|  |  |  |
| --- | --- | --- |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |
| Институт Информационных Технологий | | |
| Кафедра Вычислительной техники | | |

| **ОТЧЕТ О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ №4**  «Реализация лексического анализатора на Python» | |
| --- | --- |
|  | |
| **по дисциплине** | |
| **«**Теория формальных языков**»** | |
|  | |
| Выполнил студент группы ИКБО-43-23 | *Жаворонкова А.А.* |
| Принял ассистент | *Цынгалёв П.С.* |

| Практическая работа выполнена | «\_\_»\_\_\_\_\_\_\_2024 г. |  |
| --- | --- | --- |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_2024 г. |  |

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ](#_heading=h.4wf95dhyemmb) 3

[1 ЦЕЛЬ 4](#_heading=h.139rbsry1568)

[2 ЗАДАНИЕ 5](#_heading=h.pjw4hjhizqqe)

[2.1 Формулировка задания 5](#_heading=h.z0nv9dxm0oqk)

[2.2 Математическая модель 6](#_heading=h.aiye6uk0akm3)

[2.3 Тестирование программы 8](#_heading=h.lhwuhxvu00kg)

[ЗАКЛЮЧЕНИЕ 12](#_heading=h.2n8hmiceyi3u)

[ПРИЛОЖЕНИЯ 14](#_heading=h.bfaiz9c6bn22)

# ВВЕДЕНИЕ

Изучение теории формальных языков занимает важное место в образовательной программе по программной инженерии. Эта область, являющаяся неотъемлемой частью компьютерных наук и теоретической информатики, посвящена анализу структуры, свойств формальных языков и моделированию их обработки. В условиях современного мира, где постоянно растут объемы данных и необходимость в эффективных методах их обработки, хранения и передачи, понимание основ формальных языков становится особенно актуальным.

Формальные языки находят применение в различных областях: от описания синтаксиса языков программирования до разработки сетевых протоколов и создания компиляторов. Они позволяют формализовать взаимодействие как между человеком и компьютером, так и между различными компонентами программного обеспечения. Основу теории формальных языков составляют такие понятия, как грамматики, автоматы и алгебраические структуры, которые используются для описания синтаксиса и семантики языков.

В данном отчете мы рассмотрим программу реализацию лексического анализатора на языке Python.

# 1 ЦЕЛЬ

Реализовать лексический анализатор на языке Python, согласно варианту КР.

# 2 ЗАДАНИЕ

## 2.1 Формулировка задания

На выбранном ЯП реализовать лексический анализатор согласно варианту.

Согласно индивидуальному варианту задания на курсовую работу грамматика языка включает следующие синтаксические конструкции:

<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

<идентификатор> ::= <буква> { <буква> | <цифра> }

<число> ::= {/< цифра> /}

<ключевое\_слово> ::= read | write | if | then | else | for | to | while | do | true | false | or | and | not | as

<разделитель> ::= { | } | % | ! | $ | , | ; | [ | ] | : | ( | ) | + | - | \* | / | = | <> | < | <= | > | >= | /\* | \*/

<программа>::= «{» {/ (<описание> | <оператор>) ; /} «}»

<описание>::= <тип> <идентификатор> { , <идентификатор> }

<тип>::= % | ! | $

<оператор> ::= <присваивания> | <условный> | <фиксированного\_цикла> | <условного\_цикла> | <составной> | <ввода> | <вывода>

<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>

<составной>:: = «[» <оператор> { : <оператор> } «]»

<ввода>:: = read «(»<идентификатор> {, <идентификатор> } «)»

<вывода>:: = write «(»<выражение> {, <выражение> } «)»

<выражение>:: = <сумма> | <выражение> (< > | = | < | <= | > | >=) <сумма>

<сумма> ::= <произведение> { (+ | - | or) <произведение>}

<произведение>:: = <множитель> { ( \* | / | and) <множитель>}

<множитель>:: = <идентификатор> | <число> | <логическая\_константа> | not <множитель> | «(»<выражение>«)»

<логическая\_константа>:: = true | false

<целое>::= <двоичное> | <восьмеричное> | <десятичное> | <шестнадцатеричное>

<двоичное>::= {/ 0 | 1 /} (B | b)

<восьмеричное>::= {/ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 /} (O | o)

<десятичное>::= {/ <цифра> /} [D | d]

<шестнадцатеричное>::= <цифра> {<цифра> | A | B | C | D | E | F | a |

b | c | d | e | f} (H | h)

<действительное>::= <числовая\_строка> <порядок> |

[<числовая\_строка>] . <числовая\_строка> [порядок]

<числовая\_строка>::= {/ <цифра> /}

<порядок>::= ( E | e )[+ | -] <числовая\_строка>

## 2.2 Математическая модель

Лексический анализатор — это программа, которая обрабатывает исходный текст программы и преобразует его в последовательность лексем, представляющих собой минимальные смысловые единицы. В модельном языке программирования выделяются следующие типы лексем:

● ключевые слова;

● ограничители;

● числа;

● идентификаторы.

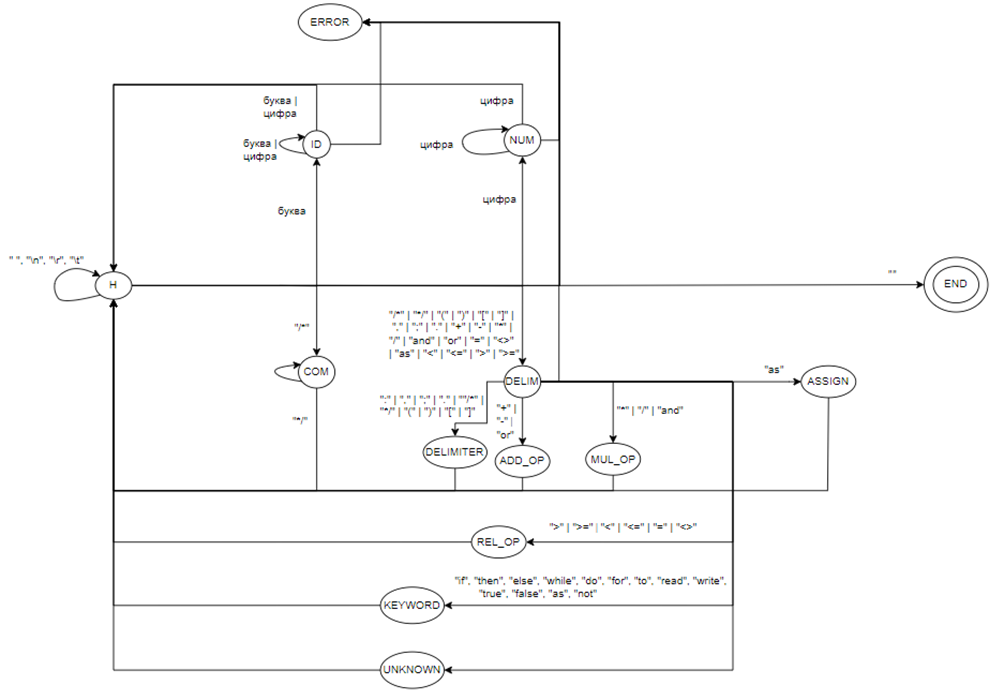
При разработке лексического анализатора ключевые слова и ограничители определяются заранее, а идентификаторы и числовые константы вычисляются в процессе анализа исходного текста.

Кроме ключевых слов, идентификаторов и числовых констант, в исходном коде могут встречаться пробельные символы (пробелы, табуляция, переносы строк, возврат каретки) и комментарии. Лексический анализатор должен уметь игнорировать эти символы и не воспринимать их как часть лексем.

Лексический анализ выполняется с использованием регулярной грамматики, которая эквивалентна конечному автомату. Для разработки лексического анализатора необходимо построить диаграмму состояний конечного автомата, которая описывает процесс анализа текста.

Лексический анализатор использует состояния для определения типа текущей лексемы и применяет различные правила для распознавания символов и конструкций языка программирования. Это позволяет эффективно обрабатывать текст программы и выделять из него смысловые единицы для дальнейшей обработки, такой как синтаксический и семантический анализ.

Исходный код лексического анализатора описан в Приложении А.



**Рисунок 1 – Диаграмма состояний лексического анализатора**

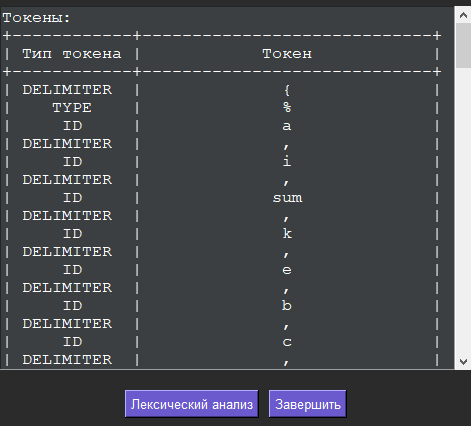
## 2.3 Тестирование программы

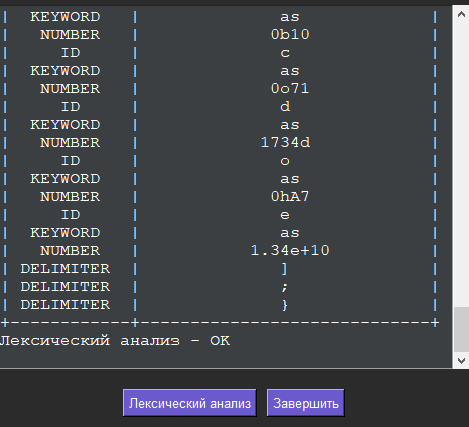
Создано графическое приложение для анализа исходного текста программы на модельном языке. Приложение выполняет лексический анализ введенного кода и выдает результат: либо подтверждает корректность программы, либо сообщает об обнаруженной ошибке. Далее представлены примеры работы приложения. Исходный код программы показан в Листинге 1.

*Листинг 1 – Тестовая программа 1*

| {  % a, i, sum, k, e, b, c, d, o;  ! r, s;  $ flag;  [  a as 5 /\* Это комментарий \*/  if (a > 5) then  write('a больше 5')  else  write('a меньше или равно 5')  sum as 0  for i as 0 to a do  write('i = ', i):  sum as sum + i:  if (sum >= 10) then  write('sum больше или равно 10'):  k as 10  while (k <> 0) do  write('k = ', k):  k as k - 1:  flag as true  if (flag) then  write('flag имеет значение true')  else  write('flag имеет значение false')  b as 0b10  c as 0o71  d as 1734d  o as 0hA7  e as 1.34e+10  ];  } |
| --- |

Данная программа лексически корректна. Это продемонстрировано на рисунке 2.1 и рисунке 2.2, где показан вывод таблицы токенов и корректность выполнения.

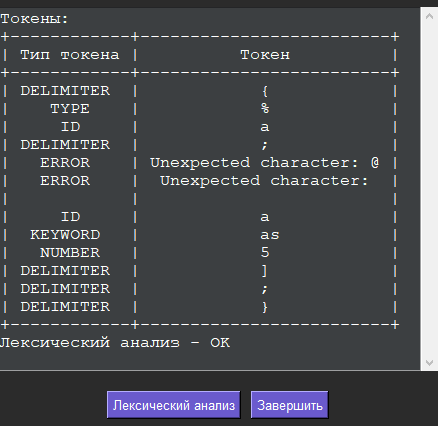
**  
Рисунок 2.1 – Тестирование лексического анализа**

**  
Рисунок 2.2 – Тестирование лексического анализа**

Исходный код программы, содержащий лексическую ошибку, приведен в листинге 2.

*Листинг 2 – Тестовая программа*

| {  % a;  @  a as 5 /\* Это комментарий \*/  ];  } |
| --- |

**  
Рисунок 3 – Тестирование ошибки лексического анализ**

# 

# ЗАКЛЮЧЕНИЕ

В ходе выполнения практической работы были достигнуты следующие результаты:

Разработан лексический анализатор, выполняющий разбиение последовательности символов исходного текста программы на лексемы. Лексический анализатор реализован на языке Python в виде класса Lexer.

Созданы графические представления: недетерминированного конечного автомата, детерминированного конечного автомата и минимального детерминированного конечного автомата, соответствующего варианту задания.

Выполнена реализация лексического анализатора в соответствии с индивидуальным вариантом курсовой работы на языке Python.

Проведено тестирование программы, результаты которого включены в отчет.

Таким образом, поставленная цель работы — создание лексического анализатора на языке Python в соответствии с вариантом курсовой работы — успешно выполнена.

**СПИСОК ИСПОЛЬЗОВАННЫХ 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 – Запуск лексического анализа

**Приложение А**

Класс лексического анализатора

*Листинг А.1 – lexer.py*

| from enum import Enum  class Lexer:  # Состояния автомата  class LexerState(Enum):  H = "H" # Начальное состояние  ID = "ID" # Идентификаторы  NUM = "NUM" # Числа  COM = "COM" # Комментарии  ALE = "ALE" # Операции отношения  NEQ = "NEQ" # Неравенство  DELIM = "DELIM" # Разделители  STR = "STR" # Строковые литералы  ERROR = "ERROR" # Состояние ошибки  END = "END" # Конечное состояние  # Ключевые слова и типы данных  TW = [  "if", "then", "else", "while", "do", "for", "to", "read", "write",  "true", "false", "as", "not", "or", "and"  ]  # Типы данных  TYPES = [  "%", "!", "$"  ]  # Разделители и операторы  TD = [  "{", "}", "[", "]", "(", ")", ",", ";", ":", "<>", "=", "<", "<=", ">", ">=", "+", "-", "\*", "/", "or", "and"  ]  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.state = self.LexerState.H # Начальное состояние  # Обработка ввода и переход к следующему символу  def advance\_position(self):  self.pos += 1  self.current\_char = self.text[self.pos] if self.pos < len(self.text) else None  def append\_token(self, token\_type, value):  self.tokens.append((token\_type, value))  def skip\_whitespace(self):  while self.current\_char and self.current\_char in ' \n\r\t':  self.advance\_position() |
| --- |

*Продолжение листинга А.1*

| # Переходы между состояниями  def process\_state(self):  if self.state == self.LexerState.H:  self.skip\_whitespace()  if not self.current\_char:  self.state = self.LexerState.END # Завершаем анализ  return  if self.current\_char in self.TYPES:  self.state = self.LexerState.ALE  elif self.current\_char.isalpha():  self.state = self.LexerState.ID  elif self.current\_char.isdigit():  self.state = self.LexerState.NUM  elif self.current\_char == "'":  self.state = self.LexerState.STR  elif self.current\_char == '/' and self.text[self.pos + 1:self.pos + 2] == '\*':  self.state = self.LexerState.COM  elif self.current\_char in self.TD:  self.state = self.LexerState.DELIM  else:  self.state = self.LexerState.ERROR # Ошибка при неправильном символе  self.append\_token('ERROR', f"Unexpected character: {self.current\_char}")  self.advance\_position()  elif self.state == self.LexerState.ID:  self.handle\_identifier\_or\_keyword()  self.state = self.LexerState.H  elif self.state == self.LexerState.NUM:  self.handle\_number()  self.state = self.LexerState.H  elif self.state == self.LexerState.STR:  self.handle\_string()  self.state = self.LexerState.H  elif self.state == self.LexerState.COM:  self.handle\_comment()  self.state = self.LexerState.H  elif self.state == self.LexerState.ALE:  self.handle\_type\_symbol()  self.state = self.LexerState.H  elif self.state == self.LexerState.DELIM:  self.handle\_delimiter\_or\_operator()  self.state = self.LexerState.H  elif self.state == self.LexerState.ERROR:  # Ошибка в состоянии, можно добавить обработку или выход из цикла  self.append\_token('ERROR', f"Unexpected character: {self.current\_char}")  self.state = self.LexerState.H # Переход к началу после ошибки  self.advance\_position() |
| --- |

*Продолжение листинга А.1*

| def handle\_identifier\_or\_keyword(self):  start = self.pos  while self.current\_char and (self.current\_char.isalnum() or self.current\_char == '\_'):  self.advance\_position()  text = self.text[start:self.pos]  if text in self.TYPES:  self.append\_token('TYPE', text)  elif text in self.TW:  self.append\_token('KEYWORD', text)  else:  self.append\_token('ID', text)  def handle\_number(self):  start = self.pos  if self.current\_char == '0': # Возможные специальные форматы  self.advance\_position()  if self.current\_char in 'Bb': # Двоичное число  self.advance\_position()  while self.current\_char and self.current\_char in '01':  self.advance\_position()  self.append\_token('NUMBER', self.text[start:self.pos])  return  elif self.current\_char in 'Oo': # Восьмеричное число  self.advance\_position()  while self.current\_char and self.current\_char in '01234567':  self.advance\_position()  self.append\_token('NUMBER', self.text[start:self.pos])  return  elif self.current\_char in 'Hh': # Шестнадцатеричное число  self.advance\_position()  while self.current\_char and (self.current\_char.isdigit() or self.current\_char in 'ABCDEFabcdef'):  self.advance\_position()  self.append\_token('NUMBER', self.text[start:self.pos])  return  is\_real = False  while self.current\_char and self.current\_char.isdigit():  self.advance\_position()  if self.current\_char == '.': # Возможное действительное число  is\_real = True  self.advance\_position()  while self.current\_char and self.current\_char.isdigit():  self.advance\_position()  if self.current\_char in 'Ee': # Порядок  is\_real = True  self.advance\_position()  if self.current\_char in '+-':  self.advance\_position()  while self.current\_char and self.current\_char.isdigit():  self.advance\_position()  if is\_real:  self.append\_token('NUMBER', self.text[start:self.pos])  return |
| --- |

*Окончание листинга А.1*

| if self.current\_char in 'Dd': # Десятичное число с суффиксом  self.advance\_position()  self.append\_token('NUMBER', self.text[start:self.pos])  return  self.append\_token('NUMBER', self.text[start:self.pos])  def handle\_comment(self):  self.advance\_position() # Спускаемся с /\*  while self.current\_char and not (self.current\_char == '\*' and self.text[self.pos + 1:self.pos + 2] == '/'):  self.advance\_position()  self.advance\_position() # Спускаемся на \*  self.advance\_position() # И на /  def handle\_delimiter\_or\_operator(self):  start = self.pos  self.advance\_position()  while self.current\_char and (self.text[start:self.pos + 1] in self.TD):  self.advance\_position()  text = self.text[start:self.pos]  if text in ["<>", "=", "<", "<=", ">", ">="]:  self.append\_token('REL\_OP', text)  elif text in ["+", "-", "or"]:  self.append\_token('ADD\_OP', text)  elif text in ["\*", "/", "and"]:  self.append\_token('MUL\_OP', text)  elif text in self.TD:  self.append\_token('DELIMITER', text)  else:  self.append\_token('UNKNOWN', text)  def handle\_type\_symbol(self):  if self.current\_char in self.TYPES:  self.append\_token('TYPE', self.current\_char)  self.advance\_position()  def handle\_string(self):  self.advance\_position()  start = self.pos  while self.current\_char and self.current\_char != "'":  self.advance\_position()  text = self.text[start:self.pos]  self.append\_token('STRING', f"'{text}'")  self.advance\_position()  # Циклическая обработка входной строки  def tokenize(self):  while self.state != self.LexerState.END:  self.process\_state()  # Если есть ошибка, токены ошибки тоже добавляются  if self.state == self.LexerState.ERROR:  self.append\_token('ERROR', "Unexpected end of input")  return self.tokens |
| --- |

**Приложение Б**

Запуск лексического анализа

*Листинг Б.1 – main.py*

| import tkinter as tk  from tkinter import scrolledtext, messagebox  from lex import Lexer  from prettytable import PrettyTable  # Исходный код  code = '''  {  % a;  @  a as 5 /\* Это комментарий \*/  ];  }  '''  def run\_lexical\_analysis():  try:  lexer = Lexer(code)  tokens = lexer.tokenize()  table = PrettyTable()  table.field\_names = ["Тип токена", "Токен"]  for token in tokens:  table.add\_row(token)  output\_text.delete(1.0, tk.END)  output\_text.insert(tk.END, "Токены:\n")  output\_text.insert(tk.END, str(table))  output\_text.insert(tk.END, "\nЛексический анализ - OK\n")  except Exception as e:  messagebox.showerror("Ошибка", f"Ошибка лексического анализа: {e}")  def exit\_application():  root.destroy()  # Создание основного окна  root = tk.Tk()  root.title("Анализатор кода")  # Цветовая схема  bg\_color = "#2b2b2b" # Темно-серый фон  frame\_color = "#3c3f41" # Серый для рамок  btn\_color = "#6a5acd" # Темно-фиолетовый  text\_color = "#ffffff" # Белый текст  root.configure(bg=bg\_color)  # Текстовое поле для вывода  output\_text = scrolledtext.ScrolledText(root, wrap=tk.WORD, width=80, height=20, bg=frame\_color, fg=text\_color, insertbackground=text\_color, font=("Courier New", 12))  output\_text.pack(pady=10) |
| --- |

*Окончание листинга Б.1*

| # Панель кнопок  button\_frame = tk.Frame(root, bg=bg\_color)  button\_frame.pack(pady=10)  btn\_lexical = tk.Button(button\_frame, text="Лексический анализ", command=run\_lexical\_analysis, bg=btn\_color, fg=text\_color, font=("Arial", 10))  btn\_lexical.grid(row=0, column=0, padx=5)  btn\_exit = tk.Button(button\_frame, text="Завершить", command=exit\_application, bg=btn\_color, fg=text\_color, font=("Arial", 10))  btn\_exit.grid(row=0, column=3, padx=5)  # Запуск главного цикла  root.mainloop() |
| --- |