Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы трансляции

ОТЧЁТ
к лабораторной работе №5
на тему

ИНТЕРПРЕТАЦИЯ ИСХОДНОГО КОДА

Выполнил: студент гр.253504
Фроленко К.Ю.
Проверил: ассистент кафедры информатики
Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

1 ФОРМУЛИРОВКА ЗАДАЧИ

Цель данной лабораторной работы заключается в разработке интерпретатора исходного кода, который выполняет программу на основе результатов анализа, полученных в лабораторных работах 1–4. Основная задача интерпретатора – осуществить динамическое выполнение инструкций, представленных в виде синтаксического дерева (*CST*), сохранённого в текстовом формате *JSON*. Это дерево, полученное на предыдущих этапах трансляции, содержит детальную структурную информацию о исходном коде, что позволяет интерпретатору точно восстановить логику работы программы.

При выполнении интерпретации исходного кода решаются следующие задачи:

1 Загрузка и анализ *CST***:** интерпретатор использует библиотеку *jsonlite* для загрузки синтаксического дерева из файла st_tree.txt. Это дерево содержит вложенную иерархию токенов, отражающую структуру программы, и служит основой для последующего исполнения команд.

2 Формирование и обновление таблиц: для корректного выполнения программы создаются и обновляются таблицы символов, указателей и базовых переменных. Эти структуры позволяют хранить информацию о типах, значениях и областях видимости переменных, а также обеспечивают разрешение ссылок через указатели. Такой подход позволяет моделировать динамическое состояние программы в процессе её выполнения.

3 Выполнение управляющих конструкций и выражений: интерпретатор осуществляет выполнение операторов управления потоком, таких как условные конструкции (*IF-THEN-ELSE*), циклы (*DO, REPEAT, TO, WHILE*), а также операторов ввода/вывода (*PUT*). При этом происходит интерпретация арифметических и логических выражений, разрешение вызовов процедур и корректное выполнение операций над агрегатными структурами.

4 Контроль корректности и отладка: благодаря встроенным функциям отладки (*debugPrint*) интерпретатор отслеживает процесс выполнения, выводя промежуточные результаты и сообщения об ошибках. Это позволяет не только обнаруживать логические и синтаксические недочёты в исходном коде, но и предоставляет возможность оперативно корректировать ошибки на этапе исполнения.

В результате реализации данного интерпретатора достигается не только воспроизведение логики исходной программы, но и создание основы для дальнейшей разработки компилятора, способного выполнять полную трансляцию кода в машинный. Такой комплексный подход подтверждает значимость интеграции предыдущих этапов трансляции (лексический, синтаксический и семантический анализ) для корректного и эффективного исполнения программ.

2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Интерпретация исходного кода представляет собой один из ключевых этапов трансляции программ, на котором осуществляется непосредственное выполнение инструкций программы в режиме реального времени. В отличие от компиляции, при которой исходный код преобразуется в машинный код с последующим выполнением, интерпретация позволяет динамически анализировать и исполнять программу «на лету». Такой подход существенно упрощает отладку, позволяет получать мгновенную обратную связь, а также поддерживает возможность интерактивного ввода команд, что особенно ценно для разработки прототипов и образовательных целей [1].

Одним из фундаментальных аспектов интерпретации является загрузка и анализ синтаксического дерева (*CST*), сохранённого в текстовом формате JSON. Это дерево, сформированное на этапах лексического, синтаксического и семантического анализа, содержит полное описание структуры исходного кода – от отдельных токенов до сложных управляющих конструкций. Благодаря библиотеке *jsonlite* интерпретатор может эффективно загрузить и обработать это дерево, что служит прочной основой для дальнейшего динамического исполнения команд [2]. Такой метод позволяет интерпретировать не только простые выражения, но и сложные программные конструкции, обеспечивая точное восстановление логики работы исходного кода.

После загрузки *CST* интерпретатор переходит к формированию внутренних структур данных, таких как таблицы символов, указателей и базовых переменных. Таблица символов аккумулирует информацию о переменных: их имена, типы, значения и области видимости. Таблицы указателей и базовых переменных позволяют отслеживать динамические связи между элементами программы, корректно разрешать ссылки и управлять памятью в процессе выполнения. Такой подход обеспечивает моделирование динамического состояния программы и позволяет реализовывать сложные алгоритмы обработки данных, поддерживая целостность и согласованность при изменении значений переменных [3].

Основная функциональность интерпретатора заключается в последовательном выполнении инструкций, представленных в *CST*. Для этого реализуются специализированные функции, отвечающие за интерпретацию различных типов команд:

1 Управляющие конструкции: интерпретатор обрабатывает условные операторы (*IF-THEN-ELSE*), циклы (*DO, REPEAT, TO, WHILE*) и операторы перехода (*GOTO, CALL*). Каждый из этих элементов подвергается динамической оценке, что позволяет интерпретировать сложные логические выражения и изменять порядок выполнения команд в зависимости от текущего состояния программы. Такой механизм управления потоком исполнения является основой для реализации сложных алгоритмов и принятия решений на лету [4].

2 Арифметические и логические выражения**:** при выполнении арифметических операций и логических сравнений интерпретатор анализирует входящие выражения, преобразует строковые значения в числовые и проводит необходимые вычисления. Функции для обработки арифметических операторов, таких как сложение, вычитание, умножение и деление, а также логических операторов (например, сравнения «=», «<», «>=»), гарантируют корректное исполнение математических операций, что критически важно для динамической интерпретации кода.

3 Операции ввода/вывода и вызова процедур: команды вывода, такие как *PUT*, используются для отображения промежуточных результатов выполнения, что значительно упрощает процесс отладки и тестирования программы. Дополнительно, поддержка вызова процедур (*CALL*) и переходов по меткам (*GOTO*) позволяет реализовывать модульную архитектуру программы и управлять последовательностью исполнения кода, обеспечивая гибкость и масштабируемость системы интерпретации.

4 Отладка и контроль корректности: встроенные функции отладки, такие как *debugPrint*, выводят подробные сообщения о состоянии таблиц, промежуточных результатах вычислений и ходе выполнения управляющих конструкций. Такой механизм позволяет оперативно обнаруживать как логические, так и синтаксические ошибки, что значительно сокращает время на поиск и устранение проблем. Это особенно полезно при разработке сложных систем, где даже небольшая ошибка может привести к нарушению логики работы программы [5].

Интерпретация исходного кода также играет важную роль в реализации интерактивных сред разработки, таких как *REPL* (*Read-Eval-Print Loop*). Благодаря этому подходу программисты могут вводить команды по одной, сразу получать результаты их выполнения и изменять код в режиме реального времени. Такая интерактивность способствует более глубокому пониманию работы программы и позволяет быстрее экспериментировать с алгоритмами и структурами данных.

В совокупности, методы интерпретации исходного кода обеспечивают динамическое и корректное выполнение программ, что позволяет не только анализировать и исполнять инструкции в режиме реального времени, но и эффективно оптимизировать работу системы. Интеграция процессов загрузки синтаксического дерева, формирования таблиц символов, динамического управления потоком исполнения и отладки создаёт прочную основу для дальнейшего развития систем трансляции программ и реализации полноценных компиляторов. Такой комплексный подход способствует созданию высококачественных, надёжных и оптимизированных программных продуктов, способных удовлетворить требования самых сложных вычислительных задач.

3 РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ

В ходе выполнения лабораторной работы интерпретатор исходного кода принимает на вход текстовое синтаксическое дерево (*CST*), сохранённое в файле *st_tree.txt* в формате *JSON*. Это дерево, полученное на предыдущих этапах трансляции, содержит подробное описание исходного кода – от отдельных токенов до вложенных грамматических конструкций. Интерпретатор анализирует структуру *CST*, формирует таблицы символов, указателей и базовых переменных, а затем динамически выполняет команды, отражённые в дереве, что позволяет воспроизвести логику работы исходной программы.

Программа последовательно обрабатывает *CST* следующим образом:

1 Загрузка и разбор синтаксического дерева: интерпретатор использует библиотеку *jsonlite* для загрузки *CST* из файла *st_tree.txt*. Это дерево демонстрирует, как исходный код разбит на блоки деклараций, операторов и управляющих конструкций, что позволяет наглядно проследить его структуру и иерархию.

2 Формирование внутренних таблиц: на основе *CST* интерпретатор формирует таблицу символов, в которой для каждой переменной фиксируется её имя, тип и начальное значение (если оно задано). Дополнительно создаются таблицы указателей и базовых переменных, позволяющие отслеживать динамические связи между элементами программы и корректно разрешать ссылки. Такой подход обеспечивает моделирование текущего состояния программы во время её исполнения.

3 Интерпретация управляющих конструкций и выражений**:** Интерпретатор выполняет инструкции, представленные в *CST*, обрабатывая условные конструкции (*IF-THEN-ELSE*), циклы (*DO, REPEAT, TO, WHILE*) и операторы перехода (*GOTO, CALL*). При этом происходит динамическая оценка арифметических и логических выражений, разрешение вызовов процедур и корректное выполнение операций ввода/вывода (например, через команду *PUT*). Это позволяет точно воспроизвести логику исходной программы, даже если в ней присутствуют сложные вложенные конструкции.

4 Контроль корректности выполнения и отладка: Встроенные функции отладки, такие как *debugPrint*, позволяют отслеживать промежуточные результаты выполнения, состояние таблиц символов и ход интерпретации. В случае обнаружения ошибок или некорректных вычислений интерпретатор выводит соответствующие сообщения, что существенно облегчает процесс отладки и тестирования программы. Все сообщения и результаты исполнения выводятся в консоль, а итоговый результат работы интерпретатора фиксируется сообщением «=== Интерпретация завершена ===».

На рисунке 3.1 представлено входное текстовое дерево (*CST*), сохранённое в файле *st_tree.txt***.** Это дерево иллюстрирует, как исходный код программы разбит на логические блоки и как отдельные токены группируются в грамматические конструкции, что служит основой для дальнейшей интерпретации.

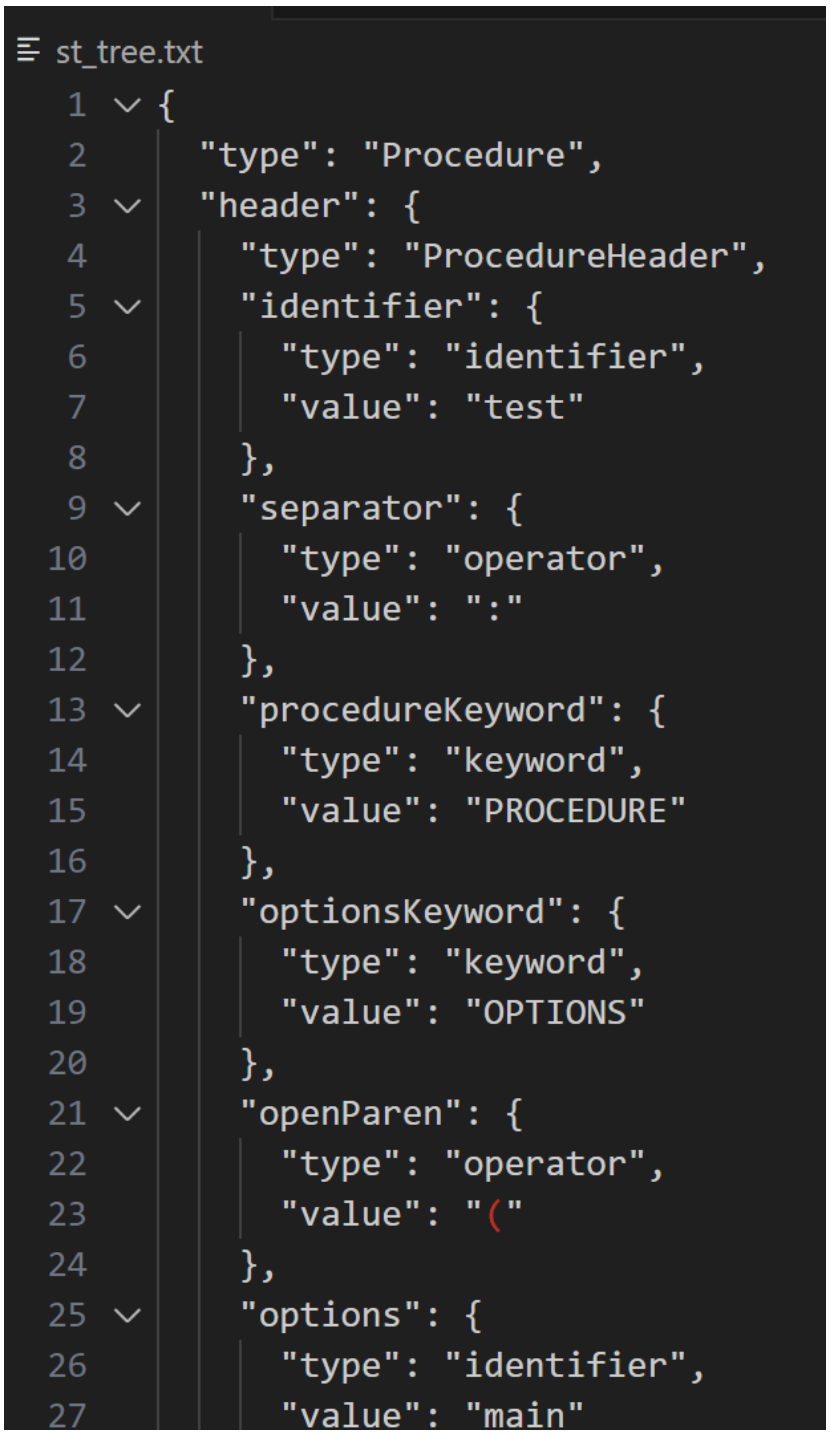

Рисунок 3.1 – Входное текстовое дерево (*CST*), сохранённое в файле *st_tree.txt*

На рисунке 3.2 показан пример вывода результатов работы интерпретатора, полученный в консольном режиме. Здесь видно, как интерпретатор последовательно выполняет команды исходного кода, выводит промежуточные сообщения, обрабатывает условия и циклы, а также завершает выполнение с финальным сообщением «=== Интерпретация завершена ===». Такой подробный отчёт позволяет оценить корректность работы интерпретатора и выявить возможные логические или синтаксические ошибки, которые могут возникать в процессе исполнения программы.

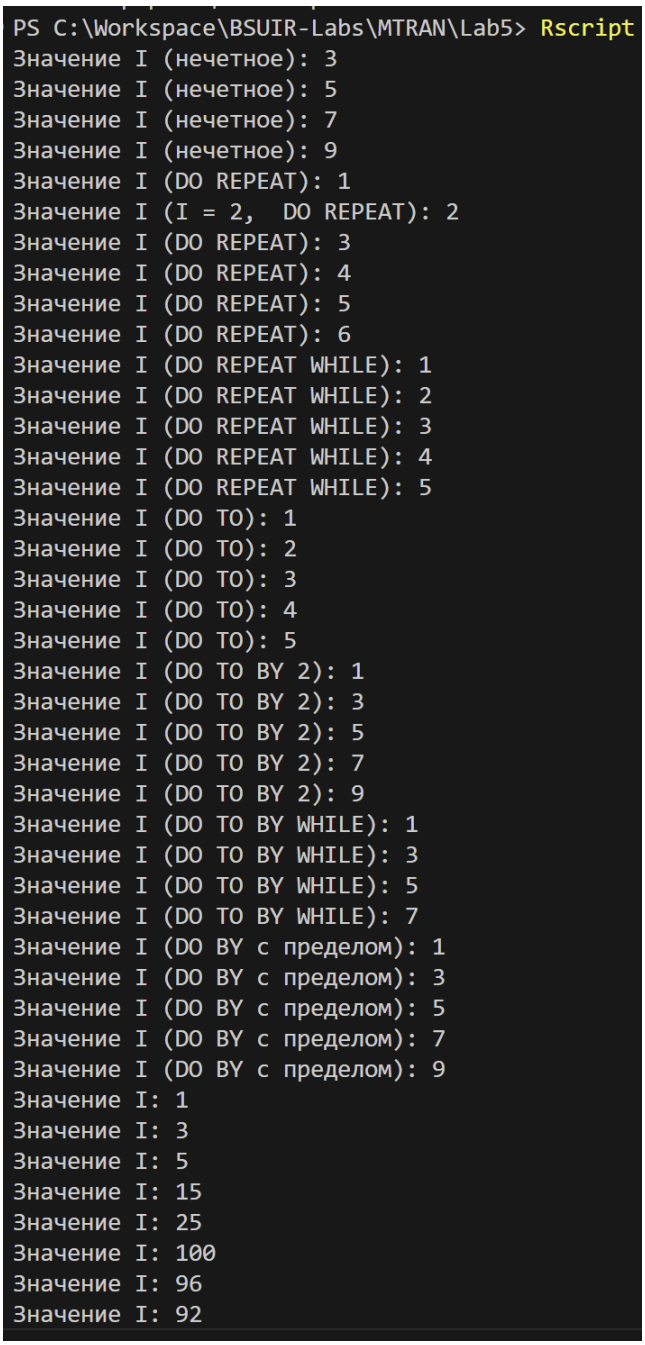

Рисунок 3.2 – Пример вывода результатов работы интерпретатора

Таким образом, разработанный интерпретатор успешно преобразует входное синтаксическое дерево в динамическое исполнение программы. Полученные результаты демонстрируют, что интерпретатор корректно выполняет все инструкции, обеспечивает управление потоком выполнения, правильно обрабатывает выражения и контролирует состояние переменных. Возможность оперативного вывода сообщений об ошибках и промежуточных результатов значительно сокращает время отладки и тестирования, способствуя созданию эффективных и надёжных программных систем. Такой комплексный подход к интерпретации исходного кода является важной основой для дальнейшей разработки полноценного компилятора и интегрированных систем трансляции программ.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была решена задача разработки интерпретатора исходного кода, который динамически исполняет программу на основе результатов анализа, полученных в лабораторных работах 1–4. Использование текстового синтаксического дерева (*CST*), сохранённого в формате *JSON*, позволило восстановить детальную структуру исходного кода и обеспечить его корректное выполнение. Интерпретатор последовательно обрабатывает *CST*, формируя внутренние таблицы символов, указателей и базовых переменных, что позволяет моделировать динамическое состояние программы в режиме реального времени.

Особое внимание было уделено реализации механизмов управления потоком исполнения. Интерпретатор корректно обрабатывает условные конструкции (*IF-THEN-ELSE*), циклы (*DO, REPEAT, TO, WHILE*) и операторы перехода (*GOTO, CALL*), динамически оценивая арифметические и логические выражения. Это позволяет воспроизводить сложную логику работы исходного кода, обеспечивая точное выполнение инструкций, независимо от вложенности конструкций и условий их исполнения.

Благодаря встроенным функциям отладки, таким как *debugPrint*, разработанный интерпретатор выводит подробные сообщения о состоянии таблиц символов и промежуточных результатах вычислений. Возможность оперативного выявления логических и синтаксических ошибок существенно упрощает процесс отладки и тестирования программы, что повышает надёжность системы в целом. Итоговое сообщение «=== Интерпретация завершена ===» свидетельствует о корректном выполнении всех команд и успешном завершении работы интерпретатора.

Таким образом, проведённое исследование подтвердило значимость интеграции предыдущих этапов трансляции – лексического, синтаксического и семантического анализа – для динамической интерпретации исходного кода. Разработанный интерпретатор не только воспроизводит логику работы исходной программы, но и служит прочной основой для дальнейшего развития системы трансляции, что открывает перспективы для реализации полноценных компиляторов. Полученные результаты могут стать отправной точкой для дальнейшего расширения функциональности инструмента, повышения его эффективности и создания надежных программных систем, способных удовлетворить требования современных вычислительных задач.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Pierce, B. (2002). *Types and Programming Languages*. – Режим доступа: https://mitpress.mit.edu/books/types-and-programming-languages. – Дата доступа: 13.03.2025.

[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: 13.03.2025.

[3] Appel, A. W. (1998). *Modern Compiler Implementation in Java*. – Режим доступа: https://www.amazon.com/Modern-Compiler-Implementation-Java-2nd/dp/0521820537. – Дата доступа: 13.03.2025.

[4] Scott, M. L. (2009). *Programming Language Pragmatics*. – Режим доступа: https://www.elsevier.com/books/programming-language-pragmatics/scott/978-0-12-374514-3. – Дата доступа: 14.03.2025.

[5] Winskel, G. (1993). *The Formal Semantics of Programming Languages*. – Режим доступа: https://www.cambridge.org/core/books/formal-semantics-of-programming-languages/7F02B50D0DE50E7AAE86ED15A9B3AB4C. – Дата доступа: 14.03.2025.

ПРИЛОЖЕНИЕ А
(обязательное)
Исходный код программы

```
library(jsonlite)

ast_obj <- fromJSON("st_tree.txt",
    flatten           = FALSE,
    simplifyDataFrame = FALSE,
    simplifyMatrix    = FALSE,
    simplifyVector    = FALSE
)

emit <- function(msg) {
    cat(msg, "\n")
}

debugFlag <- FALSE

debugPrint <- function(...) {
    if (debugFlag) {
        args <- lapply(list(...), function(x) {
            if   (is.list(x))   paste(unlist(x),   collapse   =   "   ")   else
as.character(x)
        })
        cat("[DEBUG]", paste(args, collapse = " "), "\n")
    }
}

symbol_table <- new.env(parent = emptyenv())
pointer_table <- new.env(parent = emptyenv())
based_table <- new.env(parent = emptyenv())

determineCodeType <- function(ast) {
    if (is.null(ast$declarations)) {
        return("Unknown")
    }

    for (decl in ast$declarations) {
        tokens <- decl$tokens
        if (length(tokens) > 0 && tokens[[1]]$type == "numeric_constant") {
            formatValue <- function(val) {
                if (is.list(val)) {
                    if (length(val) == 0) {
                        return("")
                    }
                    if (!is.null(names(val))) {
                        return(paste(sapply(val, formatValue), collapse = " "))
                    } else {
                        return(paste(sapply(val, formatValue), collapse = " "))
                    }
                } else {
                    return(as.character(val))
                }
            }

            createAggregate <- function(declTokens) {
                debugPrint("createAggregate: declTokens =", sapply(declTokens,
function(t) t$value))
                if (length(declTokens) >= 2 && declTokens[[2]]$value == "(") {
                    sizes <- c()
                    i <- 3
```

```
                    while (i <= length(declTokens) && declTokens[[i]]$value !=
")") {
                        if (declTokens[[i]]$type == "numeric_constant") {
                            sizes                 <-                 c(sizes,
as.integer(declTokens[[i]]$value))
                        }
                        i <- i + 1
                    }
                    debugPrint("Создаем массив с размерами:", sizes)
                    createArray <- function(dims) {
                        if (length(dims) == 0) {
                            return(NA)
                        }
                        res <- vector("list", dims[1])
                        if (length(dims) > 1) {
                            for (j in seq_len(dims[1])) {
                                res[[j]] <- createArray(dims[-1])
                            }
                        } else {
                            for (j in seq_len(dims[1])) {
                                res[[j]] <- NA
                            }
                        }
                        return(res)
                    }
                    return(createArray(sizes))
                }
                debugPrint("Создаем структуру (пустой список)")
                return(list())
            }

            parseCompoundIdentifier <- function(tokens) {
                debugPrint("parseCompoundIdentifier: tokens =", sapply(tokens,
function(t) t$value))
                parts <- list()
                i <- 1
                if (tokens[[i]]$type %in% c("identifier", "numeric_constant"))
{
                    parts[[length(parts) + 1]] <- list(type = "name", value =
tokens[[i]]$value)
                    i <- i + 1
                }
                while (i <= length(tokens)) {
                    if (tokens[[i]]$value == "(") {
                        i <- i + 1
                        indices <- c()
                        while (i <= length(tokens) && tokens[[i]]$value != ")")
{
                            if (tokens[[i]]$type == "numeric_constant") {
                                indices             <-             c(indices,
as.integer(tokens[[i]]$value))
                            }
                            i <- i + 1
                        }
                        parts[[length(parts)  +  1]]  <-  list(type  =  "index",
value = indices)
                        i <- i + 1
                    } else if (tokens[[i]]$value == ".") {
                        i <- i + 1
                        if  (i  <=  length(tokens)  &&  tokens[[i]]$type  ==
"identifier") {
                            parts[[length(parts) + 1]] <- list(type = "field",
value = tokens[[i]]$value)
                            i <- i + 1
```

```
                    }
                } else {
                    i <- i + 1
                }
            }
            debugPrint("parseCompoundIdentifier:", parts)
            return(parts)
        }

        getCompoundValue <- function(parts) {
            debugPrint("getCompoundValue: parts =", parts)
            curVal <- if (exists(parts[[1]]$value, envir = symbol_table,
inherits = FALSE)) {
                get(parts[[1]]$value, envir = symbol_table)
            } else {
                NA
            }
            debugPrint("Начальное   значение",   parts[[1]]$value,   "=",
curVal)
            if (is.atomic(curVal) && length(curVal) == 1 && is.na(curVal))
{
                return(NA)
            }
            if (length(parts) > 1) {
                for (i in 2:length(parts)) {
                    p <- parts[[i]]
                    if (p$type == "index") {
                        debugPrint("Обрабатываем индекс:", p$value)
                        for (idx in p$value) {
                            if (is.list(curVal) && length(curVal) >= idx)
{
                                curVal <- curVal[[idx]]
                            } else {
                                debugPrint("Индекс", idx, "вне диапазона")
                                return(NA)
                            }
                        }
                    } else if (p$type == "field") {
                        debugPrint("Обрабатываем поле:", p$value)
                        if               (is.list(curVal)               &&
!is.null(curVal[[p$value]])) {
                            curVal <- curVal[[p$value]]
                        } else {
                            debugPrint("Поле", p$value, "не найдено")
                            return(NA)
                        }
                    }
                }
            }
            debugPrint("getCompoundValue возвращает:", curVal)
            return(curVal)
        }

        setCompoundValue <- function(parts, newVal) {
            debugPrint("setCompoundValue: parts =", parts, "новое значение
=", newVal)
            if (length(parts) == 0) {
                return()
            }
            baseName <- parts[[1]]$value
            if (!exists(baseName, envir = symbol_table, inherits = FALSE))
{
                assign(baseName, NA, envir = symbol_table)
            }
```

```
                curVal <- get(baseName, envir = symbol_table)
                if ((length(parts) >= 2) &&
                    ((is.atomic(curVal)    &&    length(curVal)    ==    1    &&
is.na(curVal)) || is.null(curVal))) {
                    curVal <- list()
                }
                setNested <- function(cur, parts, pos, newVal) {
                    if (pos > length(parts)) {
                        return(newVal)
                    }
                    p <- parts[[pos]]
                    if (p$type == "index") {
                        indices <- p$value
                        if (!is.list(cur)) cur <- list()
                        idx <- indices[1]
                        if (length(cur) < idx || is.null(cur[[idx]])) {
                            cur[[idx]] <- if (length(indices) == 1 && pos ==
length(parts)) NA else list()
                        }
                        if (length(indices) > 1) {
                            cur[[idx]] <- setNested(cur[[idx]], list(list(type
= "index", value = indices[-1])), 1, newVal)
                        } else {
                            cur[[idx]] <- setNested(cur[[idx]], parts, pos + 1,
newVal)
                        }
                        return(cur)
                    } else if (p$type == "field") {
                        if (!is.list(cur)) cur <- list()
                        fieldName <- p$value
                        debugPrint("Устанавливаем поле", fieldName)
                        cur[[fieldName]]            <-            setNested(if
(!is.null(cur[[fieldName]])) cur[[fieldName]] else NA, parts, pos + 1, newVal)
                        return(cur)
                    } else {
                        return(newVal)
                    }
                }
                newBaseVal <- setNested(curVal, parts, 2, newVal)
                assign(baseName, newBaseVal, envir = symbol_table)
                debugPrint("После установки, базовое значение", baseName, "=",
get(baseName, envir = symbol_table))
            }

            parseNumeric <- function(x) {
                debugPrint("parseNumeric: x =", x)
                if (is.null(x) || length(x) == 0) {
                    debugPrint("parseNumeric: x is null or empty")
                    return(NULL)
                }
                x <- as.character(x)
                x <- trimws(x)
                if (nchar(x) == 0) {
                    debugPrint("parseNumeric: trimmed x is empty")
                    return(NULL)
                }
                val <- suppressWarnings(as.numeric(x))
                debugPrint("parseNumeric: val =", val)
                if (is.na(val)) {
                    return(NULL)
                }
                return(val)
            }
```

```
            getValue <- function(varToken) {
                debugPrint("getValue: varToken =", varToken)
                if (is.list(varToken) && length(varToken) > 1) {
                    compound <- parseCompoundIdentifier(varToken)
                    return(getCompoundValue(compound))
                }
                if (exists(varToken, envir = symbol_table, inherits = FALSE))
{
                    value <- get(varToken, envir = symbol_table)
                    debugPrint("getValue:", varToken, "=", value)
                    return(value)
                } else {
                    debugPrint("getValue:", varToken, "не найдено")
                    return(NA)
                }
            }

            setValue <- function(varToken, newVal) {
                debugPrint("setValue:  varToken  =",  varToken,  "newVal  =",
newVal)
                if (is.list(varToken) && length(varToken) > 1) {
                    compound <- parseCompoundIdentifier(varToken)
                    setCompoundValue(compound, newVal)
                    return()
                }
                assign(varToken, newVal, envir = symbol_table)
            }

            setPointer <- function(ptrName, varName) {
                debugPrint("setPointer:", ptrName, "->", varName)
                assign(ptrName, varName, envir = pointer_table)
            }

            declareBased <- function(basedVar, ptrName) {
                debugPrint("declareBased:", basedVar, "->", ptrName)
                assign(basedVar, ptrName, envir = based_table)
            }

            handleOpenFile <- function(tokens) {
                titleIdx <- which(toupper(sapply(tokens, function(t) t$value))
== "TITLE")
                if (length(titleIdx) == 0) {
                    debugPrint("handleOpenFile: TITLE не найден")
                    return()
                }
                closeParenIdx <- which(sapply(tokens, function(t) t$value) ==
")")
                validClose <- closeParenIdx[closeParenIdx > titleIdx[1]]
                if (length(validClose) == 0) {
                    debugPrint("handleOpenFile:    закрывающая    скобка    не
найдена")
                    return()
                }
                fileNameToken <- tokens[[min(validClose) - 1]]
                fileName <- fileNameToken$value
                fileName <- gsub("^['\"](.*)['\"]$", "\\1", fileName)
                debugPrint("handleOpenFile: filename =", fileName)
                if (!file.exists(fileName)) {
                    file.create(fileName)
                    debugPrint("handleOpenFile: файл создан")
                } else {
                    debugPrint("handleOpenFile: файл уже существует")
                }
            }
```

```
            if (!is.null(ast_obj$declarations)) {
                for (decl in ast_obj$declarations) {
                    tokens <- decl$tokens
                    if (length(tokens) < 1) next
                    varNameToken <- tokens[[1]]
                    if (varNameToken$type == "numeric_constant") {
                        varNameToken <- tokens[[2]]
                        setValue(varNameToken$value, list())
                        debugPrint("Обработка     декларации     структуры:",
varNameToken$value)
                    } else {
                        if (length(tokens) >= 2 && tokens[[2]]$value == "(") {
                            agg <- createAggregate(tokens)
                            setValue(varNameToken$value, agg)
                            debugPrint("Обработка    декларации    массива:",
varNameToken$value)
                        } else {
                            setValue(varNameToken$value, NA)
                            debugPrint("Обработка     декларации     скалярной
переменной:", varNameToken$value)
                        }
                    }
                    inInit <- FALSE
                    buf <- c()
                    for (iTok in seq_along(tokens)) {
                        t <- tokens[[iTok]]
                        if  (t$type  ==  "keyword"  &&  toupper(t$value)  ==
"INITIAL") {
                            inInit <- TRUE
                            next
                        }
                        if (inInit) {
                            if (t$type == "operator" && t$value == ")") {
                                inInit <- FALSE
                                raw <- paste0(buf, collapse = "")
                                raw <- trimws(raw)
                                raw <- gsub("^['\"](.*)['\"]$", "\\1", raw)
                                debugPrint("Обнаружено     значение     для",
varNameToken$value, ":", raw)
                                maybeN <- parseNumeric(raw)
                                initVal  <-  if  (!is.null(maybeN))  maybeN  else
raw
                                setValue(varNameToken$value, initVal)
                                buf <- c()
                            } else if (!(t$type == "operator" && t$value ==
"(")) {
                                buf <- c(buf, t$value)
                            }
                        }
                    }
                    debugPrint("Установлено   значение",   varNameToken$value,
"=", getValue(varNameToken$value))
                }
            }

            processPutTokens <- function(tokens) {
                outParts <- c()
                i <- 1
                parenCount <- 0
                compoundTokens <- list()
                while (i <= length(tokens)) {
                    t <- tokens[[i]]
                    if (t$value == "(") {
```

```
                        parenCount <- parenCount + 1
                        compoundTokens[[length(compoundTokens) + 1]] <- t
                    } else if (t$value == ")") {
                        parenCount <- parenCount - 1
                        compoundTokens[[length(compoundTokens) + 1]] <- t
                    } else if (t$value == "," && parenCount == 0) {
                        if (length(compoundTokens) > 0) {
                            outParts              <-              c(outParts,
formatValue(getCompoundValue(parseCompoundIdentifier(compoundTokens))))
                            compoundTokens <- list()
                        }
                    } else if (t$type == "string_constant") {
                        if (length(compoundTokens) > 0) {
                            outParts              <-              c(outParts,
formatValue(getCompoundValue(parseCompoundIdentifier(compoundTokens))))
                            compoundTokens <- list()
                        }
                        outParts   <-   c(outParts,   gsub("^['\"](.*)['\"]$",
"\\1", t$value))
                    } else {
                        compoundTokens[[length(compoundTokens) + 1]] <- t
                    }
                    i <- i + 1
                }
                if (length(compoundTokens) > 0) {
                    outParts                  <-                  c(outParts,
formatValue(getCompoundValue(parseCompoundIdentifier(compoundTokens))))
                }
                return(outParts)
            }

            findMatchingParen <- function(tokens, startIndex) {
                count <- 0
                for (i in startIndex:length(tokens)) {
                    if (tokens[[i]]$value == "(") {
                        count <- count + 1
                    } else if (tokens[[i]]$value == ")") {
                        count <- count - 1
                        if (count == 0) {
                            return(i)
                        }
                    }
                }
                return(NA)
            }

            evaluateCondition <- function(tokens) {
                debugPrint("evaluateCondition:   tokens   =",   sapply(tokens,
function(t) t$value))
                if (length(tokens) < 3) {
                    return(FALSE)
                }
                leftVal <- parseLeftExpression(tokens, 1, 1)
                op <- tokens[[2]]$value
                rightVal <- parseCompareValue(tokens[[3]])
                debugPrint("evaluateCondition: leftVal =", leftVal, "op =", op,
"rightVal =", rightVal)
                if (is.null(leftVal) || is.null(rightVal) || is.na(leftVal) ||
is.na(rightVal)) {
                    return(FALSE)
                }
                if (op == "<") {
                    return(leftVal < rightVal)
                }
```

```
            if (op == "<=") {
                return(leftVal <= rightVal)
            }
            if (op == ">") {
                return(leftVal > rightVal)
            }
            if (op == ">=") {
                return(leftVal >= rightVal)
            }
            if (op == "=") {
                return(leftVal == rightVal)
            }
            if (op == "<>") {
                return(leftVal != rightVal)
            }
            return(FALSE)
        }

        evaluateExpression <- function(tokens, env) {
            debugPrint("evaluateExpression:  tokens  =",  sapply(tokens,
function(t) t$value))
            if (length(tokens) == 0) {
                return(NULL)
            }
            result <- NULL
            current_op <- NULL
            for (token in tokens) {
                if      (token$type      %in%      c("numeric_constant",
"string_constant")) {
                    value <- token$value
                    if (token$type == "numeric_constant") {
                        value <- if (grepl("\\.", value)) as.numeric(value)
else as.integer(value)
                    }
                    result <- if (is.null(result)) value else result + (if
(!is.null(current_op) && current_op == "+") value else 0)
                } else if (token$type == "identifier") {
                    value <- getValue(token$value)
                    if (is.null(value) || is.na(value)) value <- 0
                    result <- if (is.null(result)) value else result + (if
(!is.null(current_op) && current_op == "+") value else 0)
                } else if (token$type == "operator" && token$value == "+")
{
                    current_op <- token$value
                }
            }
            debugPrint("evaluateExpression: result =", result)
            return(result)
        }

        parseCompareValue <- function(tok) {
            debugPrint("parseCompareValue:  ток  =",  tok$value,  "тип  =",
tok$type)
            if (tok$type == "numeric_constant") {
                maybe <- parseNumeric(tok$value)
                if (!is.null(maybe)) {
                    return(maybe)
                } else {
                    return(NA)
                }
            } else if (tok$type == "string_constant") {
                return(gsub("^['\"](.*)['\"]$", "\\1", tok$value))
            } else if (tok$type == "identifier") {
                return(getValue(tok$value))
```

```
            }
            return(NA)
        }

        parseLeftExpression <- function(tokens, startPos, endPos) {
            debugPrint("parseLeftExpression:          tokens          =",
sapply(tokens[startPos:endPos], function(t) t$value))
            if (startPos > length(tokens) || startPos > endPos) {
                return(NA)
            }
            if (toupper(tokens[[startPos]]$value) == "MOD") {
                debugPrint("parseLeftExpression: обнаружен вызов MOD")
                if ((endPos - startPos + 1) < 6) {
                    return(NA)
                }
                arg1 <- getValue(tokens[[4]]$value)
                arg2 <- parseNumeric(tokens[[6]]$value)
                debugPrint("MOD args:", arg1, arg2)
                if (is.null(arg1) || is.null(arg2)) {
                    return(NA)
                }
                return(as.numeric(arg1) %% arg2)
            } else {
                varName <- tokens[[startPos]]$value
                v <- getValue(varName)
                maybeNum <- parseNumeric(v)
                debugPrint("parseLeftExpression: var", varName, "=", v)
                return(if (!is.null(maybeNum)) maybeNum else v)
            }
        }

        interpretIf <- function(tokens) {
            debugPrint("interpretIf: tokens =", sapply(tokens, function(t)
t$value))
            condOpPos <- NA
            for (i in seq_along(tokens)) {
                if (tokens[[i]]$type == "operator" && tokens[[i]]$value
%in% c("=", "<", "<=", ">", ">=", "<>")) {
                    condOpPos <- i
                    break
                }
            }
            if (is.na(condOpPos) || condOpPos + 1 > length(tokens)) {
                return(list(action = "normal"))
            }
            rightVal <- parseCompareValue(tokens[[condOpPos + 1]])
            leftVal <- parseLeftExpression(tokens, 2, condOpPos - 1)
            op <- tokens[[condOpPos]]$value
            pass <- FALSE
            if (!is.null(leftVal) && !is.null(rightVal) && !is.na(leftVal)
&& !is.na(rightVal)) {
                if (op == "=") {
                    pass <- (leftVal == rightVal)
                } else if (op == "<") {
                    pass <- (leftVal < rightVal)
                } else if (op == "<=") {
                    pass <- (leftVal <= rightVal)
                } else if (op == ">") {
                    pass <- (leftVal > rightVal)
                } else if (op == ">=") {
                    pass <- (leftVal >= rightVal)
                } else if (op == "<>") pass <- (leftVal != rightVal)
            }
            debugPrint("interpretIf: условие =", pass)
```

```
                idxThen <- which(sapply(tokens, function(t) t$value) == "THEN")
                if (length(idxThen) == 0) {
                    return(list(action = "normal"))
                }
                idxThen <- idxThen[1]
                if (any(sapply(tokens, function(t) t$value) == "ELSE")) {
                    idxElse <- which(sapply(tokens, function(t) t$value) ==
"ELSE")[1]
                    branchTokens <- if (pass) tokens[(idxThen + 1):(idxElse -
1)] else tokens[(idxElse + 1):length(tokens)]
                } else {
                    branchTokens <- if (pass) tokens[(idxThen +
1):length(tokens)] else list()
                }
                if (length(branchTokens) == 0) {
                    return(list(action = "normal"))
                }
                firstWord <- toupper(trimws(branchTokens[[1]]$value))
                if (firstWord %in% c("LEAVE", "CONTINUE")) {
                    return(list(action = tolower(firstWord)))
                }
                interpretPut(branchTokens)
                return(list(action = "normal"))
            }

            interpretPut <- function(tokens) {
                debugPrint("interpretPut: tokens =", sapply(tokens,
function(t) t$value))
                opens <- which(sapply(tokens, function(t) t$value) == "(")
                closes <- which(sapply(tokens, function(t) t$value) == ")")
                if (length(opens) > 0 && length(closes) > 0 && opens[1] <
closes[length(closes)]) {
                    mid <- tokens[(opens[1] + 1):(closes[length(closes)] - 1)]
                    out <- processPutTokens(mid)
                    emit(paste(out, collapse = " "))
                }
            }

            interpretAssignment <- function(tokens) {
                debugPrint("interpretAssignment: tokens =", sapply(tokens,
function(t) t$value))
                eqPos <- which(sapply(tokens, function(t) t$value) == "=")[1]
                lhsTokens <- tokens[1:(eqPos - 1)]
                rhsTokens <- tokens[(eqPos + 1):(length(tokens) - 1)]
                raw <- paste(sapply(rhsTokens, function(t) t$value), collapse
= " ")
                maybeN <- parseNumeric(raw)
                if (!is.null(maybeN)) {
                    setCompoundValue(parseCompoundIdentifier(lhsTokens),
maybeN)
                    debugPrint("interpretAssignment: присваиваем число",
maybeN)
                } else {

    emit("=== Интерпретация завершена ===")

    return("Code 1")
}

codeType <- determineCodeType(ast_obj)
```