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 №4

на тему

**СЕМАНТИЧЕСКИЙ 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 6](#_Toc178067891)

Заключение8

[Список использованных источников 9](#_Toc178067900)

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

# ВВЕДЕНИЕ

В процессе разработки компиляторов ключевую роль играет стадия семантического анализа, которая непосредственно следует за синтаксическим разбором. Именно на этом этапе происходит проверка смысла программы: анализируются типы данных, переменные, идентификаторы, выражения и инструкции, чтобы убедиться, что они используются в допустимых для языка программирования контекстах.

Цель семантического анализа — не просто убедиться, что программа «грамматически» корректна, но и удостовериться в её логической непротиворечивости и соответствии правилам языка. Он выявляет ошибки, которые не могут быть обнаружены на предыдущих этапах, например, использование необъявленных переменных, несоответствие типов, неправильное количество аргументов в вызовах процедур и функций.

Важным аспектом работы является выбор подходящего метода семантического анализа и его реализация на языке Kotlin. Язык FOCAL, используемый в рамках данной лабораторной работы, отличается строгостью в отношении типов и требует аккуратной работы с семантическими правилами. Это позволяет наглядно продемонстрировать, как именно компилятор производит контроль типов и какие механизмы для этого задействуются.

Реализация семантического анализатора средствами языка Kotlin предоставляет возможность сочетать современный подход к программированию с лаконичностью и безопасностью типов, присущей платформе JVM. Такой выбор инструментов помогает не только глубже понять принципы построения компиляторов, но и получить практический опыт анализа и обработки языков программирования, приближенный к реальным условиям промышленной разработки.

1 ЦЕЛЬ РАБОТЫ

Целью данной лабораторной работы является разработка и исследование семантического анализатора для подмножества языка программирования Focal с использованием языка Kotlin. Семантический анализатор представляет собой критически важный компонент компилятора, предназначенный для выявления смысловых ошибок в исходной программе и формирования информации о типах, необходимой для последующих стадий компиляции, включая генерацию промежуточного или объектного кода. В отличие от лексического и синтаксического анализа, сосредоточенных на форме программы, семантический анализ работает с её содержанием — он проверяет корректность использования переменных, операций и выражений с точки зрения правил и ограничений языка Focal, таких как совместимость типов, область видимости идентификаторов, корректность вызовов функций и соблюдение соглашений о передаче параметров.

В рамках работы необходимо реализовать обработку информации, полученной от синтаксического анализатора, и построить структуру, способную распознавать и интерпретировать типы данных, проводить проверку допустимости операций, а также осуществлять автоматическое приведение типов, если это предусмотрено спецификацией языка. Особое внимание уделяется таким аспектам, как статическая типизация, контроль соответствия типов в выражениях и операциях, работа с неявными преобразованиями и выявление недопустимых действий, которые могли бы привести к ошибкам на этапе выполнения программы. Работа с языком Focal предоставляет возможность более строго формализовать семантические правила и исследовать особенности статического анализа программ.

Реализация семантического анализатора на языке Kotlin позволяет применить его современные средства, включая безопасную работу с типами, объектно-ориентированную структуру, расширенные возможности по обработке данных и построению абстрактных синтаксических деревьев. Это делает процесс разработки более наглядным и приближенным к практике создания реальных компиляторов и инструментов анализа программ. Выполнение лабораторной работы способствует формированию устойчивых навыков проектирования и реализации компонентов компилятора, пониманию роли и сложности семантического анализа, а также развитию способности выявлять и предотвращать логические ошибки в программном коде на ранних этапах его обработки. Полученные знания и опыт могут быть применены при разработке статических анализаторов, интерпретаторов, трансляторов, а также при решении задач, связанных с обеспечением надежности и безопасности программного обеспечения.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Семантический анализ — это важный этап в процессе компиляции программного кода, который проверяет смысловую корректность программы. В отличие от синтаксического анализа, который проверяет правильность структуры кода, семантический анализ фокусируется на логике и значениях, чтобы убедиться, что программа делает то, что задумано разработчиком. Это позволяет избежать множества ошибок, которые могут возникнуть при выполнении программы, и делает код более надежным и предсказуемым. [1]

Семантический анализ включает в себя множество проверок, таких как соответствие типов данных, корректность использования идентификаторов, проверка области видимости и корректность выполняемых операций. Эти проверки помогают разработчику убедиться, что программа работает так, как задумано, и позволяет избежать множества ошибок, которые могут возникнуть при выполнении программы.

Ппосле анализа синтаксиса программы следующим шагом является анализ семантики программы. Это включает в себя использование набора правил, чтобы определить, является ли программа семантически правильной, и создать промежуточное представление программы. Семантический анализ — это процесс использования набора правил для определения правильности программы с точки зрения семантики и создания промежуточного представления программы.

Цель семантического анализа двояка: во-первых, он проверяет, что все операции допустимы в соответствии с их определениями; во-вторых, он создает внутреннее представление, которое может использоваться другими частями компилятора или интерпретатора. Чтобы семантический анализ работал правильно, каждая операция должна иметь собственное определение в терминах того, какие входные данные она получает и какие выходные данные производит.

Результатом этого этапа должно быть абстрактное синтаксическое дерево (AST), содержащее информацию о том, как каждая операция работает в своем контексте. После создания этого AST может выполняться дальнейшая обработка, например проверка типов или оптимизация. Семантический анализ помогает убедиться, что программы синтаксически корректны до их выполнения, что позволяет обнаруживать ошибки на ранних этапах разработки. [2]

Семантический анализ выполняет несколько ключевых задач, каждая из которых направлена на обеспечение корректности и надежности кода

Одна из основных задач семантического анализа — проверка соответствия типов данных. Например, если вы пытаетесь сложить число и строку, компилятор должен выдать ошибку. Это предотвращает множество ошибок, связанных с некорректным использованием типов данных. Например, в языке Python попытка сложить число и строку приведет к ошибке времени выполнения, тогда как компилятор языка C++ может выявить такую ошибку на этапе компиляции.

Семантический анализ также проверяет, что все переменные и функции объявлены перед использованием. Это помогает избежать ошибок, связанных с неопределенными идентификаторами. Например, если вы попытаетесь использовать переменную, которая не была объявлена, компилятор выдаст ошибку. Это позволяет избежать множества ошибок, связанных с неправильным использованием идентификаторов.

Компилятор проверяет, что переменные и функции используются в пределах своей области видимости. Это предотвращает ошибки, связанные с доступом к переменным из других блоков кода. Например, если вы попытаетесь использовать переменную, объявленную внутри функции, за пределами этой функции, компилятор выдаст ошибку. Это помогает избежать множества ошибок, связанных с неправильным использованием областей видимости.

Семантический анализ также проверяет, что операции выполняются над корректными типами данных. Например, деление на ноль или использование неинициализированных переменных. Это позволяет избежать множества ошибок, связанных с некорректными операциями. Например, в языке Python попытка деления на ноль приведет к ошибке времени выполнения, тогда как компилятор языка C++ может выявить такую ошибку на этапе компиляции.

Семантический анализатор использует синтаксическое дерево и информацию из таблицы идентификаторов для проверки входной программы на семантическую согласованность с определением языка программирования. Он также собирает информацию о типах и сохраняет её в синтаксическом дереве или в таблице идентификаторов для последующего использования в процессе генерации промежуточного кода. [3]

После синтаксического и семантического анализа исходной программы компиляторы генерируют низкоуровневое промежуточное представление входной программы, которое можно рассматривать как программу для абстрактной вычислительной машины. Такое промежуточное представление должно обладать двумя важными свойствами: оно должно легко генерироваться и легко транслироваться в целевой машинный язык.

Семантика в программировании — это фундаментальное понятие, которое помогает разработчикам понимать и предсказывать поведение программного кода. Понимание семантики позволяет писать более надежные и предсказуемые программы. Для дальнейшего изучения рекомендуется ознакомиться с книгами и статьями по теории языков программирования, а также практиковаться в написании кода на различных языках программирования.

Изучение семантики также помогает лучше понять, как работают компиляторы и интерпретаторы. Это знание может быть полезным при оптимизации кода и создании более эффективных программ. Кроме того, понимание семантики помогает лучше понимать документацию и примеры кода, что делает процесс обучения новым языкам программирования более быстрым и эффективным.

**3 ОПИСАНИЕ И ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ**

Разработанная программа представляет собой семантический анализатор для подмножества языка программирования Focal, реализованный на языке Kotlin. Её основная задача — выполнять глубокий анализ исходного кода программ, выявляя семантические ошибки, такие как использование необъявленных переменных, некорректные типы данных, нарушения правил работы с массивами, недопустимые идентификаторы, а также логические ошибки в структуре программы.

Семантический анализатор работает с предварительно разобранной последовательностью токенов, полученной в результате лексического и синтаксического анализа, и осуществляет проверку корректности семантики каждой инструкции. В частности, анализируются объявления переменных, операции присваивания, вызовы встроенных функций, использование массивов и управляющих конструкций. При обнаружении ошибок формируются понятные сообщения, содержащие номер строки и описание нарушения, что облегчает отладку и улучшает читаемость обратной связи.

В ходе выполнения лабораторной работы программа была протестирована на трех входных файлах: input\_1.txt, input\_2.txt и input\_3.txt. Ниже представлены выдержки из содержимого этих файлов, результаты семантической проверки.

Первый тестовый файл содержит множество намеренно допущенных семантических ошибок: использование неинициализированных и необъявленных переменных, обращение к массиву без индекса и другие. Программа успешно зафиксировала все нарушения, предоставив подробный список ошибок с указанием строк и описанием. Он изображён на рисунке 3.1.

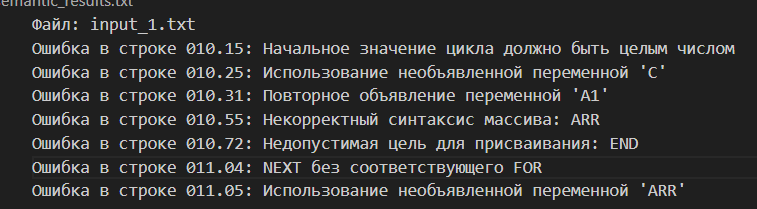


Рисунок 3.1 – Результат первого теста

В языке Focal переменная цикла (K) должна принимать только целочисленные значения. Указание 1.5 нарушает это требование, поскольку оно является вещественным числом.

Использование необъявленной переменной. C. Переменная C не была предварительно объявлена (через ACCEPT или LET) и не входит в список встроенных.

Повторное объявление переменной A1. Переменная A1 уже была объявлена в строке 010.30. Повторное её объявление через ACCEPT нарушает правило уникальности имён переменных.

Использование необъявленного массива ARR.До этого момента массив ARR не был объявлен. Семантический анализ требует, чтобы массивы, как и переменные, были либо явно объявлены, либо проинициализированы до использования.

Использование ключевого слова END как идентификатора. END является зарезервированным словом в языке Focal, обозначающим конец программы. Использование его в качестве имени переменной нарушает правила языка.

Корректная программа — input\_2.txt. В этом файле содержится корректная Focal-программа, выполняющая тригонометрические и логарифмические вычисления, использование встроенных функций и операций с числовыми значениями. Семантический анализ показал, что ошибок не обнаружено (рисунок 3.2).



Рисунок 3.2 – Результат второго теста

Ошибки в работе с массивами — input\_3.txt. Данный тестовый файл содержит программу с использованием массивов и циклов. Семантический анализ выявил ошибки, связанные с некорректным синтаксисом присваивания в массив и использованием необъявленного массива. Эти нарушения были корректно зафиксированы анализатором и отражены на рисунке 3.3.

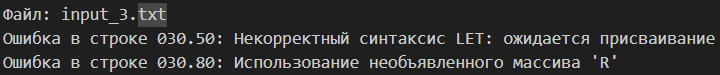


Рисунок 3.3 – Результат третьего теста

В данной строке между R и (I) стоит пробел, из-за чего синтаксический разбор нарушается, и выражение трактуется как попытка присвоения переменной с именем R, за которой неожиданно следует открывающая скобка — это вызывает ошибку парсера и семантического анализа.

Таким образом, разработанный семантический анализатор успешно справляется с задачей обнаружения смысловых ошибок в программах на языке Focal. Он демонстрирует устойчивую работу на различных входных данных и предоставляет подробную диагностическую информацию, необходимую для последующей корректировки программного кода.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы №4 была реализована важнейшая стадия компиляции — семантический анализ, задача которого заключается в выявлении логических и смысловых ошибок в программном коде, не охватываемых лексическим и синтаксическим анализом. Работа была сосредоточена на анализе программ, написанных на языке Focal — учебном языке, отличающемся строгими правилами объявления переменных, индексирования массивов и использования ключевых слов. Реализация семантического анализатора на языке Kotlin позволила глубоко погрузиться в процесс автоматической проверки корректности программ, выявления неинициализированных переменных, недопустимых операций между типами, нарушения правил приведения типов, ошибок в индексировании массивов и других типичных логических нарушений.

Проверка программ на этом этапе осуществляется не только на основе синтаксической структуры, но и с учетом хранимой информации о типах, идентификаторах и операциях, что требует использования таблиц символов, контекстного отслеживания и умения распознавать зависимости между выражениями и инструкциями. Лабораторная работа также показала, насколько важно точно интерпретировать спецификации языка: недопустимые преобразования типов, повторные объявления и использование ключевых слов в качестве идентификаторов могут не только привести к ошибкам, но и нарушить логику всей программы.

Особую роль в процессе работы сыграло тестирование на реальных фрагментах кода. Сравнение программ с ошибками и без них продемонстрировало надежность реализованного семантического анализатора и подчеркнуло его способность локализовать проблемы, давая точные и понятные диагностические сообщения. Это особенно важно при анализе программ, в которых ошибки могут быть неочевидны на этапе компиляции без глубокого контекстного разбора.

Завершение данной лабораторной работы позволило не только расширить теоретические знания в области построения компиляторов, но и развить практические навыки в разработке и отладке сложных программных компонентов, связанных с анализом структуры и смысла исходного кода. Полученный опыт будет полезен при создании более сложных компиляторов и статических анализаторов кода, а также при понимании глубинных механизмов работы современных языков программирования.

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] Компиляторы: принципы, методы и инструменты. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульманрёв [Электронный ресурс]. – Режим доступа: https://vsuhom.ru/pdf/ru/book/www.vsuhom.ru\_744\_abstrakt-Ko mpilyatory\_princip.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"

    )

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

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

    parseTreesFile.writeText("")

    semanticResultsFile.writeText("")

    for (fileName in inputFiles) {

        val file = File(fileName)

        if (!file.exists()) {

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

            continue

        }

        val code = file.readLines()

        val tokens = tokenize(code)

        // 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")

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

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

            println("ОШИБКА: Программа должна заканчиваться ключевым словом 'END'!")

            println("Файл: $fileName")

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

        }

        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 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 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, true, lineNumber, isArray, mutableSetOf(), forLoopVariable)

    }

    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"  // Хранить как String

    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) {  // Используем type вместо value

                "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)

                }

            }

            i++

        }

        checkUninitializedVariables()

        checkArrayElements()

        return errors

    }

    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)

        }

    }

    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.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" -> analyzeVariableAssignment(target.value)

            "ARRAY" -> analyzeArrayAssignment(target.value)

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

        }

        val exprType = parseExpression(tokens.subList(index + 3, tokens.size), tokens).first.getType(symbolTable, this)

        val targetType = when (target.type) {

            "IDENTIFIER" -> symbolTable.getVariable(target.value)?.type

            "ARRAY" -> Type.NUMBER

            else -> null

        }

        if (exprType != targetType && !isCoercionAllowed(exprType, targetType)) {

        }

    }

    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)

    }

}

// Abstract syntax tree

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)  // Propagate error

                }

                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("!.\*")

    // Улучшенные регулярные выражения

    val functionRegex = Regex("([A-Z]+)\\(([^()]\*)\\)") // Более точное соответствие аргументам

    // This regex MUST match the entire array access, including the parentheses and index

    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() // Изменено для поддержки массивов

        if (match("OPERATOR", "=")) {

            addNode("OPERATOR: =")

            parseExpression()

        }

        endNode()

    }

    private fun parseVariable() {

        when {

            match("IDENTIFIER") -> parseIdentifier()

            match("ARRAY") -> parseArrayAccess()

            else -> {

                addNode("ERROR: Expected variable")

                index++

            }

        }

    }

    private fun parseExpression(): Unit {

        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() // Only call when it's an ARRAY token

            else -> index++

        }

        endNode()

    }

    private fun parseFunctionCall() {

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

        index++

        if (match("SYMBOL", "(")) {

            addNode("SYMBOL: (")

            index++

            parseExpression() // Разбор аргументов (упрощенно, без разделения запятыми)

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

                addNode("SYMBOL: )")

                index++

            }

        }

        endNode()

    }

       private fun parseArrayAccess() {

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

        index++ // Пропускаем имя массива

        if (match("SYMBOL", "(")) {

            addNode("SYMBOL: (")

            index++

            parseExpression()

            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)

    }

}