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
Фроленко К.Ю.
Проверил: ассистент кафедры информатики
Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

1 Формулировка задачи 3
2 Краткие теоритические сведения 4
3 Результат работы программы..... 6
Заключение 9
Список использованных источников 10
Приложение А (обязательное) 11

1 ФОРМУЛИРОВКА ЗАДАЧИ

Цель данной лабораторной работы заключается в глубоком изучении принципов синтаксического анализа и разработке собственного синтаксического анализатора, способного не только проверять исходный текст программы на соответствие синтаксическим нормам языка, но и формировать наглядное синтаксическое дерево, отражающее структурную организацию входной последовательности токенов. Эта задача позволяет исследовать фундаментальные методы построения грамматических структур и улучшить навыки работы с формальными грамматиками и методами анализа.

Исходный текст программы, представленный в виде потока лексем, включает различные элементы – идентификаторы, литералы, операторы, скобки и разделители. Анализатор должен на основе заданных правил синтаксиса выделять грамматические конструкции, такие как арифметические выражения, операторы присваивания и прочие синтаксические единицы, путем группировки последовательностей токенов в логически связанные фрагменты. Это обеспечивает возможность последующего анализа и трансформации кода, а также облегчает понимание его структурной организации.

Особое внимание в работе уделяется правильному определению порядка выполнения операций, когда вспомогательные элементы вроде скобок утрачивают свою семантическую нагрузку, уступая место непосредственно выполняемым операциям. Таким образом, синтаксическое дерево представляет собой иерархическую структуру, в которой внутренние вершины указывают на выполняемые операции, а листья – на операнды. Это позволяет наглядно проследить последовательность вычислительных шагов и установить взаимосвязь между элементами программы.

Кроме того, важным аспектом работы является обработка синтаксических ошибок. Разработанный анализатор должен иметь возможность своевременно обнаруживать ошибки, такие как неправильное расположение операторов, отсутствие необходимых разделителей или некорректное использование скобок, и генерировать информативные сообщения об ошибках. Такой механизм обеспечивает возможность быстрой диагностики и исправления ошибок, что является существенным для повышения надежности и качества программного обеспечения.

Реализация синтаксического анализатора выполняется с применением одного из табличных методов, таких как *LL*- или *LR*-анализ, либо с использованием метода рекурсивного спуска. Выбор метода определяется сложностью синтаксиса и требуемой степенью точности анализа. Применение выбранного метода позволяет не только корректно разбить поток токенов на грамматические фразы, но и обеспечить раннее обнаружение ошибок, что является важным этапом для последующего синтаксического и семантического анализа программы.

2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Синтаксический анализ представляет собой один из ключевых этапов трансляции программ, на котором поток лексем, полученный после лексического анализа, преобразуется в иерархическую структуру, отражающую грамматическую организацию исходного кода [1]. Этот процесс основывается на формальных грамматиках, которые определяют набор правил для построения корректных программных конструкций. Благодаря этим правилам происходит систематизация входного потока токенов, что позволяет обнаружить структурные ошибки еще до перехода к этапу семантического анализа или генерации промежуточного кода. Синтаксическое дерево, являющееся итогом анализа, представляет собой граф, где каждая вершина соответствует определённой грамматической единице, а ребра отражают логические взаимосвязи между различными компонентами программы.

Важнейшей задачей синтаксического анализа является группировка последовательностей лексем в грамматические фразы, что обеспечивает корректное понимание логической структуры программы. Например, рассмотрим выражение «*COST = (PRICE+TAX)*0.98*». При его разборе анализатор сначала идентифицирует идентификатор *COST*, затем знак присваивания, после чего обрабатывает выражение в скобках, в котором идентификаторы *PRICE* и *TAX* объединяются посредством оператора сложения, а полученный результат далее умножается на числовой литерал 0.98. В процессе построения синтаксического дерева скобки, служащие для задания приоритета операций, исключаются, уступая место узлам, которые явно представляют выполняемые арифметические действия. Такой метод позволяет точно определить последовательность вычислительных операций и тем самым гарантирует правильное выполнение программного кода [2].

Методы синтаксического анализа условно делятся на нисходящие (*top-down*) и восходящие (*bottom-up*) подходы. При нисходящем анализе, который включает метод рекурсивного спуска или *LL*-анализ, разбор начинается со стартового символа грамматики, после чего последовательно выводится цепочка токенов в соответствии с установленными правилами. В отличие от этого, восходящий анализ, представленный такими методами, как *LR*-анализ, начинает обработку с отдельных лексем и постепенно сводит их к стартовому символу, тем самым восстанавливая всю синтаксическую структуру программы [3]. Выбор между этими методами зачастую определяется сложностью грамматики конкретного языка и требованиями к точности разбора, что позволяет адаптировать анализатор для решения специфических задач, возникающих при обработке программ разной сложности.

Одним из важнейших аспектов синтаксического анализа является обнаружение синтаксических ошибок. Если поток токенов не соответствует заданным правилам формальной грамматики, анализатор фиксирует ошибку и предоставляет подробную информацию о её локализации в исходном коде. Такой механизм позволяет не только указывать на место нарушения синтаксиса, но и давать рекомендации по его исправлению, что существенно

облегчает процесс отладки и повышает надежность программного обеспечения [4]. Ранняя диагностика синтаксических ошибок помогает предотвратить накопление проблем на последующих этапах трансляции и оптимизации кода, что является важным условием для создания качественных компиляторов и интерпретаторов.

Практическая реализация синтаксического анализатора предполагает использование специализированных генераторов, способных автоматически строить необходимый разбор на основе заданной грамматики. Такие генераторы не только ускоряют процесс разработки, но и обеспечивают модульность и масштабируемость инструментов трансляции. Кроме того, современные компиляторы часто используют гибридные методы, сочетающие преимущества как нисходящего, так и восходящего подходов, что позволяет добиться высокой точности анализа даже для языков с комплексной структурой. Возможность интеграции синтаксического анализатора с другими компонентами системы трансляции делает его незаменимым инструментом при создании современных систем программирования.

Расширяя теоретическую базу, можно отметить, что синтаксический анализ является фундаментальным звеном не только в компиляторостроении, но и в разработке средств автоматической обработки естественных языков. Применение аналогичных методов позволяет анализировать и структурировать текстовую информацию, что находит применение в системах машинного перевода, обработке запросов и поисковых системах. Такой междисциплинарный характер синтаксического анализа подчеркивает его важность и универсальность, позволяя разрабатывать эффективные алгоритмы для различных областей информатики.

Комплексный подход к синтаксическому анализу, включающий в себя теоретическую модель, практические методы и средства автоматизации, является залогом создания высококачественных систем трансляции программ. Современные исследования в области компиляторостроения подтверждают, что правильно реализованный синтаксический анализ значительно упрощает последующие этапы трансляции, такие как семантический анализ и генерация кода, что в конечном итоге влияет на эффективность и производительность программных систем. Таким образом, синтаксический анализатор, интегрированный в систему компиляции, выполняет ключевую роль в обеспечении корректности, надежности и масштабируемости программ.

3 РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ

В ходе выполнения лабораторной работы был разработан синтаксический анализатор на языке *R*, использующий библиотеки *jsonlite*, *DiagrammeR*, *DiagrammeRsvg* и *rsvg*. Программа принимает на вход файл *tokens_output.json*, содержащий массив токенов, сформированных на предыдущем этапе лексического анализа, и последовательно обрабатывает его для построения конкретного синтаксического дерева (*CST*). В данном дереве сохраняется информация обо всех токенах, что позволяет детально проследить процесс группировки лексем в соответствии с синтаксическими правилами языка *PL/1*.

CST сохраняется в текстовом виде в файле *st_tree.txt* в формате *JSON*, что даёт возможность изучить полную структуру разбора, включая все токены и грамматические конструкции. Для наглядности работы анализатора синтаксическое дерево дополнительно визуализируется с помощью библиотеки *DiagrammeR*: формируется граф в формате *Graphviz*, который экспортируется в *SVG*-файл *st_tree.svg*. Это графическое представление демонстрирует, как отдельные токены группируются в соответствии с правилами языка, и позволяет легко выявить возможные ошибки в синтаксическом разборе.

На рисунке 3.1 представлен анализируемый код, подготовленный для тестирования работы разработанного анализатора.

Рисунок 3.1 – Анализируемый код

На рисунке 3.2 показано текстовое представление конкретного синтаксического дерева (*CST*), сохранённого в файле *st_tree.txt*. Здесь видно, как все токены, поступающие на вход, структурированы в виде вложенной *JSON*-структуры, отражающей детальную организацию исходного кода.

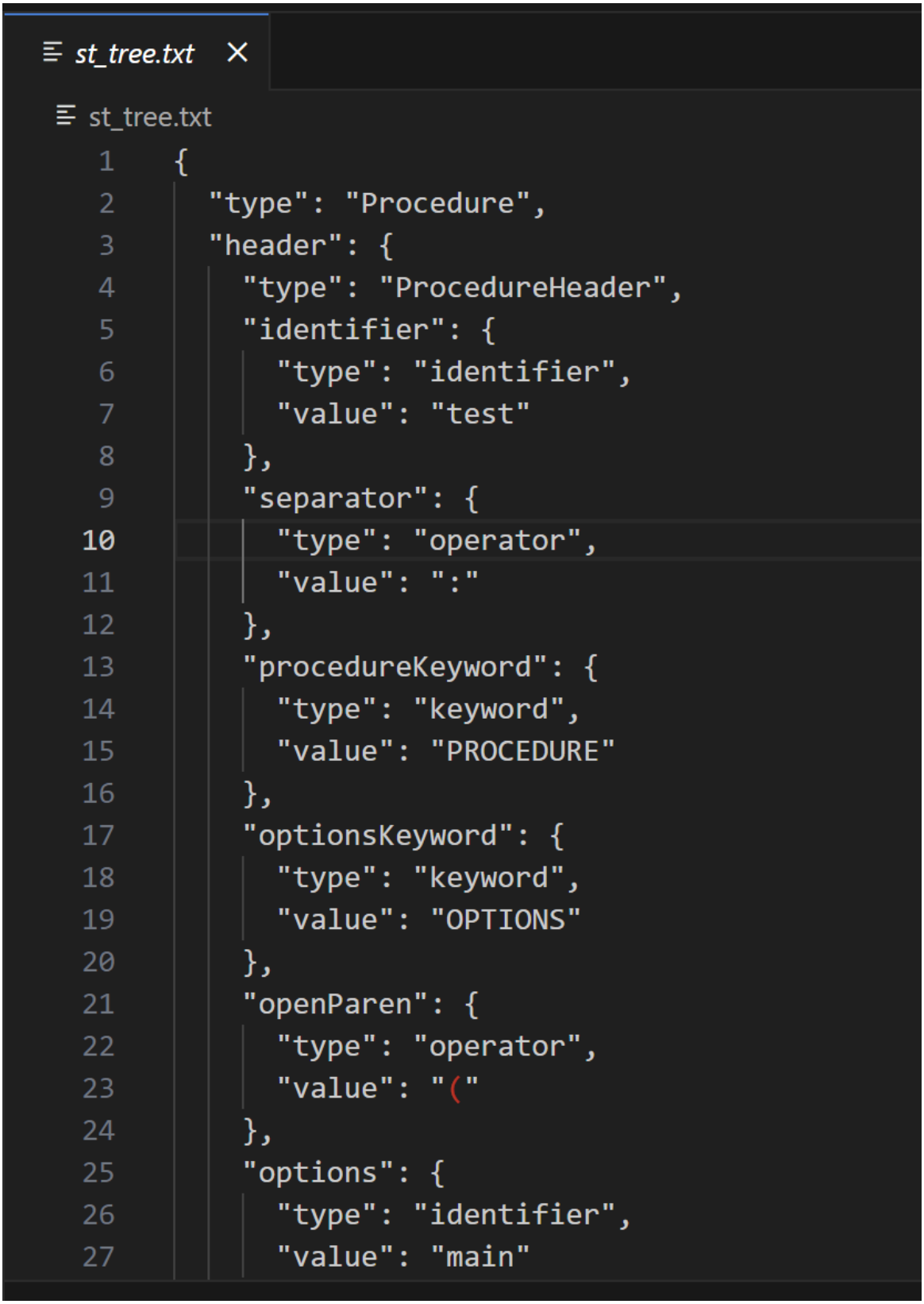

Рисунок 3.2 – Текстовое представление конкретного синтаксического дерева

На рисунке 3.3 демонстрируется визуальное представление конкретного синтаксического дерева, сохранённого в виде файла *st_tree.svg*. Графическое отображение *CST* позволяет наглядно проследить последовательность группировки токенов в грамматические конструкции, что значительно упрощает процесс отладки и проверки корректности разбора исходного кода.

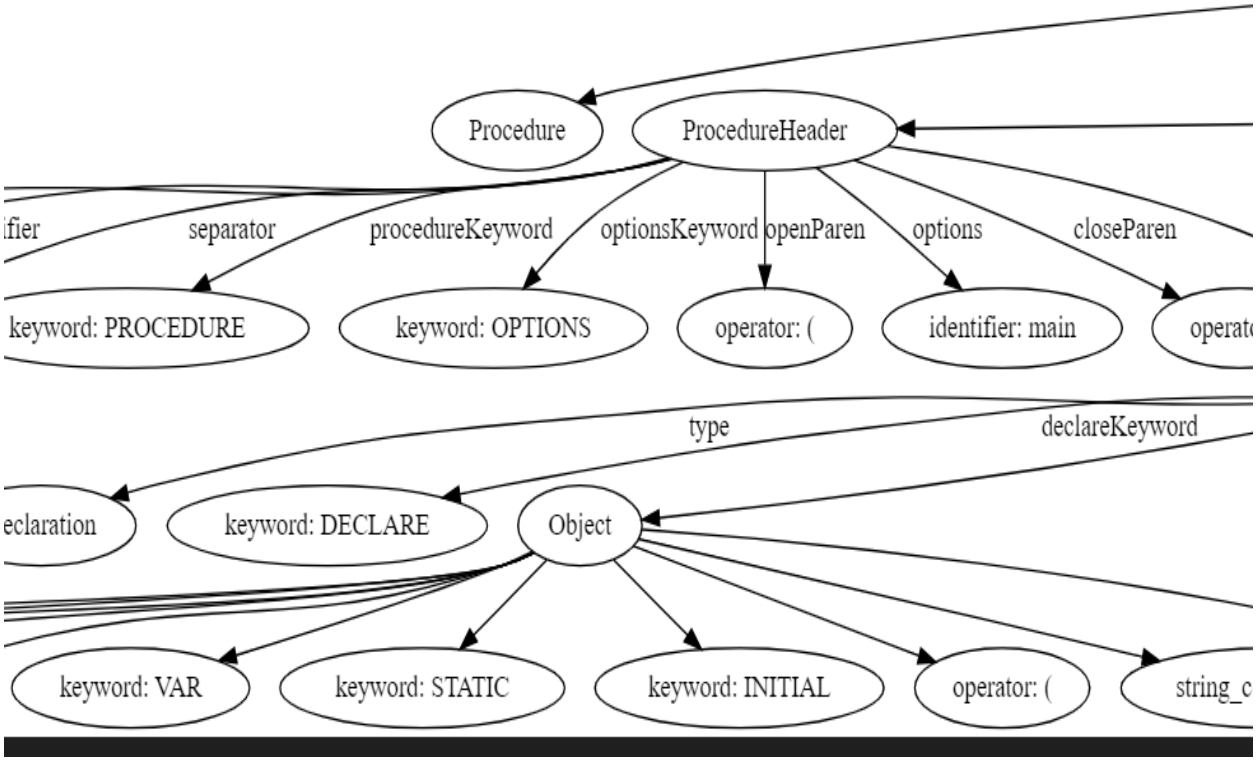

Рисунок 3.3 – Визуальное представление конкретного синтаксического дерева

Таким образом, разработанный синтаксический анализатор успешно выполняет задачу по разбору исходного кода на языке *PL/1*, преобразуя входной поток токенов в конкретное синтаксическое дерево (*CST*) и обеспечивая наглядное представление результатов работы как в текстовом, так и в графическом виде. Это позволяет детально анализировать структуру исходного кода и оперативно выявлять ошибки на этапе синтаксического анализа.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была решена задача разработки синтаксического анализатора для языка *PL/1* с использованием языка *R*. На вход анализатора поступает поток токенов, полученных на предыдущем этапе лексического анализа, который затем группируется в конкретное синтаксическое дерево (*CST*). Такой подход позволяет восстановить детальную структуру исходного кода, а также выявить синтаксические ошибки на ранних этапах трансляции.

Особое внимание в работе уделялось корректной обработке синтаксических ошибок, таких как неправильное расположение операторов, отсутствие необходимых разделителей или некорректное использование скобок. Реализованный анализатор не только фиксирует подобные несоответствия, но и предоставляет информативные сообщения, что существенно облегчает процесс отладки и повышает надежность программы.

Разработанная программа демонстрирует возможность получения как текстового представления *CST* (файл *st_tree.txt*), так и его визуализации в виде графического файла (*st_tree.svg*) с использованием средств, предоставляемых библиотеками *DiagrammeR* и *DiagrammeRsvg*. Модульная структура кода обеспечивает гибкость и масштабируемость анализатора, позволяя в дальнейшем расширять функциональность за счёт добавления поддержки новых синтаксических конструкций и интеграции с последующими этапами трансляции, такими как семантический анализ и оптимизация программного кода.

Таким образом, проведённое исследование подтвердило значимость корректного синтаксического анализа на ранних этапах трансляции. Разработанный синтаксический анализатор позволяет детально структурировать исходный код и служит надежной основой для дальнейшей обработки, что является важным условием для создания полноценных систем компиляции. Полученные результаты могут служить отправной точкой для дальнейшего развития инструмента и реализации полноценного компилятора для языка *PL/1*.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Синтаксический анализ – Википедия [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Синтаксический_анализ. – Дата доступа: 10.03.2025.

[2] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2-е изд.). – Режим доступа: https://www.pearson.com/us/higher-education/program/Aho-Compilers-Principles-Techniques-and-Tools-2nd-Edition/PGM132973.html. – Дата доступа: 10.03.2025.

[3] Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J. H., & Langendoen, K. (2012). *Modern Compiler Design*. – Режим доступа: https://www.springer.com/gp/book/9783642193729. – Дата доступа: 10.03.2025.

[4] Appel, A. W. (1998). *Modern Compiler Implementation in Java*. – Режим доступа: https://www.amazon.com/Modern-Compiler-Implementation-Java-2nd/dp/0521820537. – Дата доступа: 11.03.2025.

ПРИЛОЖЕНИЕ А
(обязательное)
Исходный код программы

```
library(jsonlite)
library(DiagrammeR)
library(DiagrammeRsvg)
library(rsvg)

tokens <- fromJSON("tokens_output.json", simplifyDataFrame = FALSE)
pos <- 1

currentToken <- function() {
    if (pos <= length(tokens)) {
        return(tokens[[pos]])
    } else {
        return(NULL)
    }
}

peekToken <- function() {
    if (pos + 1 <= length(tokens)) {
        return(tokens[[pos + 1]])
    } else {
        return(NULL)
    }
}

nextToken <- function() {
    pos <<- pos + 1
    return(currentToken())
}

createTokenNode <- function(expectedType = NULL) {
    token <- currentToken()
    if (is.null(token)) {
        stop("Неожиданный конец токенов")
    }
    cat(sprintf(
        "Parsed token: id=%s, type=%s, value=%s\n",
        token$tokenID, token$type, token$value
    ))
    node <- list(type = token$type, value = token$value)
    pos <<- pos + 1
    return(node)
}

parseProcedureHeader <- function() {
    header <- list(type = "ProcedureHeader")
    header$identifier <- createTokenNode("identifier")
    header$separator <- createTokenNode("operator")
    header$procedureKeyword <- createTokenNode("keyword")
    header$optionsKeyword <- createTokenNode("keyword")
    header$openParen <- createTokenNode("operator")
    header$options <- createTokenNode("identifier")
    header$closeParen <- createTokenNode("operator")
    header$endHeader <- createTokenNode("operator")
    return(header)
}

parseDeclaration <- function() {
    decl <- list(type = "Declaration")
```

```
    decl$declareKeyword <- createTokenNode("keyword")
    decl$tokens <- list()
    while (!is.null(currentToken())) {
        token <- currentToken()
        if (token$type == "operator" && token$value == ";") {
            decl$endDeclaration <- createTokenNode("operator")
            break
        }
        decl$tokens <- c(decl$tokens, list(createTokenNode()))
    }
    return(decl)
}

parseDeclarations <- function() {
    decls <- list()
    while (!is.null(currentToken()) &&
        currentToken()$type == "keyword" &&
        currentToken()$value == "DECLARE") {
        decls <- c(decls, list(parseDeclaration()))
    }
    return(decls)
}

parseStatementBlock <- function() {
    block <- list(type = "StatementBlock", tokens = list())
    token <- currentToken()
    if (!is.null(token) && token$type == "keyword" && token$value == "END") {
        block$tokens <- c(block$tokens, list(createTokenNode()))
        return(block)
    }
    while (!is.null(currentToken())) {
        token <- currentToken()
        if (token$type == "operator" && token$value == ";") {
            block$tokens <- c(block$tokens, list(createTokenNode()))
            break
        }
        block$tokens <- c(block$tokens, list(createTokenNode()))
    }
    return(block)
}

parseStatements <- function(procName) {
    stmts <- list()
    while (!is.null(currentToken())) {
        token <- currentToken()
        if (token$type == "keyword" && token$value == "END") {
            nextTok <- peekToken()
            if (!is.null(nextTok) &&
                nextTok$type == "identifier" &&
                trimws(nextTok$value) == procName) {
                break
            }
        }
        stmts <- c(stmts, list(parseStatementBlock()))
    }
    return(stmts)
}

parseProcedureEnd <- function(expectedName) {
    procEnd <- list(type = "ProcedureEnd")
    procEnd$endKeyword <- createTokenNode("keyword")
    procEnd$identifier <- createTokenNode("identifier")
    if (procEnd$identifier$value != expectedName) {
```

```
        stop("Идентификатор завершения процедуры не совпадает с именем
процедуры")
    }
    procEnd$endSymbol <- createTokenNode("operator")
    return(procEnd)
}

parseProcedure <- function() {
    ast <- list(type = "Procedure")
    ast$header <- parseProcedureHeader()
    ast$declarations <- parseDeclarations()
    procName <- ast$header$identifier$value
    ast$statements <- parseStatements(procName)
    ast$procedureEnd <- parseProcedureEnd(procName)
    return(ast)
}

ast_tree <- parseProcedure()

ast_text <- toJSON(ast_tree, pretty = TRUE, auto_unbox = TRUE)
write(ast_text, file = "st_tree.txt")

dotNodes <- c()
dotEdges <- c()
nodeIdCounter <- 0

newNodeId <- function() {
    nodeIdCounter <<- nodeIdCounter + 1
    return(paste0("node", nodeIdCounter))
}

traverseAst <- function(ast, parent = NULL, edgeLabel = NULL) {
    currentId <- newNodeId()
    label <- ""
    if (is.list(ast)) {
        if (length(ast) == 2 && all(c("type", "value") %in% names(ast))) {
            label <- paste0(ast$type, ": ", ast$value)
        } else if (!is.null(ast$type)) {
            label <- ast$type
        } else {
            label <- "Object"
        }
    } else {
        label <- as.character(ast)
    }
    label <- gsub('"', '\\"', label)
    dotNodes <<- c(dotNodes, sprintf('  %s [label="%s"];', currentId, label))

    if (!is.null(parent)) {
        if (!is.null(edgeLabel)) {
            dotEdges <<- c(dotEdges, sprintf('  %s -> %s [label="%s"];', parent,
currentId, edgeLabel))
        } else {
            dotEdges <<- c(dotEdges, sprintf("  %s -> %s;", parent, currentId))
        }
    }

    if (is.list(ast) && !(length(ast) == 2 && all(c("type", "value") %in%
names(ast)))) {
        if (!is.null(names(ast))) {
            for (name in names(ast)) {
                if (name == "pointer") next
                child <- ast[[name]]
                if (!is.null(child)) {
```

```
                    if (is.list(child)) {
                        traverseAst(child, currentId, name)
                    } else {
                        childId <- newNodeId()
                        dotNodes <<- c(dotNodes, sprintf('  %s [label="%s"];',
childId, as.character(child)))
                        dotEdges  <<-  c(dotEdges,  sprintf('    %s  ->  %s
[label="%s"];', currentId, childId, name))
                    }
                }
            }
        } else {
            for (child in ast) {
                if (!is.null(child)) {
                    if (is.list(child)) {
                        traverseAst(child, currentId)
                    } else {
                        childId <- newNodeId()
                        dotNodes <<- c(dotNodes, sprintf('  %s [label="%s"];',
childId, as.character(child)))
                        dotEdges  <<-  c(dotEdges,  sprintf("    %s  ->  %s;",
currentId, childId))
                    }
                }
            }
        }
    }
    return(currentId)
}

dotNodes <<- c()
dotEdges <<- c()
nodeIdCounter <<- 0
rootId <- traverseAst(ast_tree)
dotGraphString <- paste("digraph ST {",
    paste(dotNodes, collapse = "\n"),
    paste(dotEdges, collapse = "\n"),
    "}",
    sep = "\n"
)

svg_code <- export_svg(grViz(dotGraphString))
write(svg_code, file = "st_tree.svg")

cat("ST дерево сохранено в 'st_tree.txt'\n")
cat("Визуальное представление ST сохранено в 'st_tree.svg'\n")
```