**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 4](#_gjdgxs)

[1 ПОСТАНОВКА ЗАДАЧИ 5](#_30j0zll)

[2 ПОРЯДОК ВЫПОЛНЕНИЯ 6](#_1fob9te)

[3 ГРАММАТИКА МОДЕЛЬНОГО ЯЗЫКА 7](#_3znysh7)

[4 ЛЕКСИЧЕСКИЙ АНАЛИЗАТОР 8](#_2et92p0)

[5 СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР 10](#_tyjcwt)

[6 СЕМАНТИЧЕСКИЙ АНАЛИЗ 11](#_3dy6vkm)

[7 ТЕСТИРОВАНИЕ ПРОГРАММЫ 12](#_1t3h5sf)

[ЗАКЛЮЧЕНИЕ 17](#_4d34og8)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 18](#_2s8eyo1)

[ПРИЛОЖЕНИЯ 19](#_17dp8vu)

# ВВЕДЕНИЕ

Хотя вычислительная техника существует уже более полувека, теория формальных языков начала свою историю в 1957 году. В этом году американский ученый Джон Бэкус создал первый компилятор для языка Фортран, применив теорию формальных языков, основывающуюся на работах Н. Хомского, который разработал классификацию формальных языков. Хомский в основном изучал естественные языки, а Бэкус адаптировал его теорию для создания языков программирования. Это стало важным шагом в развитии множества языков программирования.

Несмотря на наличие множества алгоритмов, упрощающих создание трансляторов для формальных языков, процесс разработки нового языка требует творческого подхода. Особенно это касается синтаксиса языка, который должен быть как удобным для прикладного программирования, так и соответствовать области контекстно-свободных языков, для которых существуют проверенные методы анализа.

Знания основ теории формальных языков и практических методов разработки распознавателей формальных языков являются неотъемлемой частью образования современного инженера-программиста.

Цели данной курсовой работы:

* освоение ключевых методов разработки распознавателей формальных языков на примере модельного языка программирования;
* получение практических навыков разработки трансляторов для языков программирования;
* закрепление умений решать инженерные задачи, работать с технической литературой и документацией.

# 1 ПОСТАНОВКА ЗАДАЧИ

Разработка распознавателя модельного языка программирования включает выполнение следующих шагов:

Создать распознаватель модельного языка программирования в соответствии с заданной формальной грамматикой.

Распознаватель представляет собой алгоритм, который определяет, принадлежит ли цепочка символов определённому языку.

Трансляция исходного текста программы выполняется поэтапно. Основные этапы:

1. Лексический анализ.
2. Синтаксический анализ.
3. Семантический анализ.
4. Генерация целевого кода.

Лексический анализ выполняемая с использованием регулярной грамматики, что эквивалентно разработке конечного автомата и его диаграммы состояний.

Синтаксический анализ опирается на контекстно-свободные грамматики и предназначен для сопоставления структуры программы с формальным описанием языка.

Семантический анализ проверяет аспекты, не описываемые КС-грамматикой, такие как обработка описаний, анализ выражений и проверка корректности операторов. Он гарантирует, что:

* каждая переменная описана единожды;
* переменные в выражениях корректно определены, а типы операндов соответствуют операциям.

# 2 ПОРЯДОК ВЫПОЛНЕНИЯ

Разработка распознавателя модельного языка программирования включает следующие этапы:

1. В соответствии с номером варианта составить описание модельного языка программирования в виде правил вывода формальной грамматики.
2. Составить таблицу лексем и нарисовать диаграмму состояний для распознавания и формирования лексем языка.
3. Разработать процедуру лексического анализа исходного текста программы на языке высокого уровня.
4. Разработать процедуру синтаксического анализа исходного текста методом рекурсивного спуска на языке высокого уровня.
5. Построить программный продукт, который читает текст программы, написанной на модельном языке, в виде консольного приложения.
6. Протестировать работу программного продукта с помощью серии тестов, демонстрирующих все основные особенности модельного языка программирования, включая возможные лексические и синтаксические ошибки.

# 3 ГРАММАТИКА МОДЕЛЬНОГО ЯЗЫКА

Согласно индивидуальному варианту задания на курсовую работу грамматика языка включает следующие синтаксические конструкции, которые представлены в Листинге 1

*Листинг 1 – Грамматика модельного языка*

| <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> |
| --- |

# 4 ЛЕКСИЧЕСКИЙ АНАЛИЗАТОР

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.

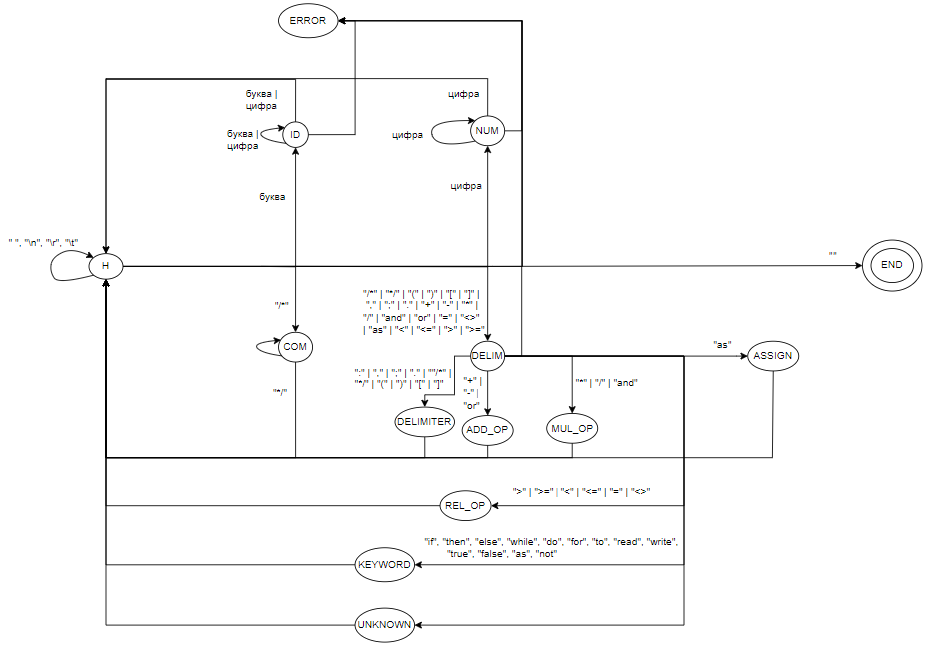
При разработке лексического анализатора ключевые слова и ограничители определяются заранее, а идентификаторы и числовые константы вычисляются в процессе анализа исходного текста.

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 4.1 – Диаграмма состояний лексического анализатора**

# 5 СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР

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 Б.

# 6 СЕМАНТИЧЕСКИЙ АНАЛИЗ

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 В.

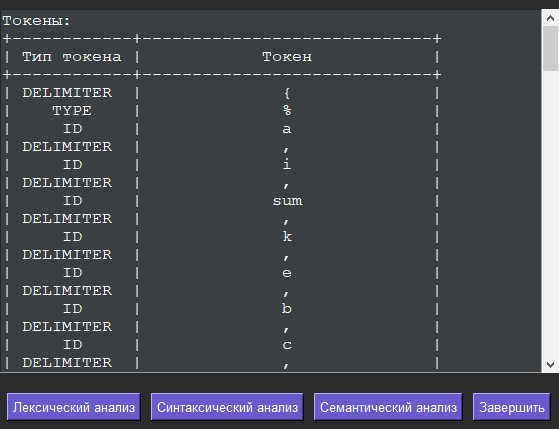
# 7 ТЕСТИРОВАНИЕ ПРОГРАММЫ

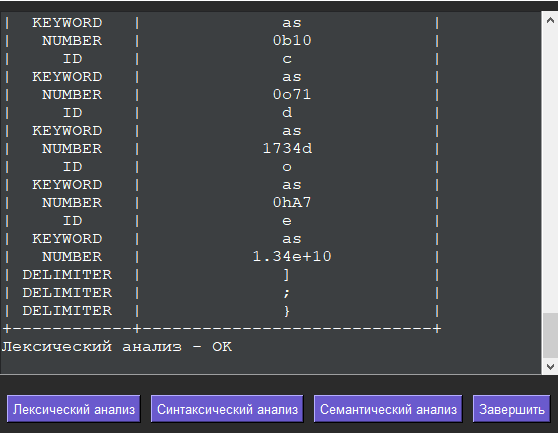
Было разработано графическое приложение, которое предназначено для лексического, синтаксического и семантического анализа исходного текста программы. В результате приложение сообщает о правильности программы, либо сообщает об ошибке, где написана причина, тип токена и токен. Рассмотрим примеры работы приложения.

1. Исходный код программы приведен в листинге 2.

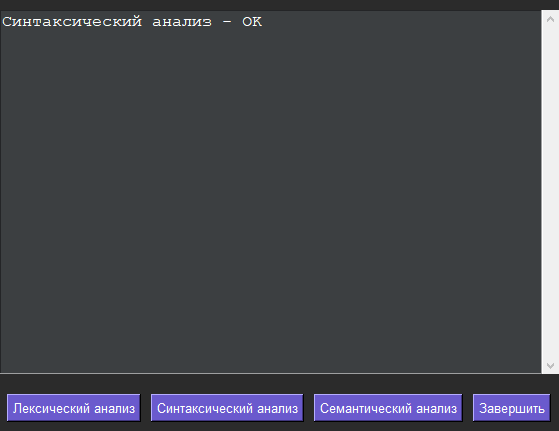
*Листинг 2 – Тестовая программа 1*

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

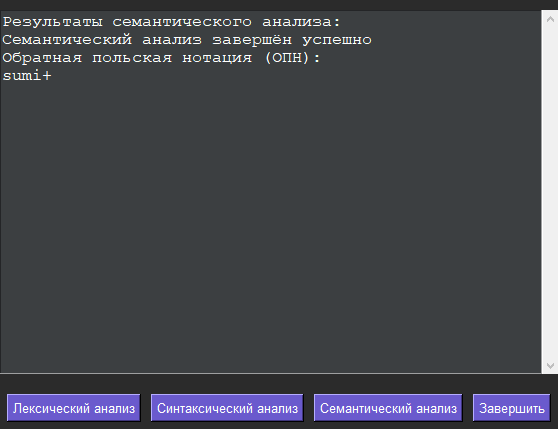
**  
Рисунок 7.1 – Тестирование лексического анализа**

**  
Рисунок 7.2 – Тестирование лексического анализа**

Данная программа синтаксически корректна, поэтому анализатор выдает следующее сообщение.

**  
Рисунок 7.3 – Тестирование синтаксического анализа**

Данная программа семантически корректна, поэтому анализатор выдает следующее сообщение.

**  
Рисунок 7.4 – Тестирование семантического анализа**

2. Исходный код программы, содержащий синтаксическую и семантическую ошибку, приведен в листинге 3.

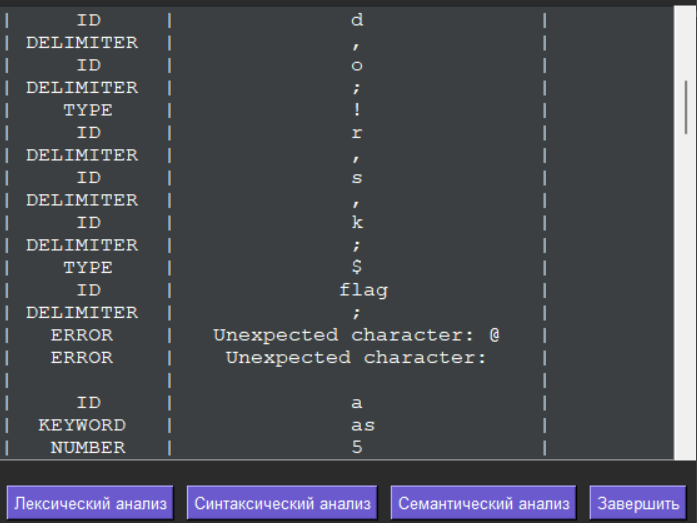
*Листинг 3 – Тестовая программа 2*

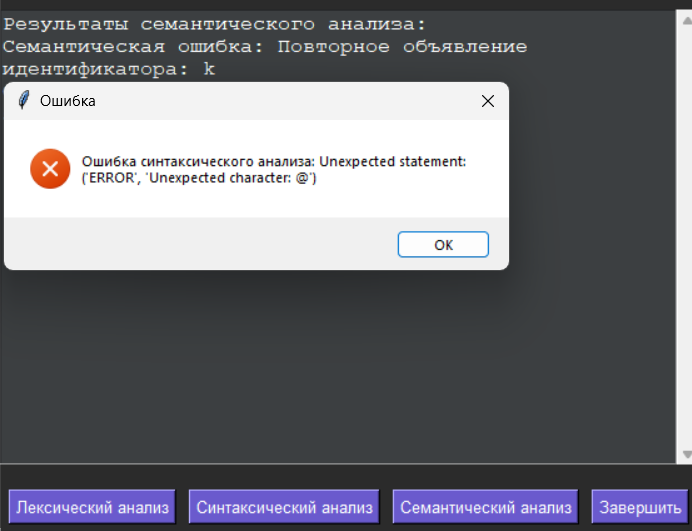
| {  % a, i, sum, k, e, b, c, d, o;  ! r, s, k; /\* Дважды объявили k - семантическая ошибка \*/  $ flag;  @  a as 5 /\* Это комментарий \*/  if (a > 5) then  write('a больше 5')  else  write('a меньше или равно 5')  sum as 0  for i as 0 to a do  write('i = ', i):  sum as sum + i:  if (sum >= 10) then  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  ];  } |
| --- |

Лексическая ошибка допущена в строке 3: @ не распознается. Вывод об ошибке продемонстрирован на рисунке 7.5.

Синтаксическая ошибка допущена в строке 5: неожиданная переменная (@). Вывод об ошибке продемонстрирован на рисунке 7.6.

Семантическая ошибка допущена в строке 3: переменная k объявлена дважды. Вывод об ошибке продемонстрирован на рисунке 7.6.

**  
Рисунок 7.5 – Тестирование ошибки лексического анализа**

**  
Рисунок 7.6 – Тестирование ошибки синтаксического и семантического анализа**

# ЗАКЛЮЧЕНИЕ

Разработан распознаватель модельного языка программирования.

Грамматика языка задана с помощью правил вывода и описана в форме Бэкуса-Наура (БНФ). Согласно этой грамматике, в языке присутствуют лексемы следующих базовых типов: числовые константы, переменные, разделители и ключевые слова.

Разработан лексический анализатор, позволяющий разделить последовательность символов исходного текста программы на последовательность лексем.

Разработан синтаксический анализатор, который занимается разбором исходного текста программы. Анализатор распознает входной язык по методу рекурсивного спуска.

Разработан семантический анализатор, в который включены проверки дополнительных условий, в частности, проверка на повторное объявление одной и той же переменной.

Тестирование программного продукта показало, что синтаксически и семантически корректно написанная программа успешно распознается анализатором, а программа, содержащая ошибки, отвергается.

В ходе работы изучены основные принципы построения интеллектуальных систем на основе теории автоматов и формальных грамматик, приобретены навыки лексического, синтаксического и семантического анализа предложений языков программирования.

# 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 – Класс синтаксического анализатора

Приложение В – Класс семантического анализатора

**Приложение А**

Класс лексического анализатора

*Листинг А.1 – lex.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() |
| --- |

*Листинг А.1 - Продолжение*

| # 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() |
| --- |

*Листинг А.1 - Продолжение*

| 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  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]) |
| --- |

*Листинг А.1 - Окончание*

| 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**

Класс синтаксического анализатора

*Листинг Б.1 – syntax.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' |
| --- |

*Листинг Б.1 – Продолжение*

| 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', ')') |
| --- |

*Листинг Б.1 – Продолжение*

| 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}") |
| --- |

*Листинг Б.1 – Окончание*

| 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 В**

Класс семантического анализатора

*Листинг В.1 – semantic.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" |
| --- |

*Листинг В.1 – Продолжение*

| except Exception as e:  return f"Семантическая ошибка: {e}"  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}") |
| --- |

*Листинг В.1 – Продолжение*

| 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  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() |
| --- |

*Листинг В.1 – Продолжение*

| end\_position = self.current\_position  identifiers = self.collect\_identifiers(start\_position, end\_position)  # 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 = '%' |
| --- |

*Листинг В.1 – Продолжение*

| # Строковый литерал  elif token[0] == 'STRING':  self.advance\_token()  left\_type = 'STRING'  # 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 |
| --- |

*Листинг В.1 – Продолжение*

| def handle\_if\_statement(self):  self.expect\_token('KEYWORD', 'if')  self.expect\_token('DELIMITER', '(')  condition\_type = self.handle\_expression()  if condition\_type != '$':  raise Exception("Условие должно быть логическим выражением.")  self.expect\_token('DELIMITER', ')')  self.expect\_token('KEYWORD', 'then')  self.handle\_statement()  if self.get\_current\_token() and self.get\_current\_token()[0] == 'KEYWORD' and self.get\_current\_token()[1] == 'else':  self.advance\_token()  self.handle\_statement()  def handle\_for\_loop(self):  self.expect\_token('KEYWORD', 'for')  self.handle\_assignment()  self.expect\_token('KEYWORD', 'to')  expr\_type = self.handle\_expression()  if expr\_type != '%':  raise Exception("Выражение в операторе for должно быть числовым.")  self.expect\_token('KEYWORD', 'do')  self.handle\_statement()  def handle\_while\_loop(self):  self.expect\_token('KEYWORD', 'while')  self.expect\_token('DELIMITER', '(')  condition\_type = self.handle\_expression()  if condition\_type != '$':  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 |
| --- |

*Листинг В.1 – Продолжение*

| 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()  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) # Добавляем операнд в выходной список  # 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 |
| --- |

*Листинг В.1 – Окончание*

| # 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)  # Неизвестный токен  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())  # Собираем и возвращаем строку ОПН  return ''.join(output) |
| --- |