Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы трансляции

ОТЧЁТ

к лабораторной работе №2

на тему

**ЛЕКСИЧЕСКИЙ АНАЛИЗ**

Выполнил: студент гр.253504 Лавренова А.С.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

Введение3

[1  [Ц](#_agswffz6g6ei)ель работы](#_Toc178067889) 4

[2  Теоретические сведения 5](#_Toc178067890)

[3  Описание и пример выполнения программы 7](#_Toc178067891)

Заключение [11](#_Toc178067899)

[Список использованных источников 12](#_Toc178067900)

[Приложение А (обязательное) 13](#_Toc178067901)

# ВВЕДЕНИЕ

Лексический анализ является первым этапом обработки исходного кода в компиляторе или интерпретаторе. Его задача — преобразование входного потока символов в последовательность лексем, что позволяет следующему этапу (синтаксическому анализу) работать с более структурированными данными.

В данной лабораторной работе требуется разработать лексический анализатор для подмножества языка программирования *Focal*, определенного в рамках лабораторной работы 1. Анализатор должен обрабатывать входной текст, выделять ключевые слова, идентификаторы, операторы, константы и другие элементы языка, используя заранее определенные лексические правила. Также необходимо реализовать механизм обработки ошибок, который позволит выявлять и корректно сообщать о некорректных последовательностях символов.

В процессе выполнения работы потребуется:

* определить и формализовать лексические правила языка *Focal*;
* разработать механизм чтения входного файла, содержащего код на *Focal*;
* создать таблицы ключевых слов, констант, разделителей, операторов;
* реализовать алгоритм преобразования входного потока символов в поток лексем;
* разработать обработку ошибок и продемонстрировать обнаружение четырех различных лексических ошибок с соответствующими сообщениями.

Лексический анализатор будет реализован на языке *Kotlin*, что позволит использовать его мощные инструменты работы со строками, регулярными выражениями и коллекциями, упрощающие реализацию обработки кода.

1 ЦЕЛЬ РАБОТЫ

Цель данной лабораторной работы заключается в разработке и исследовании лексического анализатора для подмножества языка программирования *Focal*, определенного в рамках предыдущей лабораторной работы, с использованием языка программирования *Kotlin*. Лексический анализатор является ключевым компонентом компиляторов и интерпретаторов, так как он выполняет первичную обработку исходного кода, разбивая поток символов на осмысленные единицы — лексемы. Создание такого анализатора требует понимания принципов построения языков программирования, знания структур данных и алгоритмов обработки текста, а также навыков работы с инструментами для анализа и трансформации текстовых данных.

В ходе выполнения работы предстоит изучить основные подходы к лексическому анализу, включая ручную разработку анализатора и возможности использования существующих инструментов для автоматической генерации лексеров. Необходимо определить лексические правила, формализовать их в виде набора регулярных выражений или конечных автоматов, разработать алгоритм обработки входного потока символов и выделения лексем. Важной частью работы является создание структурированных таблиц ключевых слов, операторов, разделителей, числовых и строковых констант, что позволит унифицировать процесс анализа и упростить последующую обработку лексем.

Отдельное внимание уделяется обработке ошибок, возникающих при разборе кода. Лексический анализатор должен не только корректно распознавать допустимые элементы языка, но и эффективно выявлять некорректные последовательности символов, выдавая информативные сообщения об ошибках. Для демонстрации возможностей анализатора в отчете необходимо привести примеры нахождения и обработки четырех различных типов лексических ошибок.

Разработка лексического анализатора на языке *Kotlin* дает возможность использовать его мощные средства работы со строками, коллекциями и регулярными выражениями, что значительно упрощает реализацию обработки текста. Освоение этих инструментов и принципов лексического анализа является важным шагом в изучении компиляторостроения и языковых технологий. Выполнение данной лабораторной работы позволит приобрести практический опыт создания инструментов для анализа и обработки языков программирования, что особенно ценно при разработке компиляторов, интерпретаторов, анализаторов кода и других систем, связанных с обработкой текстовых данных.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Лексический анализ, также известный как сканирование, — это первый этап компиляции, который включает в себя чтение исходной программы посимвольно слева направо и объединение символов в лексемы. Лексемы — это значимые последовательности символов. Обычно в языке программирования существует лишь небольшое количество лексем, включая константы (например, целые числа, числа с плавающей точкой, символы и строки), операторы (арифметические, реляционные и логические), знаки препинания и зарезервированные ключевые слова. [1]

Лексический токен — это последовательность символов, которую можно рассматривать как единое целое в грамматике языков программирования.

Категории токенов:

1. Ключевые слова. В программировании ключевые слова — это зарезервированные слова с определённым значением, которые используются для определения структуры языка, например *if, else, for* и *void*. Их нельзя использовать в качестве имён переменных или идентификаторов, так как это приводит к ошибкам компиляции.
2. Идентификаторы.Идентификаторы — это имена переменных, функций, массивов или других пользовательских элементов. Они должны начинаться с буквы или символа подчеркивания и могут содержать буквы, цифры и символы подчеркивания. Во многих языках программмирования учитывается регистр, поэтому прописные и строчные буквы различаются. Идентификаторы не могут совпадать с ключевыми словами, такими как *if*, *else* или *for*.
3. Константы.Константы — это фиксированные значения, которые не могут изменяться во время выполнения программы. Они также известны как литералы. В C константы включают такие типы, как целые числа, числа с плавающей запятой, символы и строки.
4. Операторы.Операторы — это символы, которые выполняют действия с переменными или другими элементами данных, называемыми операндами.
5. Специальные символы.Специальные символы — это лексемы компилятора, используемые для определённых целей, например для разделения элементов кода или определения операций. Например точка с запятой для завершения операторов, запятая для разделения значений, фигурные скобки. Эти символы играют важную роль в структуре и синтаксисе программы.

Лексема — это фактическая строка символов, которая соответствует шаблону и генерирует токен. [2]

Существует несколько подходов к разработке лексического анализатора, каждый из которых имеет свои особенности и области применения.

1. Ручная реализация с использованием регулярных выражений. Этот метод предполагает написание кода, который анализирует входную строку и выделяет токены на основе регулярных выражений. Такой подход подходит для относительно простых языков программирования и позволяет быстро реализовать анализатор, используя стандартные инструменты работы со строками в *Kotlin*.
2. Использование конечных автоматов. Конечный автомат — это математическая модель, которая позволяет описывать поведение лексического анализатора в виде состояний и переходов между ними. Этот метод более сложный, но он даёт высокую производительность и чёткую структурированность анализа. Автомат может быть реализован вручную или сгенерирован автоматически с использованием специальных инструментов.
3. Автоматическая генерация лексера. Существуют инструменты, такие как *Flex*, *JFlex*, *ANTLR*, которые позволяют описывать правила лексического анализа в виде специальных спецификаций, после чего генератор создаёт готовый код анализатора. Этот подход значительно сокращает время разработки, но требует освоения дополнительного инструментария.
4. Комбинированный подход. В реальных проектах часто используется комбинированный подход, когда часть работы выполняется вручную, а часть автоматизируется с помощью генераторов. Например, можно вручную обрабатывать специфические случаи, а основные правила задать с использованием регулярных выражений или конечных автоматов. [3]

Лексический анализ — это неотъемлемая часть процесса компиляции, которая обеспечивает правильное разбиение исходного текста на лексемы и их классификацию. Этот этап играет ключевую роль в обеспечении корректности и эффективности компиляции программного кода. Понимание принципов и задач лексического анализа поможет вам лучше разобраться в том, как работают компиляторы и другие инструменты для обработки кода.

Лексический анализ также является основой для многих других инструментов и технологий, используемых в разработке программного обеспечения. Например, инструменты статического анализа кода, подсветка синтаксиса в редакторах кода и многие другие технологии зависят от правильного и эффективного лексического анализа. Поэтому знание и понимание этого процесса является важным для любого разработчика программного обеспечения.

**3 ОПИСАНИЕ И ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ**

Программа представляет собой лексический анализатор для языка *Focal*, реализованный на языке *Kotlin*. Она обрабатывает входные файлы с кодом на *Focal*, выполняет разбор лексем и классифицирует их в соответствии с заданными категориями.

Входные данные загружаются из текстовых файлов. Лексический анализатор выделяет следующие типы лексем:

* *LINE\_NUMBER* – номер строки;
* *NUMBER* – числовые значения;
* *KEYWORD* – ключевые слова (*LET*, *END*, *FOR*, *NEXT*, *TYPE*, *ACCEPT*);
* *OPERATOR* – операторы (=, +, -, \*, /, ^, :);
* *SYMBOL* – специальные символы ((, ), {, }, ;, ", ,);
* *Literal Strings* – строковые литералы;
* *IDENTIFIER* – идентификаторы;
* *COMMENT* – комментарии;
* *ARRAY* – массивы;
* *MATH\_FUNCTION* – математические функции (*FSIN*, *FCOS*, *FATN*, FLOG, *FSQT*, и др.);
* *UNKNOWN* – некорректные или неопределённые лексемы.

После обработки входного файла программа выводит список всех лексем, распределённых по классам (рисунок 3.1).

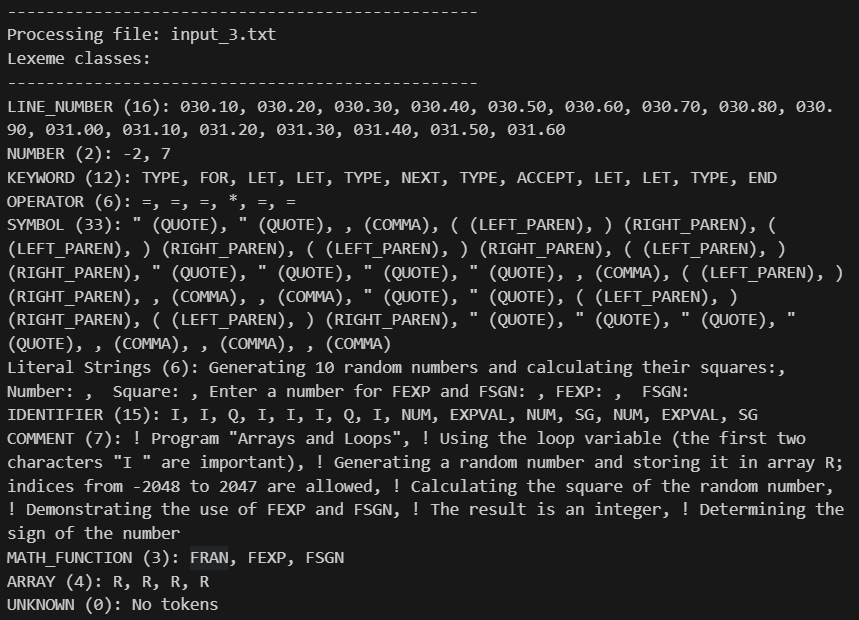


Рисунок 3.1 – Список всех лексем

На данном изображении представлен результат работы программы для одного из тестовых входных файлов. В первой части вывода указаны классы лексем, а также их количество и сами лексемы.

Следующим этапом является генерация таблицы, содержащей полный список всех найденных лексем в том порядке, в котором они встречаются в коде. Данная таблица продемонстрирована на рисунке 3.2. Каждая строка таблицы включает:

* номер элемента (позиция в исходном коде);
* значение лексемы;
* класс лексемы.

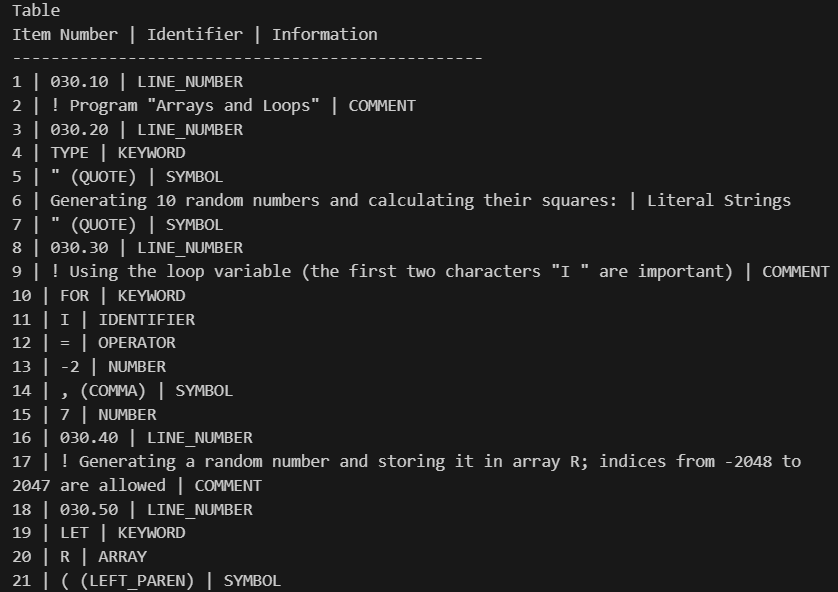


Рисунок 3.2 – Таблица всех лексем

На изображении можно увидеть табличное представление результатов анализа. Например, первая строка содержит номер строки 30 с категорией *LINE\_NUMBER*, далее идет комментарий, затем начинается новая строка и через ключевое слово *TYPE* запрашивает пользователя ввести данные.

Таблица помогает визуально определить структуру кода и выявить возможные ошибки.

После сбора всех лексем программа формирует список уникальных значений для каждой категории. Это позволяет увидеть, какие элементы использовались в коде без повторений (рисунок 3.3).

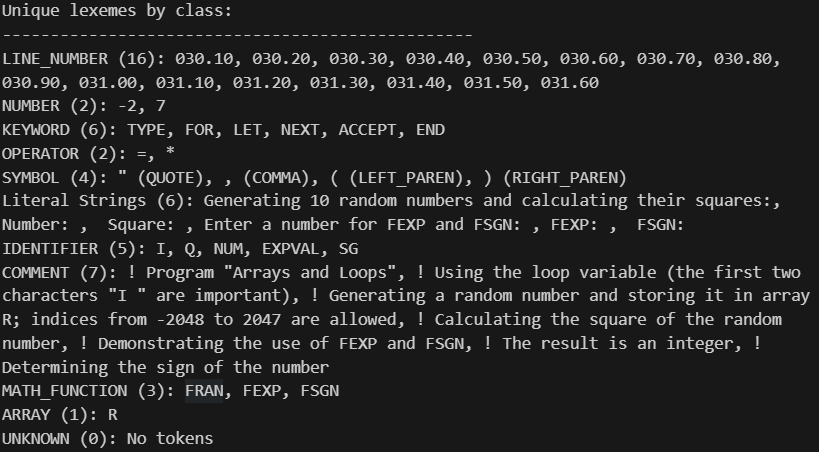


Рисунок 3.3 – Список всех уникальных символов

В данном выводе отображаются только уникальные лексемы, без дублирования. Например, если в коде несколько раз использовался оператор =, он будет отображён только один раз. Это удобно для анализа используемых конструкций.

При обработке программы анализатор также выполняет проверку корректности структуры кода.

Если рассматривать процесс выявления ошибок подробнее, то следует упомянуть, что лексический анализатор проверяет, что номера строк идут в порядке возрастания. Если порядок нарушен, выводится ошибка, изображённая на рисунке 3.4.

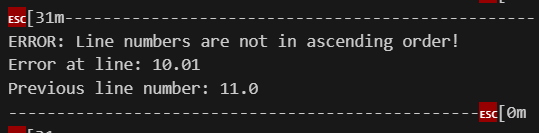


Рисунок 3.4 – Ошибка в нумерации строк

На изображении показан фрагмент кода, где строка с номером 11 встречается перед строкой 10, что является нарушением правил языка *Focal*.

Также следует упомянуть, что программа должна заканчиваться ключевым словом *END*. Если оно отсутствует, выводится ошибка, изображённая на рисунке 3.5.

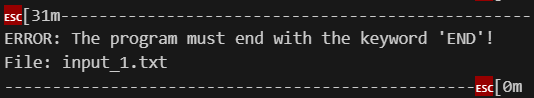


Рисунок 3.5 – Ошибка ключевого слова *END*

В этом примере видно, что последний оператор не является END, что приводит к ошибке.

Также после ключевого слова *TYPE* должны следовать корректно закрытые строковые литералы (рисунок 3.6).

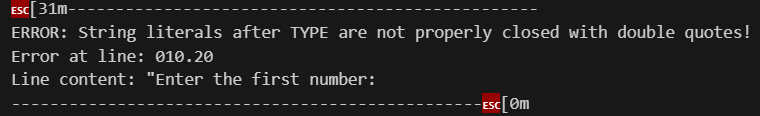


Рисунок 3.6 – Ошибка ключевого слова *TYPE*

На изображении видно, что строковый литерал начинается с ", но не имеет закрывающей кавычки.

Любая лексема, которая не соответствует определённым правилам, помечается как *UNKNOWN* (рисунок 3.7).



Рисунок 3.7 – Неизвестный символ

В данном случае программа обнаружила символ, который не относится ни к одной из известных категорий.

Программа была протестирована на трёх различных кодах на языке *Focal*, разработанных в первой лабораторной работе (*input1.txt*, *input2.txt*, *input3.txt*). В результате выполнения были получены следующие результаты:

Вывод классифицированных лексем по категориям;

* список всех лексем по категориям;
* таблица лексем с номерами, идентификаторами и типами в той последовательности, в которой они встречаются в коде;
* список уникальных лексем по категориям;
* обнаружение возможных ошибок в коде.

В ходе тестирования лексического анализатора программа корректно классифицировала лексемы, сформировала таблицу, выделила уникальные элементы и обнаружила четыре типа ошибок. Данный анализ показывает эффективность разработанного метода лексического разбора и может быть использован для дальнейшей работы с языком *Focal*.

# ЗАКЛЮЧЕНИЕ

В ходе данной лабораторной работы был разработан и протестирован лексический анализатор для языка *Focal*, реализованный на языке программирования *Kotlin*. Основная цель заключалась в анализе кода, представленном в текстовых файлах, разбиении его на лексемы, классификации по категориям и выявлении возможных синтаксических ошибок.

Разработка анализатора включала несколько ключевых этапов. В первую очередь были определены правила лексического анализа, в соответствии с которыми выделены основные классы лексем, характерные для языка *Focal*. К ним относятся номера строк, ключевые слова, операторы, идентификаторы, комментарии, строковые литералы, массивы и специальные символы.

На следующем этапе был реализован механизм разбора кода, основанный на алгоритме последовательного анализа строк, их токенизации и распределения по соответствующим классам. В рамках реализации выполнялась проверка корректного завершения кода ключевым словом *END*, контроль за правильным порядком номеров строк, а также валидация строковых литералов после оператора *TYPE*.

После этого программа была протестирована на трёх различных примерах кода на языке *Focal*, созданных в первой лабораторной работе. В результате выполнения были получены данные, содержащие классифицированный список лексем, таблицу с последовательностью всех лексем и их идентификаторами, список уникальных лексем по категориям и выявленные ошибки.

В процессе анализа первого тестового кода были зафиксированы четыре типа лексических ошибок. Среди них отсутствие завершающего ключевого слова *END*, нарушение порядка номеров строк, незакрытая строка после оператора *TYPE*, а также неопределённые символы, не соответствующие стандартному синтаксису *Focal*. Каждая из этих ошибок была успешно обнаружена программой, а их описание и возможные причины подробно рассмотрены в отчёте.

Проведённая лабораторная работа продемонстрировала важность лексического анализа как неотъемлемой части компиляции и интерпретации программного кода. Разработанный анализатор успешно справился с задачей классификации лексем и выявления ошибок, что подтверждается результатами тестирования. Применение языка *Kotlin* позволило реализовать эффективный и читаемый код, а также использовать мощные инструменты обработки строк и коллекций. В процессе выполнения работы были приобретены практические навыки работы с регулярными выражениями, анализа кода и автоматического выявления ошибок.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Основы построения трансляторов языков программирования. Е. В. Шостак, И. М. Марина, Д. Е. Оношко [Электронный ресурс]. – Режим доступа: https://libeldoc.bsuir.by/bitstram/123456789/35077/1/Shostak\_2019.pdf.

[2] Генерация автоматных лексических анализаторов по регулярным выражениям. И. И. Чернявский [Электронный ресурс]. – Режим доступа: https ://is.ifmo.ru/projects/lex/doc.pdf.

[3] Кнспектное изложение теории языков программирования и методов трансляции. Вл. Понаморёв [Электронный ресурс]. – Режим доступа: https://t imeweb.cloud/tutorials/linux/regulyarnye—vyrazheniya—bash—gajd?ysclid=m82 zxlerds129168253.

ПРИЛОЖЕНИЕ А

(обязательное)

Исходный код

import java.io.File

fun main() {

    val inputFiles = listOf("input\_1.txt", "input\_2.txt", "input\_3.txt")

    val lexemeOrder = listOf(

        "LINE\_NUMBER", "NUMBER", "KEYWORD", "OPERATOR", "SYMBOL",

        "Literal Strings", "IDENTIFIER", "COMMENT", "MATH\_FUNCTION", "ARRAY", "UNKNOWN"

    )

    for (fileName in inputFiles) {

        val file = File(fileName)

        if (!file.exists()) {

            println("Error: File $fileName not found.")

            continue

        }

        val code = file.readLines()

        val tokens = tokenize(code)

        // Проверка, что последняя строка содержит END

        if (code.lastOrNull()?.trim()?.endsWith("END") != true) {

            println("\u001B[31m-------------------------------------------------")

            println("ERROR: The program must end with the keyword 'END'!")

            println("File: $fileName")

            println("-------------------------------------------------\u001B[0m")

        }

        println("-------------------------------------------------")

        println("Processing file: $fileName")

        println("Lexeme classes: ")

        println("-------------------------------------------------")

        // Группируем лексемы по классам (все элементы)

        val lexemeClasses = mutableMapOf<String, MutableList<String>>()

        for (token in tokens) {

            lexemeClasses.computeIfAbsent(token.type) { mutableListOf() }.add(token.value)

        }

        // Выводим лексемы по классам в порядке lexemeOrder (все элементы)

        for (type in lexemeOrder) {

            val lexemes = lexemeClasses[type] ?: emptyList()

            val lexemeList = if (lexemes.isEmpty()) "No tokens" else lexemes.joinToString(", ")

            println("$type (${lexemes.size}): $lexemeList")

        }

        println()

        println("Table ")

        println("Item Number | Identifier | Information")

        println("-------------------------------------------------")

        tokens.forEachIndexed { index, token ->

            println("${index + 1} | ${token.value} | ${token.type}")

        }

        println("\n")

        // Третий вывод: уникальные лексемы по классам

        println("Unique lexemes by class: ")

        println("-------------------------------------------------")

        // Группируем лексемы по классам (уникальные элементы)

        val uniqueLexemeClasses = mutableMapOf<String, MutableSet<String>>()

        for (token in tokens) {

            uniqueLexemeClasses.computeIfAbsent(token.type) { mutableSetOf() }.add(token.value)

        }

        // Выводим уникальные лексемы по классам в порядке lexemeOrder

        for (type in lexemeOrder) {

            val lexemes = uniqueLexemeClasses[type] ?: emptySet()

            val lexemeList = if (lexemes.isEmpty()) "No tokens" else lexemes.joinToString(", ")

            println("$type (${lexemes.size}): $lexemeList")

        }

        println("\n")

    }

}

data class Token(val type: String, val value: String)

fun tokenize(codeLines: List<String>): List<Token> {

    val tokens = mutableListOf<Token>()

    val operators = setOf("=", "+", "-", "\*", "/", "^", ":")

    val keywords = setOf("LET", "END", "FOR", "NEXT", "TYPE", "ACCEPT")

    val symbols = mapOf(

        "(" to "LEFT\_PAREN", ")" to "RIGHT\_PAREN", "{" to "LEFT\_BRACE", "}" to "RIGHT\_BRACE",

        ";" to "SEMICOLON", "\"" to "QUOTE", "," to "COMMA"

    )

    val mathFunctions = setOf("FSIN", "FCOS", "FATN", "FLOG", "FSQT", "FABS", "FITR", "ABSVAL", "FEXP", "FSGN", "EXPVAL", "FRAN")

    val numberRegex = Regex("[-]?\\d+(\\.\\d+)?")

    val lineNumberRegex = Regex("^(\\d+\\.\\d+|\\d+)\\s") // Номер строки должен заканчиваться пробелом

    val identifierRegex = Regex("^[A-Za-z\_][A-Za-z0-9\_]\*$")

    val commentRegex = Regex("!.\*")

    val functionRegex = Regex("([A-Z]+)\\((.\*)\\)") // Регулярное выражение для математических функций

    val arrayRegex = Regex("([A-Za-z\_][A-Za-z0-9\_]\*)\\(([^)]\*)\\)") // Регулярное выражение для массивов

    var lastLineNumber: Double? = null // Переменная для хранения последнего номера строки

    for (line in codeLines) {

        var remainingLine = line.trim()

        if (remainingLine.isEmpty()) continue

        // Сначала обрабатываем номер строки

        val lineNumberMatch = lineNumberRegex.find(remainingLine)

        if (lineNumberMatch != null) {

            val currentLineNumber = lineNumberMatch.value.trim().toDouble()

            // Проверяем, что номер строки возрастает

            if (lastLineNumber != null && currentLineNumber <= lastLineNumber) {

                // Яркое оформление ошибки

                println("\u001B[31m-------------------------------------------------")

                println("ERROR: Line numbers are not in ascending order!")

                println("Error at line: $currentLineNumber")

                println("Previous line number: $lastLineNumber")

                println("-------------------------------------------------\u001B[0m")

            }

            lastLineNumber = currentLineNumber // Обновляем последний номер строки

            tokens.add(Token("LINE\_NUMBER", lineNumberMatch.value.trim()))

            remainingLine = remainingLine.substring(lineNumberMatch.range.last + 1).trim()

        }

        // Затем обрабатываем комментарий

        val commentMatch = commentRegex.find(remainingLine)

        if (commentMatch != null) {

            tokens.add(Token("COMMENT", commentMatch.value))

            remainingLine = remainingLine.split("!").first().trim()

        }

        // Обрабатываем строки после TYPE

        if (remainingLine.startsWith("TYPE")) {

            // Добавляем TYPE как KEYWORD

            tokens.add(Token("KEYWORD", "TYPE"))

            remainingLine = remainingLine.substringAfter("TYPE").trim()

            // Проверяем, что все кавычки закрыты

            val quoteCount = remainingLine.count { it == '"' }

            if (quoteCount % 2 != 0) {

                // Яркое оформление ошибки

                println("\u001B[31m-------------------------------------------------")

                println("ERROR: String literals after TYPE are not properly closed with double quotes!")

                println("Error at line: ${lineNumberMatch?.value?.trim() ?: "unknown"}")

                println("Line content: $remainingLine")

                println("-------------------------------------------------\u001B[0m")

            } else {

                val parts = remainingLine.split("\"")

                for (i in parts.indices step 2) {

                    if (i + 1 < parts.size) {

                        tokens.add(Token("SYMBOL", "\" (QUOTE)"))

                        tokens.add(Token("Literal Strings", parts[i + 1]))

                        tokens.add(Token("SYMBOL", "\" (QUOTE)"))

                    }

                }

               оставшейся части строки

                remainingLine = parts.filterIndexed { index, \_ -> index % 2 == 0 }.joinToString("").trim()

            }

        }

        // Обрабатываем оставшуюся часть строки

        val words = mutableListOf<String>()

        var insideString = false

        val stringBuffer = StringBuilder()

        for (char in remainingLine) {

            when {

                char == '"' -> {

                    if (insideString) {

                        // Если строка заканчивается, добавляем всё содержимое как одну лексему

                        words.add(stringBuffer.toString())

                        stringBuffer.clear()

                        insideString = false

                    } else {

                        // Если строка начинается, добавляем открывающую кавычку

                        if (stringBuffer.isNotEmpty()) {

                            words.add(stringBuffer.toString())

                            stringBuffer.clear()

                        }

                        insideString = true

                    }

                }

                insideString -> stringBuffer.append(char) // Внутри строки добавляем символы в буфер

                char.isWhitespace() -> if (stringBuffer.isNotEmpty()) {

                    words.add(stringBuffer.toString())

                    stringBuffer.clear()

                }

                else -> stringBuffer.append(char)

            }

        }

        if (stringBuffer.isNotEmpty()) words.add(stringBuffer.toString())

        // Разделяем идентификаторы и запятые, если они идут вместе без пробела

        val processedWords = mutableListOf<String>()

        for (word in words) {

            if (word.endsWith(",") && word.length > 1) {

                // Разделяем идентификатор и запятую

                processedWords.add(word.dropLast(1)) // Идентификатор

                processedWords.add(",") // Запятая

            } else {

                processedWords.add(word)

            }

        }

        for (word in processedWords) {

            when {

                word == "\"" -> tokens.add(Token("SYMBOL", "\" (QUOTE)"))

                numberRegex.matches(word) -> tokens.add(Token("NUMBER", word)) // Обрабатываем числа

                word in keywords -> tokens.add(Token("KEYWORD", word))

                word in operators -> tokens.add(Token("OPERATOR", word))

                symbols.containsKey(word) -> tokens.add(Token("SYMBOL", "${word} (${symbols[word]})"))

                functionRegex.matches(word) -> {

                    // Разбиваем математическую функцию на компоненты

                    val functionMatch = functionRegex.find(word)

                    if (functionMatch != null) {

                        val functionName = functionMatch.groupValues[1]

                        val arguments = functionMatch.groupValues[2]

                        // Добавляем имя функции как MATH\_FUNCTION, если оно есть в списке

                        if (functionName in mathFunctions) {

                            tokens.add(Token("MATH\_FUNCTION", functionName))

                        } else {

                            // Если имя функции не в списке, добавляем как ARRAY

                            tokens.add(Token("ARRAY", functionName))

                        }

                        // Добавляем открывающую скобку как SYMBOL

                        tokens.add(Token("SYMBOL", "( (LEFT\_PAREN)"))

                        // Обрабатываем аргументы

                        val argumentTokens = tokenize(listOf(arguments)) // Рекурсивно обрабатываем аргументы

                        tokens.addAll(argumentTokens)

                        // Добавляем закрывающую скобку как SYMBOL

                        tokens.add(Token("SYMBOL", ") (RIGHT\_PAREN)"))

                    }

                }

                arrayRegex.matches(word) -> {

                    // Обрабатываем массивы

                    val arrayMatch = arrayRegex.find(word)

                    if (arrayMatch != null) {

                        val arrayName = arrayMatch.groupValues[1]

                        val index = arrayMatch.groupValues[2]

                        // Добавляем имя массива как ARRAY

                        tokens.add(Token("ARRAY", arrayName))

                        // Добавляем открывающую скобку как SYMBOL

                        tokens.add(Token("SYMBOL", "( (LEFT\_PAREN)"))

                        // Обрабатываем индекс

                        val indexTokens = tokenize(listOf(index)) // Рекурсивно обрабатываем индекс

                        tokens.addAll(indexTokens)

                        // Добавляем закрывающую скобку как SYMBOL

                        tokens.add(Token("SYMBOL", ") (RIGHT\_PAREN)"))

                    }

                }

                word.startsWith("(") || word.endsWith(")") -> {

                    // Обрабатываем выражения в скобках

                    if (word.startsWith("(")) {

                        tokens.add(Token("SYMBOL", "( (LEFT\_PAREN)"))

                        val content = word.substring(1)

                        if (content.isNotEmpty()) {

                            val contentTokens = tokenize(listOf(content)) // Рекурсивно обрабатываем содержимое

                            tokens.addAll(contentTokens)

                        }

                    }

                    if (word.endsWith(")")) {

                        val content = word.dropLast(1)

                        if (content.isNotEmpty()) {

                            val contentTokens = tokenize(listOf(content)) // Рекурсивно обрабатываем содержимое

                            tokens.addAll(contentTokens)

                        }

                        tokens.add(Token("SYMBOL", ") (RIGHT\_PAREN)"))

                    }

                }

                identifierRegex.matches(word) -> tokens.add(Token("IDENTIFIER", word))

                word.startsWith("-") && identifierRegex.matches(word.drop(1)) -> tokens.add(Token("IDENTIFIER", word))

                word == "," -> tokens.add(Token("SYMBOL", ", (COMMA)"))

                else -> tokens.add(Token("UNKNOWN", word))

            }

        }

    }

    return tokens

}