МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение
высшего 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 №1
«Создание лексического анализатора»
по дисциплине
«Математическая логика и теория автоматов»
Вариант 15

Студент,
группы 5130201/20102 _____ Тищенко А. А.

Преподаватель _____ Востров А. В.

«_____»_____ 2025г.

Санкт-Петербург, 2025

Содержание

Введение

Лабораторная №1 по дисциплине «Математическая логика» заключается в следующем. Необходимо написать программу, которая выполняет лексический анализ входного текста в соответствии с вариантом задания и порождает таблицу лексем с указанием их типов и значений. Также необходимо подготовить несколько вариантов программы в виде текста на входном языке. Программа должна выдавать сообщения о наличие во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа. Длина идентификатора и строковых констант ограничена 16 символами, только латиница. Программа должна допускать наличие комментариев неограниченной длины во входном файле.

Вариант 15. Входной язык содержит арифметические выражения, разделенные символом | (вертикальная полоса). Арифметические выражения состоят из идентификаторов, комплексных чисел (в показательной форме), знака присваивания (:=), знаков операций +, –, *, /, ^и круглых скобок.

1 Математическое описание

1.1 Структура транслятора

Транслятор выполняет преобразование исходного текста L на каком-либо языке в какую-либо структуру данных с сохранением смысла входного текста.

Транслятор можно разделить на три основные части (см. Рис. 1):

- Лексический анализатор – представляет исходную программу как последовательность лексем. Лексемы - минимальные единицы языка, имеющие смысл.
- Распознаватель – строит структуру исходного текста (например, в виде дерева) по полученной от лексического анализатора цепочке лексем.
- Генератор – использует построенную структуру для создания выходных данных, отражающих семантику входной цепочки.

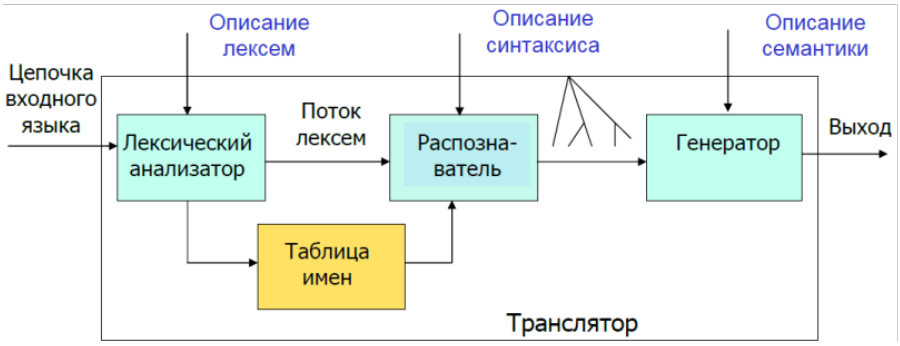

Рис. 1. Структура транслятора.

В данной лабораторной работе необходимо написать программу лексический анализатор для заданного языка.

1.2 Формальное определение лексического анализа

Лексический анализ — первая фаза трансляции программ на языках программирования, цель которой состоит в преобразовании входного текста программы в последовательность структурно значимых элементов — *лексем*.

Лексема — это минимальная единица языка программирования, обладающая самостоятельным смыслом. В отличие от естественных языков, где лексемами выступают слова или словоформы, в языках программирования лексемами являются имена, ключевые (служебные) слова, числовые и строковые константы, а также операторы (в том числе составные, например, `:=`).

Цели лексического анализа включают:

- определение класса каждой лексемы (например, идентификатор, число, строка, оператор и т.п.);
- определение значения лексемы;
- преобразование значений некоторых лексем (например, чисел) во внутреннее представление;
- фильтрация и классификация символов для облегчения анализа.

В зависимости от класса, значение лексемы может быть преобразовано во внутреннее представление уже на этапе лексического анализа. Например, числа преобразуют в двоичное машинное представление, что обеспечивает более компактное хранение и проверку правильности диапазона на ранней стадии трансляции.

Перед лексическим анализом производится этап транслитерации, на котором каждому символу сопоставляется его класс. Типичные классы символов:

- *буква* — символы алфавита;
- *цифра* — символы от `0` до `9`;
- *разделитель* — пробел, перевод строки, табуляция и др.;
- *игнорируемый* — символы, не несущие смысловой нагрузки (например, сигнальные коды);
- *запрещённый* — символы, не входящие в алфавит языка;
- *прочие* — остальные символы, не попавшие в предыдущие категории.

Для данного варианта лабораторной работы были выбраны следующие классы лексем:

1. **Идентификаторы** — последовательности латинских букв, цифр и символов нижнего подчёркивания. Не могут начинаться с цифры. Максимальная длина идентификатора — 16 символов. Примеры: `X`, `alpha1`, `_private`.
2. **Комплексные числа в показательной форме** — числа вида `aEb`, где `a` и `b` — действительные числа (целые или с плавающей точкой), знак 'E' указывает на показатель степени. Примеры: `1.5E3`, `-2E-4`, `3.0E+2`.
3. **Оператор присваивания** — составной символ `:=`, обозначающий операцию присваивания значения.
4. **Арифметические операторы** — знаки `+`, `-`, `*`, `/`, `^` для выполнения соответствующих математических операций.
5. **Левая скобка** — символ `(`, используемый для задания порядка вычислений в арифметических выражениях.
6. **Правая скобка** — символ `)`, используемый для задания порядка вычислений в арифметических выражениях.

7. **Разделитель выражений** — символ вертикальной черты `|`, отделяющий одно арифметическое выражение от другого.

8. **Комментарии** — начинаются с символа `#` и продолжаются до конца строки. Комментарии могут быть произвольной длины и содержать любые символы кроме переноса строки. Пример: `# это комментарий`.

9. **Ошибки** — это случаи, когда входной текст нарушает правила формирования лексем. К таким ошибкам относятся идентификаторы, длина которых превышает 16 символов, идентификаторы, начинающиеся с цифры, и прочие неопознанные лексемы.

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)

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:

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

1.4 Регулярные выражения

Регулярные выражения – формальный язык шаблонов для поиска и выполнения манипуляций с подстроками в тексте. Регулярное выражение - это формула (pattern, шаблон), задающая правило поиска подстрок в потоке символов.

Все лексемы языка можно описать регулярными выражениями и для каждой задать семантику (какую функцию вызывать, если распознана эта лексема – например, обратиться к таблице служебных слов и искать там, если нет, то ...).

Классы лексем для этого варианта лабораторной работы определяются через регулярные выражения следующим образом.

1. **Идентификаторы.**

$$R_{\text{Identifier}} = \text{[A-Za-z][A-Za-z\_0-9]\{0,15\}(?![A-Za-z\_0-9])}$$

   Первый символ — латинская буква. Далее допускается до 15 символов, включая буквы, цифры и подчёркивания. В конце используется *negative lookahead*, чтобы за идентификатором не следовал символ, допустимый внутри него (иначе это была бы часть более длинного идентификатора).

2. **Комплексные числа в показательной форме.**

$$R_{\text{Complex}} = \text{[+-]?\textbackslash d+(?:\textbackslash .\textbackslash d+)?E[+-]?\textbackslash d+(?:\textbackslash .\textbackslash d+)?}$$

   Опциональный знак в начале числа, целое или десятичное число до `E`, затем обязательный символ `E`, снова опциональный знак, и целое или десятичное число.

3. **Оператор присваивания.**

$$R_{\text{Assign}} = \text{\textbackslash :=}$$

4. **Арифметические операторы.**

$$R_{\text{ArithmeticOp}} = \text{[+\textbackslash -\textbackslash */\textbackslash \^{}]}$$

5. **Левая скобка.**

$$R_{\text{LeftParen}} = \text{\textbackslash (}$$

6. **Правая скобка.**

$$R_{\text{RightParen}} = \text{\textbackslash)}$$

7. **Разделитель выражений.**

$$R_{\text{Separator}} = \text{\textbackslash |}$$

8. **Комментарии.**

$$R_{\text{Comment}} = \#.*$$

В класс ошибок попадают все лексемы, неподошедшие ни под какой другой класс. Некорректная лексема считывается с помощью следующего регулярного выражения:

$$R_{\text{Error}} = .[\hat{}|()+\text{-}*/\hat{}:=\text{s}]*$$

Это регулярное выражение считывает как минимум один символ, чтобы анализ текста не завис. Затем регулярное выражение считывает все символы до первого пробела, арифметического оператора, левой или правой скобки, или оператора присваивания.

1.5 Алгоритм лексического анализа

Алгоритм лексического анализа состоит из следующих шагов:

1. **Инициализация:** указатель устанавливается на начало строки.
2. **Определение типа лексемы:** регулярные выражения, соответствующие различным типам лексем, перебираются в цикле до первого совпадения. Если регулярное выражение не найдено, то применяется регулярное выражение R_{Error}.
3. **Извлечение лексемы:** с помощью регулярного выражения лексема извлекается из строки и сохраняется в списке лексем с указанием типа лексемы.
4. **Вычисление значения лексемы:** значение каждой лексемы определяется посредством вызова функции обработчика, соответствующей типу лексемы. Функция обработчик получает на вход текст лексемы и возвращает некоторый объект, представляющий её значение.
5. **Сохранение лексемы:** текст лексемы, её тип и значение сохраняются в итоговый список лексем.
6. **Сдвиг указателя:** указатель сдвигается на следующий непросмотренный символ.
7. **Завершение анализа:** алгоритм завершается, когда указатель сдвигается за пределы строки. То есть, когда вся строка разобрана на лексемы.

Сложность такого алгортма без учёта сложности функций вычисления значений лексем – $O(n)$, где n – количество символов в исходном тексте программы.

1.6 Синтаксическая диаграмма

Синтаксическая диаграмма реализованной программы представлена на Рис. 2.

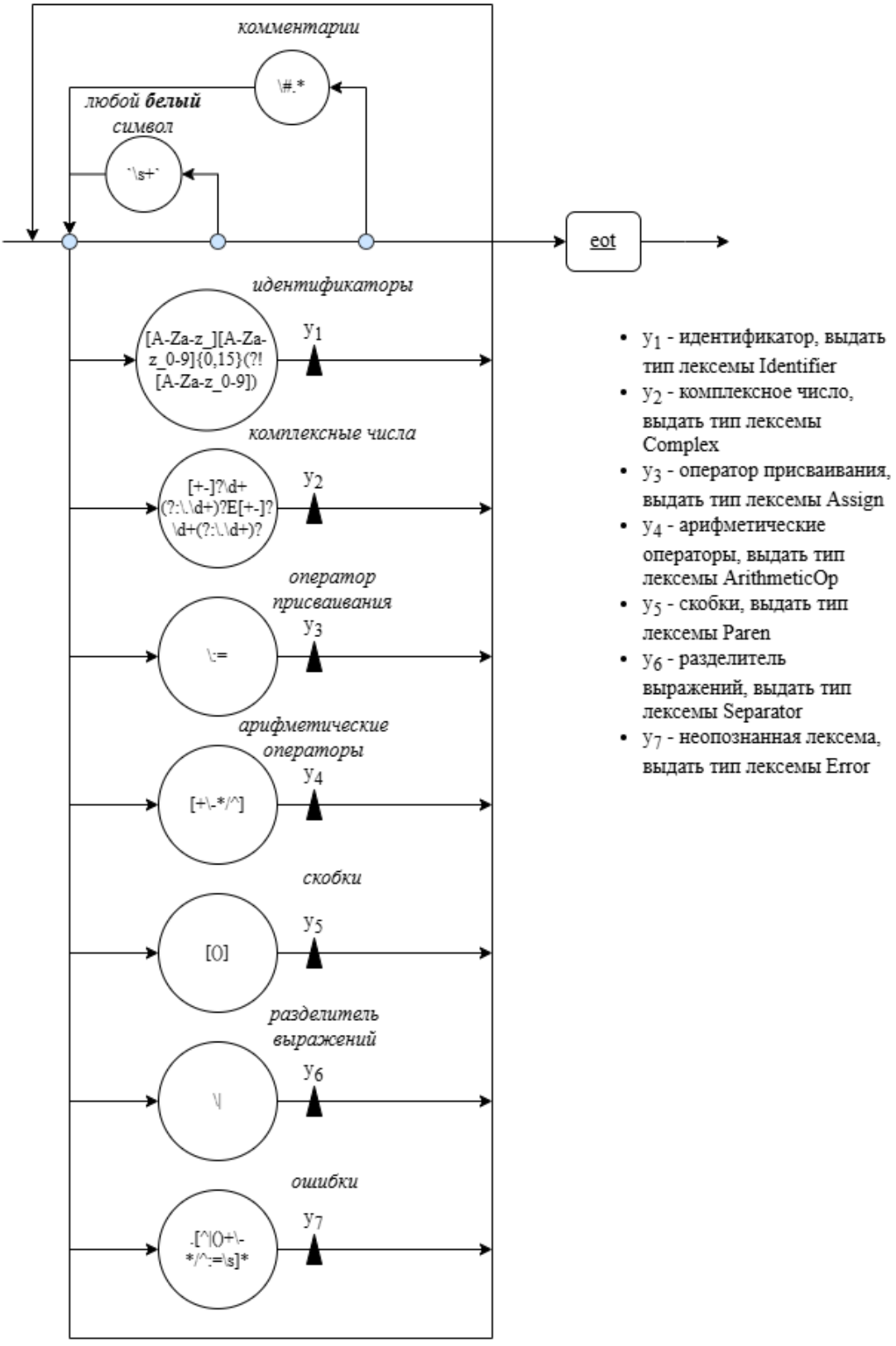

Рис. 2. Синтаксическая диаграмма реализованной программы.

2 Особенности реализации

2.1 Общая структура программы

Программа состоит из двух файлов:

- `lexer.py` – содержит небольшую библиотеку для создания лексических анализаторов для произвольных языков. Файл содержит классы `Lexem`, `LexemeType` и `Lexer`.
- `main.py` – файл содержит список типов лексем и их регулярных выражений – (`LEXEME_TYPES`), а также два обработчика для вычисления значений лексем – `exp_form_to_complex` и `IdentifierMapper`. Также в файле определены функции: `analyze_and_print_table` – запускает лексический анализ и выводит результаты в консоль в виде таблицы, `main` – обрабатывает пользовательский ввод.

2.2 Класс Lexem

Класс Lexem это простой датакласс для представления лексем. Код определения класса представлен в листинге 1. В классе есть всего три поля строкового типа: `text` – текст лексемы, `type_name` – название типа лексемы, `value` – значение лексемы.

Листинг 1. Определение класса Lexem.

```
1 @dataclass
2 class Lexem:
3     text: str
4     type_name: str
5     value: str
```

2.3 Класс LexemeType

Класс LexemeType представляет собой тип лексемы. Код определения класса представлен в листинге 2. В классе всего три поля: `name` – строка с названием типа лексемы, `regex` – регулярное выражение, соответствующее типу лексемы, `value_func` – функция для вычисления значения лексемы.

В классе определён единственный метод `consume`. Он принимает два параметра: `self` – ссылку на объект класса, и строку `text` с текстом программы, из которого нужно попытаться извлечь лексему. Возвращает кортеж из двух элементов: объект класса Lexem, если лексему удалось извлечь, иначе None, и строку с текстом программы, оставшимся после извлечения лексемы.

Листинг 2. Определение класса LexemeType.

```
1 class LexemeType:
2     def __init__(
3         self,
```

```
4           name: str,
5           pattern: str,
6           value_func: Callable[[str], str] = lambda _: "",
7       ):
8           self.name = name
9           self.regex = re.compile(r"\s*(" + pattern + ")")
10          self.value_func = value_func
11
12      def consume(self, text: str) -> tuple[Lexem | None, str]:
13          match = self.regex.match(text)
14          if match:
15              lexeme_text = match.group(1)
16              value = self.value_func(lexeme_text)
17              rest = text[match.end() :]
18              return Lexem(lexeme_text, self.name, value), rest
19          return None, text
```

2.4 Класс Lexer

Класс Lexer представляет собой лексический анализатор. Код определения класса представлен в листинге 3. В классе определены следующие поля:

- `lexeme_types` – список объектов класса `LexemeType`
- `error_regex` – строка, содержащая регулярное выражение для захвата ошибочных лексем
- `skip_types` – список строк с названиями типов лексем, которые следует пропускать при анализе (например, комментарии)

В классе определено два метода: `analyze` и вспомогательный `_consume_error`.

Метод `analyze` выполняет лексический разбор входного текста. Он принимает строку `text`, содержащую текст программы, и возвращает список объектов типа `Lexem`. Метод поочерёдно применяет каждый тип лексемы из `lexeme_types`, пытаясь извлечь очередную лексему. Если хотя бы один тип лексемы успешно извлекает лексему, она добавляется в результат (если её тип не входит в `skip_types`), а оставшийся текст анализируется далее. Если ни одна лексема не подошла, вызывается метод `_consume_error` для обработки ошибки.

Метод `_consume_error` используется для обработки ситуаций, когда входной фрагмент не соответствует ни одному из допустимых шаблонов. Он использует `error_regex` для поиска ошибочной последовательности символов, сообщает об ошибке в консоль и создаёт лексему с типом `"ERROR"`. Возвращает эту ошибочную лексему и оставшийся текст.

Листинг 3. Определение класса Lexer.

```
1   class Lexer:
2       def __init__(
3           self,
4           lexeme_types: Iterable[LexemeType],
5           error_regex: str,
```

```
6           skip_types: Iterable[str] = [],
7       ):
8           self.lexeme_types = lexeme_types
9           self.skip_types = skip_types
10          self.error_regex = re.compile(r"\s*(" + error_regex + ")")
11
12      def analyze(self, text: str) -> list[Lexem]:
13          lexems: list[Lexem] = []
14          while text.strip():
15              for lex_type in self.lexeme_types:
16                  lexem, new_text = lex_type.consume(text)
17                  if lexem:
18                      if lexem.type_name not in self.skip_types:
19                          lexems.append(lexem)
20                      text = new_text
21                      break
22              else:
23                  error_lexeme, text = self._consume_error(text)
24          return lexems
25
26      def _consume_error(self, text: str) -> tuple[Lexem, str]:
27          match = self.error_regex.match(text)
28          err_text = match.group(1) if match else text.strip()
29          print(f"Недопустимая лексема: {err_text}")
30          rest = text[match.end() :] if match else ""
31          return Lexem(err_text, "ERROR", ""), rest
```

2.5 Класс `IdentifierMapper`

Класс `IdentifierMapper` используется для сопоставления идентификаторов с уникальными значениями. Код класса приведён в листинге 4. В классе определены два поля: `id_table: dict[str, str]` — таблица, содержащая отображения идентификаторов на строки с их порядковыми номерами, и `counter: int` — счётчик, увеличиваемый при каждом новом идентификаторе.

Метод `__call__(self, lex_text: str) → str` проверяет, был ли ранее встречен переданный идентификатор. Если нет — добавляет его в таблицу, присваивая уникальный номер, и возвращает соответствующее строковое представление.

Листинг 4. Класс IdentifierMapper.

```
1   class IdentifierMapper:
2       def __init__(self):
3           self.id_table = {}
4           self.counter = 0
5
6       def __call__(self, lex_text: str) -> str:
7           if lex_text not in self.id_table:
8               self.id_table[lex_text] = f"{lex_text} : {self.counter}"
9               self.counter += 1
10          return self.id_table[lex_text]
```

2.6 Функция `exp_form_to_complex`

Функция `exp_form_to_complex`, показанная в листинге 5, принимает один параметр `exp_str: str` — строку с числом в экспоненциальной форме. Возвращает строку `str`, представляющую это число в виде комплексного числа в алгебраической форме $a + i \cdot b$, где $a = r \cdot \cos(\varphi)$, $b = r \cdot \sin(\varphi)$.

Листинг 5. Функция exp_form_to_complex.

```
def exp_form_to_complex(exp_str: str) -> str:
    base, exponent = exp_str.split("E")
    r = float(base)
    phi = float(exponent)

    a = r * math.cos(phi)
    b = r * math.sin(phi)

    return f"{a:.2f} + i * {b:.2f}"
```

2.7 Список `LEXEME_TYPES`

Словарь `LEXEME_TYPES` (листинг 6) содержит определения всех типов лексем, используемых в лексическом анализе. Ключами являются строковые названия типов, значениями — экземпляры класса `LexemeType`. Каждая лексема описывается с помощью регулярного выражения и, при необходимости, функцией обработки значения (например, `IdentifierMapper` или `exp_form_to_complex`).

Листинг 6. Определение лексем в LEXEME_TYPES.

```
LEXEME_TYPES: dict[str, LexemeType] = {
    "IDENTIFIER": LexemeType(
        "IDENTIFIER", r"[A-Za-z][A-Za-z_0-9]{0,15}(?![A-Za-z_0-9])", IdentifierMapper()
    ),
    "COMPLEX": LexemeType(
        "COMPLEX",
        r"[+-]?\d+(?:\.\d+)?E[+-]?\d+(?:\.\d+)?",
        exp_form_to_complex,
    ),
    "ASSIGN": LexemeType("ASSIGN", r"\:="),
    "ARITHMETIC_OP": LexemeType("ARITHMETIC_OP", r"[+\-*/^]"),
    "LPAREN": LexemeType("LPAREN", r"\("),
    "RPAREN": LexemeType("RPAREN", r"\)"),
    "SEPARATOR": LexemeType("SEPARATOR", r"\|"),
    "COMMENT": LexemeType("COMMENT", r"\#.*"),
}
```

2.8 Функция `analyze_and_print_table`

Функция `analyze_and_print_table`, представленная в листинге 7, принимает один параметр `code: str` — строку с текстом программы. Она создаёт объект `Lexer`, выполняет анализ текста на лексемы и выводит результат в виде таблицы, используя

библиотеку `PrettyTable`. Каждая строка таблицы содержит текст лексемы, её тип и вычисленное значение.

Листинг 7. Функция analyze_and_print_table.

```
def analyze_and_print_table(code: str):
    lexer = Lexer(LEXEME_TYPES.values(), ERROR_REGEX, skip_types=["COMMENT"])
    lexemes = lexer.analyze(code)

    table = PrettyTable(["Лексема", "Тип лексемы", "Значение"])
    for l in lexemes:
        table.add_row([l.text, l.type_name, l.value])

    print(table)
    print()

    LEXEME_TYPES["IDENTIFIER"].value_func = IdentifierMapper()
```

2.9 Функция `main`

Функция `main` (листинг 8) организует интерактивный интерфейс пользователя. Не принимает параметров. Пользователь вводит строку `file_name`, содержащую путь к файлу. Если файл существует, его содержимое читается и передаётся в функцию `analyze_and_print_table`. При вводе строки `"exit"` программа завершает выполнение. Если файл не найден, пользователю выводится сообщение об ошибке.

Листинг 8. Функция main.

```
def main():
    while True:
        file_name = input(
            "Введите название файла для анализа или( 'exit' для выхода): "
        )

        if file_name.lower() == "exit":
            print("Завершаю программу.")
            break

        if not os.path.isfile(file_name):
            print(f"Файл '{file_name}' не найден. Попробуйте снова.")
            continue

        with open(file_name, "r", encoding="utf-8") as file:
            code = file.read()

        analyze_and_print_table(code)
```

3 Результаты работы программы

На Рис. 3-5 представлены результаты работы лексического анализатора, запущенного на файлах с тремя различными программами (см. листинги 9-11).

Листинг 9. Программа 1.

```
1  alpha:=2E3.1415|beta:=4E0|theta:=alpha+beta+−3E−1.57
2   | dzeta := alpha + (2.1E2.1 − 22E2)
```

Лексема	Тип лексемы	Значение
alpha	IDENTIFIER	alpha : 0
:=	ASSIGN	
2E3.1415	COMPLEX	-2.00 + i * 0.00
|	SEPARATOR	
beta	IDENTIFIER	beta : 1
:=	ASSIGN	
4E0	COMPLEX	4.00 + i * 0.00
|	SEPARATOR	
theta	IDENTIFIER	theta : 2
:=	ASSIGN	
alpha	IDENTIFIER	alpha : 0
+	ARITHMETIC_OP	
beta	IDENTIFIER	beta : 1
+	ARITHMETIC_OP	
-3E-1.57	COMPLEX	-0.00 + i * 3.00
|	SEPARATOR	
dzeta	IDENTIFIER	dzeta : 3
:=	ASSIGN	
alpha	IDENTIFIER	alpha : 0
+	ARITHMETIC_OP	
(	LPAREN	
2.1E2.1	COMPLEX	-1.06 + i * 1.81
-	ARITHMETIC_OP	
22E2	COMPLEX	-9.16 + i * 20.00
)	RPAREN	

Рис. 3. Результат работы лексического анализатора на программе 1 (листинг 9).

Листинг 10. Программа 2.

```
1  abc + +100E−3.1415
2  # some_id_ :=:= 100
3  ThisIdentifierIsTooLong :=:=
```

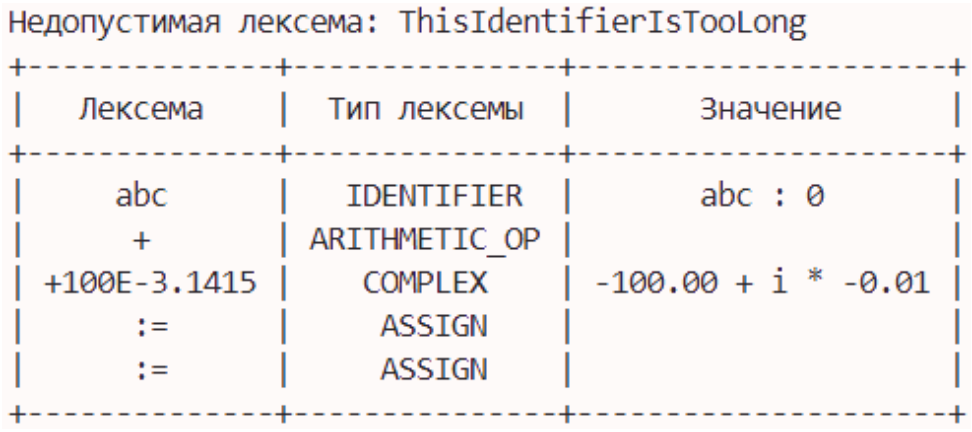

Рис. 4. Результат работы лексического анализатора на программе 2 (листинг 10).

Листинг 11. Программа 3.

```
1  x := 2.5E+3 + y1 |
2  z := 3.1E+ | # число с ошибкой
3  x := x + −2.5E+3 |
4  1_first # идентификатор с цифры
```

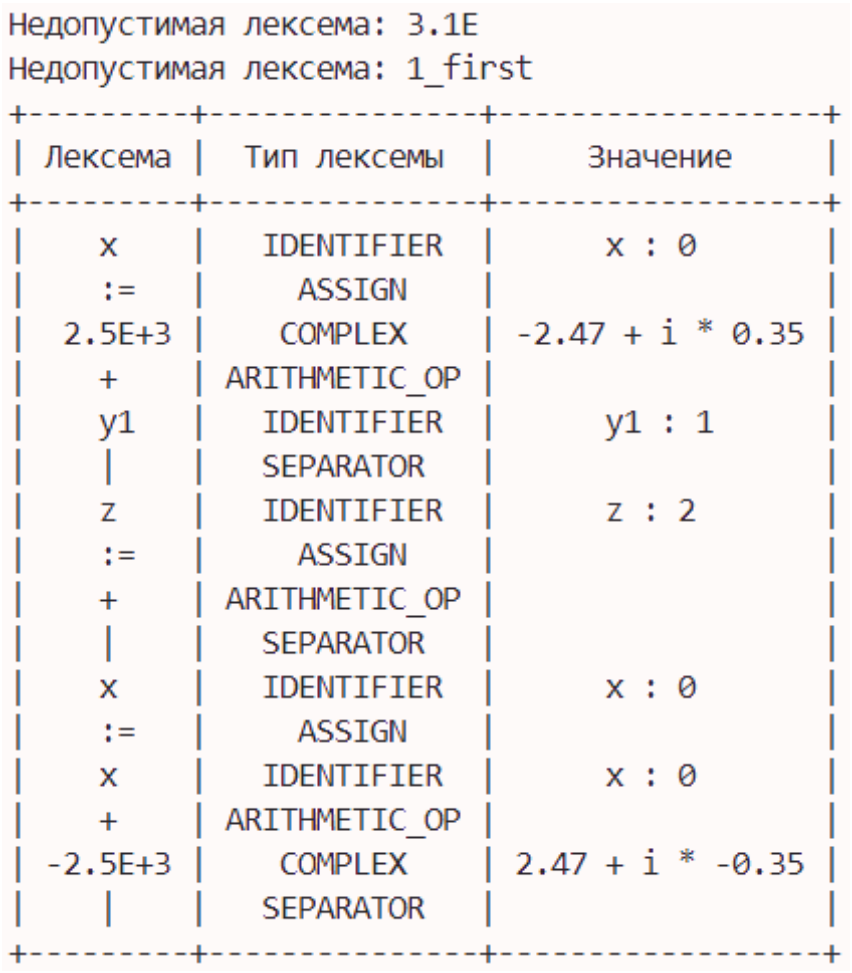

Рис. 5. Результат работы лексического анализатора на программе 3 (листинг 11).

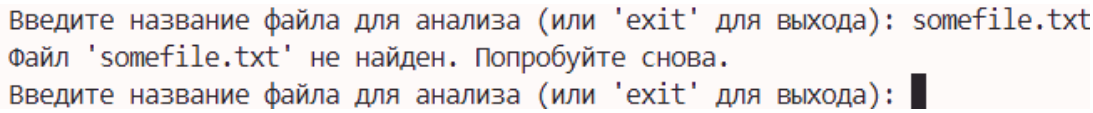

Рис. 6. Реакция программы на некорректный пользовательский ввод.

На Рис. 6 представлена реакция программы на некорректный пользовательский ввод.

Заключение

В ходе выполнения лабораторной работы был разработан лексический анализатор для языка, поддерживающего идентификаторы до 16 символов, комплексные числа в экспоненциальной форме, оператор присваивания (`:=`), арифметические операторы (`+`, `-`, `*`, `/`, `^`), скобки (`(`, `)`), разделитель выражений (`|`), комментарии (`#`). Для каждого типа лексем было разработано отдельное регулярное выражение. Предложенная программная реализация лексического анализатора выводит в консоль таблицу лексем с указанием их типов и значений. Также программа отдельно выводит сообщения об ошибочных лексемах.

Данный язык является автоматным, так как представим в виде синтаксической диаграммы. Автоматные грамматики — самые простые из формальных грамматик. Они являются подмножеством контекстно-свободных грамматик.

Из достоинств выполнения лабораторной работы можно выделить структурирование кода за счёт использования ООП. Вся логика работы лексического анализатора вынесена в отдельный класс `Lexer`. Логика работы с типами лексем в класс `LexemeType`. Также в качестве достоинства можно отметить удобочитаемый вывод таблиц в консоли с помощью библиотеки `PrettyTable`.

Можно выделить два недостатка текущей реализации. Во-первых, классы `Lexer` и `LexemeType` не поддерживают лексемы, в состав которых входят пробельные символы. Это ограничение также касается и лексем из класса «Ошибка», поэтому предложенный лексический анализатор не может обнаружить, например, следующую ошибку: «`:= :=`» – между повторяющимися операторами присваивания находится пробельный символ. Во-вторых, алгоритм, лежащий в основе реализации лексического анализатора, не предусматривает анализ структуры входного текста, поэтому способен обнаружить ошибки только на уровне отдельных лексем.

Функционал программы несложно масштабировать. Классы `Lexer`, `LexemeType` и `Lexem` получились достаточно универсальными. Для того, чтобы добавить обработку дополнительных типов лексем, достаточно расширить словарь `LEXEME_TYPES`. Кроме того, класс `LexemeType` позволяет использовать любые `Callable` объекты в качестве обработчиков для получения значений лексем. Таким образом код, написанный для решения конкретного варианта лабораторной работы, легко можно адаптировать для решения других вариантов.

На выполнение лабораторной работы ушло около 12 часов. Работа была выполнена в среде разработки Visual Studio Code. Программа написана на Python версии 3.13.

Список литературы

[1] Востров, А.В. Курс лекций по дисциплине «Математическая логика». URL https://tema.spbstu.ru/compiler/ (дата обращения 01.04.2025 г.)

[2] Лутц, М. Изучаем Python. 5-е изд. / М. Лутц. — СПб.: Питер, 2019. — 1216 с.

[3] Фридл, Дж. Регулярные выражения = Mastering Regular Expressions / Дж. Фридл. — СПб.: Питер, 2001. — 352 с. — (Библиотека программиста).