

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  
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 разработке семантического анализатора, основная задача которого – проверка исходного кода программы на соответствие семантическим нормам языка. В процессе работы анализатора осуществляется выявление логических ошибок, связанных с типизацией и использованием идентификаторов, а также производится накопление информации о типах, необходимой для последующих этапов компиляции, таких как генерация кода.

Семантический анализ играет ключевую роль в обеспечении корректности программного обеспечения, поскольку на этом этапе проверяется соответствие между операторами и операндами, а также их типами в рамках спецификации языка. Такой анализ позволяет обнаруживать ошибки на ранней стадии, что существенно снижает вероятность возникновения проблем во время выполнения программы. При этом допускается неявное приведение типов, если этого требует спецификация, либо генерируются сообщения об ошибках в случаях обнаружения несовместимости типов или некорректного использования операторов.

Важной задачей анализатора является контроль за правильностью использования идентификаторов. Создаваемая таблица символов аккумулирует информацию о переменных, включая их типы, начальные значения и факт предварительного объявления. Это позволяет не только выявлять попытки обращения к неинициализированным или не объявленным элементам, но и способствует выработке мер по предотвращению логических ошибок, что положительно сказывается на надежности разрабатываемого программного продукта.

Реализация семантического анализатора строится на данных, полученных в ходе синтаксического анализа, что дает возможность глубже изучить структурную организацию исходного текста программы. Такой подход обеспечивает детальное понимание работы компилятора, позволяя на основе выявленных структур формировать необходимые сведения для последующей трансформации кода в машинную форму. В результате разрабатывается мощный инструмент, который не только обеспечивает строгий контроль за соблюдением семантических правил, но и повышает качество конечного программного продукта, гарантируя своевременное обнаружение и обработку ошибок.

## 2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Семантический анализ – это важнейший этап трансляции программ, направленный на проверку смысловой корректности исходного кода. В отличие от синтаксического анализа, который занимается построением грамматической структуры программы, семантический анализ отвечает за выявление логических ошибок, связанных с типизацией, использованием идентификаторов и другими аспектами, влияющими на смысловую нагрузку программы [1]. На данном этапе особое внимание уделяется тому, чтобы каждое выражение и каждая операция соответствовали строгим требованиям языка, что позволяет обнаруживать ошибки, способные привести к некорректному выполнению программы, даже если синтаксис сформирован правильно.

Одним из основных направлений семантического анализа является проверка типов. Компилятор на этом этапе удостоверяется, что каждому оператору соответствуют операнды, обладающие допустимыми типами. Например, если язык программирования не допускает использование действительного числа в качестве индекса массива, то попытка такой операции приводит к генерации сообщения об ошибке. Подобная проверка обеспечивает корректное применение операторов и предотвращает выполнение некорректных вычислений, что в свою очередь способствует стабильной работе программного продукта [2]. Многие современные языки программирования реализуют механизм неявного приведения типов, позволяющий автоматически преобразовывать один тип данных в другой при выполнении арифметических или логических операций. Однако автоматическое преобразование требует строгого контроля, так как оно может привести к потере точности или появлению логических ошибок, если применяется без должного анализа.

Другим важным компонентом семантического анализа является построение таблицы символов – структуры данных, которая аккумулирует сведения об идентификаторах, их типах, областях видимости и начальных значениях. Такая таблица позволяет компилятору контролировать корректность использования переменных, функций и других сущностей, предотвращая ситуации, когда идентификаторы используются без предварительного объявления или инициализации. Помимо обнаружения ошибок, таблица символов служит базой для последующих этапов трансляции, способствуя оптимизации кода и повышению эффективности работы программы [3].

Семантический анализатор выполняет интеграционную функцию, связывая результаты синтаксического анализа с этапом генерации кода. Информация о типах и структуре программы, собранная на предыдущих этапах, используется для формирования промежуточного представления, которое затем преобразуется в машинный код. Такой комплексный подход позволяет не только обнаружить ошибки на ранней стадии, но и оптимизировать дальнейшие процессы трансляции, что существенно

повышает общую производительность компилятора и надежность конечного программного продукта [4].

Кроме того, современные семантические анализаторы включают в себя методы контроля областей видимости, проверки вызовов функций и анализа константных выражений. Эти методы позволяют учитывать контекст использования идентификаторов и корректно обрабатывать даже сложные конструкции, где могут возникать тонкие логические ошибки. Дополнительный анализ условий ветвления и циклических зависимостей между переменными помогает улучшить оптимизацию и обеспечить корректность выполнения программ, особенно в системах с жесткой типизацией, где каждая ошибка может повлечь за собой серьезные последствия [5].

Более того, семантический анализ охватывает вопросы, связанные с разрешением перегрузки операторов и функций, а также с определением областей видимости и жизненного цикла переменных. Анализатор проверяет, чтобы использование функций, методов и операторов соответствовало определенным в языке правилам, исключая неоднозначности, которые могут возникнуть в результате перегрузки имен или неправильного связывания вызовов с определениями. Такой контроль обеспечивает точность и однозначность интерпретации кода, что является критически важным для языков с развитой системой типов и сложными структурами наследования. Благодаря этому достигается более высокая степень надежности компилятора, а также упрощается процесс отладки и поддержки программных систем.

В дополнение к базовой проверке типов и идентификаторов, семантический анализатор осуществляет оптимизацию промежуточного представления программы. Анализ полученной информации позволяет выявлять неэффективные конструкции, устранять избыточные вычисления и проводить другие трансформации, направленные на улучшение производительности конечного машинного кода. Такой подход не только снижает количество ошибок на этапах выполнения программы, но и значительно ускоряет процесс трансляции, обеспечивая эффективное использование ресурсов системы. В совокупности, все методы семантического анализа способствуют созданию высококачественных, надежных и оптимизированных программных продуктов, способных удовлетворить требования самых сложных вычислительных задач.

### 3 РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ

В ходе выполнения лабораторной работы семантический анализатор принимает на вход текстовое синтаксическое дерево (*CST*), сохранённое в файле *st\_tree.txt* в формате *JSON*. Это дерево содержит подробное описание исходного кода, сформированное на этапе синтаксического анализа, где все токены и грамматические конструкции представлены в виде вложенной иерархии. На основе этой информации анализатор проводит проверку семантической корректности, выявляя ошибки, связанные с неправильным использованием типов и идентификаторов.

Программа последовательно обрабатывает *CST*, выделяя блоки объявлений и операторов, что позволяет формировать таблицу символов, где для каждой переменной фиксируется её тип и начальное значение (если оно задано). Семантический анализатор проверяет код по следующим основным направлениям:

#### 1 Ошибки соответствия типов:

Если переменной типа *CHAR* вместо строки присваивается числовое значение, генерируется сообщение вида: «Переменная *name*: Ошибка: для *CHAR/CHARACTER* ожидается строка, получено числовое: 123».

Анализатор проверяет корректность значений для типа *BIT* — если значение не содержит завершающий символ *B* или содержит недопустимые символы, выводится ошибка: «Переменная *flag*: Ошибка: для *BIT(...)* нужно двоичное значение с *B* на конце, получено: 102B».

При обработке переменных типа *FIXED BINARY* ожидается строго целочисленное значение, и если обнаруживается число с десятичной точкой, регистрируется ошибка: «Переменная *counter*: Ошибка: для *FIXED BINARY* ожидается целое число, получено: 12.34».

Для типов *FIXED DECIMAL*, *FLOAT* или *DECIMAL FLOAT* анализатор требует корректный числовой формат, допускающий наличие десятичной точки, а при неверном формате, например, при передаче строки, выводится сообщение: «Переменная *price*: Ошибка: для *FIXED DECIMAL* ожидается числовое значение, получено: *abc*».

Для типа *COMPLEX FLOAT* значение должно иметь формат *a+bI*, иначе сообщение будет таким: «Переменная *complex\_num*: Ошибка: для *COMPLEX FLOAT* ожидается формат *a+bI*, получено: 3+4».

#### 2 Ошибки, связанные с отсутствием объявления идентификаторов:

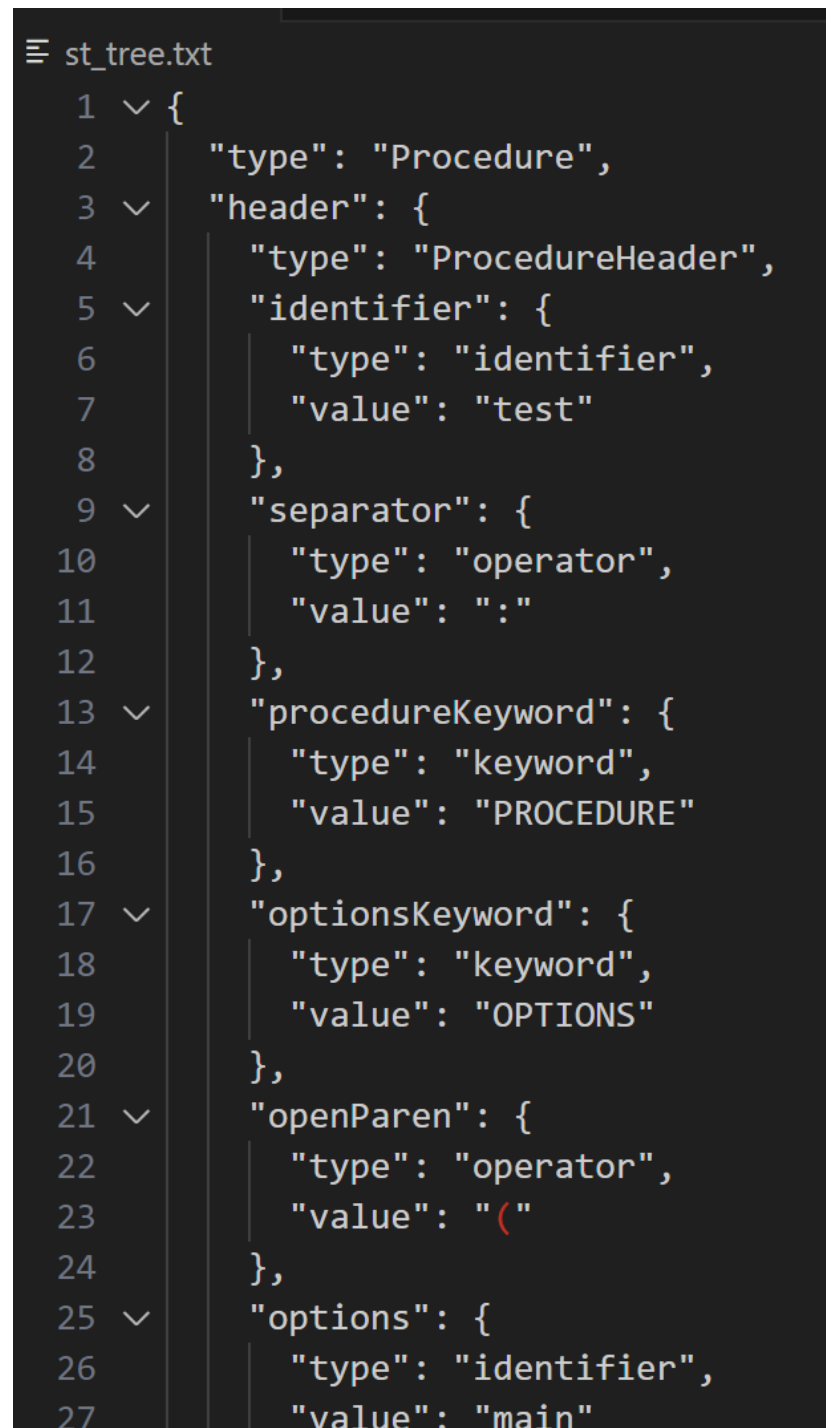
Если анализатор обнаруживает использование идентификатора, который не был предварительно объявлен и отсутствует в таблице символов, в отчёт вносится предупреждение вида: «Предупреждение: идентификатор *temp* использован, но не объявлен (как переменная/метка/процедура).»

#### 3 Ошибки в синтаксическом контексте:

Если в блоке объявления отсутствует корректный идентификатор, например, после ключевого слова *DECLARE*, анализатор фиксирует ошибку: «Ошибка: *DECLARE* не содержит нормального идентификатора!»

Все обнаруженные ошибки и предупреждения записываются в файл *semantic\_errors.txt*. При отсутствии ошибок в этом файле записывается сообщение «Семантических ошибок не обнаружено», что свидетельствует о корректности исходного кода с точки зрения семантики.

На рисунке 3.1 представлено входное текстовое дерево (CST), которое подается на вход семантическому анализатору. Это дерево демонстрирует, как отдельные токены группируются в грамматические конструкции и позволяет наглядно проследить структуру исходного кода.



```

≡ st_tree.txt
1  {
2    "type": "Procedure",
3    "header": {
4      "type": "ProcedureHeader",
5      "identifier": {
6        "type": "identifier",
7        "value": "test"
8      },
9      "separator": {
10     "type": "operator",
11     "value": ":"
12   },
13   "procedureKeyword": {
14     "type": "keyword",
15     "value": "PROCEDURE"
16   },
17   "optionsKeyword": {
18     "type": "keyword",
19     "value": "OPTIONS"
20   },
21   "openParen": {
22     "type": "operator",
23     "value": "("
24   },
25   "options": {
26     "type": "identifier",
27     "value": "main"

```

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

На рисунке 3.2 показан пример вывода ошибок, полученный в файле *semantic\_errors.txt*. Здесь видно, как анализатор фиксирует различные семантические нарушения: например, обнаруживаются ошибки, связанные с несоответствием типов – вместо ожидаемой строки для переменной типа *CHAR* было передано числовое значение, для переменной типа *BIT* обнаружен некорректный формат, где присутствуют недопустимые символы, а также для типа *FIXED DECIMAL* зафиксировано неверное числовое значение. Помимо этого, анализатор регистрирует предупреждения об использовании идентификаторов, которые не были предварительно объявлены в таблице символов, что позволяет быстро обнаружить пропущенные объявления или опечатки в именах переменных.

```
Найдены семантические ошибки:
- Переменная S1: Ошибка: для CHAR/CHARACTER ожидается строка, получено числовое: 123
- Переменная S3: Ошибка: для BIT(...) нужно двоичное значение с 'B' на конце, получено: 010210101B
- Переменная S4: Ошибка: для FIXED BINARY ожидается целое число, получено: 12345.67
- Переменная S6: Ошибка: для FLOAT ожидается вещественное число, получено: abc
- Переменная S7: Ошибка: для COMPLEX FLOAT ожидается a+bI, получено: 3+4
- Предупреждение: идентификатор undeclared_var использован, но не объявлен (как переменная/метка/процедура)
```

Рисунок 3.2 – Пример текстового представления отчёта об обнаруженных семантических ошибках

Таким образом, разработанный семантический анализатор не только преобразует входное текстовое дерево в детальную информацию о семантических ошибках исходного кода, но и демонстрирует свою эффективность в выявлении критических нарушений. Выявленные ошибки охватывают как нарушения соответствия типов, так и неправильное использование идентификаторов. Это позволяет оперативно устранять недочёты, улучшая качество и надёжность последующего процесса генерации машинного кода. Более того, возможность раннего обнаружения ошибок на этапе семантического анализа значительно сокращает затраты времени на отладку и тестирование, способствуя созданию эффективных и безопасных программных систем. Подобный подход обеспечивает комплексную защиту от логических ошибок, делая систему компиляции максимально надёжной и устойчивой к ошибкам, возникающим в процессе разработки.



## ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была решена задача разработки семантического анализатора, направленного на проверку исходного кода программы на соответствие семантическим нормам языка. На вход анализатора поступает текстовое синтаксическое дерево (*CST*), сформированное на предыдущем этапе синтаксического анализа, что позволяет восстановить детальную структуру исходного кода и выявить семантические ошибки уже на ранних этапах трансляции.

Особое внимание уделялось обнаружению логических ошибок, связанных с типизацией и использованием идентификаторов. Разработанный анализатор проверяет корректность значений для различных типов (например, *CHAR*, *BIT*, *FIXED BINARY*, *FIXED DECIMAL*, *FLOAT*, *DECIMAL FLOAT*, *COMPLEX FLOAT*) и регистрирует ошибки в случае несоответствия типов или неверного формата входных данных. Кроме того, инструмент фиксирует случаи использования не объявленных идентификаторов, что существенно облегчает процесс отладки и повышает надежность программы.

Разработанная программа демонстрирует возможность получения подробного отчёта о семантических ошибках, который сохраняется в виде текстового файла (*semantic\_errors.txt*). Такой подход позволяет не только оперативно выявлять и устранять ошибки, но и обеспечивает корректную генерацию машинного кода для дальнейшей трансляции. Модульная структура реализации семантического анализатора обеспечивает его гибкость и масштабируемость, что открывает перспективы для дальнейшего расширения функциональности за счёт поддержки дополнительных семантических конструкций и интеграции с последующими этапами трансляции, такими как оптимизация программного кода.

Таким образом, проведённое исследование подтвердило значимость семантического анализа на ранних этапах трансляции программ. Разработанный семантический анализатор позволяет детально структурировать исходный код и служит надёжной основой для дальнейшей обработки, что является важным условием для создания полноценных систем компиляции. Полученные результаты могут стать отправной точкой для дальнейшего развития инструмента и реализации полноценного компилятора, способного обеспечивать высокое качество и надёжность конечного программного продукта.

## 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

```
if (!require(jsonlite)) {
  install.packages("jsonlite", repos = "http://cran.us.r-project.org")
  library(jsonlite)
}

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

output_file <- "semantic_errors.txt"
semantic_errors <- c()

type_keywords <- c(
  "CHAR", "CHARACTER", "BIT", "FIXED", "DECIMAL", "BINARY",
  "FLOAT", "COMPLEX", "LABEL", "ENTRY", "FILE",
  "POINTER", "VAR", "STATIC", "BASED", "INITIAL", "LIKE"
)

known_labels_or_procs <- c("test", "geometric_mean", "simple_procedure")

extractTypeAndInitialValue <- function(tokens) {
  typeTokens <- c()
  initValue <- NULL

  inInitial <- FALSE
  bufInit <- c()

  for (tok in tokens) {
    if (tok$type == "keyword" && toupper(tok$value) == "INITIAL") {
      inInitial <- TRUE
      next
    }
    if (inInitial) {
      if (tok$type == "operator" && tok$value == "(") {
        next
      } else if (tok$type == "operator" && tok$value == ")") {
        inInitial <- FALSE
        rawInit <- paste0(bufInit, collapse = "")
        rawInit <- trimws(rawInit)

        if (grepl("^['\"]", rawInit) && grepl("['\"]$", rawInit)) {
          rawInit <- substring(rawInit, 2, nchar(rawInit) - 1)
        }

        initValue <- rawInit
        bufInit <- c()
      } else {
        bufInit <- c(bufInit, tok$value)
      }
    } else {
      if (!(toupper(tok$value) %in% c(
        "DECLARE", "VAR", "STATIC", "BASED", "INITIAL",
        "LABEL", "ENTRY", "POINTER", "FILE", "LIKE", "CHARACTER"
      ))) {

```

```

        typeTokens <- c(typeTokens, tok$value)
    }
}

rawType <- paste(typeTokens, collapse = " ")
rawType <- gsub("\\s*\\(\\s*", "(", rawType)
rawType <- gsub("\\s*\\)\\s*", ")", rawType)
rawType <- gsub("\\s*", "", rawType)

list(type = rawType, init = initValue)
}

checkTypeCompatibility <- function(declaredType, initVal) {
  if (is.null(initVal)) {
    return(NULL)
  }

  if (grepl("CHAR", declaredType, ignore.case = TRUE)) {
    if (!is.na(suppressWarnings(as.numeric(initVal)))) {
      return(paste("Ошибка: для CHAR/CHARACTER ожидается строка, получено
числовое:", initVal))
    }
  }

  if (grepl("BIT", declaredType, ignore.case = TRUE)) {
    valNoSpace <- gsub("\\s+", "", initVal)
    if (!grepl("^[01]+B$", valNoSpace)) {
      return(paste("Ошибка: для BIT(...) нужно двоичное значение с 'B' на
конце, получено:", initVal))
    }
  }

  if (grepl("FIXED BINARY", declaredType, ignore.case = TRUE)) {
    if (!grepl("^[0-9]+$", initVal)) {
      return(paste("Ошибка: для FIXED BINARY ожидается целое число,
получено:", initVal))
    }
  }

  if (grepl("FIXED DECIMAL", declaredType, ignore.case = TRUE)) {
    if (!grepl("^[0-9]+(\\. [0-9]+)?$", initVal)) {
      return(paste("Ошибка: для FIXED DECIMAL ожидается числовое
значение, получено:", initVal))
    }
  }

  if (grepl("\\bFLOAT\\b", declaredType, ignore.case = TRUE) &&
    !grepl("COMPLEX FLOAT", declaredType, ignore.case = TRUE)) {
    if (!grepl("^[0-9]+(\\. [0-9]+)?$", initVal)) {
      return(paste("Ошибка: для FLOAT ожидается вещественное число,
получено:", initVal))
    }
  }

  if (grepl("COMPLEX FLOAT", declaredType, ignore.case = TRUE)) {
    if (!grepl("^[0-9]+\\+ [0-9]+I$", initVal)) {
      return(paste("Ошибка: для COMPLEX FLOAT ожидается a+bI, получено:",
initVal))
    }
  }

  if (grepl("DECIMAL FLOAT", declaredType, ignore.case = TRUE)) {
    if (!grepl("^[0-9]+(\\. [0-9]+)?$", initVal)) {

```

```

        return(paste("Ошибка: для DECIMAL FLOAT ожидается вещественное
число, получено:", initVal))
    }
}

return(NULL)
}

symbol_table <- list()

processDeclarationTokens <- function(tokens) {
  idx_identifier <- NULL
  for (i in seq_along(tokens)) {
    if (tokens[[i]]$type == "identifier" &&
        !(toupper(tokens[[i]]$value) %in% toupper(type_keywords))) {
      idx_identifier <- i
      break
    }
  }
  if (is.null(idx_identifier)) {
    semantic_errors <- c(semantic_errors, "Ошибка: DECLARE не содержит
нормального идентификатора!")
    return()
  }

  varName <- tokens[[idx_identifier]]$value
  leftover <- tokens[-(1:idx_identifier)]

  info <- extractTypeAndInitialValue(leftover)
  err <- checkTypeCompatibility(info$type, info$init)
  if (!is.null(err)) {
    semantic_errors <- c(
      semantic_errors,
      paste0("Переменная ", varName, ": ", err)
    )
  }

  symbol_table[[varName]] <- list(type = info$type, init = info$init)
}

if (!is.null(tree$declarations)) {
  for (decl in tree$declarations) {
    tokens <- decl$tokens
    processDeclarationTokens(tokens)
  }
}

if (!is.null(tree$statements)) {
  for (stmt in tree$statements) {
    tokens <- stmt$tokens
    if (length(tokens) == 0) next

    if (tokens[[1]]$type == "keyword" && toupper(tokens[[1]]$value) ==
"DECLARE") {
      processDeclarationTokens(tokens)
    } else {
      i <- 1
      while (i <= length(tokens)) {
        if (tokens[[i]]$type == "identifier") {
          ident <- tokens[[i]]$value

          if (toupper(ident) %in% toupper(type_keywords)) {
            i <- i + 1
            next
          }
        }
      }
    }
  }
}

```

```

    }
    if (ident %in% known_labels_or_procs) {
      i <- i + 1
      next
    }

    if (!(ident %in% names(symbol_table))) {
      semantic_errors <- c(
        semantic_errors,
        paste(
          "Предупреждение: идентификатор", ident,
          "использован, но не объявлен (как
переменная/метка/процедура)."
        )
      )
    }

    j <- i + 1
    while (j <= length(tokens)) {
      if (tokens[[j]]$type == "operator" &&
        tokens[[j]]$value %in% c(".", "(", ")", ":", ",", "))
{
      j <- j + 1
    } else if (tokens[[j]]$type %in% c("numeric_constant",
"identifier")) {
      j <- j + 1
    } else {
      break
    }
  }

  i <- j
  next
}

i <- i + 1
}
}
}

if (length(semantic_errors) == 0) {
  cat("Семантических ошибок не обнаружено.\n", file = output_file)
  message("Семантических ошибок не обнаружено.")
} else {
  cat("Найдены семантические ошибки:\n\n", file = output_file)
  for (err in semantic_errors) {
    cat("- ", err, "\n", file = output_file, append = TRUE)
  }
  message("Семантические ошибки записаны в файл 'semantic_errors.txt'.")
}

```