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 №3
на тему

СИНТАКСИЧЕСКИЙ АНАЛИЗ

Выполнил: студент гр.253504 Лавренова А.С.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данной лабораторной работе рассматривается процесс синтаксического анализа, являющегося ключевым этапом компиляции программ. В этой работе будет рассматриваться разработка синтаксического анализатора, способного проверить исходный текст программы на соответствие синтаксическим правилам заданного языка и построить дерево разбора (синтаксическое дерево). Это дерево представляет собой иерархическую структуру, отражающую синтаксическую организацию входной последовательности токенов, что делает его удобным для дальнейшей обработки, например, при генерации кода или выполнении семантического анализа. В случае обнаружения синтаксических ошибок, анализатор должен предоставить информативные сообщения, облегчающие процесс отладки.

В основе синтаксического анализа лежит группировка токенов, полученных от лексического анализатора, в грамматические фразы, соответствующие правилам языка. Результатом анализа является представление синтаксической структуры программы в виде дерева, где узлы соответствуют грамматическим конструкциям, а ветви – отношениям между ними. В данной работе предлагается реализовать синтаксический анализатор, используя один из табличных методов, таких как LL-анализ, LR-анализ или метод предшествования.

В данной работе используется язык Kotlin для реализации и Focal для тестирования.

Использование LL- или LR-методов позволяет обнаруживать синтаксические ошибки на ранних этапах анализа, что способствует более быстрой идентификации и исправлению проблем в исходном коде. Анализатор может быть реализован как нисходящий (top-down), начинающий разбор с начального символа грамматики, или как восходящий (bottom-up), восстанавливающий продукцию из правых частей, начиная с токенов. В первом случае можно использовать метод рекурсивного спуска или LL-анализатор, а во втором – LR-анализатор.

1 ЦЕЛЬ РАБОТЫ

Целью данной лабораторной работы является разработка и исследование синтаксического анализатора для подмножества языка программирования Focal, определенного в предыдущих работах, с использованием языка Kotlin. Синтаксический анализатор, или парсер, является неотъемлемой частью компиляторов и интерпретаторов, следующим звеном после лексического анализатора. Он отвечает за проверку соответствия последовательности токенов, полученных от лексического анализатора, синтаксическим правилам языка и построение синтаксического дерева, отражающего структуру программы.

В ходе выполнения работы необходимо глубоко изучить принципы синтаксического анализа, включая различные методы, такие как LL- и LR-анализ, а также метод предшествования. Необходимо формализовать грамматику подмножества Focal, определив правила, которым должны соответствовать допустимые конструкции языка. На основе этой грамматики предстоит разработать алгоритм синтаксического анализа, способный построить синтаксическое дерево для входной последовательности токенов.

Важным аспектом работы является выбор подходящего метода синтаксического анализа и его реализация на языке Kotlin. Необходимо рассмотреть преимущества и недостатки различных методов, таких как LL-анализ (нисходящий анализ) и LR-анализ (восходящий анализ), и выбрать наиболее подходящий для заданного подмножества Focal. Реализация должна быть эффективной и обеспечивать корректное построение синтаксического дерева для допустимых программ.

Разработка синтаксического анализатора на Kotlin позволит использовать современные средства разработки, такие как мощные библиотеки для работы с данными и алгоритмами, что облегчит реализацию алгоритма синтаксического анализа и построение синтаксического дерева.

Выполнение данной лабораторной работы позволит получить практический опыт разработки синтаксических анализаторов, углубить знания в области компиляторостроения и языковых технологий, а также развить навыки работы с формальными грамматиками и алгоритмами синтаксического анализа. Приобретенные знания и навыки будут полезны при разработке компиляторов, интерпретаторов, статических анализаторов кода и других инструментов, связанных с обработкой языков программирования.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

В иерархии грамматик Хомского выделено 4 основных группы языков (и описывающих их грамматик). При этом наибольший интерес представляют регулярные и контекстно-свободные (КС) грамматики и языки. Они используются при описании синтаксиса языков программирования. С помощью регулярных грамматик можно описать лексемы языка – идентификаторы, константы, служебные слова и прочие. На основе КС-грамматик строятся более крупные синтаксические конструкции: описания типов и переменных, арифметические и логические выражения, управляющие операторы, и, наконец, полностью вся программа на входном языке. Входные цепочки регулярных языков распознаются с помощью конечных автоматов (КА). Они лежат в основе сканеров, выполняющих лексический анализ и выделение слов в тексте программы на входном языке. Результатом работы сканера является преобразование исходной программы в список или таблицу лексем. Дальнейшую её обработку выполняет другая часть компилятора – синтаксический анализатор. Его работа основана на использовании правил КС грамматики, описывающих конструкции исходного языка. [1]

Синтаксический анализ — это второй этап процесса разработки компилятора, на котором данная входная строка проверяется на предмет подтверждения правил и структуры формальной грамматики. Он анализирует синтаксическую структуру и проверяет, соответствует ли данный ввод правильному синтаксису языка программирования или нет.

Синтаксический анализ в процессе проектирования компилятора происходит после этапа лексического анализа. Оно также известно как дерево разбора или синтаксическое дерево. Дерево разбора разрабатывается с помощью заранее определенной грамматики языка. Синтаксический анализатор также проверяет, соответствует ли данная программа правилам, подразумеваемым контекстно-свободной грамматикой. Если это удовлетворяет, синтаксический анализатор создает дерево разбора этой исходной программы. В противном случае будут отображаться сообщения об ошибках. [2]

Синтаксический анализ — это важнейший процесс, который позволяет компьютерам анализировать необработанные данные. Разбивая данные на более мелкие структурированные компоненты, синтаксический анализ облегчает точную интерпретацию, понимание и осмысленные действия. Он находит применение, среди прочего, в языках программирования, обработке естественного языка, анализе данных и веб-скрапинге. Освоение тонкостей синтаксического анализа открывает мир возможностей для эффективной организации и использования данных.

3 ОПИСАНИЕ И ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ

Данная лабораторная работа посвящена разработке синтаксического анализатора для подмножества языка Focal. Программа, разработанная в рамках этой работы, выполняет лексический и синтаксический анализ исходного кода Focal, представленного в текстовых файлах.

Функциональность программы:

1. Программа принимает на вход список текстовых файлов, содержащих исходный код на языке Focal. Каждый файл обрабатывается отдельно.
2. Исходный код каждого файла подвергается лексическому анализу, в результате которого формируется последовательность токенов. Каждый токен представляет собой осмысленную единицу языка, такую как ключевое слово, идентификатор, число, оператор или символ.
3. Последовательность токенов, полученная на этапе лексического анализа, передается на вход синтаксическому анализатору. Синтаксический анализатор проверяет соответствие последовательности токенов синтаксическим правилам языка Focal. В случае успешной проверки строится синтаксическое дерево, отражающее структуру программы.
4. Программа выводит информацию о процессе анализа для каждого файла. Эта информация включает в себя список лексем с указанием их типов и значений, таблицу символов, содержащую информацию об идентификаторах, используемых в программе и синтаксическое дерево, отображающее структуру программы в иерархической форме.
5. В случае обнаружения лексических, программа выводит сообщения об ошибках, указывающие на место и причину ошибки.

Для демонстрации работы программы были подготовлены три тестовых файла: input_1.txt, input_2.txt и input_3.txt. Каждый файл содержит пример кода на языке Focal, демонстрирующий различные возможности языка, такие как арифметические операции, тригонометрические функции, массивы и циклы.

При запуске программы с этими файлами, она выполняет лексический и синтаксический анализ каждого файла и выводит результаты в консоль и в файл parser_trees.txt.

На рисунке 3.1 представлен пример вывода программы.

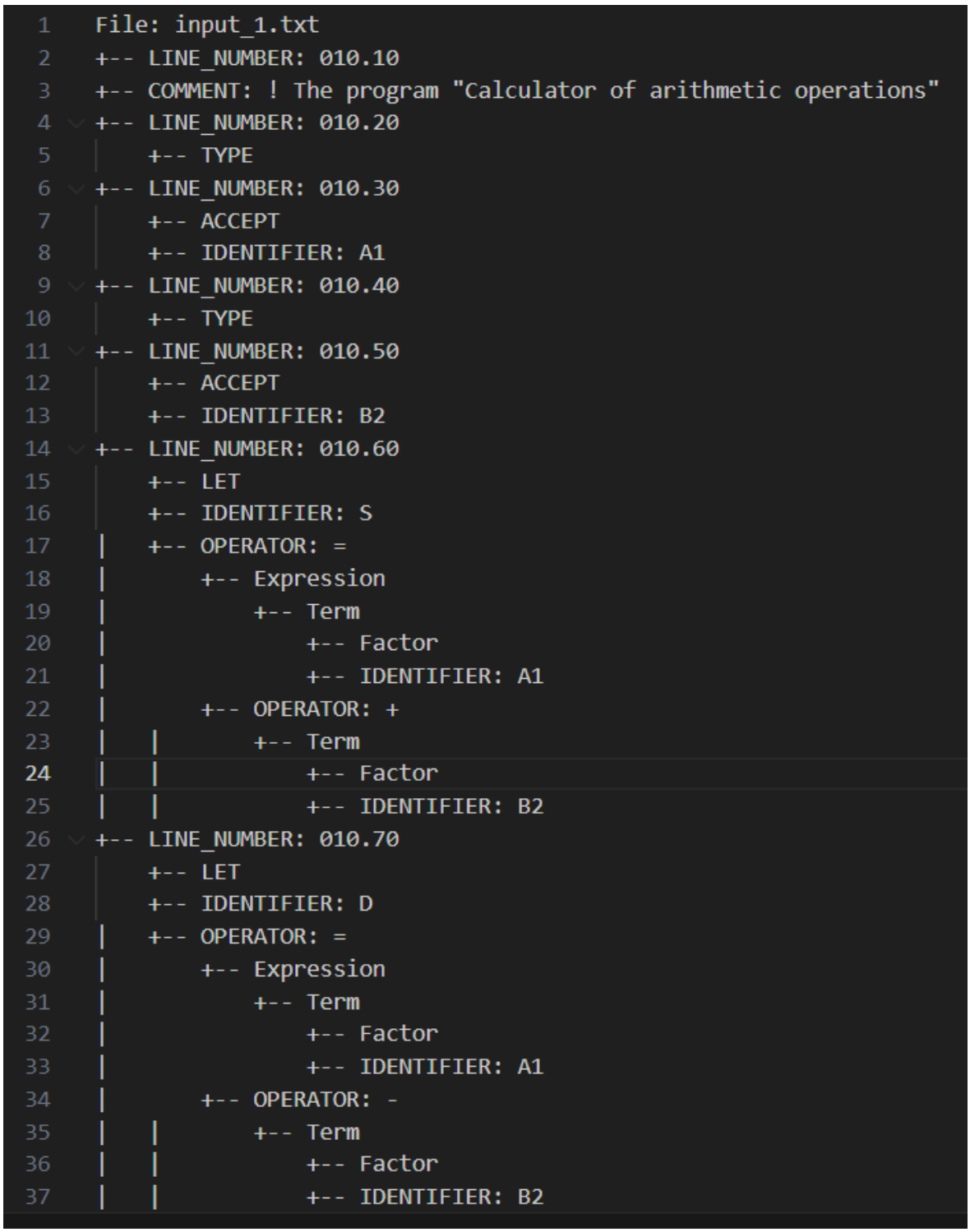

Рисунок 3.1 – Результат работы программного продукта

Разработанная программа успешно выполняет лексический и синтаксический анализ исходного кода на языке Focal, строит синтаксическое дерево и выводит информацию о процессе анализа. Программа может быть использована для дальнейшей разработки компилятора или интерпретатора языка Focal.

ЗАКЛЮЧЕНИЕ

В заключение данной лабораторной работы был разработан и исследован синтаксический анализатор для подмножества языка Focal, реализованный на языке программирования Kotlin. Основной целью являлось создание инструмента, способного не только проверять соответствие исходного кода синтаксическим правилам языка, но и строить синтаксическое дерево, отражающее структуру программы.

В процессе разработки были изучены и применены различные методы синтаксического анализа, такие как LL- и LR-анализ. Была формализована грамматика подмножества Focal, определены правила, которым должны соответствовать допустимые конструкции языка. На основе этой грамматики был разработан алгоритм синтаксического анализа, способный построить синтаксическое дерево для входной последовательности токенов, полученных от лексического анализатора.

Программа была протестирована на трех различных примерах кода на языке Focal, демонстрирующих различные возможности языка, такие как арифметические операции, тригонометрические функции, массивы и циклы. Результаты тестирования показали, что разработанный анализатор успешно справляется с задачей синтаксического анализа и построения синтаксических деревьев для допустимых программ.

Разработка синтаксического анализатора на Kotlin позволила использовать современные средства разработки, такие как мощные библиотеки для работы с данными и алгоритмами, что облегчило реализацию алгоритма синтаксического анализа и построение синтаксического дерева. Применение языка Kotlin также способствовало созданию эффективного и читаемого кода.

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.

[3] Кнспектное изложение теории языков программирования и методов трансляции. Вл. Понаморёв [Электронный ресурс]. – Режим доступа: https://timeweb.cloud/tutorials/linux/regulyarnye—vyrazheniya—bash—gajd?ysclid=m82zxlerds129168253.

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")
    parseTreesFile.writeText("")

    for (fileName in inputFiles) {
        val file = File(fileName)
        if (!file.exists()) {
            println("Error: File $fileName not found.")
            continue
        }

        val code = file.readLines()
        val tokens = tokenize(code)

        if (code.lastOrNull()?.trim()?.endsWith("END") != true) {
            println("\u001B[31m----------------------------------------
----------")
            println("ERROR: The program must end with the keyword
'END'!")
            println("File: $fileName")
            println("------------------------------------------------
\u001B[0m")
        }

        println("--------------------------------------------------")
        println("Processing file: $fileName")
        println("Lexeme classes: ")
        println("--------------------------------------------------")

        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()) "No tokens" else
lexemes.joinToString(", ")
            println("$type (${lexemes.size}): $lexemeList")
        }
```

```
        println()

        println("Table ")
        println("Item Number | Identifier | Information")
        println("-----------------------------------------------")

        tokens.forEachIndexed { index, token ->
            println("${index + 1} | ${token.value} | ${token.type}")
        }
        println("\n")

        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()) "No tokens" else
lexemes.joinToString(", ")
            println("$type (${lexemes.size}): $lexemeList")
        }
        println("\n")

        val parseTree = generateParseTree(tokens)
        parseTreesFile.appendText("File: $fileName\n")
        parseTreesFile.appendText(parseTree)
        parseTreesFile.appendText("\n--------------------------------
----------------\n\n")
    }
}

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

fun tokenize(codeLines: List<String>): List<Token> {
    val tokens = mutableListOf<Token>()
    val operators = setOf("=", "+", "-", "*", "/", "^", ":")
    val keywords = setOf("LET", "END", "FOR", "NEXT", "TYPE",
"ACCEPT")
    val symbols = mapOf(
        "(" to "LEFT_PAREN", ")" to "RIGHT_PAREN", "{" to
"LEFT_BRACE", "}" to "RIGHT_BRACE",
        ";" to "SEMICOLON", "\"" to "QUOTE", "," to "COMMA"
    )
    val mathFunctions = setOf("FSIN", "FCOS", "FATN", "FLOG", "FSQT",
"FABS", "FITR", "ABSVAL", "FEXP", "FSGN", "EXPVAL", "FRAN")

    val numberRegex = Regex("[-]?\\d+(\\.\\d+)?")
    val lineNumberRegex = Regex("^(\\d+\\.\\d+|\\d+)\\s")
    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("ERROR: Line numbers are not in ascending
order!")
                println("Error at line: $currentLineNumber")
                println("Previous line number: $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("ERROR: String literals after TYPE are not
properly closed with double quotes!")
                println("Error at line:
${lineNumberMatch?.value?.trim() ?: "unknown"}")
                println("Line content: $remainingLine")
                println("-----------------------------------------
----\u001B[0m")
            } else {
                val parts = remainingLine.split("\"")
                for (i in parts.indices step 2) {
                    if (i + 1 < parts.size) {
                        tokens.add(Token("SYMBOL", "\" (QUOTE)"))
                        tokens.add(Token("Literal Strings", parts[i +
1]))
                        tokens.add(Token("SYMBOL", "\" (QUOTE)"))
                    }
                }
```

```
                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) {
        "Error parsing expression: ${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()
            else -> index++
        }
    }

    private fun parseAssignment() {
        startNode("LET")
        index++
        parseIdentifier()
        if (match("OPERATOR", "=")) {
            addNode("OPERATOR: =")
            index++
            parseExpression()
        }
```

```
        endNode()
    }

    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()
            else -> index++
        }
        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)
        }
    }
```