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 №5

на тему

**ИНТЕРПРЕТАЦИЯ ИСХОДНОГО КОДА**

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 6](#_Toc178067891)

Заключение8

[Список использованных источников 9](#_Toc178067900)

[Приложение А (обязательное) 10](#_Toc178067901)

# ВВЕДЕНИЕ

В процессе разработки систем программирования одним из завершающих этапов становится интерпретация исходного кода — непосредственное выполнение программы на основе результатов лексического, синтаксического и семантического анализа. Интерпретация предполагает не просто проверку структуры или смысла текста программы, но активную реализацию её логики: вычисление выражений, выполнение управляющих конструкций, управление памятью и обработку данных в рамках заданной семантики языка. На данном этапе абстрактное представление программы, сформированное ранее в виде дерева разбора или внутреннего представления, превращается в набор конкретных вычислений и операций, воспроизводимых в среде выполнения.

Основная задача данной лабораторной работы заключается в реализации интерпретатора, способного выполнять исходный код, написанный на учебном языке FOCAL. Этот язык, использовавшийся в предыдущих лабораторных работах для анализа лексических и синтаксических структур, теперь становится объектом интерпретации — то есть интерпретатор должен не только распознавать конструкции языка, но и корректно исполнять их в соответствии с определённой моделью вычислений. Такой подход обеспечивает сквозное понимание процесса трансляции программ: от анализа текста до его запуска и получения результата.

Реализация интерпретатора на языке Kotlin предоставляет удобный и безопасный способ моделирования поведения программ FOCAL. Kotlin, ориентированный на работу с JVM, позволяет эффективно управлять памятью, использовать функциональные и объектно-ориентированные приёмы, а также интегрировать различные компоненты, разработанные на предыдущих этапах.

Таким образом, интерпретатор становится логическим продолжением всей системы анализа и обработки программного кода, формируя завершённую цепочку от исходного текста до выполнения. Работа с интерпретацией требует точного воспроизведения семантики языка: правильной обработки арифметики, тригонометрии, присваиваний, условных операторов и циклов, что требует глубокой интеграции знаний о структуре языка и построения надёжной внутренней модели исполнения.

Выполнение данной лабораторной работы позволяет не только закрепить понимание внутренних процессов работы компиляторов и интерпретаторов, но и получить практический опыт реализации исполняющей системы, ориентированной на конкретный язык.

1 ЦЕЛЬ РАБОТЫ

Целью данной лабораторной работы является разработка интерпретатора для подмножества языка программирования Focal с использованием языка Kotlin, а также исследование механизмов исполнения программного кода на основе ранее проведённого лексического, синтаксического и семантического анализа. В рамках выполнения задания необходимо обеспечить корректную интерпретацию конструкций исходного языка, включая выражения, операторы присваивания, управляющие структуры и встроенные функции, что позволит воспроизвести логическое поведение программ Focal без генерации промежуточного кода или трансляции в машинные инструкции.

Реализация интерпретатора предполагает построение среды выполнения, способной обрабатывать дерево разбора или аналогичное внутреннее представление программы, вычислять значения выражений, управлять областью видимости переменных, выполнять контроль типов и динамически отслеживать состояние выполнения. Такой подход позволяет полноценно смоделировать работу абстрактной вычислительной машины, соответствующей спецификации языка, и обеспечивает точное соответствие между описанием программы и её поведением во время интерпретации.

Интерпретатор является завершающим звеном в цепочке анализа исходного кода и демонстрирует, каким образом ранее собранная информация используется для непосредственного исполнения программы. Работа с интерпретацией развивает навыки построения исполняемых моделей языков программирования, помогает глубже понять принципы работы интерпретируемых систем и служит подготовкой к созданию полноценных виртуальных машин или инструментов выполнения кода. Использование языка Kotlin позволяет реализовать интерпретатор в безопасной и выразительной среде, обладающей широкими возможностями для обработки данных и моделирования вычислений, что делает процесс разработки максимально приближенным к современным практикам проектирования языковых инструментов.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Интерпретация — это процесс выполнения кода программы, когда вся работа делается в момент исполнения. При интерпретации код сразу переводится в машинный язык и выполняется построчно, тогда как при компиляции код сначала полностью переводится в машинный язык и только потом выполняется. [1]

Интерпретация позволяет более гибко контролировать выполнение программы и отлаживать ее на ходу. Но, конечно, у нее есть и свои недостатки, так как интерпретация работает медленнее, чем компиляция.

Один из ключевых преимуществ интерпретации — это возможность быстрой разработки программного обеспечения. Поскольку интерпретируемые языки обычно более высокого уровня, программистам требуется меньше времени для написания кода. Интерпретация также позволяет быстро проверять результаты, что может быть полезно во время разработки и отладки программ.

Однако, у интерпретации есть и некоторые недостатки. Интерпретация обычно медленнее, чем компиляция, поскольку на каждом шаге программы требуется «переводить» код на язык машины. Это может привести к потере производительности в некоторых случаях. Также интерпретируемый код часто требует наличия интерпретатора на целевой платформе, что может быть неудобно для некоторых пользователей.

Одним из основных случаев использования интерпретации является программирование на высоком уровне. В этом случае, интерпретатор выполняет код программы на основе его текстового представления. Это позволяет разработчикам писать код на более человеческом языке, который затем будет интерпретироваться и выполняться компьютером. [2]

Использование интерпретации в информатике также позволяет разработчикам быстро создавать прототипы приложений или систем. Благодаря этому процессу, разработчики могут быстро тестировать и проверять идеи, не создавая полноценные компилируемые версии программы.

Интерпретация играет важную роль в разработке программного обеспечения. Она помогает программистам работать более эффективно и удобно, предоставляя возможность выполнять код на разных языках программирования без необходимости компиляции. В результате, интерпретация ускоряет процесс разработки, позволяет быстро тестировать и отлаживать программы.

Кроме того, интерпретация позволяет создавать более гибкие программы, которые могут быть изменены в процессе выполнения. Это особенно важно для программного обеспечения с открытым исходным кодом, где интерпретатор изначально предоставляется пользователям для изменения и расширения функциональности программы.

**3 ОПИСАНИЕ И ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ**

Разработанный интерпретатор выполняет пошаговую интерпретацию исходных программ на языке Focal, опираясь на предварительно проведённые лексический, синтаксический и семантический анализы. На вход интерпретатору подаётся текстовая программа в формате, соответствующем синтаксису языка Focal, содержащая последовательность нумерованных операторов, разделённых по строкам. Каждый оператор проходит последовательную обработку: сначала распознаётся его тип, затем проверяются синтаксические и семантические ограничения, и, в случае успешной проверки, осуществляется выполнение соответствующего действия. Для вывода значений используется оператор TYPE, позволяющий отображать строки и значения переменных в заданной последовательности.

Рассмотрим примеры работы интерпретатора на двух тестовых входных программах.

Первый тест демонстрирует базовые операции с числами. Программа принимает значения переменных A и B, выполняет арифметические вычисления суммы, разности, произведения, деления и возведения в степень, а затем выводит результаты, которые отображены на рисунке 3.1.

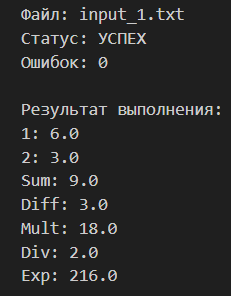


Рисунок 3.1 – Результат первого теста

Все вычисления были успешно выполнены, значения переменных корректно сохранены и выведены. Ошибки в ходе интерпретации отсутствуют.

Второй тест демонстрирует использование встроенных тригонометрических и математических функций. Значение переменной ANGLE устанавливается в 1.5708 (приблизительно π/2), затем вычисляются синус и косинус этого угла, логарифм от 10, а также квадратный корень из 25 (рисунок 3.2)

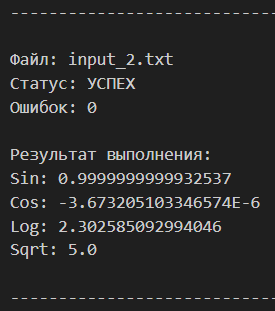


Рисунок 3.2 – Результат второго теста

Функции FSIN, FCOS, FLOG и FSQT отработали корректно, интерпретатор успешно интерпретировал каждое выражение и вывел точные числовые результаты. Погрешности связаны с особенностями работы с числами с плавающей точкой.

Оба теста подтверждают правильность работы интерпретатора, его способность обрабатывать базовые конструкции языка Focal и корректно исполнять программы, соответствующие формальному синтаксису.

Таким образом, проведённые тесты демонстрируют корректную и стабильную работу интерпретатора на языке Focal. Интерпретатор успешно справляется с обработкой числовых переменных, арифметических операций, встроенных математических функций и операторов ввода-вывода. Все конструкции языка интерпретируются согласно правилам, установленным на предыдущих этапах анализа, что подтверждается отсутствием ошибок и корректностью выводимых результатов. Полученные итоги подтверждают надёжность реализованного интерпретатора и его готовность к дальнейшему расширению функциональности или применению в рамках более сложных программ.

# ЗАКЛЮЧЕНИЕ

В результате выполнения данной лабораторной работы была реализована завершающая и наиболее практикоориентированная стадия процесса трансляции — интерпретация исходного кода. Этот этап позволяет превратить предварительно разобранный и проверенный текст программы в реальные вычисления, непосредственно исполняемые в среде исполнения. Основываясь на результатах лексического, синтаксического и семантического анализа, интерпретатор обеспечивает последовательное выполнение команд, заданных пользователем на языке Focal, демонстрируя тем самым полное прохождение всех стадий обработки программного кода — от анализа структуры до получения конкретного результата.

Работа над интерпретатором потребовала внимательного проектирования внутреннего представления программы, разработки механизмов выполнения операторов, управления памятью и обработки встроенных математических функций. Особое внимание было уделено реализации арифметических и логических выражений, поддержке базовых управляющих конструкций и обработке строк. Язык Focal, используемый в лабораторной работе, хорошо подходит для учебных целей, так как его простота сочетается с возможностью выразить основные идеи процедурного программирования. Это позволило сосредоточиться на ключевых аспектах интерпретации, избегая излишней сложности, связанной с синтаксическими исключениями или многоуровневыми контекстами.

В процессе выполнения работы была успешно подтверждена возможность реализации полноценного интерпретатора на языке Kotlin. Благодаря выразительности и типовой безопасности Kotlin, удалось создать стабильную и удобную архитектуру исполнения, обеспечивающую корректное выполнение программ, в том числе с использованием встроенных математических функций, операций над переменными и процедур вывода. Проведённые тесты показали, что интерпретатор точно и последовательно реализует логику исходных программ, обеспечивая при этом корректное поведение даже в случае более сложных выражений и вычислений.

Лабораторная работа стала важным шагом в понимании того, как исходный код языка высокого уровня может быть превращён в конкретное поведение на уровне исполняемой программы. Полученные знания и навыки формируют прочную основу для дальнейшего изучения интерпретируемых и компилируемых языков, разработки специализированных систем анализа и исполнения кода, а также понимания архитектуры современных языков программирования. Результаты работы не только закрепляют принципы трансляции, но и демонстрируют их практическую ценность при создании систем, способных выполнять реальный программный код.

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] Структура и интерпретация компьютерных программ. Абельсон Х. [Электронный ресурс]. – Режим доступа: https://rusneb.ru/catalog /000199\_000 009 \_004650731/.

ПРИЛОЖЕНИЕ А

(обязательное)

Исходный код

import java.io.File

import kotlin.math.\*

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"

    )

    val parseTreesFile = File("parser\_trees.txt")

    val semanticResultsFile = File("semantic\_results.txt")

    val testResultsFile = File("test\_results.txt")

    parseTreesFile.writeText("")

    semanticResultsFile.writeText("")

    testResultsFile.writeText("")

    for (fileName in inputFiles) {

        val file = File(fileName)

        if (!file.exists()) {

            println("Ошибка: Файл $fileName не найден.")

            testResultsFile.appendText("Файл: $fileName\nСтатус: ОШИБКА\nОшибка: Файл не найден\n\n")

            continue

        }

        val code = file.readLines()

        val tokens = tokenize(code)

        var hasEndError = false

        // Синтаксический анализ

        val parseTree = generateParseTree(tokens)

        parseTreesFile.appendText("Файл: $fileName\n$parseTree\n-------------------------------------------------\n\n")

        // Семантический анализ

        val symbolTable = SymbolTable()

        val semanticAnalyzer = SemanticAnalyzer(symbolTable)

        val semanticErrors = semanticAnalyzer.analyze(tokens)

        semanticResultsFile.appendText("Файл: $fileName\n")

        if (semanticErrors.isEmpty()) {

            semanticResultsFile.appendText("Семантических ошибок не найдено\n")

        } else {

            semanticErrors.forEach { error ->

                semanticResultsFile.appendText("Ошибка в строке ${error.lineNumber}: ${error.message}\n")

            }

        }

        semanticResultsFile.appendText("\n-------------------------------------------------\n\n")

        // Проверка END

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

            hasEndError = true

            println("\u001B[31m-------------------------------------------------")            println("ОШИБКА: Программа должна заканчиваться ключевым словом 'END'!")

            println("Файл: $fileName")

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

        }

        // Запись результатов теста

        val errorMessages = mutableListOf<String>().apply {

            if (hasEndError) add("Программа не заканчивается на END")

            addAll(semanticErrors.map { "Строка ${it.lineNumber}: ${it.message}" })

        }

        testResultsFile.appendText(

            """

            |Файл: $fileName

            |Статус: ${if (errorMessages.isEmpty()) "УСПЕХ" else "ОШИБКА"}

            |Ошибок: ${errorMessages.size}

            |${if (errorMessages.isNotEmpty()) "Список ошибок:\n" + errorMessages.joinToString("\n") { "- $it" } else ""}

        """.trimMargin()

        )

        // Интерпретация кода если нет ошибок

        if (errorMessages.isEmpty()) {

            val interpreter = Interpreter(tokens, symbolTable)

            val output = interpreter.interpret()

            testResultsFile.appendText("\nРезультат выполнения:\n$output")

        }

        testResultsFile.appendText("\n-------------------------------------------------\n\n")

        // Остальной вывод в консоль

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

        println("Обработка файла: $fileName")

        println("Классы лексем: ")

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

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

        for (token in tokens) {

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

        }

        for (type in lexemeOrder) {

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

            val lexemeList = if (lexemes.isEmpty()) "Нет токенов" else lexemes.joinToString(", ")

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

        }

        println()

        println("Таблица ")

        println("Номер элемента | Идентификатор | Информация")

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

        tokens.forEachIndexed { index, token ->

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

        }

        println("\n")

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

        for (token in tokens) {

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

        }

        for (type in lexemeOrder) {

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

            val lexemeList = if (lexemes.isEmpty()) "Нет токенов" else lexemes.joinToString(", ")

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

        }

        println("\n")

    }

}

data class VariableInfo(

    var type: Type = Type.UNKNOWN,

    var value: Double = 0.0,

    var initialized: Boolean = false,

    var lineNumber: String = "0.0",

    var isArray: Boolean = false,

    var arrayIndices: MutableSet<Int> = mutableSetOf(),

    var forLoopVariable: Boolean = false

)

enum class Type { NUMBER, INTEGER, STRING, UNKNOWN }

class Interpreter(private val tokens: List<Token>, private val symbolTable: SymbolTable) {

    private val output = StringBuilder()

    private var currentLineNumber = "0.0"

    private var currentIndex = 0

    private val variables = mutableMapOf<String, Double>()

    fun interpret(): String {

        // Сначала выполняем все присваивания и вычисления

        performAssignmentsAndCalculations()

        // Потом выводим результат

        currentIndex = 0

        while (currentIndex < tokens.size) {

            val token = tokens[currentIndex]

            when (token.type) {

                "LINE\_NUMBER" -> currentLineNumber = token.value

                "KEYWORD" -> when (token.value) {

                    "TYPE" -> handleType()

                    "END" -> break

                }

            }

            currentIndex++

        }

        return output.toString()

    }

    private fun parseExpression(tokens: List<Token>): Expression {

        var index = 0

        return parseAdditive(tokens, index).first

    }

    private fun parseAdditive(tokens: List<Token>, startIndex: Int): Pair<Expression, Int> {

        var (left, index) = parseMultiplicative(tokens, startIndex)

        while (index < tokens.size) {

            val token = tokens[index]

            if (token.type != "OPERATOR" || (token.value != "+" && token.value != "-")) break

            val operator = token.value

            val (right, nextIndex) = parseMultiplicative(tokens, index + 1)

            left = Expression.BinaryOperation(left, operator, right)

            index = nextIndex

        }

        return Pair(left, index)

    }

    private fun parseMultiplicative(tokens: List<Token>, startIndex: Int): Pair<Expression, Int> {

        var (left, index) = parsePrimary(tokens, startIndex)

        while (index < tokens.size) {

            val token = tokens[index]

            if (token.type != "OPERATOR" || (token.value != "\*" && token.value != "/" && token.value != "^")) break

            val operator = token.value

            val (right, nextIndex) = parsePrimary(tokens, index + 1)

            left = Expression.BinaryOperation(left, operator, right)

            index = nextIndex

        }

        return Pair(left, index)

    }

    private fun parsePrimary(tokens: List<Token>, startIndex: Int): Pair<Expression, Int> {

        if (startIndex >= tokens.size) return Pair(Expression.Error("Unexpected end"), startIndex)

        val token = tokens[startIndex]

        return when (token.type) {

            "NUMBER" -> Pair(Expression.Number(token.value.toDouble()), startIndex + 1)

            "IDENTIFIER" -> Pair(Expression.Variable(token.value), startIndex + 1)

            "SYMBOL" -> when (token.value) {

                "( (LEFT\_PAREN)" -> {

                    val (expr, nextIndex) = parseAdditive(tokens, startIndex + 1)

                    if (nextIndex < tokens.size && tokens[nextIndex].value == ") (RIGHT\_PAREN)") {

                        Pair(expr, nextIndex + 1)

                    } else {

                        Pair(Expression.Error("Missing closing parenthesis"), nextIndex)

                    }

                }

                else -> Pair(Expression.Error("Unexpected symbol: ${token.value}"), startIndex + 1)

            }

            "MATH\_FUNCTION" -> {

                if (startIndex + 1 >= tokens.size || tokens[startIndex + 1].value != "( (LEFT\_PAREN)") {

                    return Pair(Expression.Error("Invalid function syntax"), startIndex)

                }

                val (args, nextIndex) = parseFunctionArguments(tokens, startIndex + 2)

                if (nextIndex >= tokens.size || tokens[nextIndex].value != ") (RIGHT\_PAREN)") {

                    return Pair(Expression.Error("Missing closing parenthesis"), nextIndex)

                }

                Pair(Expression.FunctionCall(token.value, args), nextIndex + 1)

            }

            else -> Pair(Expression.Error("Unexpected token: ${token.type}"), startIndex + 1)

        }

    }

    private fun parseFunctionArguments(tokens: List<Token>, startIndex: Int): Pair<List<Expression>, Int> {

        val args = mutableListOf<Expression>()

        var index = startIndex

        while (index < tokens.size) {

            if (tokens[index].value == ") (RIGHT\_PAREN)") break

            val (arg, nextIndex) = parseAdditive(tokens, index)

            args.add(arg)

            index = nextIndex

            if (index < tokens.size && tokens[index].value == ", (COMMA)") index++

        }

        return Pair(args, index)

    }

    private fun handleLet() {

        val varName = tokens[currentIndex + 1].value

        val exprStart = currentIndex + 3

        val exprTokens = mutableListOf<Token>()

        var i = exprStart

        while (i < tokens.size && tokens[i].type != "LINE\_NUMBER" && tokens[i].value != "END") {

            exprTokens.add(tokens[i])

            i++

        }

        val expression = parseExpression(exprTokens)

        val value = evaluate(expression)

        variables[varName] = value

        symbolTable.getVariable(varName)?.let {

            it.value = value

            it.initialized = true

        } ?: run {

            symbolTable.addVariable(varName, Type.NUMBER, currentLineNumber)

            symbolTable.getVariable(varName)!!.value = value

        }

        currentIndex = i

    }

    private fun handleAssignment() {

        val varName = tokens[currentIndex].value

        currentIndex += 2 // Skip IDENTIFIER and =

        val exprTokens = mutableListOf<Token>()

        var i = currentIndex

        while (i < tokens.size && tokens[i].type != "LINE\_NUMBER" && tokens[i].value != "END") {

            exprTokens.add(tokens[i])

            i++

        }

        val expression = parseExpression(exprTokens)

        val value = evaluate(expression)

        variables[varName] = value

        symbolTable.getVariable(varName)?.let {

            it.value = value

            it.initialized = true

        } ?: run {

            symbolTable.addVariable(varName, Type.NUMBER, currentLineNumber)

            symbolTable.getVariable(varName)!!.value = value

        }

        currentIndex = i

    }

    private fun performAssignmentsAndCalculations() {

        currentIndex = 0

        while (currentIndex < tokens.size) {

            val token = tokens[currentIndex]

            when (token.type) {

                "LINE\_NUMBER" -> currentLineNumber = token.value

                "KEYWORD" -> when (token.value) {

                    "LET" -> handleLet()

                    "ACCEPT" -> handleAccept()

                }

                "IDENTIFIER" -> {

                    if (currentIndex + 1 < tokens.size &&

                        tokens[currentIndex + 1].type == "OPERATOR" &&

                        tokens[currentIndex + 1].value == "=") {

                        handleAssignment()

                    }

                }

            }

            currentIndex++

        }

    }

    private fun handleType() {

        val sb = StringBuilder()

        var i = currentIndex + 1

        while (i < tokens.size && tokens[i].type != "LINE\_NUMBER") {

            when (tokens[i].type) {

                "Literal Strings" -> sb.append(tokens[i].value.removeSurrounding("\""))

                "SYMBOL" -> if (tokens[i].value == ", (COMMA)") sb.append(" ")

                "IDENTIFIER" -> {

                    val varName = tokens[i].value

                    val variableValue = variables[varName] ?: symbolTable.getVariable(varName)?.value ?: 0.0

                    sb.append(variableValue.toString())

                }

            }

            i++

        }

        output.appendLine(sb.toString().trim())

        currentIndex = i - 1

    }

    private fun handleAccept() {

        val varName = tokens[currentIndex + 1].value

        currentIndex += 2

        if (currentIndex < tokens.size && tokens[currentIndex].type == "OPERATOR" && tokens[currentIndex].value == "=") {

            println("Ошибка: ACCEPT не должен содержать выражений. Строка: $currentLineNumber")

            return

        }

        var assignedValue: Double? = null

        if (currentIndex < tokens.size && tokens[currentIndex].type == "NUMBER") {

            assignedValue = tokens[currentIndex].value.toDouble()

            currentIndex++ // Переходим к следующему токену

        } else

неинициализированной

            assignedValue = 0.0 // Или можно оставить null, если хотите проверять на null

        }

        val value = assignedValue

        variables[varName] = value

        symbolTable.getVariable(varName)?.let {

            it.value = value

            it.initialized = true

        } ?: run {

            symbolTable.addVariable(varName, Type.NUMBER, currentLineNumber)

            symbolTable.getVariable(varName)!!.value = value

        }

    }

    private fun evaluate(expr: Expression): Double = when (expr) {

        is Expression.Number -> expr.value

        is Expression.Variable -> {

            // Сначала ищем в variables, потом в symbolTable

            variables[expr.name] ?: symbolTable.getVariable(expr.name)?.value ?: 0.0

        }

        is Expression.BinaryOperation -> when (expr.operator) {

            "+" -> evaluate(expr.left) + evaluate(expr.right)

            "-" -> evaluate(expr.left) - evaluate(expr.right)

            "\*" -> evaluate(expr.left) \* evaluate(expr.right)

            "/" -> {

                val rightVal = evaluate(expr.right)

                if (rightVal == 0.0) {

                    println("Деление на ноль!")

                    0.0

                } else {

                    evaluate(expr.left) / rightVal

                }

            }

            "^" -> Math.pow(evaluate(expr.left), evaluate(expr.right))

            else -> 0.0

        }

        is Expression.FunctionCall -> when (expr.name) {

            "FSIN" -> sin(evaluate(expr.arguments[0]))

            "FCOS" -> cos(evaluate(expr.arguments[0]))

            "FATN" -> atan(evaluate(expr.arguments[0]))

            "FLOG" -> ln(evaluate(expr.arguments[0]))

            "FSQT" -> sqrt(evaluate(expr.arguments[0]))

            "FABS" -> abs(evaluate(expr.arguments[0]))

            "FITR" -> evaluate(expr.arguments[0]).toInt().toDouble()

            else -> 0.0

        }

        else -> 0.0

    }

}

class SymbolTable {

    private val table = mutableMapOf<String, VariableInfo>()

    private val arrays = mutableMapOf<String, MutableMap<Int, Type>>()

    fun addVariable(name: String, type: Type, lineNumber: String, isArray: Boolean = false, forLoopVariable: Boolean = false) {

        table[name] = VariableInfo(

            type = type,

            lineNumber = lineNumber,

            isArray = isArray,

            forLoopVariable = forLoopVariable,

            value = 0.0,

            initialized = false

        )

    }

    fun getVariable(name: String): VariableInfo? = table[name]

    fun getAllVariables(): Map<String, VariableInfo> = table.toMap()

    fun addArrayElement(arrayName: String, index: Int, type: Type) {

        arrays.getOrPut(arrayName) { mutableMapOf() }[index] = type

        table[arrayName]?.arrayIndices?.add(index)

    }

    fun getArrayElementType(arrayName: String, index: Int): Type? {

        return arrays[arrayName]?.get(index)

    }

    fun setVariableInitialized(name: String) {

        table[name]?.initialized = true

    }

    fun isKeyword(name: String): Boolean {

        return name in setOf("LET", "END", "FOR", "NEXT", "TYPE", "ACCEPT", "THEN", "IF")

    }

}

data class SemanticError(val lineNumber: String, val message: String)

class SemanticAnalyzer(private val symbolTable: SymbolTable) {

    private var currentLineNumber = "0.0"

    private val errors = mutableListOf<SemanticError>()

    private var inForLoop = false

    private var loopVariable: String? = null

    fun analyze(tokens: List<Token>): List<SemanticError> {

        var i = 0

        while (i < tokens.size) {

            when (tokens[i].type) {

                "LINE\_NUMBER" -> currentLineNumber = tokens[i].value

                "KEYWORD" -> when (tokens[i].value) {

                    "ACCEPT" -> analyzeAccept(tokens, i)

                    "LET" -> analyzeLet(tokens, i)

                    "TYPE" -> analyzeType(tokens, i)

                    "FOR" -> analyzeFor(tokens, i)

                    "NEXT" -> analyzeNext(tokens, i)

                    "IF" -> analyzeIf(tokens, i)

                }

                "IDENTIFIER" -> analyzeIdentifier(tokens, i)

            }

            i++

        }

        checkUninitializedVariables()

        checkArrayElements()

        return errors

    }

    private fun analyzeIdentifier(tokens: List<Token>, index: Int) {

        val token = tokens[index]

        if (index + 1 < tokens.size && tokens[index + 1].type == "OPERATOR" && tokens[index + 1].value == "=") {

            if (symbolTable.getVariable(token.value) == null) {

                symbolTable.addVariable(token.value, Type.NUMBER, currentLineNumber)

            }

        }

    }

    private fun analyzeAccept(tokens: List<Token>, index: Int) {

        if (index + 1 >= tokens.size) {

            addError("Некорректный синтаксис ACCEPT: отсутствует идентификатор")

            return

        }

        val target = tokens[index + 1]

        when (target.type) {

            "IDENTIFIER" -> handleVariableDeclaration(target.value, currentLineNumber)

            "ARRAY" -> handleArrayDeclaration(target.value)

            else -> addError("Недопустимая цель для ACCEPT: ${target.value}")

        }

    }

    private fun handleVariableDeclaration(name: String, line: String) {

        if (symbolTable.getVariable(name) != null) {

            addError("Повторное объявление переменной '$name'")

        } else {

            symbolTable.addVariable(name, Type.NUMBER, line)

            symbolTable.setVariableInitialized(name) // Помечаем как инициализированную

        }

    }

    private fun handleArrayDeclaration(fullName: String) {

        val (name, index) = parseArrayAccess(fullName)

        if (index == null) {

            addError("Некорректный синтаксис массива: $fullName")

            return

        }

        if (symbolTable.getVariable(name) == null) {

            symbolTable.addVariable(name, Type.NUMBER, currentLineNumber, true)

            symbolTable.setVariableInitialized(name) // Помечаем массив как инициализированный

        }

        symbolTable.addArrayElement(name, index.toInt(), Type.NUMBER)

    }

    private fun analyzeLet(tokens: List<Token>, index: Int) {

        if (index + 3 >= tokens.size || tokens[index + 2].value != "=") {

            addError("Некорректный синтаксис LET: ожидается присваивание")

            return

        }

        val target = tokens[index + 1]

        when (target.type) {

            "IDENTIFIER", "ARRAY" -> {

                symbolTable.setVariableInitialized(target.value)

                parseExpression(tokens.subList(index + 3, tokens.size), tokens)

            }

            else -> addError("Недопустимая цель для присваивания")

        }

    }

    private fun analyzeVariableAssignment(name: String) {

        if (symbolTable.isKeyword(name)) {

            addError("Использование ключевого слова '$name' в качестве идентификатора")

        }

        if (symbolTable.getVariable(name) == null) {

            symbolTable.addVariable(name, Type.NUMBER, currentLineNumber)

        }

    }

    private fun analyzeArrayAssignment(fullName: String) {

        val (name, index) = parseArrayAccess(fullName)

        if (index == null) {

            addError("Некорректный синтаксис массива: $fullName")

            return

        }

        if (symbolTable.getVariable(name) == null) {

            addError("Массив '$name' не объявлен")

            return

        }

        if (index < -2048 || index > 2047) {

            addError("Индекс массива вне допустимого диапазона (-2048..2047)")

        }

    }

    private fun parseArrayAccess(fullName: String): Pair<String, Int?> {

        return try {

            val parts = fullName.split("(", ")")

            val name = parts[0]

            val index = parts[1].toDouble().toInt()

            Pair(name, index)

        } catch (e: Exception) {

            Pair(fullName, null)

        }

    }

    private fun analyzeType(tokens: List<Token>, index: Int) {

        var i = index + 1

        while (i < tokens.size && tokens[i].type != "LINE\_NUMBER") {

            when (tokens[i].type) {

                "IDENTIFIER" -> checkVariableExistence(tokens[i].value)

                "ARRAY" -> checkArrayExistence(tokens[i].value)

                "Literal Strings" -> {}

            }

            i++

        }

    }

    private fun checkVariableExistence(name: String) {

        if (symbolTable.getVariable(name) == null) {

            addError("Использование необъявленной переменной '$name'")

        }

    }

    private fun checkArrayExistence(fullName: String) {

        val (name, \_) = parseArrayAccess(fullName)

        if (symbolTable.getVariable(name) == null) {

            addError("Использование необъявленного массива '$name'")

        }

    }

    private fun analyzeFor(tokens: List<Token>, index: Int) {

        if (index + 5 >= tokens.size) {

            addError("Некорректный синтаксис FOR")

            return

        }

        val varName = tokens[index + 1].value

        if (varName.toDoubleOrNull() != null) {

            addError("Недопустимое имя переменной цикла: $varName")

        }

        val startValue = tokens[index + 3].value.toDoubleOrNull()

        val endValue = tokens[index + 5].value.toDoubleOrNull()

        when {

            startValue == null -> addError("Недопустимое начальное значение цикла")

            endValue == null -> addError("Недопустимое конечное значение цикла")

            startValue % 1 != 0.0 -> addError("Начальное значение цикла должно быть целым числом")

            endValue % 1 != 0.0 -> addError("Конечное значение цикла должно быть целым числом")

            else -> {

                symbolTable.addVariable(varName, Type.INTEGER, currentLineNumber, forLoopVariable = true)

                inForLoop = true

                loopVariable = varName

            }

        }

    }

    private fun analyzeNext(tokens: List<Token>, index: Int) {

        if (!inForLoop) {

            addError("NEXT без соответствующего FOR")

            return

        }

        val varName = tokens.getOrNull(index + 1)?.value

        when {

            varName == null -> addError("Отсутствует переменная цикла после NEXT")

            varName != loopVariable -> addError("Несоответствие переменной цикла: ожидалось $loopVariable, получено $varName")

            else -> {

                inForLoop = false

                loopVariable = null

            }

        }

    }

    private fun analyzeIf(tokens: List<Token>, index: Int) {

        var i = index + 1

        while (i < tokens.size && tokens[i].value != "THEN") {

            if (tokens[i].type == "Literal Strings") {

                addError("Строковые литералы недопустимы в условиях IF")

            }

            i++

        }

    }

    private fun checkUninitializedVariables() {

        symbolTable.getAllVariables().forEach { (name, info) ->

            if (!info.initialized && !info.isArray && !info.forLoopVariable) {

                addError("Переменная '$name' объявлена, но не инициализирована", info.lineNumber)

            }

        }

    }

    private fun checkArrayElements() {

        symbolTable.getAllVariables().filter { it.value.isArray }.forEach { (name, info) ->

            if (info.arrayIndices.isEmpty()) {

                addError("Массив '$name' объявлен, но не инициализирован", info.lineNumber)

            }

        }

    }

    private fun isCoercionAllowed(from: Type?, to: Type?): Boolean {

        return from == Type.INTEGER && to == Type.NUMBER

    }

    private fun addError(message: String) {

        errors.add(SemanticError(currentLineNumber, message))

    }

    private fun addError(message: String, lineNumber: String) {

        errors.add(SemanticError(lineNumber, message))

    }

    fun getVariable(name: String): VariableInfo? {

        return symbolTable.getVariable(name)

    }

}

sealed class Expression {

    abstract fun getType(symbolTable: SymbolTable, semanticAnalyzer: SemanticAnalyzer): Type

    data class Number(val value: Double) : Expression() {

        override fun getType(symbolTable: SymbolTable, semanticAnalyzer: SemanticAnalyzer): Type =

            if (value % 1 == 0.0) Type.INTEGER else Type.NUMBER

    }

    data class Variable(val name: String) : Expression() {

        override fun getType(symbolTable: SymbolTable, semanticAnalyzer: SemanticAnalyzer): Type {

            val variableInfo = (semanticAnalyzer as SemanticAnalyzer).getVariable(name)

            return variableInfo?.type ?: Type.UNKNOWN

        }

    }

    data class FunctionCall(val name: String, val arguments: List<Expression>) : Expression() {

        override fun getType(symbolTable: SymbolTable, semanticAnalyzer: SemanticAnalyzer): Type {

            return when (name) {

                "FSIN", "FCOS", "FATN", "FLOG", "FSQT", "FABS" -> Type.NUMBER

                "FITR", "FEXP", "FSGN" -> Type.INTEGER

                else -> Type.UNKNOWN

            }

        }

    }

    data class BinaryOperation(val left: Expression, val operator: String, val right: Expression) : Expression() {

        override fun getType(symbolTable: SymbolTable, semanticAnalyzer: SemanticAnalyzer): Type {

            val leftType = left.getType(symbolTable, semanticAnalyzer)

            val rightType = right.getType(symbolTable, semanticAnalyzer)

            if (leftType == Type.STRING || rightType == Type.STRING) return Type.UNKNOWN

            return if (leftType == Type.NUMBER || rightType == Type.NUMBER) Type.NUMBER else Type.INTEGER

        }

    }

    data class ArrayAccess(val arrayName: String, val index: Expression) : Expression() {

        override fun getType(symbolTable: SymbolTable, semanticAnalyzer: SemanticAnalyzer): Type {

            val arrayInfo = symbolTable.getVariable(arrayName)

            return arrayInfo?.type ?: Type.UNKNOWN

        }

    }

    data class Error(val message: String) : Expression() {

        override fun getType(symbolTable: SymbolTable, semanticAnalyzer: SemanticAnalyzer): Type = Type.UNKNOWN

    }

}

fun parseExpression(tokens: List<Token>, allTokens: List<Token>): Pair<Expression, Int> {

    if (tokens.isEmpty()) return Pair(Expression.Error("Пустое выражение"), 0)

    var i = 0

    var expression: Expression? = null

    while (i < tokens.size) {

        val token = tokens[i]

        expression = when (token.type) {

            "NUMBER" -> {

                Expression.Number(token.value.toDouble())

            }

            "IDENTIFIER" -> {

                Expression.Variable(token.value)

            }

            "MATH\_FUNCTION" -> {

                val functionName = token.value

                if (i + 1 >= tokens.size || tokens[i + 1].value != "(") {

                    return Pair(Expression.Error("Неверный синтаксис вызова функции"), i)

                }

                val (arguments, j) = parseFunctionArguments(tokens, i + 2, allTokens)

                i = j

                Expression.FunctionCall(functionName, arguments)

            }

            "OPERATOR" -> {

                if (expression == null) {

                    return Pair(Expression.Error("Отсутствует левый операнд"), i)

                }

                if (i + 1 >= tokens.size) {

                    return Pair(Expression.Error("Отсутствует правый операнд"), i)

                }

                val (right, rightEndIndex) = parseExpression(tokens.subList(i + 1, tokens.size), allTokens)

                if (right is Expression.Error) {

                    return Pair(right, i)

                }

                i += rightEndIndex

                Expression.BinaryOperation(expression, token.value, right)

            }

            "ARRAY" -> {

                val arrayName = token.value

                if (i + 1 >= tokens.size || tokens[i + 1].value != "(") {

                    return Pair(Expression.Error("Неверный синтаксис доступа к массиву"), i)

                }

                val (indexExpression, j) = parseArrayAccessIndex(tokens, i + 2, allTokens)

                i = j

                Expression.ArrayAccess(arrayName, indexExpression)

            }

            "SYMBOL" -> {

                when (token.value) {

                    "(" -> {

                        val (innerExpression, innerEndIndex) = parseInnerExpression(tokens, i + 1, allTokens)

                        if (innerExpression is Expression.Error) {

                            return Pair(innerExpression, i)

                        }

                        i += innerEndIndex

                        innerExpression

                    }

                    ")" -> {

                        return Pair(expression ?: Expression.Error("Неожиданная закрывающая скобка"), i)

                    }

                    "-" -> {

                        if (i + 1 < tokens.size) {

                            val (operand, operandEndIndex) = parseExpression(tokens.subList(i + 1, tokens.size), allTokens)

                            if (operand is Expression.Error) {

                                return Pair(operand, i)

                            }

                            i += operandEndIndex

                            Expression.BinaryOperation(Expression.Number(0.0), "-", operand)

                        } else {

                            return Pair(Expression.Error("Отсутствует операнд для унарного минуса"), i)

                        }

                    }

                    else -> return Pair(Expression.Error("Неожиданный символ: ${token.value}"), i)

                }

            }

            else -> {

                return Pair(Expression.Error("Неожиданный токен: ${token.value}"), i)

            }

        }

        i++

    }

    return Pair(expression ?: Expression.Error("Неверное выражение"), i - 1)

}

fun parseFunctionArguments(tokens: List<Token>, startIndex: Int, allTokens: List<Token>): Pair<List<Expression>, Int> {

    val arguments = mutableListOf<Expression>()

    var i = startIndex

    var parenCount = 1

    while (i < tokens.size && parenCount > 0) {

        if (tokens[i].value == "(") {

            parenCount++

        } else if (tokens[i].value == ")") {

            parenCount--

            if (parenCount == 0) {

                break

            }

        }

        val argumentTokens = mutableListOf<Token>()

        var k = i

        var innerParenCount = 0

        while (k < tokens.size && parenCount > 0) {

            if (tokens[k].value == "(") {

                innerParenCount++

            } else if (tokens[k].value == ")") {

                innerParenCount--

                if (innerParenCount < 0) {

                    break

                }

            }

            if (innerParenCount >= 0) {

                argumentTokens.add(tokens[k])

            }

            k++

        }

        val (argument, argumentEndIndex) = parseExpression(argumentTokens, allTokens)

        if (argument is Expression.Error) {

            return Pair(emptyList(), i)

        }

        arguments.add(argument)

        i += argumentEndIndex + 1

    }

    return Pair(arguments, i)

}

fun parseArrayAccessIndex(tokens: List<Token>, startIndex: Int, allTokens: List<Token>): Pair<Expression, Int> {

    var i = startIndex

    var parenCount = 1

    while (i < tokens.size && parenCount > 0) {

        if (tokens[i].value == "(") {

            parenCount++

        } else if (tokens[i].value == ")") {

            parenCount--

            if (parenCount == 0) {

                break

            }

        }

        i++

    }

    val indexTokens = tokens.subList(startIndex, i - 1)

    val (indexExpression, indexEndIndex) = parseExpression(indexTokens, allTokens)

    if (indexExpression is Expression.Error) {

        return Pair(indexExpression, i)

    }

    return Pair(indexExpression, i)

}

fun parseInnerExpression(tokens: List<Token>, startIndex: Int, allTokens: List<Token>): Pair<Expression, Int> {

    var i = startIndex

    var parenCount = 1

    while (i < tokens.size && parenCount > 0) {

        if (tokens[i].value == "(") {

            parenCount++

        } else if (tokens[i].value == ")") {

            parenCount--

            if (parenCount == 0) {

                break

            }

        }

        i++

    }

    val innerTokens = tokens.subList(startIndex, i - 1)

    val (innerExpression, innerEndIndex) = parseExpression(innerTokens, allTokens)

    if (innerExpression is Expression.Error) {

        return Pair(innerExpression, i)

    }

    return Pair(innerExpression, i)

}

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", "THEN", "IF")

    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")

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

    val functionRegex = Regex("([A-Z]+)\\(([^()]\*)\\)")

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

    var lastLineNumber: Double? = null

    for (line in codeLines) {

        var remainingLine = line.trim()

        if (remainingLine.isEmpty()) continue

        val lineNumberMatch = lineNumberRegex.find(remainingLine)

        if (lineNumberMatch != null) {

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

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

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

                println("ОШИБКА: Номера строк не упорядочены по возрастанию!")

                println("Ошибка в строке: $currentLineNumber")

                println("Предыдущий номер строки: $lastLineNumber")

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

            }

            lastLineNumber = currentLineNumber

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

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

        }

        val commentMatch = commentRegex.find(remainingLine)

        if (commentMatch != null) {

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

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

        }

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

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

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

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

            if (quoteCount % 2 != 0) {

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

                println("ОШИБКА: Строковые литералы после TYPE не закрыты двойными кавычками!")

                println("Ошибка в строке: ${lineNumberMatch?.value?.trim() ?: "unknown"}")

                println("Содержимое строки: $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)"))

                    }

                }

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

            }

        }

        val words = mutableListOf<String>()

        var insideString = false

        val stringBuffer = StringBuilder()

        for (char in remainingLine) {

            when {

                char == '"' -> {

                    if (insideString) {

                        words.add(stringBuffer.toString())

                        stringBuffer.clear()

                        insideString = false

                    } else {

                        if (stringBuffer.isNotEmpty()) {

                            words.add(stringBuffer.toString())

                            stringBuffer.clear()

                        }

                        insideString = true

                    }

                }

                insideString -> stringBuffer.append(char)

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

                    words.add(stringBuffer.toString())

                    stringBuffer.clear()

                }

                else -> stringBuffer.append(char)

            }

        }

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

        val processedWords = mutableListOf<String>()

        for (word in words) {

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

                processedWords.add(word.dropLast(1))

                processedWords.add(",")

            } else {

                processedWords.add(word)

            }

        }

        for (word in processedWords) {

            when {

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

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

                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) -> {

                    val functionMatch = functionRegex.find(word)

                    if (functionMatch != null) {

                        val functionName = functionMatch.groupValues[1]

                        val arguments = functionMatch.groupValues[2]

                        if (functionName in mathFunctions) {

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

                        } else {

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

                        }

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

                        val argumentTokens = tokenize(listOf(arguments))

                        tokens.addAll(argumentTokens)

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

                    }

                }

                arrayRegex.matches(word) -> {

                    val arrayMatch = arrayRegex.find(word)

                    if (arrayMatch != null) {

                        val arrayName = arrayMatch.groupValues[1]

                        val index = arrayMatch.groupValues[2]

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

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

                        val indexTokens = tokenize(listOf(index))

                        tokens.addAll(indexTokens)

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

                    }

                }

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

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

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

                        val content = word.substring(1)

                        if (content.isNotEmpty()) {

                            val contentTokens = tokenize(listOf(content))

                            tokens.addAll(contentTokens)

                        }

                    }

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

                        val content = word.dropLast(1)

                        if (content.isNotEmpty()) {

                            val contentTokens = tokenize(listOf(content))

                            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

}

fun generateParseTree(tokens: List<Token>): String {

    val parser = EnhancedParser(tokens)

    return try {

        parser.parse()

    } catch (e: Exception) {

        "Ошибка при разборе выражения: ${e.message}"

    }

}

class EnhancedParser(private val tokens: List<Token>) {

    private var index = 0

    private val tree = StringBuilder()

    private val indentStack = mutableListOf<String>()

    private val parentHasSibling = mutableListOf<Boolean>()

    fun parse(): String {

        while (index < tokens.size) {

            when (tokens[index].type) {

                "LINE\_NUMBER" -> parseLineNumber()

                "KEYWORD" -> parseKeyword()

                "COMMENT" -> parseComment()

                else -> index++

            }

        }

        return tree.toString()

    }

    private fun parseLineNumber() {

        addNode("LINE\_NUMBER: ${tokens[index].value}", isRoot = true)

        index++

    } private fun parseKeyword() {

        when (tokens[index].value) {

            "LET" -> parseAssignment()

            "TYPE" -> parseType()

            "ACCEPT" -> parseAccept()

            "END" -> parseEnd()

            "FOR" -> parseFor()

            "NEXT" -> parseNext()

            else -> index++

        }

    }private fun parseAssignment() {

        startNode("LET")

        index++

        parseVariable()

        if (match("OPERATOR", "=")) {

            addNode("OPERATOR: =")

            index++ // Увеличиваем индекс после обработки оператора =

            parseExpression()

        }

        endNode()

    }private fun parseVariable() {

        when {

            match("IDENTIFIER") -> parseIdentifier()

            match("ARRAY") -> parseArrayAccess()

            else -> {

                addNode("ERROR: Expected variable")

                index++

            }

        }

    }  private fun parseExpression() {

        startNode("Expression")

        parseTerm()

        while (match("OPERATOR", "+") || match("OPERATOR", "-")) {

            addOperatorNode()

            parseTerm()

        }

        endNode()

    }private fun parseTerm() {

        startNode("Term")

        parseFactor()

        while (match("OPERATOR", "\*") || match("OPERATOR", "/") || match("OPERATOR", "^")) {

            addOperatorNode()

            parseFactor()

        }

        endNode()

    }

    private fun parseFactor() {

        startNode("Factor")

        when {

            match("IDENTIFIER") -> parseIdentifier()

            match("NUMBER") -> parseNumber()

            match("SYMBOL", "(") -> parseParentheses()

            match("MATH\_FUNCTION") -> parseFunctionCall()

            tokens[index].type == "ARRAY" -> parseArrayAccess()

            else -> {

                index++

            }

        }

        endNode()

    } private fun parseFunctionCall() {

        startNode("FunctionCall: ${tokens[index].value}")

        index++

        if (match("SYMBOL", "(")) {

            addNode("SYMBOL: (")

            index++

            parseExpression() // Разбор аргументов

            if (match("SYMBOL", ")")) {

                addNode("SYMBOL: )")

                index++

            } else {

                addNode("ERROR: Expected ')'")

            }

        }

        endNode()

    }private fun parseArrayAccess() {

        startNode("ArrayAccess: ${tokens[index].value}")

        index++

        if (match("SYMBOL", "(")) {

            addNode("SYMBOL: (")

            index++

            parseExpression() // Разбор индекса

            if (match("SYMBOL", ")")) {

                addNode("SYMBOL: )")

                index++

            } else {

                addNode("ERROR: Expected ')'")

            }

        } else {

            addNode("ERROR: Expected '('")

        }

        endNode()

    } private fun parseFor() {

        startNode("FOR")

        index++

        parseIdentifier()

        if (match("OPERATOR", "=")) {

            addNode("OPERATOR: =")

            parseExpression()

            if (match("SYMBOL", ",")) {

                addNode("SYMBOL: ,")

                parseExpression()

            }

        }

        endNode()

    }private fun parseNext() {

        startNode("NEXT")

        index++

        parseIdentifier()

        endNode()

    } private fun parseIdentifier() {

        if (match("IDENTIFIER")) {

            addNode("IDENTIFIER: ${tokens[index].value}")

            index++

        }

    } private fun parseNumber() {

        if (match("NUMBER")) {

            addNode("NUMBER: ${tokens[index].value}")

            index++

        }

    }private fun parseType() {

        startNode("TYPE")

        index++

        while (index < tokens.size) {

            when {

                match("SYMBOL", "\"") -> {

                    addNode("SYMBOL: \"")

                    index++

                }

                match("Literal Strings") -> {

                    addNode("LITERAL: ${tokens[index].value}")

                    index++

                }

                match("IDENTIFIER") -> parseIdentifier()

                else -> break

            }

        }

        endNode()

    } private fun parseAccept() {

        startNode("ACCEPT")

        index++

        parseIdentifier()

        endNode()

    }private fun parseComment() {

        addNode("COMMENT: ${tokens[index].value}", isRoot = true)

        index++

    }private fun parseParentheses() {

        addNode("SYMBOL: (")

        index++

        parseExpression()

        if (match("SYMBOL", ")")) {

            addNode("SYMBOL: )")

            index++

        }

    }private fun parseEnd() {

        addNode("END", isRoot = true)

        index++

    }private fun startNode(label: String) {

        indentStack.add(label)

        parentHasSibling.add(false)

        tree.append("${currentIndent}+-- $label\n")

    } private fun endNode() {

        indentStack.removeLast()

        parentHasSibling.removeLastOrNull()

    }private fun addNode(text: String, isRoot: Boolean = false) {

        val indent = if (isRoot) "" else currentIndent

        tree.append("$indent+-- $text\n")

        if (!isRoot && indentStack.isNotEmpty()) {

            parentHasSibling[parentHasSibling.lastIndex] = true

        }

    }

    private val currentIndent: String

        get() = indentStack.indices.joinToString("") {

            if (parentHasSibling.getOrNull(it) == true) "|   " else "    "

        }

    private fun addOperatorNode() {

        if (index < tokens.size) {

            addNode("OPERATOR: ${tokens[index].value}")

            index++

        }

    }

    private fun match(type: String, value: String? = null): Boolean {

        return index < tokens.size && tokens[index].type == type &&

                (value == null || tokens[index].value == value)

    }

}