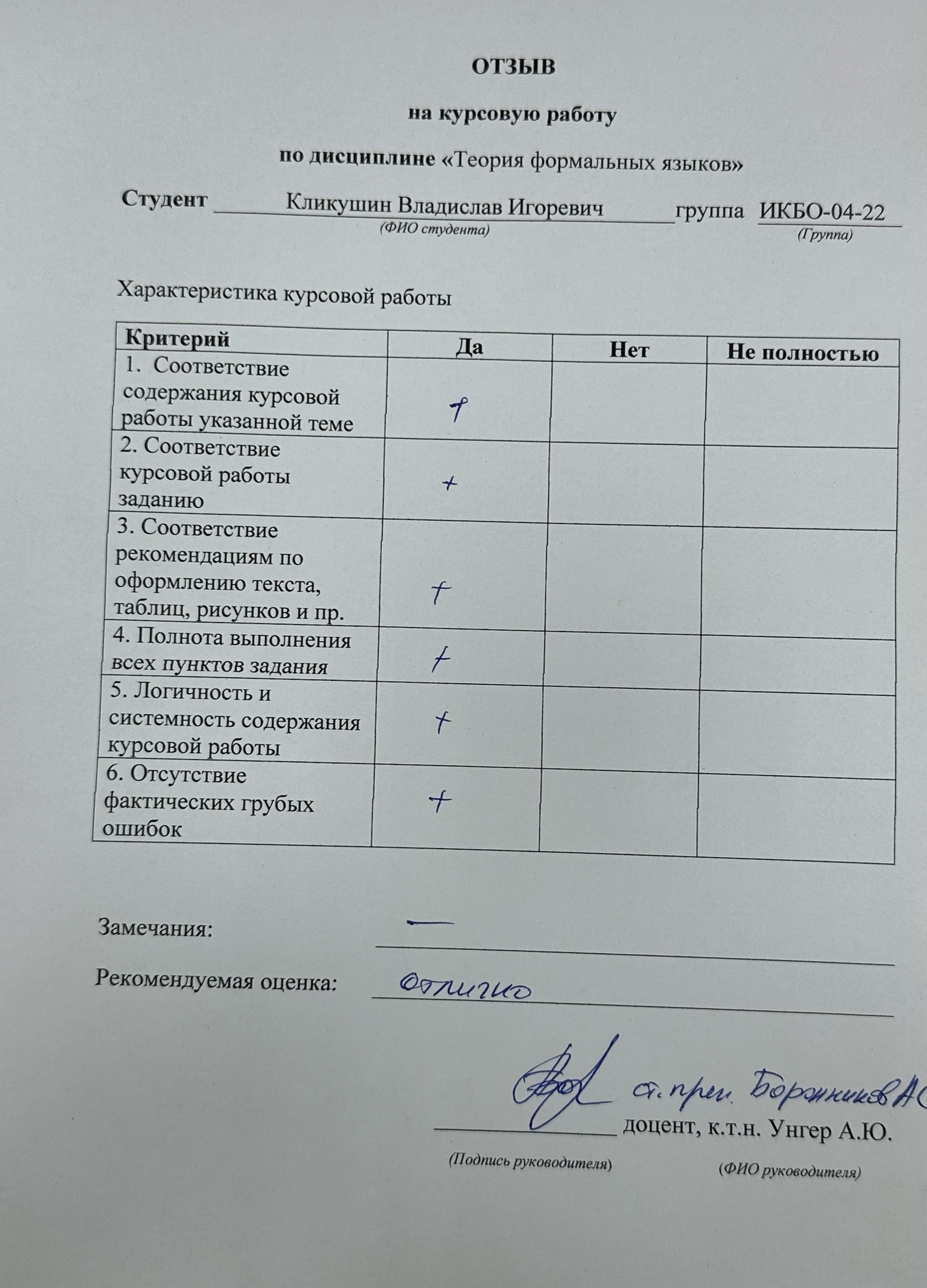
Изображение выглядит как текст, документ, меню, письмо

Автоматически созданное описание

Изображение выглядит как текст, письмо, чернила, бумага

Автоматически созданное описание



# СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 5](#_Toc153450567)

[1 ПОСТАНОВКА ЗАДАЧИ 7](#_Toc153450568)

[2 ПОРЯДОК ВЫПОЛНЕНИЯ 8](#_Toc153450569)

[3 ГРАММАТКА МОДЕЛЬНОГО ЯЗЫКА 9](#_Toc153450570)

[4 РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА 11](#_Toc153450571)

[5 РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА 13](#_Toc153450572)

[6 СЕМАНТИЧЕСКИЙ АНАЛИЗ 15](#_Toc153450573)

[7 ТЕСТИРОВАНИЕ ПРОГРАММЫ 16](#_Toc153450574)

[ЗАКЛЮЧЕНИЕ 24](#_Toc153450575)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 25](#_Toc153450576)

[ПРИЛОЖЕНИЯ 26](#_Toc153450577)

[Приложение А 27](#_Toc153450578)

[Приложение Б 37](#_Toc153450579)

[Приложение В 42](#_Toc153450580)

# ВВЕДЕНИЕ

Развитие информационных технологий в последние десятилетия привело к увеличению объема программного кода, написанного на различных языках программирования. Для обеспечения правильной работы программ, необходимо не только их синтаксический анализ, но и оценка соответствия кода стандартам программирования и наличие правильных структурных элементов.

Первые языки программирования появились в середине XX века и имели свои особенности и ограничения. Вплоть до появления высокоуровневых языков программирования программирование было очень трудоёмким процессом, требующим внимательности и точности. С развитием компьютерных технологий важность языков программирования только возрастала, и разработчики начали активно стремиться к созданию более эффективных и удобных языков для решения сложных задач.

Однако, при разработке новых языков программирования возникает задача интерпретации и компиляции данных языков. Компиляторы — это программные средства, которые преобразуют программы на определенном языке программирования в машинный код, понятный компьютеру. Правильная работа компилятора обеспечивает эффективность и корректность выполнения программы.

Распознаватель модельного языка программирования играет важную роль в процессе разработки языков программирования и компиляторов. Он позволяет анализировать структуру и синтаксические правила языка программирования, что необходимо для создания компиляторов и других инструментов разработки. Без распознавателей модельного языка программирования процесс создания новых языков программирования и компиляторов стал бы гораздо более сложным и трудоемким.

Цель курсовой работы:

* закрепление теоретических знаний в области теории формальных языков, грамматик и автоматов;
* формирование практических умений и навыков разработки собственного распознавателя модельного языка программирования;
* закрепление практических навыков самостоятельного решения инженерных задач, развитие творческих способностей студентов и умений пользоваться технической, нормативной и справочной литературой.

# 1 ПОСТАНОВКА ЗАДАЧИ

Разработать распознаватель модельного языка программирования согласно заданной грамматике варианта.

Распознаватель – это специальный алгоритм, который позволяет определить принадлежность цепочки символов некоторому языку. Распознаватель состоит из входной ленты, читающей головки, устройства управления и дополнительной памяти (стек). Конфигурацией распознавателя есть совокупность трех элементов: состояния управляющего устройства, содержимого входной ленты и положения входной головки, содержимого вспомогательной памяти.

В рамках курсовой работы были выполнены первые три этапа трансляции — лексический, синтаксический и семантический анализы.

Лексический анализ является необязательным этапом трансляции, однако крайне желательным по следующим причинам: замена идентификаторов, констант, ограничителей и служебных слов лексемами делает программу более удобной для дальнейшей обработки, лексический анализатор уменьшает длину программы, устраняя из ее исходного представления несущественные пробелы и комментарии. Лексический анализатор выполняется с использованием регулярных грамматик, поэтому принцип его работы эквивалентен разработке конечного автомата и его диаграммы состояний.

Для синтаксического разбора используются контекстно-свободные грамматики. Его задача - провести разбор текста программы, сопоставив его с эталоном, данным в описании языка.

В ходе семантического анализа проверяются отдельные правила записи исходных программ, которые не описываются КС-грамматикой.

# 2 ПОРЯДОК ВЫПОЛНЕНИЯ

1. В соответствии с номером варианта составить формальное описание модельного языка программирования с помощью:
   1. РБНФ;
   2. диаграмм Вирта;
   3. формальных грамматик;
2. Составить таблицу лексем и диаграмму состояний для распознавания и формирования лексем языка.
3. Разработать процедуру лексического анализа исходного текста программы на выбранном языке высокого уровня.
4. Разработать процедуру синтаксического анализа исходного текста методом рекурсивного спуска на выбранном языке высокого уровня.
5. Построить программный продукт, анализирующий текст программы, написанной на модельном языке, в виде консольного приложения.
6. Протестировать работу программного продукта в полном объёме при помощи серии тестов, демонстрирующих все основные особенности модельного языка программирования, включая возможные лексические и синтаксические ошибки, которые могут быть допущены.

# 3 ГРАММАТКА МОДЕЛЬНОГО ЯЗЫКА

Для описания грамматики модельного языка воспользуемся расширенными формами Бэкуса-Наура (РБНФ), где символ «::=» отделяет левую часть правила от правой. Согласно индивидуальному варианту №2 задания на курсовую работы язык состоит из следующих синтаксических конструкций. Для удобства ключевые слова выделены полужирным шрифтом:

* <операции\_группы\_отношения>:: = < > | = | < | <= | > | >=
* <операции\_группы\_сложения>:: = + | - | **or**
* <операции\_группы\_умножения>::= \* | / | **and**
* <унарная\_операция>::= **not**
* <выражение>::=<операнд>{<операции\_группы\_отношения> <операнд>}
* <операнд>::=<слагаемое>{<операции\_группы\_сложения> <слагаемое>}
* <слагаемое>::=<множитель>{<операции\_группы\_умножения> <множитель>}
* <множитель>::=<идентификатор> | <число> | <логическая\_константа> | <унарная\_операция> <множитель> | «(»<выражение>«)»
* <число>::= <целое> | <действительное>
* <логическая\_константа>::= **true** | **false**
* <идентификатор>::= <буква> {<буква> | <цифра>}
* <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
* <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>
* <программа>::= «{» {/ (<описание> | <оператор>) ; /} «}»
* <описание>::= **dim** <идентификатор> {, <идентификатор> } <тип>
* <тип>::= **integer** | **real** | **boolean**
* <оператор>::= <составной> | <присваивания> | <условный> | <фиксированного\_цикла> | <условного\_цикла> | <ввода> | <вывода>
* <составной>::= «[» <оператор> { ( : | перевод строки) <оператор> } «]»
* <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>
* <ввода>::= **read** «(»<идентификатор> {, <идентификатор> } «)»
* <вывода>::= **write** «(»<выражение> {, <выражение> } «)»
* <начало\_комментария>::= «/\*»
* <конец\_комментария>::= «\*/»

# 4 ЛЕКСИЧЕСКий АНАЛИЗАТОР

Лексический анализатор – это подпрограмма, принимающая на вход исходный текст программы и выдающая последовательность лексем — минимальных элементов программы, несущих смысловую нагрузку.

Можно выделить следующие типы лексем:

* служебные (ключевые) слова;
* ограничители (разделители);
* числа;
* идентификаторы.

При разработке лексического анализатора, ключевые слова и ограничителя известны заранее, идентификаторы и числовые константы – вычисляются в момент разбора исходного текста.

Для лексического анализа написана функция Lexical, которая читает поток символов из файла и разбивает его на лексемы. Особенности выбранного языка Python позволяют создавать вложенные функции – этот подход используется для написания вспомогательных функций для работы лексического анализатора, например функция считывания символа.

Для хранения лексем введён массив, который содержит словари и является результатом работы лексического анализатора. Каждый словарь имеет четыре элемента:

1. Ключ «Тип лексемы» и значение по ключу - идентификатор значения типа лексемы
2. Ключ «Значение» и значение по ключу – значение лексемы
3. Ключ «Номер строки» и значение по ключу – порядковый номер строки, где находится текущая лексема
4. Ключ «Позиция в строке» и значение по ключу – порядковый номер текущей лексемы в строке.

В массив лексем не сохраняется информация о символах перехода на новую строку, символах табуляции и пробела, а также весь текст, заключенный между символами «/\*» и «\*/» — это комментарии. Исключением являются символы перехода на новую строку, которые идут после открывающейся квадратной скобки – их необходимо сохранять в массив, так как они являются разделителями в составных операторах.

Для разработки лексического анализатора была построена диаграмма состояний, представленная на Рисунке 4.1.

Исходный код лексического анализатора представлен в Приложении А.

Изображение выглядит как диаграмма, карта, шаблон

Автоматически созданное описание

Рисунок 4.1 — Диаграмма состояний лексического анализатора

# 5 СИНТАКСИЧЕСКий АНАЛИЗАТОР

Синтаксический анализатор — это подпрограмма, отвечающая за проверку текста программы на соответствие установленным синтаксическим правилам. На вход ему подаётся таблица лексем, сформированная на этапе лексического анализа, а результатом работы является сообщение об успешном завершении анализа или сообщение об ошибке с её описанием.

Синтаксический анализатор по аналогии с лексическим представлен в виде функции, которая работает с массивом лексем, полученным при лексическом анализе. Функция содержит внутренние функции, отвечающие за считывание лексемы и сам синтаксический анализ.

Разработку синтаксического анализатора проведем с помощью метода рекурсивного спуска (РС). В основе метода лежит тот факт, что каждому нетерминалу ставится в соответствие рекурсивная функция. Для того, чтобы в явном виде представить множество рекурсивных функций, перепишем грамматические правила следующим образом:

* program → «{» body«}»
* body → {/ (description | operator) «;» /}
* description → id\_sequence (integer|real|boolean)
* id\_sequence → id {,id}
* operator → «[» operator {(«:» | перевод строки) operator} «]» | ID **as** expression | **if** expression **then** operator [**else** operator] | **while** expression **do** operator | **for** ID **as** expression **to** expression **do** operator | **read** «(» id\_sequence «)» | **write** «(» expression\_sequence «)»
* expression\_sequence → expression {,expression}
* expression → operand [ (> | < | >= | <= | <> | =) operand ]
* operand → summand [ (+ | - | **or**) summand ]
* summand → multiplier [ (\* | / | **and**) multiplier ]
* multiplier → ID | NUMBER | **true** | **false** | **not** multiplier | «(» expression «)»

Код синтаксического анализатора представлен в Приложении Б.

# 6 СЕМАНТИЧЕСКИЙ АНАЛИЗ

Некоторые особенности языка нельзя описать контекстно-свободной грамматикой. К таким особенностям относятся:

* любой идентификатор, используемый в теле программы должен быть описан;
* повторное описание одного и того же идентификатора не разрешается;
* в операторе присваивания типы идентификаторов должны совпадать;
* в условном операторе и операторе цикла в качестве условия допустимы только логические выражения;
* операнды операций отношения должны быть целочисленными.

Эти особенности проверяются на этапе семантического анализа, который удобно совестить с синтаксическим анализом.

Подход к каждой семантической проверке может кардинально отличаться. Например, для проверки объявленной переменной необходимо на этапе лексического анализа сохранять для каждого идентификатора информацию об объявлении. Информация же будет обновляться в синтаксическом анализаторе при каждом новом разборе описания. Важно помнить, что повторное объявление переменной недопустимо.

# 7 ТЕСТИРОВАНИЕ ПРОГРАММЫ

Программным продуктом выступает программа *main.py*, которая запускается в консоли командой *python main.py* на Windows или *python3 main.py* на Linux или MacOS. После запуска программа предлагает пользователю ввести путь до анализируемой программы. Исходный код программы *main.py* представлен в Приложении В.

Программа выводит сообщение об успешном запуске в случае отсутствия каких-либо ошибок. Если на этапе лексического анализа появилась неизвестная лексема, синтаксический анализ не будет произведен и выведется сообщение об ошибке. Если лексический анализ прошел успешно, в дело вступает синтаксический, который сообщает об ошибках или их отсутствии. Для начала проверим работу распознавателя на полностью корректной программе, написанной на модельном языке.

1. Исходный код синтаксически верной программы (не несущей какой-либо смысл) представлен на Рисунке 7.1 совместно с сообщением об успешной проверке.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 7.1 — Пример синтаксически верной программы

1. Исходный текст синтаксически верной программы представлен на Рисунке 7.2 совместно с сообщением о неизвестной лексеме с указанием номера строки и позиции в строке.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 7.2 — Пример программы с лексической ошибкой

Здесь ошибка заключается в лексеме «1C». Идентификатор не может начинаться с цифры. Но и число не содержит «С» в конце.

1. Исходный текст программы, содержащей лексическую ошибку, представлен на Рисунке 7.3 совместно с сообщением об неизвестной лексеме.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 7.3 — Пример программы с лексической ошибкой

Данное действительное число не является корректным, потому что содержит дополнительный символ «e» в конце.

1. Исходный текст программы, содержащей синтаксическую ошибку, представлен на Рисунке 7.4 совместно с сообщением об ошибке.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 7.4 — Пример программы с синтаксической ошибкой

После ключевого слова dim, использующегося для объявления переменной был пропущен идентификатор.

1. Исходный текст программы, содержащей синтаксическую ошибку, представлен на Рисунке 7.5 совместно с сообщением об ошибке.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 7.5 — Пример программы с синтаксической ошибкой

В операторе while была поставлена лишняя запятая, требующая еще один идентификатор для вывода.

1. Исходный текст программы, содержащей синтаксическую ошибку, представлен на Рисунке 7.6 совместно с сообщением об ошибке.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 7.6 — Пример программы с синтаксической ошибкой

В составном операторе отсутствует разделитель между операторами. Ожидался символ перехода на новую строку или двоеточие.

1. Исходный текст программы, содержащей синтаксическую ошибку, представлен на Рисунке 7.7 совместно с сообщением об ошибке.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 7.7 — Пример программы с синтаксической ошибкой

Пропущено ключевое слово do в операторе for.

# ЗАКЛЮЧЕНИЕ

В работе представлены результаты разработки анализатора языка программирования. Грамматика языка задана с помощью правил вывода и описана в форме Бэкуса-Наура (БНФ). Согласно 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. Лексический анализатор реализован на языке высокого уровня Python в виде функции Lexical.

Разбором исходного текста программы занимается синтаксический анализатор, принимающий на вход массив лексем, сгенерированный при лексической проверке, который реализован в виде функции Syntactic на языке Python. Анализатор распознает входной язык по методу рекурсивного спуска. Для применимости необходимо было преобразовать грамматику, в частности, специальным образом обрабатывать встречающиеся итеративные синтаксически конструкции.

Тестирование программного продукта показало, что синтаксически корректно написанная программа успешно распознается анализатором, а программа, содержащая ошибки, отвергается. Программа выдаёт сообщение об успешной сборке при отсутствии ошибок и указывает на ошибку с информацией о номере строки и позиции в строке.

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.

Подводя итог, курс теории формальных языков был усвоен в полном объёме, я получил ценный опыт и знания, которые будут полезны в моей дальнейшей профессиональной деятельности.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

Свердлов С. З. Языки программирования и методы трансляции: учебное пособие. – Санкт-Петербург: Лань, 2019.

Малявко А. А. Формальные языки и компиляторы: учебное пособие для вузов. – М.: Юрайт, 2020.

Миронов С. В. Формальные языки и грамматики: учебное пособие для студентов факультета компьютерных наук и информационных технологий. – Саратов: СГУ, 2019.

Унгер А.Ю. Основы теории трансляции: учебник. – М.: МИРЭА – Российский технологический университет, 2022.

Антик М. И., Казанцева Л. В. Теория формальных языков в проектировании трансляторов: учебное пособие. – М.: МИРЭА, 2020.

Ахо А. В., Лам М. С., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструментарий. – М.: Вильямс, 2008.

Ишакова Е.Н. Теория языков программирования и методов трансляции: учебное пособие. – Оренбург: ИПК ГОУ ОГУ, 2007.

# ПРИЛОЖЕНИЯ

Приложение А — Исходный код лексического анализатора

Приложение Б — Исходный код синтаксического анализатора

Приложение В — Исходный код запуска анализа программы

### Приложение А

Исходный код лексического анализатора

Листинг А — Код функции Lexical

def Lexical():

def gc():

nonlocal total, line\_number, on\_line\_number, letter

if total < len(data)-1:

if data[total] == '\n':

line\_number += 1

on\_line\_number = 1

else:

on\_line\_number += 1

total += 1

letter = data[total]

return True

else:

return False

flag, FLAG\_READ\_FILE = False, True

Lex\_result = list()

buffer = str()

status = 'H'

total, line\_number, on\_line\_number = 0, 1, 1

key\_words = ("not", "or", "and", "dim", "integer", "real", "boolean", "while",

"as", "if", "else", "then", "for", "to", "do", "read", "write", "true", "false")

separators = ('{', '}', '<>', '=', '<', '>', '<=', '[', ']', ':', '/\*', '\*/'

'>=', '+', '-', '\*', '/', ',', ';', '(', ')')

define\_dict = {"not": "NOT", "or": "OR", "and": "AND", "dim": "DIM", "integer": "INTEGER", "real": "REAL", "boolean": "BOOLEAN", "while": "WHILE",

"as": "ASSIGN", "if": "IF", "else": "ELSE", "then": "THEN", "for": "FOR", "to": "TO", "do": "DO", "read": "READ", "write": "WRITE",

"true": "TRUE", "false": "FALSE", "{": "BEGIN\_PROG", "}": "END\_PROG", "<>": "NEQ", "=": "EQUAL", "<": "LESS", ">": "GREATER",

"<=": "LEQ", ">=": "GEQ", "[": "BEGIN\_COMPLEX", "]": "END\_COMPLEX", ":": "COLON", "+": "PLUS", "-": "MINUS", "\*": "MULT",

"/": "DIVIDE", ",": "COMMA", ";": "SEMICOLON", "(": "LEFT\_PAREN", ")": "RIGHT\_PAREN"}

# ID, KEYWORD, TYPE, LINE\_BREAK, BEGIN\_COMMENT,END\_COMMENT, ERR,TYPE\_I, TYPE\_F, TYPE\_B, EMPTY

data = file.read()

while True:

letter = data[total]

match status:

case 'H':

while letter in (' ', '\n', '\t') and FLAG\_READ\_FILE:

if letter == '\n' and flag:

Lex\_result.append({"Тип лексемы": "LINE\_BREAK", "Значение": "LINE\_BREAK",

"Номер строки": line\_number, "Позиция в строке": on\_line\_number})

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

if letter.isalpha():

buffer = letter

FLAG\_READ\_FILE = gc()

Продолжение листинга А

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'ID'

elif letter in ('0', '1'):

buffer = letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = "BINARY NUMBER"

elif letter in map(str, list(range(2, 8))):

buffer = letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = "OCTAL NUMBER"

elif letter in ('8', '9'):

status = 'DECIMAL NUMBER'

buffer = letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

elif letter == '.':

buffer = letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'P1'

elif letter == '/':

buffer = letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'C1'

elif letter == '<':

buffer = letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'M1'

elif letter == '>':

buffer = letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'M2'

elif letter == '}':

buffer = letter

Lex\_result.append({"Тип лексемы": define\_dict[buffer], "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number})

FLAG\_READ\_FILE = gc()

else:

status = 'SEPARATOR'

case 'ID':

while letter.isalnum():

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

if buffer in key\_words:

Продолжение листинга А

Lex\_result.append({"Тип лексемы": define\_dict[buffer], "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

else:

Lex\_result.append({"Тип лексемы": "ID", "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

case 'BINARY NUMBER':

while letter in ('0', '1'):

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

if letter in map(str, list(range(2, 8))):

status = 'OCTAL NUMBER'

elif letter in ('8', '9'):

status = 'DECIMAL NUMBER'

elif letter in ('a', 'c', 'f', 'A', 'C', 'F'):

status = 'HEX NUMBER'

elif letter in ('e', 'E'):

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'E11'

elif letter in ('d', 'D'):

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'D'

elif letter in ('o', 'O'):

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'O'

elif letter in ('H', 'h'):

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'HX'

elif letter == '.':

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'P1'

elif letter in ('b', 'B'):

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'B'

elif letter.isalpha():

status = 'ERROR'

else:

Продолжение листинга А

status = 'DECIMAL NUMBER'

case 'OCTAL NUMBER':

while letter in map(str, list(range(0, 8))):

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

if letter in ('8', '9'):

status = 'DECIMAL NUMBER'

elif letter in ('a', 'c', 'b', 'f', 'A', 'B', 'C', 'F'):

status = 'HEX NUMBER'

elif letter in ('e', 'E'):

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'E11'

elif letter in ('d', 'D'):

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'D'

elif letter in ('H', 'h'):

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'HX'

elif letter == '.':

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'P1'

elif letter in ('o', 'O'):

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'O'

elif letter.isalpha():

status = 'ERROR'

else:

status = 'DECIMAL NUMBER'

case 'DECIMAL NUMBER':

while letter in map(str, list(range(0, 10))):

buffer += letter

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

if letter in ('a', 'b', 'c', 'f', 'A', 'B', 'C', 'F'):

status = 'HEX NUMBER'

elif letter in ('e', 'E'):

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'E11'

elif letter in ('h', 'H'):

buffer += letter

Продолжение листинга А

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'HX'

elif letter == '.':

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'P1'

elif letter in ('d', 'D'):

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'D'

elif letter.isalpha():

status = 'ERROR'

else:

Lex\_result.append({"Тип лексемы": "десятичное число", "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

case 'HEX NUMBER':

while letter.isdigit() or letter in ('a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E', 'F'):

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

if letter in ('h', 'H'):

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'HX'

else:

status = 'ERROR'

case 'B':

if letter.isdigit() or letter in ('a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E', 'F'):

status = 'HEX NUMBER'

elif letter in ('h', 'H'):

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'HX'

elif letter.isalpha():

status = 'ERROR'

else:

Lex\_result.append({"Тип лексемы": "число в двоичной системе", "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

case 'O':

if letter.isalnum():

status = 'ERROR'

else:

Продолжение листинга А

Lex\_result.append({"Тип лексемы": "число в восьмеричной системе", "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

case 'D':

if letter in ('h', 'H'):

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'HX'

elif letter.isdigit() or letter in ('a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E', 'F'):

status = 'HEX NUMBER'

elif letter.isdigit():

status = 'ERROR'

else:

Lex\_result.append({"Тип лексемы": "десятичное число", "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

case 'HX':

if letter.isalnum():

status = 'ERROR'

else:

Lex\_result.append({"Тип лексемы": "число в шестнадцатеричной системе", "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

case 'E11':

if letter.isdigit():

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'E12'

elif letter in ('+', '-'):

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'ZN'

elif letter in ('h', 'H'):

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'HX'

elif letter.isdigit() or letter in ('a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E', 'F'):

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'HEX NUMBER'

else:

status = 'ERROR'

case 'ZN':

if letter.isdigit():

Продолжение листинга А

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'E13'

else:

status = 'ERROR'

case 'E12':

while letter.isdigit():

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

if letter.isdigit() or letter in ('a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E', 'F'):

status = 'HEX NUMBER'

elif letter in ('h', 'H'):

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'HX'

elif letter.isalpha():

status = 'ERROR'

else:

Lex\_result.append({"Тип лексемы": "действительное число", "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

case 'E13':

while letter.isdigit():

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

if letter.isalpha() or letter == '.':

status = 'ERROR'

else:

Lex\_result.append({"Тип лексемы": "действительное число", "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

case 'P1':

if letter.isdigit():

status = 'P2'

else:

status = 'ERROR'

case 'P2':

while letter.isdigit():

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

if letter in ('e', 'E'):

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'E21'

Продолжение листинга А

elif letter.isalpha() or letter == '.':

status = 'ERROR'

else:

Lex\_result.append({"Тип лексемы": "действительное число", "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

case 'E21':

if letter in ('+', '-'):

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'ZN'

elif letter.isdigit():

status = 'E22'

else:

status = 'ERROR'

case 'E22':

while letter.isdigit():

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

if (letter.isalpha() or letter == '.'):

status = 'ERROR'

else:

Lex\_result.append({"Тип лексемы": "действительное число", "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

case 'C1':

if letter == '\*':

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'C2'

else:

Lex\_result.append({"Тип лексемы": define\_dict[buffer], "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

case 'C2':

while total != len(data) and letter != '\*':

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

buffer = ''

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'C3'

case 'C3':

if letter == '/':

buffer += letter

gc()

Продолжение листинга А

if not FLAG\_READ\_FILE:

return Lex\_result

status = 'H'

else:

status = 'C2'

case 'M1':

if letter == '>':

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

Lex\_result.append({"Тип лексемы": define\_dict[buffer], "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

elif letter == '=':

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

Lex\_result.append({"Тип лексемы": define\_dict[buffer], "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

else:

Lex\_result.append({"Тип лексемы": define\_dict[buffer], "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

case 'M2':

if letter == '>':

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

Lex\_result.append({"Тип лексемы": define\_dict[buffer], "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

elif letter == '=':

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

Lex\_result.append({"Тип лексемы": define\_dict[buffer], "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

else:

Lex\_result.append({"Тип лексемы": define\_dict[buffer], "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

case 'SEPARATOR':

buffer = letter

Продолжение листинга А

if buffer in separators:

FLAG\_READ\_FILE = gc()

if not FLAG\_READ\_FILE:

return Lex\_result

if buffer == '[':

flag = True

elif buffer == ']':

flag = False

Lex\_result.append({"Тип лексемы": define\_dict[buffer], "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

else:

status = 'ERROR'

case 'ERROR':

while FLAG\_READ\_FILE and letter not in (' ', '\n', '\t') and letter not in separators:

buffer += letter

gc()

if not FLAG\_READ\_FILE:

return Lex\_result

Lex\_result.append({"Тип лексемы": "ERR", "Значение": buffer,

"Номер строки": line\_number, "Позиция в строке": on\_line\_number-len(buffer)})

status = 'H'

### Приложение Б

Исходный код синтаксического анализатора

Листинг Б — Код функции Syntactic

def Syntactic():

def GL():

nonlocal total, prev\_lex, current\_lex

if total < len(data):

prev\_lex = current\_lex

current\_lex = data[total]

total += 1

return True

else:

print(

"========== Сборка: успешно: 1, сбой: 0, в актуальном состоянии: 0, пропущено: 0==========")

exit(0)

def EXPRESSION\_SEQUENCE():

GL()

EXPRESSION()

while current\_lex['Тип лексемы'] == 'COMMA':

GL()

EXPRESSION()

def ID\_SEQUENCE():

if current\_lex['Тип лексемы'] == 'ID':

GL()

while current\_lex['Тип лексемы'] == 'COMMA':

GL()

if current\_lex['Тип лексемы'] == 'ID':

GL()

else:

ERROR\_HANDLER(5)

else:

ERROR\_HANDLER(6)

def ERROR\_HANDLER(n: int):

match n:

case 1:

print("Отсутсвует символ }")

case 2:

print("Отсутствует символ {")

case 3:

print("Требуется оператор или описание переменной")

case 4:

print("Ожидалась точка с запятой")

case 5:

print("Ошибка в последовательности идентификаторов")

case 6:

print("Требуется идентификатор")

case 7:

print("Отсутствует символ )")

case 8:

print("Требуется тип переменной")

case 9:

print("Отсутствует оператор then")

case 10:

print("Отсутствует оператор do")

Продолжение листинга Б

case 11:

print("Отсутствует оператор do")

case 12:

print("Отсутствует оператор to")

case 13:

print("Отсутствует символ )")

case 14:

print("Отсутствует символ (")

case 15:

print("Отсутсвует символ )")

case 16:

print("Отсутствует символ (")

case 17:

print("Отсутствует символ ]")

case 18:

print("Ошибка в выражении")

case 19:

pass

case 20:

print("Неизвестный оператор")

case 21:

print("Требовался идентификатор")

print(

f"line {current\_lex['Номер строки']}, position {current\_lex['Позиция в строке']}")

exit(n)

def DESCRIPTION():

if current\_lex['Тип лексемы'] == "DIM":

GL()

ID\_SEQUENCE()

if current\_lex['Тип лексемы'] not in ('INTEGER', 'REAL', 'BOOLEAN'):

ERROR\_HANDLER(8)

def OPERATOR():

match current\_lex['Тип лексемы']:

case "IF":

GL()

CONDITION\_OPERATOR()

case 'WHILE':

GL()

WHILE\_OPERATOR()

case "FOR":

GL()

FOR\_OPERATOR()

case "READ":

GL()

READ\_OPERATOR()

case "WRITE":

GL()

WRITE\_OPERATOR()

case "ID": # ASSIGN -> ID as EXPR

GL()

ASSIGN\_OPERATOR()

case "BEGIN\_COMPLEX":

GL()

COMPLEX\_OPERATOR()

case \_:

ERROR\_HANDLER(20)

Продолжение листинга Б

def EXPRESSION():

OPERAND()

if current\_lex['Тип лексемы'] in ('EQUAL', 'LESS', 'GREATER', 'LEQ', 'GEQ', 'NEQ'):

GL()

EXPRESSION()

def OPERAND():

SUMMAND()

if current\_lex['Тип лексемы'] in ('PLUS', 'MINUS', 'OR'):

GL()

OPERAND()

def SUMMAND():

MULTIPLIER()

if current\_lex['Тип лексемы'] in ('DIVIDE', 'MULT', 'AND'):

GL()

SUMMAND()

def MULTIPLIER():

if current\_lex['Тип лексемы'] in ('ID', 'действительное число', 'десятичное число', 'число в шестнадцатеричной системе',

'число в восьмеричной системе', 'число в двоичной системе', 'TRUE', 'FALSE'):

GL()

elif current\_lex['Тип лексемы'] == 'NOT':

GL()

MULTIPLIER()

elif current\_lex['Тип лексемы'] == 'LEFT\_PAREN':

GL()

EXPRESSION()

if current\_lex['Тип лексемы'] == 'RIGHT\_PAREN':

GL()

else:

ERROR\_HANDLER(7)

else:

ERROR\_HANDLER(18)

def CONDITION\_OPERATOR():

EXPRESSION()

if current\_lex['Тип лексемы'] == 'THEN':

GL()

OPERATOR()

if current\_lex['Тип лексемы'] == 'ELSE':

GL()

OPERATOR()

else:

ERROR\_HANDLER(9)

def WHILE\_OPERATOR():

EXPRESSION()

if current\_lex['Тип лексемы'] == 'DO':

GL()

OPERATOR()

else:

ERROR\_HANDLER(10)

def FOR\_OPERATOR():

if current\_lex['Тип лексемы'] != 'ID':

ERROR\_HANDLER(21)

Return

Продолжение листинга Б

GL()

ASSIGN\_OPERATOR()

if current\_lex['Тип лексемы'] == 'TO':

GL()

EXPRESSION()

if current\_lex['Тип лексемы'] == 'DO':

GL()

OPERATOR()

else:

ERROR\_HANDLER(11)

else:

ERROR\_HANDLER(12)

def ASSIGN\_OPERATOR():

if current\_lex['Тип лексемы'] == 'ASSIGN':

GL()

EXPRESSION()

def READ\_OPERATOR():

if current\_lex['Тип лексемы'] == 'LEFT\_PAREN':

GL()

ID\_SEQUENCE()

if current\_lex['Тип лексемы'] != 'RIGHT\_PAREN':

ERROR\_HANDLER(13)

else:

GL()

else:

ERROR\_HANDLER(14)

def WRITE\_OPERATOR():

if current\_lex['Тип лексемы'] == 'LEFT\_PAREN':

EXPRESSION\_SEQUENCE()

if current\_lex['Тип лексемы'] == 'RIGHT\_PAREN':

GL()

else:

ERROR\_HANDLER(15)

else:

ERROR\_HANDLER(16)

def COMPLEX\_OPERATOR():

OPERATOR()

while current\_lex['Тип лексемы'] in ('COLON', 'LINE\_BREAK'):

GL()

OPERATOR()

if current\_lex['Тип лексемы'] != 'END\_COMPLEX':

ERROR\_HANDLER(17)

else:

GL()

def ANALYZE():

PROGRAM()

def PROGRAM():

GL()

if current\_lex['Тип лексемы'] == "BEGIN\_PROG":

GL()

BODY()

ERROR\_HANDLER(1)

else:

ERROR\_HANDLER(2)

Продолжение листинга Б

def BODY():

if current\_lex['Тип лексемы'] == "DIM":

DESCRIPTION()

GL()

elif current\_lex['Тип лексемы'] in ('ID', 'FOR', 'WHILE', 'IF', 'BEGIN\_COMPLEX', 'READ', 'WRITE'):

OPERATOR()

else:

ERROR\_HANDLER(3)

while current\_lex['Тип лексемы'] == "SEMICOLON":

GL()

if current\_lex['Тип лексемы'] == "DIM":

DESCRIPTION()

GL()

elif current\_lex['Тип лексемы'] in ('ID', 'FOR', 'WHILE', 'IF', 'BEGIN\_COMPLEX', 'READ', 'WRITE'):

OPERATOR()

elif current\_lex['Тип лексемы'] == "END\_PROG":

GL()

break

else:

ERROR\_HANDLER(4)

if current\_lex['Тип лексемы'] != "SEMICOLON":

ERROR\_HANDLER(4)

total = 0

current\_lex = data[total]

prev\_lex = dict()

ANALYZE()

### Приложение В

Исходный код запуска анализа программы

Листинг В — Исходный код

with open('Lexical.txt', encoding='utf-8') as file:

data = Lexical()

for d in data:

if d['Тип лексемы'] == "ERR":

print(f"Неизвестная лексема: {d['Значение']}\nline {d['Номер строки']}, position {d['Позиция в строке']}")

exit(0)

Syntactic()