|  |  |  |
| --- | --- | --- |
| 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 №6**  «Реализация синтаксического анализатора (с минимум одной семантической проверкой) 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 3](#_heading=h.pz2177ftd759)

[1 ЦЕЛЬ 4](#_heading=h.lffuteg3hjz3)

[2 ЗАДАНИЕ 5](#_heading=h.q6qtb9qbm75o)

[2.1 Формулировка задания 5](#_heading=h.e16gdee16mpd)

[2.2 Математическая модель 6](#_heading=h.v5tgk5t4njru)

[2.3 Тестирование программы 8](#_heading=h.1owrwdzejya)

[ЗАКЛЮЧЕНИЕ 11](#_heading=h.2n8hmiceyi3u)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 12](#_heading=h.3rdcrjn)

[ПРИЛОЖЕНИЯ 13](#_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 синтаксического анализатора (с минимум одной семантической проверкой) на языке Python.

# 1 ЦЕЛЬ

Реализовать синтаксический анализатор (с минимум одной семантической проверкой) на языке Python, согласно варианту КР.

# 2 ЗАДАНИЕ

## 2.1 Формулировка задания

На выбранном ЯП реализовать синтаксический анализатор (с минимум одной семантической проверкой), согласно варианту КР.

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

1. Синтаксический анализатор в данном коде выполняет разбор исходного текста программы, используя методы рекурсивного спуска (РС). Этот метод подходит для грамматик, которые можно выразить с помощью рекурсивных функций, каждая из которых соответствует одному из нетерминалов грамматики. В данном случае каждая рекурсивная функция обрабатывает определенные конструкции исходного языка, такие как операторы, выражения и объявления.

Для синтаксического анализа, основанного на рекурсивном сплете, грамматические правила можно переписать в виде функций, где каждая функция соответствует одному нетерминалу. Пример грамматической формулы и соответствующих рекурсивных функций для данного кода:

Исходный код синтаксического анализатора приведен в Приложении А.

1. Семантический анализатор. Некоторые особенности модельного языка не могут быть выражены с помощью контекстно-свободной грамматики. К таким особенностям относятся следующие правила

* каждый идентификатор, используемый в программе, должен быть предварительно описан;
* повторное объявление одного и того же идентификатора запрещено;
* в операторе присваивания типы переменных должны быть одинаковыми;
* условные операторы и операторы циклов могут использовать только логические выражения в качестве условий;
* операнды отношений должны быть целочисленными.

Эти правила анализируются на этапе семантического анализа.

Некоторые важные особенности:

* декларация переменных;
* при присваивании проверяется, что переменная объявлена и что тип переменной совпадает с присваиваемый значением;
* для условных операторов условие является логическим выражением;
* Обработка различных типов чисел, а также проверка совместимости типов. Логические выражения проверяются на соответствие типу `$`.

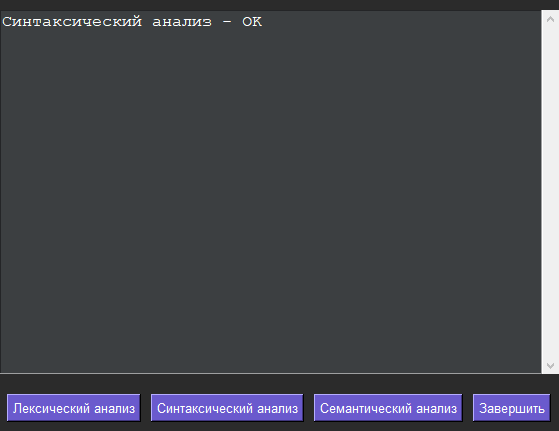
Исходный код синтаксического анализатора приведен в Приложении Б.

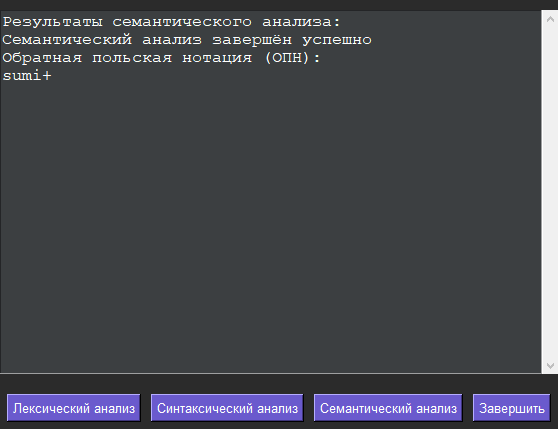
## 2.3 Тестирование программы

Разработано консольное приложение, предназначенное для анализа исходного текста программы, написанной на модельном языке. Программа делает синтаксический и семантический анализ введенного кода. В результате выполнения приложение либо подтверждает правильность программы, либо выводит сообщение об ошибке. Рассмотрим несколько примеров работы приложения. Рассмотрим несколько примеров работы приложения. Исходный код программы приведен в Листинге 1.

*Листинг 1 – Входные данные*

| {  % 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  ];  } |
| --- |

**  
Рисунок 1 – Тестирование синтаксического анализа**

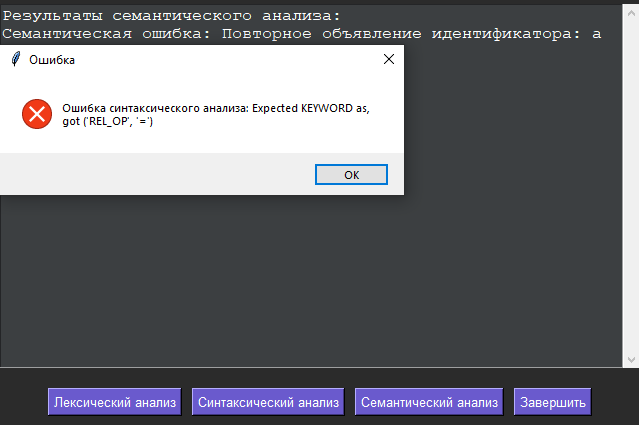
****

**Рисунок 2 – Тестирование семантического анализа**

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

*Листинг 2 – Тестовая программа*

| {  % a;  ! a;  $ flag;  [  a as 5 /\* Это комментарий \*/  flag = true  ];  } |
| --- |

**  
Рисунок 3 – Тестирование ошибки синтаксического и семантического анализа**

# ЗАКЛЮЧЕНИЕ

В ходе выполнения практической работы были достигнуты следующие результаты:

Разработан синтаксический анализатор, реализованный в виде класса Syntax на языке Python. Анализатор использует метод рекурсивного спуска для распознавания входного языка. Для его корректной работы была модифицирована грамматика, включая обработку итеративных синтаксических конструкций.

В код рекурсивных функций добавлены проверки дополнительных семантических условий, таких как контроль повторного объявления переменных.

Проведенное тестирование показало, что программы, написанные синтаксически и семантически корректно, успешно распознаются анализатором, в то время как программы с ошибками отвергаются.

Таким образом, поставленная цель работы — разработка синтаксического анализатора с реализацией как минимум одной семантической проверки 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 синтаксического анализатора

Приложение Б – Класс семантического анализатора

**Приложение А**

Класс синтаксического анализатора

*Листинг А.1 – syntax\_analyzer.py*

| class Syntax:  def \_\_init\_\_(self, tokens):  self.tokens = tokens  self.position = 0  def get\_current\_token(self):  return self.tokens[self.position] if self.position < len(self.tokens) else None  def expect\_token(self, token\_type, value=None):  token = self.get\_current\_token()  if token and token[0] == token\_type and (value is None or token[1] == value):  self.position += 1  return token  raise SyntaxError(f"Expected {token\_type} {value}, got {token}")  def move\_to\_next\_token(self):  self.position += 1  def parse\_program(self):  self.expect\_token('DELIMITER', '{')  while self.get\_current\_token() and self.get\_current\_token()[1] != '}':  token = self.get\_current\_token()  if token[0] == 'TYPE':  self.parse\_declaration()  else:  self.parse\_statement()  self.expect\_token('DELIMITER', ';')  self.expect\_token('DELIMITER', '}')  def parse\_declaration(self):  self.expect\_token('TYPE')  self.parse\_identifier\_list()  def parse\_identifier\_list(self):  self.expect\_token('ID')  while self.get\_current\_token() and self.get\_current\_token()[1] == ',':  self.expect\_token('DELIMITER', ',')  self.expect\_token('ID')  def parse\_if\_statement(self):  self.expect\_token('KEYWORD', 'if')  self.expect\_token('DELIMITER', '(')  self.parse\_expression()  self.expect\_token('DELIMITER', ')')  self.expect\_token('KEYWORD', 'then')  self.parse\_statement()  if self.get\_current\_token() and self.get\_current\_token()[1] == 'else':  self.move\_to\_next\_token() # Move to 'else'  self.parse\_statement() # Handle statement after 'else' |
| --- |

*Продолжение листинга А.1*

| def parse\_statement(self):  token = self.get\_current\_token()  if token[0] == 'DELIMITER' and token[1] == '[':  self.parse\_compound\_statement()  elif token[0] == 'ID':  self.parse\_assignment()  elif token[0] == 'KEYWORD' and token[1] == 'if':  self.parse\_if\_statement()  elif token[0] == 'KEYWORD' and token[1] == 'while':  self.parse\_while\_statement()  elif token[0] == 'KEYWORD' and token[1] == 'for':  self.parse\_for\_loop()  elif token[0] == 'KEYWORD' and token[1] == 'read':  self.parse\_input\_statement()  elif token[0] == 'KEYWORD' and token[1] == 'write':  self.parse\_output\_statement()  elif token[0] == 'KEYWORD' and token[1] == 'else':  raise SyntaxError(f"Unexpected 'else' statement without matching 'if'")  else:  raise SyntaxError(f"Unexpected statement: {token}")  def parse\_compound\_statement(self):  self.expect\_token('DELIMITER', '[')  while self.get\_current\_token() and self.get\_current\_token()[1] != ']':  self.parse\_statement()  if self.get\_current\_token() and self.get\_current\_token()[1] in {':', '\n'}:  self.move\_to\_next\_token()  self.expect\_token('DELIMITER', ']')  def parse\_assignment(self):  self.expect\_token('ID')  self.expect\_token('KEYWORD', 'as')  self.parse\_expression()  def parse\_while\_statement(self):  self.expect\_token('KEYWORD', 'while')  self.parse\_expression()  self.expect\_token('KEYWORD', 'do')  self.parse\_statement()  def parse\_for\_loop(self):  self.expect\_token('KEYWORD', 'for')  self.parse\_assignment()  self.expect\_token('KEYWORD', 'to')  self.parse\_expression()  self.expect\_token('KEYWORD', 'do')  self.parse\_statement()  def parse\_input\_statement(self):  self.expect\_token('KEYWORD', 'read')  self.expect\_token('DELIMITER', '(')  self.expect\_token('ID')  while self.get\_current\_token() and self.get\_current\_token()[1] == ',':  self.expect\_token('DELIMITER', ',')  self.expect\_token('ID')  self.expect\_token('DELIMITER', ')') |
| --- |

*Продолжение листинга А.1*

| def parse\_output\_statement(self):  self.expect\_token('KEYWORD', 'write')  self.expect\_token('DELIMITER', '(')  self.parse\_expression()  while self.get\_current\_token() and self.get\_current\_token()[1] == ',':  self.expect\_token('DELIMITER', ',')  self.parse\_expression()  self.expect\_token('DELIMITER', ')')  def parse\_expression(self):  self.parse\_term()  while self.get\_current\_token() and self.get\_current\_token()[1] in {'+', '-', 'or'}:  self.move\_to\_next\_token()  self.parse\_term()  while self.get\_current\_token() and self.get\_current\_token()[0] == 'REL\_OP': # Handle relational operators  self.move\_to\_next\_token()  self.parse\_term()  def parse\_term(self):  self.parse\_factor()  while self.get\_current\_token() and self.get\_current\_token()[1] in {'\*', '/', 'and'}:  self.move\_to\_next\_token()  self.parse\_factor()  def parse\_factor(self):  token = self.get\_current\_token()  if token[0] in {'ID', 'STRING'}:  self.move\_to\_next\_token()  elif token[0] == 'NUMBER':  self.parse\_number()  elif token[0] == 'KEYWORD' and token[1] in {'true', 'false'}:  self.parse\_boolean\_literal()  elif token[0] == 'DELIMITER' and token[1] == '(':  self.move\_to\_next\_token()  self.parse\_expression()  self.expect\_token('DELIMITER', ')')  elif token[0] == 'KEYWORD' and token[1] == 'not':  self.move\_to\_next\_token()  self.parse\_factor()  else:  raise SyntaxError(f"Unexpected factor: {token}")  def parse\_number(self):  token = self.get\_current\_token()  if self.is\_binary\_number(token):  self.move\_to\_next\_token()  elif self.is\_octal\_number(token):  self.move\_to\_next\_token()  elif self.is\_decimal\_number(token):  self.move\_to\_next\_token()  elif self.is\_hexadecimal\_number(token):  self.move\_to\_next\_token()  elif self.is\_real\_number(token):  self.move\_to\_next\_token()  else:  raise SyntaxError(f"Unexpected number format: {token}") |
| --- |

*Окончание листинга А.1*

| def parse\_boolean\_literal(self):  token = self.get\_current\_token()  if token[0] == 'KEYWORD' and token[1] in {'true', 'false'}:  self.move\_to\_next\_token()  else:  raise SyntaxError(f"Unexpected boolean literal: {token}")  def is\_binary\_number(self, token):  if token[0] != 'NUMBER':  return False  value = token[1]  if value.startswith(('0b', '0B')):  return all(c in '01' for c in value[2:])  return False  def is\_octal\_number(self, token):  if token[0] != 'NUMBER':  return False  value = token[1]  if value.startswith(('0o', '0O')):  return all(c in '01234567' for c in value[2:])  return False  def is\_decimal\_number(self, token):  if token[0] != 'NUMBER':  return False  value = token[1]  if value.isdigit():  return True  if value.endswith(('d', 'D')):  return value[:-1].isdigit()  return False  def is\_hexadecimal\_number(self, token):  if token[0] != 'NUMBER':  return False  value = token[1]  if value.startswith(('0h', '0H')):  valid\_chars = set('0123456789ABCDEFabcdef')  return all(c in valid\_chars for c in value[2:])  return False  def is\_real\_number(self, token):  if token[0] != 'NUMBER':  return False  value = token[1]  try:  float(value)  return True  except ValueError:  return False |
| --- |

**Приложение Б**

Класс семантического анализатора

*Листинг Б.1 – semantic\_analyzer.py*

| class Semantic:  def \_\_init\_\_(self, tokens=None):  self.symbol\_table = {}  self.current\_position = 0  self.tokens = tokens  def get\_current\_token(self):  return self.tokens[self.current\_position] if self.current\_position < len(self.tokens) else None  def advance\_token(self):  self.current\_position += 1  def analyze\_tokens(self):  try:  while self.current\_position < len(self.tokens):  token = self.get\_current\_token()  # Обработка объявления переменных  if token[0] == 'TYPE':  self.handle\_variable\_declaration()  # Обработка операторов (например, присваивания)  elif token[0] == 'ID' and self.peek\_next\_token() and self.peek\_next\_token()[0] == 'KEYWORD' and self.peek\_next\_token()[1] == 'as':  self.handle\_assignment()  # Логическое значение (true/false)  elif token[0] == 'KEYWORD' and token[1] in ['true', 'false']:  self.advance\_token()  return '$'  # Обработка условного оператора if  elif token[0] == 'KEYWORD' and token[1] == 'if':  self.handle\_if\_statement()  # Обработка цикла for  elif token[0] == 'KEYWORD' and token[1] == 'for':  self.handle\_for\_loop()  # Обработка цикла while  elif token[0] == 'KEYWORD' and token[1] == 'while':  self.handle\_while\_loop()  # Игнорирование остальных токенов (например, ';', '{', '}')  elif token[0] in {'DELIMITER', 'COMMENT'}:  self.advance\_token()  else:  raise Exception(f"Неожиданный токен: {token}")  return "Семантический анализ завершён успешно"  except Exception as e:  return f"Семантическая ошибка: {e}" |
| --- |

*Продолжение листинга Б.1*

| def handle\_integer(self):  token = self.get\_current\_token()  if not token or token[0] != 'NUMBER':  raise Exception(f"Ожидалось целое число, но найдено: {token}")  number = token[1]  if self.is\_binary\_number(number):  self.advance\_token()  return '%'  elif self.is\_octal\_number(number):  self.advance\_token()  return '%'  elif self.is\_decimal\_number(number):  self.advance\_token()  return '%'  elif self.is\_hexadecimal\_number(number):  self.advance\_token()  return '%'  else:  raise Exception(f"Недопустимый формат числа: {number}")  def is\_binary\_number(self, number):  return number[:-1].isdigit() and all(c in '01' for c in number[:-1]) and number[-1] in 'Bb'  def is\_octal\_number(self, number):  return number[:-1].isdigit() and all(c in '01234567' for c in number[:-1]) and number[-1] in 'Oo'  def is\_decimal\_number(self, number):  return number[:-1].isdigit() and number[-1] in 'Dd' or number.isdigit()  def is\_hexadecimal\_number(self, number):  prefix = number[:-1]  return all(c in '0123456789ABCDEFabcdef' for c in prefix) and number[-1] in 'Hh'  def handle\_real\_number(self):  token = self.get\_current\_token()  if not token or token[0] != 'REAL':  raise Exception(f"Ожидалось действительное число, но найдено: {token}")  number = token[1]  if self.is\_real\_format(number):  self.advance\_token()  return '%'  else:  raise Exception(f"Недопустимый формат действительного числа: {number}")  def is\_real\_format(self, number):  if '.' in number:  parts = number.split('.')  if len(parts) != 2 or not parts[0].isdigit() or not parts[1].isdigit():  return False  return True |
| --- |

*Продолжение листинга Б.1*

| elif 'E' in number or 'e' in number:  parts = number.split('E' if 'E' in number else 'e')  if len(parts) != 2 or not parts[0].isdigit() or not parts[1].lstrip('+-').isdigit():  return False  return True  return False  def handle\_program(self):  self.expect\_token('DELIMITER', '{')  while self.get\_current\_token() and self.get\_current\_token()[1] != '}':  if self.get\_current\_token()[0] in {'KEYWORD', 'ID'}:  self.handle\_statement()  self.expect\_token('DELIMITER', ';')  else:  raise Exception(f"Неожиданный токен: {self.get\_current\_token()}")  self.expect\_token('DELIMITER', '}')  def handle\_statement(self):  token = self.get\_current\_token()  if token[0] == 'KEYWORD':  if token[1] == 'if':  self.handle\_if\_statement()  elif token[1] == 'for':  self.handle\_for\_loop()  elif token[1] == 'while':  self.handle\_while\_loop()  elif token[1] == 'read':  self.handle\_input()  elif token[1] == 'write':  self.handle\_output()  else:  raise Exception(f"Неизвестная ключевая команда: {token[1]}")  elif token[0] == 'ID':  self.handle\_assignment()  else:  raise Exception(f"Неподдерживаемый оператор: {token}")  # Парсинг декларации переменных. Например: % a, b, c;  def handle\_variable\_declaration(self):  token = self.get\_current\_token()  if token[0] != 'TYPE':  raise Exception(f"Ожидался токен TYPE, но найден: {token}")  var\_type = token[1] # Сохраняем тип переменной: %, ! или $  self.advance\_token() # Пропускаем токен TYPE  start\_position = self.current\_position  # Собираем идентификаторы до конца строки (;)  while self.get\_current\_token()[0] != 'DELIMITER' or self.get\_current\_token()[1] != ';':  self.advance\_token()  end\_position = self.current\_position  identifiers = self.collect\_identifiers(start\_position, end\_position) |
| --- |

*Продолжение листинга Б.1*

| # Проверяем на наличие повторных идентификаторов  for identifier in identifiers:  if identifier in self.symbol\_table:  raise Exception(f"Повторное объявление идентификатора: {identifier}")  # Добавляем идентификатор в таблицу символов  self.symbol\_table[identifier] = {'type': var\_type}  self.advance\_token() # Пропускаем токен ';'  # Собирает идентификаторы (ID) из токенов в заданном диапазоне  def collect\_identifiers(self, start\_index, end\_index):  identifiers = []  for i in range(start\_index, end\_index):  token = self.tokens[i]  if token[0] == 'ID': # Проверяем тип токена  identifiers.append(token[1]) # Добавляем имя идентификатора  return identifiers  # Возвращает следующий токен без изменения текущей позиции.  def peek\_next\_token(self):  if self.current\_position + 1 < len(self.tokens):  return self.tokens[self.current\_position + 1]  return None # Возвращаем None, если следующий токен отсутствует  def handle\_assignment(self):  variable = self.expect\_token('ID')  if variable[1] not in self.symbol\_table:  raise Exception(f"Переменная '{variable[1]}' не была объявлена.")  self.expect\_token('KEYWORD', 'as')  expr\_type = self.handle\_expression()  var\_type = self.symbol\_table[variable[1]]['type']  if expr\_type != var\_type:  raise Exception(  f"Несоответствие типов: переменная '{variable[1]}' имеет тип {var\_type}, но ей присваивается значение типа {expr\_type}"  )  # Анализирует выражение и возвращает его тип.  def handle\_expression(self):  token = self.get\_current\_token()  if not token:  raise Exception("Ожидалось выражение, но токены закончились.")  # Логические значения true/false  if token[0] == 'KEYWORD' and token[1] in ['true', 'false']:  self.advance\_token()  return '$' # Логические значения имеют тип $  # Числовой литерал  if token[0] == 'NUMBER':  self.advance\_token()  left\_type = '%'  # Строковый литерал  elif token[0] == 'STRING':  self.advance\_token()  left\_type = 'STRING' |
| --- |

*Продолжение листинга Б.1*

| # Переменная  elif token[0] == 'ID':  var\_name = token[1]  if var\_name not in self.symbol\_table:  raise Exception(f"Переменная '{var\_name}' не была объявлена.")  left\_type = self.symbol\_table[var\_name]['type']  self.advance\_token()  # Проверяем на наличие оператора REL\_OP (сравнительного)  if self.get\_current\_token() and self.get\_current\_token()[0] == 'REL\_OP':  operator = self.get\_current\_token()  self.advance\_token()  right\_type = self.handle\_expression() # Обрабатываем правый операнд  if left\_type != '%' or right\_type != '%':  raise Exception(  f"Операторы сравнения применимы только к числовым значениям, найдено: {left\_type} и {right\_type}"  )  return '$' # Логическое выражение возвращает тип '$'  # Скобки  elif token[0] == 'DELIMITER' and token[1] == '(':  self.advance\_token()  left\_type = self.handle\_expression()  self.expect\_token('DELIMITER', ')')  else:  raise Exception(f"Неподдерживаемый токен в выражении: {token}")  # Обработка бинарных операций  while self.get\_current\_token() and self.get\_current\_token()[0] in ['ADD\_OP', 'MUL\_OP']:  op\_token = self.get\_current\_token()  self.advance\_token()  right\_type = self.handle\_expression() # Рекурсивный вызов для правого операнда  if left\_type != '%' or right\_type != '%':  raise Exception(  f"Операторы {op\_token[1]} применимы только к числовым значениям, найдено: {left\_type} и {right\_type}"  )  left\_type = '%' # Результат бинарной операции — всегда числовой тип  return left\_type  def handle\_if\_statement(self):  self.expect\_token('KEYWORD', 'if')  self.expect\_token('DELIMITER', '(')  condition\_type = self.handle\_expression()  if condition\_type != '$':  raise Exception("Условие должно быть логическим выражением.")  self.expect\_token('DELIMITER', ')')  self.expect\_token('KEYWORD', 'then')  self.handle\_statement()  if self.get\_current\_token() and self.get\_current\_token()[0] == 'KEYWORD' and self.get\_current\_token()[1] == 'else':  self.advance\_token()  self.handle\_statement() |
| --- |

*Продолжение листинга Б.1*

| def handle\_for\_loop(self):  self.expect\_token('KEYWORD', 'for')  self.handle\_assignment()  self.expect\_token('KEYWORD', 'to')  expr\_type = self.handle\_expression()  if expr\_type != '%':  raise Exception("Выражение в операторе for должно быть числовым.")  self.expect\_token('KEYWORD', 'do')  self.handle\_statement()  def handle\_while\_loop(self):  self.expect\_token('KEYWORD', 'while')  self.expect\_token('DELIMITER', '(')  condition\_type = self.handle\_expression()  if condition\_type != '$':  raise Exception("Условие должно быть логическим выражением.")  self.expect\_token('DELIMITER', ')')  self.expect\_token('KEYWORD', 'do')  self.handle\_statement()  # Обработка инструкции read(<переменные>).  def handle\_input(self):  self.expect\_token('KEYWORD', 'read') # Проверяем наличие ключевого слова read  self.expect\_token('DELIMITER', '(') # Открывающая скобка  while True:  token = self.get\_current\_token()  if not token or token[0] != 'ID': # Проверяем, что токен — идентификатор  raise Exception(f"Ожидалась переменная в read(), но найдено: {token}")  var\_name = token[1]  if var\_name not in self.symbol\_table: # Проверяем, что переменная объявлена  raise Exception(f"Переменная '{var\_name}' не была объявлена до вызова read().")  self.advance\_token() # Переход к следующему токену  # Проверяем на запятые между переменными  if self.get\_current\_token() and self.get\_current\_token()[0] == 'DELIMITER' and self.get\_current\_token()[1] == ',':  self.advance\_token() # Пропускаем запятую и продолжаем  else:  break # Если запятых больше нет, выходим из цикла  self.expect\_token('DELIMITER', ')') # Закрывающая скобка  def handle\_output(self):  self.expect\_token('KEYWORD', 'write')  self.expect\_token('DELIMITER', '(')  while self.get\_current\_token() and self.get\_current\_token()[0] in {'ID', 'NUMBER', 'STRING'}:  self.handle\_expression()  if self.get\_current\_token() and self.get\_current\_token()[0] == 'DELIMITER' and self.get\_current\_token()[1] == ',':  self.advance\_token() |
| --- |

*Окончание листинга Б.1*

| else:  break  self.expect\_token('DELIMITER', ')')  def expect\_token(self, token\_type, value=None):  token = self.get\_current\_token()  if token and token[0] == token\_type and (value is None or token[1] == value):  self.advance\_token()  return token  raise Exception(f"Ожидалось {token\_type} '{value}', но найдено {token}")  # Преобразует токены выражения в строку обратной польской нотации (ОПН).  def to\_rpn\_expression(self, expression\_tokens):  output = [] # Выходной список (постфиксная запись)  stack = [] # Стек для операторов и скобок  # Приоритеты операторов  precedence = {  '+': 1, '-': 1,  '\*': 2, '/': 2  }  # Проход по каждому токену выражения  for token in expression\_tokens:  token\_type, token\_value = token  # Если токен - операнд (число или переменная)  if token\_type in {'NUMBER', 'ID'}:  output.append(token\_value)  # Если токен - открывающая скобка  elif token\_type == 'DELIMITER' and token\_value == '(':  stack.append(token\_value)  # Если токен - закрывающая скобка  elif token\_type == 'DELIMITER' and token\_value == ')':  while stack and stack[-1] != '(':  output.append(stack.pop())  if not stack or stack[-1] != '(':  raise Exception("Несогласованные скобки")  stack.pop() # Удаляем '(' из стека  # Если токен - оператор  elif token\_type in {'ADD\_OP', 'MUL\_OP'}:  while (stack and stack[-1] in precedence and  precedence[token\_value] <= precedence[stack[-1]]):  output.append(stack.pop())  stack.append(token\_value)  else:  raise Exception(f"Неподдерживаемый токен в выражении: {token}")  # Перемещаем все оставшиеся операторы из стека в выходной список  while stack:  if stack[-1] in {'(', ')'}:  raise Exception("Несогласованные скобки в выражении")  output.append(stack.pop())  return ''.join(output) |
| --- |