики входного языка. Данный этап обеспечивает анализ синтаксиса и формирование промежуточной структуры.
- **Генератор** — использует построенную структуру для создания выходных данных, отражающих семантику входной цепочки.

В реальных трансляторах языков программирования этапы могут быть частично совмещены, однако ключевым принципом остаётся построение структуры входных данных (целиком или по частям).

1.2 Лексический анализ

Первая фаза трансляции — лексический анализ, который выделяет **лексемы** — минимальные единицы языка, обладающие смыслом.

- В естественном языке лексемами являются слова (словоформы).
- В языках программирования — идентификаторы, ключевые слова, константы, составные операторы (например, ":=").

На рис.1 представлена схема транслятора.

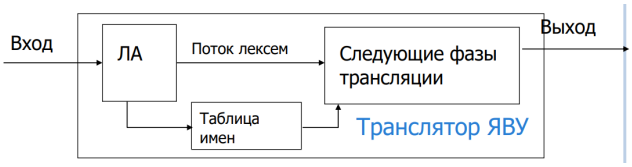

Рис. 1: Схема транслятора

1.2.1 Назначение лексического анализа

Лексический анализатор выполняет:

- Определение **значения лексемы** — подстроки символов, соответствующей распознанному классу. Для некоторых классов (например, чисел) значение преобразуется во внутреннее представление (двоичный формат) для компактности и ранней проверки корректности.
- Определение **класса лексемы** — категории элементов с общими свойствами:
  - идентификатор,
  - целое число,
  - строка символов,
  - оператор.

1.2.2 Классификация символов

Перед лексическим анализом **транслитератор** сопоставляет каждый символ входной цепочки с классом:

- *Буква* — символы алфавита (включая возможные мультиязычные наборы).
- *Цифра* — символы 0–9.
- *Разделитель* — пробел, перевод строки, возврат каретки.
- *Игнорируемый* — символы, присутствующие во входном потоке, но игнорируемые (например, служебные коды).
- *Запрещённый* — символы, не входящие в алфавит языка.
- *Прочие* — символы, не классифицированные в предыдущие категории.

1.3 Формальная модель лексического анализатора

Лексический анализатор (лексер) — это конечный автомат, который преобразует входную строку символов в последовательность токенов. Формально его можно описать следующим образом: Пусть заданы:

- Σ — входной алфавит (множество допустимых символов)
- T — множество типов токенов
- D — множество допустимых значений токенов

Тогда лексический анализатор реализует отображение:

$$F_{\text{lexer}} : \Sigma^* \to (T \times D)^*$$

где:

- Σ^* — множество всех возможных строк над алфавитом Σ
- $(T \times D)^*$ — множество последовательностей пар (тип токена, значение)

1.4 Работа лексического анализатора

Процесс лексического анализа можно представить как детерминированный конечный автомат (ДКА):

$$M = (Q, \Sigma, \delta, q_0, F)$$

где:

- Q — множество состояний автомата
- $\delta : Q \times \Sigma \to Q$ — функция переходов
- $q_0 \in Q$ — начальное состояние
- $F \subseteq Q$ — множество конечных состояний

Для каждого распознанного токена t_i выполняется:

$$t_i = (\text{type}, \text{value}), \quad \text{где type} \in T, \text{value} \in D$$

1.5 Синтаксические диаграммы

Синтаксические диаграммы визуализируют правила формирования лексем. На рис.2 представлена диаграмма работы лексического анализатора в данной работе.

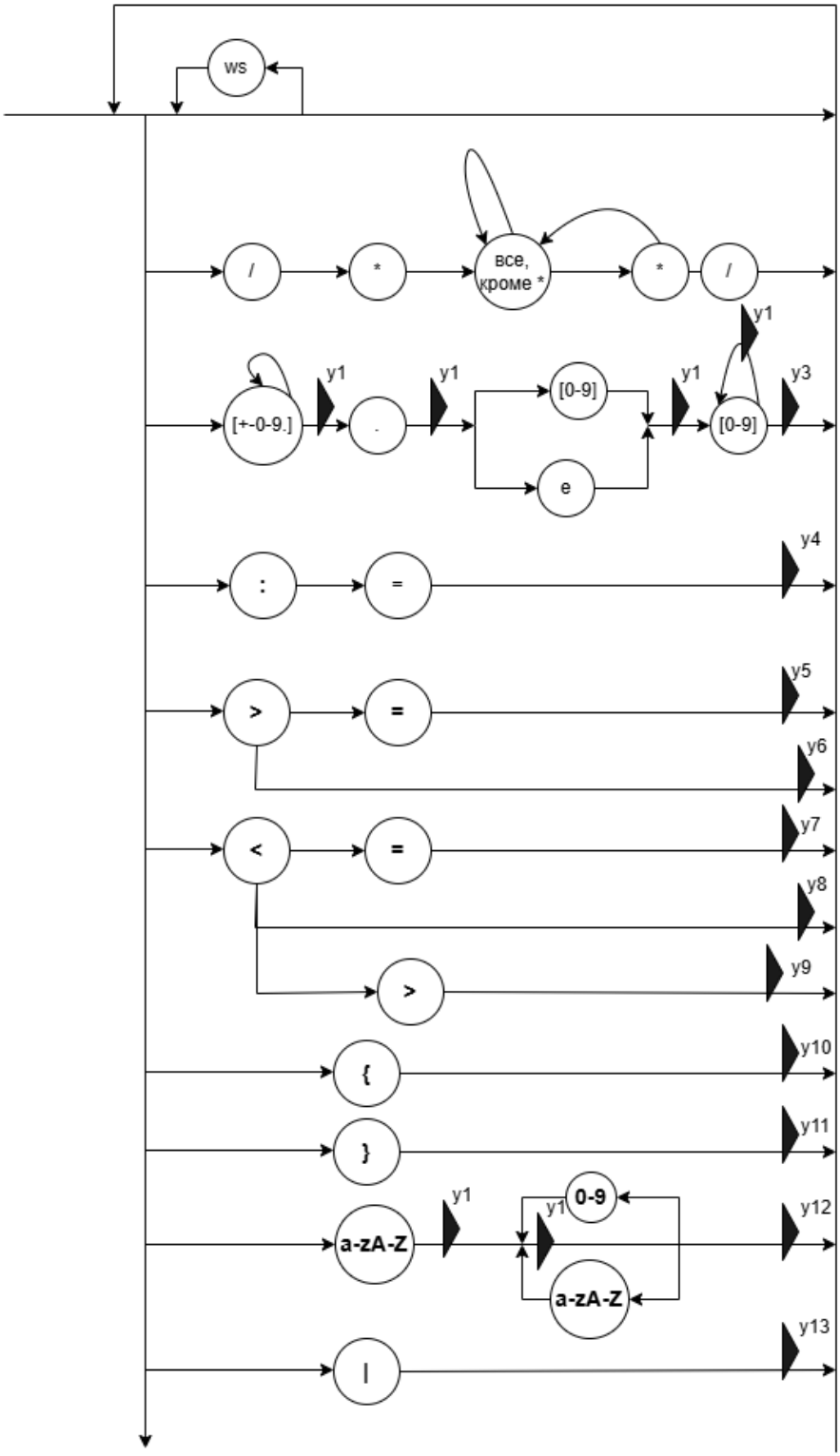

Рис. 2: Синтаксическая диаграмма

```
y1: символ в буфер
y3: записываем константу, выдаем лексему <const, значение>
y4: Выдать лексему ":="
y5:Выдать лексему ">="
```

```
y6:Выдать лексему ">"
y7:Выдать лексему "<="
y8:Выдать лексему "<"
y9:Выдать лексему "<>"
y10:Выдать лексему "{"
y11:Выдать лексему "}"
y12:Записываем идентификатор, выдаем лексему <id, n>
y13:Выдать лексему "|"
```

Лексемы в этом ЯП: идентификатор, знак присваивания, оператор(<,>,<>,<=,>=), левая скобка, правая скобка, разделитель |.

1.6 Обработка ошибок

Лексический анализатор обнаруживает следующие ошибки:

- Недопустимые символы: $\exists c \in w \mid c \notin \Sigma$
- Превышение длины идентификатора: $|\text{id}| > 16$
- Незакрытые строковые константы
- Некорректные числовые форматы

Для каждой ошибки генерируется сообщение:

$$E_{\text{lexer}} : \Sigma^* \to \mathbb{N} \times \text{String}$$

где первое значение — позиция ошибки, второе — описание.

2 Особенности реализации

2.1 Структуры данных

Код лексического анализатора использует следующие ключевые структуры данных:

2.1.1 Типы токенов

Константы для классификации лексем:

- Строковые идентификаторы типов: `TOKEN_IDENTIFIER`, `TOKEN_NUMBER`, `TOKEN_ASSIGN` и др.

2.1.2 Регулярные выражения

- `NUMBER_REGEX` — проверяет корректность вещественных чисел с экспоненциальной нотацией.
- `IDENT_REGEX` — валидирует идентификаторы (латиница + цифры, первый символ — буква).

2.1.3 Управляющие переменные

- `tokens` — список кортежей вида `(тип_токена, значение)` для хранения результата.
- `i` — индекс текущей позиции в тексте.
- `error_found` — флаг обнаружения ошибки.
- `last_was_expr` — флаг проверки последовательности выражений.

2.1.4 Наборы операторов

Множества для быстрой проверки символов:

- `multi_operators` — двухсимвольные операторы (`>=`, `<=`, `<>`, `:=`).
- `single_operators` — односимвольные операторы (`>`, `<`).
- `single_symbols` — служебные символы (`{`, `}`, `|`).

2.2 Методы лексического анализатора

2.2.1 Основные функции

- **tokenize(text)**
  Вход: строка `text` (исходный текст программы).
  Выход: список кортежей `(тип_токена, значение)`.
  Главная функция анализатора. Организует цикл разбора текста, вызывает вспомогательные методы для обработки пробелов, комментариев и лексем. Формирует итоговый список токенов с обработкой ошибок.

- **push_token(ttype, tvalue)**
  Вход: `ttype` (тип токена), `tvalue` (значение токена).
  Выход: Нет.
  Добавляет токен в список `tokens`. Управляет флагом `last_was_expr` для контроля последовательности выражений. В случае ошибки (например, два выражения без разделителя) устанавливает `error_found = True`.

- **skip_whitespace(pos)**
  Вход: `pos` (текущая позиция в тексте).
  Выход: Новая позиция `pos` после пропуска пробелов.
  Пропускает все пробельные символы (пробел, табуляция, перевод строки).

- **skip_comment(start_pos)**
  Вход: `start_pos` (начальная позиция комментария `/*`).
  Выход: Кортеж `(new_pos, not_closed)`, где:
  — `new_pos` — позиция после комментария (или конец текста),
  — `not_closed` — флаг незакрытого комментария (`True/False`).
  Обрабатывает многострочные комментарии. Если комментарий не закрыт, выводит ошибку.

- **is_delimiter(pos)**
  Вход: `pos` (текущая позиция в тексте).
  Выход: `True/False` (является ли позиция началом разделителя).
  Проверяет, находится ли в позиции `pos` пробел, оператор, служебный символ или начало комментария.

2.3 Регулярные выражения в лексическом анализаторе

В коде используются два регулярных выражения для проверки корректности лексем.

2.3.1 Проверка вещественных чисел

Регулярное выражение `NUMBER_REGEX`:

```
^[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?$
```

1. Структура
   - `^[0-9]+`
     — обязательная целая часть (одна или более цифр).
   - `(\.[0-9]+)?`
     — необязательная дробная часть: точка и одна или более цифр.
   - `([eE][+-]?[0-9]+)?` — необязательная экспонента: символ `e` или `E`, необязательный знак (`+` или `-`), одна или более цифр.
2. Примеры корректных значений
   - `123`, `45.67`, `8e10`, `3.14E-5`
3. Примеры ошибок
   - `12.3.4` — несколько точек.
   - `1e` — отсутствие цифр в экспоненте.
   - `abc` — буквы в числе.

2.3.2 Проверка идентификаторов

Регулярное выражение `IDENT_REGEX`:

`^[A-Za-z][A-Za-z0-9]*$`

1. Структура
   - `^[A-Za-z]`
     — первый символ — буква латинского алфавита.
   - `[A-Za-z0-9]*`
     — последующие символы: буквы и цифры (включая ноль символов).
2. Примеры корректных значений
   - `x`, `var1`, `MAXVALUE`
3. Примеры ошибок
   - `1var` — начинается с цифры.
   - `a&b` — содержит недопустимый символ `&`.
   - `thisisaverylongidentifier` — превышение длины (проверяется отдельно).

2.3.3 Использование в коде

- `NUMBER_REGEX.match(chunk)` — проверяет, соответствует ли строка `chunk` формату числа.
- `IDENT_REGEX.match(chunk)` — проверяет базовый формат идентификатора (без учёта длины).
- Длина идентификатора проверяется отдельно через `len(chunk) > MAX_LEN`.

2.4 Код программы

```
1
2
3  import re
4  import sys
5
6  # Типы лексем
7  TOKEN_IDENTIFIER  = "IDENTIFIER"     # Идентификатор
8  TOKEN_NUMBER      = "REAL_NUMBER"    # Вещественное число (с/без экспонен
       ты)
9  TOKEN_ASSIGN      = "ASSIGN"         # :=
10 TOKEN_OPERATOR    = "OPERATOR"       # >, <, >=, <=, <>
11 TOKEN_LBRACE      = "LBRACE"         # {
12 TOKEN_RBRACE      = "RBRACE"         # }
13 TOKEN_PIPE        = "PIPE"           # |
14 TOKEN_COMMENT     = "COMMENT"        # /* ... */ (можно не добавлять в ит
       оговую таблицу, если не нужно)
15 TOKEN_UNKNOWN     = "UNKNOWN"        # Неизвестный символ
16
17 MAX_LEN = 16
18
19 NUMBER_REGEX = re.compile(r'^[+-]?(?:\d+\.?\d*|\.\d+)([eE][+-]?\d+)?$')
20 IDENT_REGEX = re.compile(r'^[A-Za-z][A-Za-z0-9]*$')
```

```
21
22  def tokenize(text):
23
24      tokens = []
25      length = len(text)
26      i = 0
27      error_found = False
28
29      last_was_expr = False
30
31
32      brace_error_reported = False
33
34      def push_token(ttype, tvalue):
35          nonlocal last_was_expr, error_found, brace_error_reported
36
37          if ttype in [TOKEN_IDENTIFIER, TOKEN_NUMBER]:
38              if last_was_expr:
39                  print("Ошибка: два выражения не разделены '|'")
40                  error_found = True
41              last_was_expr = True
42          elif ttype == TOKEN_PIPE:
43
44              last_was_expr = False
45          else:
46              last_was_expr = False
47
48
49
50          tokens.append((ttype, tvalue))
51
52      def skip_whitespace(pos):
53          while pos < length and text[pos].isspace():
54              pos += 1
55          return pos
56
57      def skip_comment(start_pos):
58          pos = start_pos + 2
59          while pos < length:
60              if pos + 1 < length and text[pos] == '*' and text[pos+1] ==
                    '/':
61                  return pos + 2, False
62              pos += 1
63          return pos, True
64
65      multi_operators = {'>=', '<=', '<>', ':='}
66      single_operators = {'>', '<'}
67      single_symbols = {'|', '{', '}'}
68
69      while i < length:
70          i = skip_whitespace(i)
71          if i >= length:
72              break
73          if error_found:
74              pass
75
76          if i + 1 < length and text[i] == '/' and text[i+1] == '*':
77              new_i, not_closed = skip_comment(i)
78              if not_closed:
79                  print(f"Ошибка: незакрытый комментарий, начало на позиции
                        {i}")
```

```
                error_found = True
            i = new_i
            continue

        two_ch = text[i : i+2]
        if two_ch in multi_operators:
            if two_ch == ':=':
                push_token(TOKEN_ASSIGN, two_ch)
            else:
                push_token(TOKEN_OPERATOR, two_ch)
            i += 2
            continue

        ch = text[i]
        if ch in single_operators:
            push_token(TOKEN_OPERATOR, ch)
            i += 1
            continue
        if ch in single_symbols:
            if ch == '|':
                push_token(TOKEN_PIPE, ch)
            elif ch == '{':
                push_token(TOKEN_LBRACE, ch)
            elif ch == '}':
                push_token(TOKEN_RBRACE, ch)
            i += 1
            continue

        start_pos = i

        def is_delimiter(pos):
            if pos >= length:
                return True
            if text[pos].isspace():
                return True
            if pos + 1 < length and text[pos:pos+2] == '/*':
                return True
            if text[pos:pos+2] in multi_operators:
                return True
            if text[pos] in single_operators or text[pos] in single_
               symbols:
                return True
            return False

        while i < length and not is_delimiter(i):
            i += 1

        chunk = text[start_pos:i]

        if not chunk:
            continue
        if chunk[0].isdigit():
            if NUMBER_REGEX.match(chunk):
                push_token(TOKEN_NUMBER, chunk)
            else:

                if any(c.isalpha() for c in chunk):

                    if '.' in chunk or 'e' in chunk or 'E':
                        print(f"Ошибка: неверный формат вещественного чис
                           ла \"{chunk}\"")
```

```
139                         else:
140                             print("Ошибка: идентификатор не может начинаться
                                    с цифры:", chunk)
141                         error_found = True
142                     else:
143 
144                         print(f"Ошибка: неверный формат вещественного числа
                                \"{chunk}\"")
145                         error_found = True
146                 continue
147 
148             if chunk[0].isalpha():
149                 if len(chunk) > MAX_LEN:
150                     print(f"Ошибка: идентификатор \"{chunk}\" длиннее {MAX_
                            LEN} символов.")
151                     error_found = True
152                 else:
153                     if not all(c.isalnum() for c in chunk):
154                         print(f"Ошибка: в идентификаторе \"{chunk}\" обнаруже
                                ны недопустимые символы.")
155                         error_found = True
156                     else:
157                         push_token(TOKEN_IDENTIFIER, chunk)
158                 continue
159 
160             print(f"Ошибка: неизвестный символ \"{chunk}\"")
161             error_found = True
162 
163 
164         return tokens
165 
166     def gen_tbl(tokens_list):
167     # Создаем словарь для хранения номеров идентификаторов
168         identifier_table = {}
169         identifier_count = 1
170 
171         # Подготавливаем данные для вывода
172         output_data = []
173         for (tt, val) in tokens_list:
174             if tt == TOKEN_IDENTIFIER:
175                 if val not in identifier_table:
176                     identifier_table[val] = identifier_count
177                     identifier_count += 1
178                 output_data.append((val, "Идентификатор", f"{val} : {
                        identifier_table[val]}"))
179             elif tt == TOKEN_NUMBER:
180                 output_data.append((val, "Вещественная константа", val))
181             elif tt == TOKEN_ASSIGN:
182                 output_data.append((val, "Знак присваивания", val))
183             elif tt == TOKEN_OPERATOR:
184                 output_data.append((val, "Оператор", val))
185             elif tt == TOKEN_LBRACE:
186                 output_data.append((val, "Левая скобка", val))
187             elif tt == TOKEN_RBRACE:
188                 output_data.append((val, "Правая скобка", val))
189             elif tt == TOKEN_PIPE:
190                 output_data.append((val, "Разделитель", val))
191             else:
192                 output_data.append((val, tt, val))
193 
194         # Выводим таблицу лексем
```

```
195         print("\nТаблица лексем:")
196         print("    "*60)
197         print("{:<20}  {:<30}  {}".format("Лексема", "Тип лексемы", "Значение
                "))
198         print("    "*60)
199         for lexeme, lexeme_type, value in output_data:
200             print("{:<20}  {:<30}  {}".format(lexeme, lexeme_type, value))
```

3 Результаты работы программы

Далее представлен вывод для программы

```
data = "xuxa := 0|xuxa<9|x>8 /*gggg*/ ".
```

В данной программе нет ошибок, поэтому выводится только таблица лексем.

```
    Таблица лексем:
-----------------------------------------------------------------
Лексема                 Тип лексемы                     Значение
-----------------------------------------------------------------
xuxa                    Идентификатор                   xuxa : 1
:=                      Знак присваивания               :=
0                       Вещественная константа          0
|                       Разделитель                     |
xuxa                    Идентификатор                   xuxa : 1
<                       Оператор                        <
9                       Вещественная константа          9
|                       Разделитель                     |
x                       Идентификатор                   x : 2
>                       Оператор                        >
8                       Вещественная константа          8
```

Далее представлен вывод для программы

```
data = "xuxa := 0|xuxa<9|x>8 |xuxa<100 /*gggg "
```

В данной программе не закрыт комментарий, на выходе программы появляется ошибка.

```
Ошибка: незакрытый комментарий, начало на позиции 31

Таблица лексем:
-----------------------------------------------------------------
Лексема                 Тип лексемы                     Значение
-----------------------------------------------------------------
xuxa                    Идентификатор                   xuxa : 1
:=                      Знак присваивания               :=
0                       Вещественная константа          0
|                       Разделитель                     |
xuxa                    Идентификатор                   xuxa : 1
<                       Оператор                        <
9                       Вещественная константа          9
|                       Разделитель                     |
x                       Идентификатор                   x : 2
>                       Оператор                        >
8                       Вещественная константа          8
|                       Разделитель                     |
xuxa                    Идентификатор                   xuxa : 1
<                       Оператор                        <
100                     Вещественная константа          100
```

Далее представлен вывод для программы

```
data = "xuxa := 0|1xuxa<9|x>8 |xuxa<100 /*gggg*/ "
```

В данной программе идентификатор начинается с цифры, поэтому выводится ошибка и таблица лексем.

```
Ошибка: идентификатор не может начинаться с цифры: 1xuxa

Таблица лексем:
-----------------------------------------------------------------
Лексема                 Тип лексемы                     Значение
-----------------------------------------------------------------
xuxa                    Идентификатор                   xuxa : 1
:=                      Знак присваивания               :=
0                       Вещественная константа          0
|                       Разделитель                     |
<                       Оператор                        <
9                       Вещественная константа          9
|                       Разделитель                     |
x                       Идентификатор                   x : 2
>                       Оператор                        >
8                       Вещественная константа          8
|                       Разделитель                     |
xuxa                    Идентификатор                   xuxa : 1
<                       Оператор                        <
100                     Вещественная константа          100
```

Далее представлен вывод для программы

```
data = "xuxa ::= 0|{ xuxa<9 | x>8 |{xuxa<1.e.3 /*gggg*/ "
```

В данной программе есть лишнее двоеточие, незакрытые скобки, ошибка в вещественном числе, поэтому выводятся ошибки и таблица лексем.

```
Ошибка: неизвестный символ ":"
Ошибка: неверный формат вещественного числа "1.e.3"

Таблица лексем:
-----------------------------------------------------------------
Лексема                 Тип лексемы                     Значение
-----------------------------------------------------------------
xuxa                    Идентификатор                   xuxa : 1
:=                      Знак присваивания               :=
0                       Вещественная константа          0
|                       Разделитель                     |
{                       Левая скобка                    {
xuxa                    Идентификатор                   xuxa : 1
<                       Оператор                        <
9                       Вещественная константа          9
|                       Разделитель                     |
x                       Идентификатор                   x : 2
>                       Оператор                        >
8                       Вещественная константа          8
|                       Разделитель                     |
{                       Левая скобка                    {
xuxa                    Идентификатор                   xuxa : 1
<                       Оператор                        <
```

Заключение

В ходе выполнения лабораторной работы был разработан лексический анализатор для языка, поддерживающего арифметические выражения с комментариями, операторами сравнения, идентификаторами и числами в экспоненциальной форме. Данный язык является автоматным, так как представим в виде синтаксической диаграммы. Автоматные грамматики — самые простые из формальных грамматик. Они являются контекстно-свободными, но с ограниченными возможностями.

Достоинства реализации:

- Код читает входные данные последовательно, символ за символом, что позволяет обрабатывать большие файлы без загрузки всего текста в память.

- Использование предварительно скомпилированных регулярных выражений (NUMBER_REGEX, IDENT_REGEX) ускоряет проверку токенов. Регулярные выражения выполняются за O(n), где n — длина строки, что эффективно для большинства сценариев.

- Функции skip_whitespace и skip_comment пропускают пробелы и комментарии за один проход, избегая повторного анализа одних и тех же символов. Это экономит время при обработке больших текстов.

Недостатки данной реализации:

- Использование print для вывода ошибок замедляет выполнение при большом количестве ошибок.

Возможные улучшения:

Расширение набора операторов и типов данных (например, строковые константы, логические операторы).

Добавление семантического анализа для проверки совместимости типов.

Разработанный анализатор может служить основой для компилятора или интерпретатора целевого языка.

Список литературы

[1] Электронный ресурс ВШТИИ
URL:https://tema.spbstu.ru/compiler/
(Дата обращения: 13.03.2025)

[2] Карпов, Ю. Г. Теория автоматов, Санкт-Петербург : Питер, 2003.
URL:https://djvu.online/file/eeLVKnyRZPXfl
(Дата обращения: 12.03.2025)