Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

ОТЧЕТ

к лабораторной работе №4

на тему

**СЕМАНТИЧЕСКИЙ АНАЛИЗАТОР**

Выполнил: студент гр. 253505

Снежко М. А.

Проверил: ассистент кафедры

информатики Гриценко Н. Ю.

Минск 2025

**СОДЕРЖАНИЕ**

[1 Цель работы 3](https://docs.google.com/document/d/17oy0HJ6nAz99n80cSTAL3yLtqgruM1Hfs6vvzZcnsOo/edit?pli=1&tab=t.0#heading=h.30j0zll)

[2 Краткие теоретические сведения 4](https://docs.google.com/document/d/17oy0HJ6nAz99n80cSTAL3yLtqgruM1Hfs6vvzZcnsOo/edit?pli=1&tab=t.0#heading=h.1fob9te)

[3 Результаты работы 5](https://docs.google.com/document/d/17oy0HJ6nAz99n80cSTAL3yLtqgruM1Hfs6vvzZcnsOo/edit?pli=1&tab=t.0#heading=)

[Заключение](https://docs.google.com/document/d/17oy0HJ6nAz99n80cSTAL3yLtqgruM1Hfs6vvzZcnsOo/edit?pli=1&tab=t.0#heading=) 7

[Список использованных источников](https://docs.google.com/document/d/17oy0HJ6nAz99n80cSTAL3yLtqgruM1Hfs6vvzZcnsOo/edit?pli=1&tab=t.0#heading=h.1t3h5sf) 8

[Приложение А (обязательное) Листинг программного кода](https://docs.google.com/document/d/17oy0HJ6nAz99n80cSTAL3yLtqgruM1Hfs6vvzZcnsOo/edit?pli=1&tab=t.0#heading=h.2s8eyo1) 9

**1 ЦЕЛЬ РАБОТЫ**

Целью работы является создание семантического анализатора, предназначенного для проверки соответствия исходного кода семантическим правилам выбранного языка программирования. Анализатор должен не только выявлять логические и типовые ошибки в программе, но и формировать семантическое представление кода, отражающее смысловые связи между его элементами. Это представление может использоваться на этапах оптимизации, генерации кода или статического анализа. Помимо фиксации ошибок, анализатор должен предоставлять подробные сообщения об их природе, контексте возникновения и точной позиции в коде, что значительно упрощает их устранение и улучшает качество разработки.

В процессе реализации планируется использование таблиц символов, контекстной информации и правил вывода типов, что позволит эффективно анализировать идентификаторы, выражения и конструкции с учетом их области видимости, типов данных и допустимых операций. Особое внимание уделяется обработке объявлений, определений, совместимости типов и корректности вызовов функций или методов. Результатом работы анализатора должно быть формальное представление семантики программы, пригодное для дальнейших этапов компиляции, таких как промежуточное представление, оптимизация и генерация исполняемого кода. Анализатор должен быть расширяемым, чтобы поддерживать новые типы, конструкции и правила языка.

Для построения семантического анализатора может быть использован подход, основанный на промежуточных представлениях (например, абстрактных синтаксических деревьях), к которым добавляется семантическая информация, либо на правилах семантических действий, выполняемых во время обхода дерева разбора. Выбор метода зависит от структуры языка и требований к точности анализа. В результате будет разработан полноценный семантический анализатор, способный интерпретировать смысл программы, находить глубокие логические и типовые ошибки и предоставлять исчерпывающую информацию для их исправления. Такой анализатор является важнейшим компонентом в системах компиляции, статического анализа и интеллектуальной разработки программного обеспечения.

**2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Семантический анализ является ключевым этапом в процессе компиляции или интерпретации программного кода. На этом этапе проверяется, насколько исходный текст программы соответствует семантическим правилам, определённым спецификацией языка. В результате анализа формируется структурированное представление программы, обычно в виде абстрактного синтаксического дерева (*AST*), которое отображает иерархию конструкций кода и их смысловые связи. Это дерево играет важную роль на последующих этапах обработки, таких как оптимизация или генерация машинного кода. [1]

Одним из распространённых подходов к реализации семантических анализаторов является использование таблиц символов и правил проверки типов. Эти анализаторы используют информацию о типах данных, областях видимости и других семантических аспектах, чтобы обеспечить корректность программы. Главная особенность таких анализаторов заключается в том, что они принимают решения о допустимости операций и выражений на основе текущего контекста и состояния таблицы символов. Для этого строится таблица символов, которая создаётся на основе анализа программы. Однако для корректной работы семантических анализаторов необходимо, чтобы программа была свободна от синтаксических ошибок, так как они могут препятствовать точному определению семантических связей. Кроме того, для разрешения конфликтов и неоднозначностей применяется анализ контекста, который позволяет уточнить смысл конструкций программы.

Для работы семантического анализатора необходимы два ключевых компонента: таблица символов и правила проверки типов. Таблица символов содержит информацию о всех идентификаторах в программе, включая их типы, области видимости и другие свойства. Правила проверки типов определяют допустимость операций между различными типами данных. Эти компоненты используются для обеспечения семантической корректности программы, где каждая операция и выражение проверяются на соответствие правилам языка. Благодаря своей способности выявлять смысловые ошибки, семантические анализаторы широко применяются в практике разработки компиляторов и интерпретаторов. Они обеспечивают надёжную проверку программного кода, если программа соответствует необходимым требованиям.

**3 РЕЗУЛЬТАТЫ РАБОТЫ**

В ходе выполнения лабораторной работы был реализован сементический анализатор языка *F#* на языке *Swift*. На рисунке 3.1 представлен часть анализируемого кода.

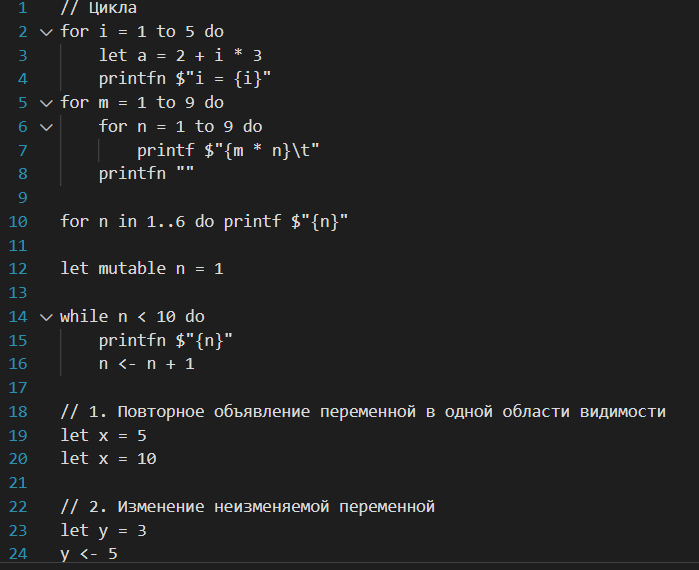
****

Рисунок 3.1 – Часть анализируемого кода

Результатом работы разработанной программы является вывод семантических ошибок на консоль если они есть.

На рисунке 3.2 представлена вывод информации об ошибках.

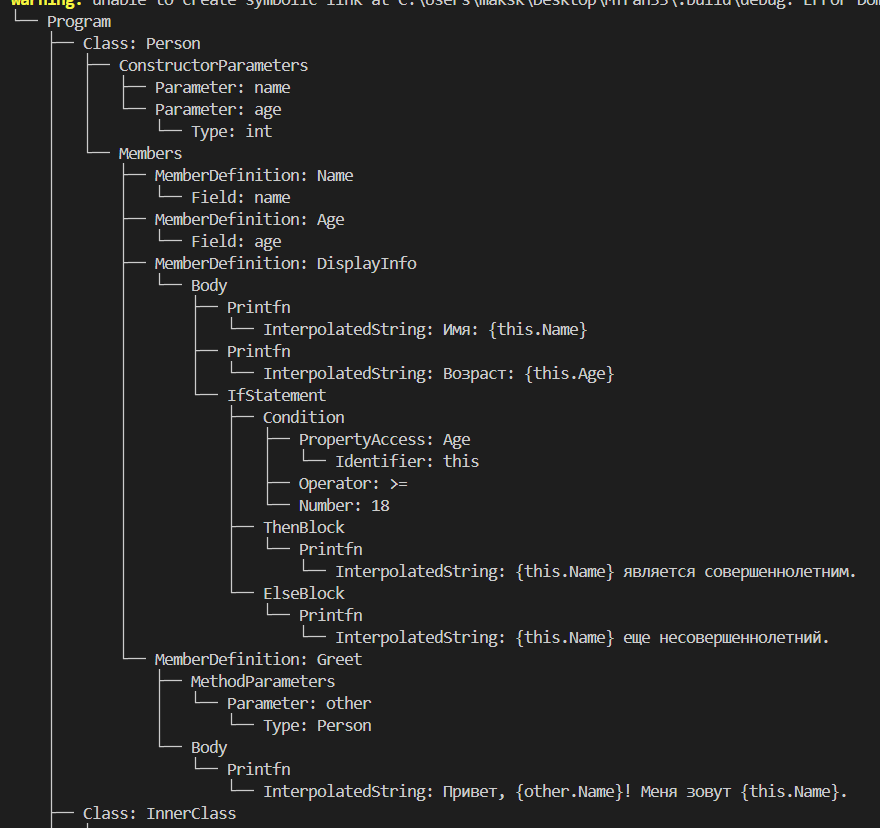


Рисунок 3.2 – Вывод семантических ошибок на консоль

В ходе выполнения работы был успешно создан и протестирован семантический анализатор. Анализатор показал высокую производительность при обработке разнообразных синтаксических конструкций, включая выражения, функции и подключение модулей.

Разработанный парсер не только корректно обрабатывает валидный код, но и демонстрирует устойчивость к различным видам некорректного ввода. Такие характеристики делают его подходящим для применения в реальных проектах, таких как компиляторы, интерпретаторы или инструменты статического анализа кода. Таким образом, выполненная работа подтвердила эффективность выбранного подхода и продемонстрировала практическую применимость созданного решения.

**ЗАКЛЮЧЕНИЕ**

В ходе выполнения лабораторной работы была разработана и реализована программа семантического анализа, предназначенная для проверки смысловой корректности программного кода. Анализатор использует абстрактное синтаксическое дерево (*AST*), построенное на предыдущем этапе синтаксического анализа, и таблицу символов для выявления семантических ошибок, таких как несоответствие типов, использование необъявленных переменных и нарушение правил области видимости.

Одним из ключевых достижений работы стало успешное внедрение механизма проверки типов, обеспечивающего корректность операций между различными типами данных. Были реализованы правила для проверки выражений, условных операторов, циклов и функций, что подтвердило универсальность подхода. Кроме того, программа была дополнена функцией визуализации аннотированного *AST*, что значительно упрощает анализ и отладку семантической структуры программы.

Таким образом, выполненная работа позволила углубить понимание принципов работы семантических анализаторов и методов их реализации, а также продемонстрировать практическое применение теоретических знаний о семантических правилах и алгоритмах анализа. Полученный результат может быть использован как база для создания более сложных систем обработки программного кода, таких как компиляторы или интерпретаторы. В перспективе возможно расширение функциональности анализатора, например, за счёт поддержки дополнительных языковых конструкций или интеграции с другими этапами компиляции для оптимизации и генерации кода.

В заключение стоит отметить, что выполненная работа не только закрепила теоретические знания, но и предоставила ценный практический опыт в разработке семантических анализаторов. Это важный шаг на пути к созданию полноценных систем обработки программного кода, которые могут найти применение в образовательных, исследовательских и промышленных целях.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1]Семантический анализ [Электронный ресурс]. – Режим доступа: https://cropas.by/seo-slovar/semanticheskij-analiz/.

[2] What is Semantic Analysis [Электронный ресурс]. – Режим доступа: https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-semantic-analysis/.

**ПРИЛОЖЕНИЕ А (обязательное) Листинг программного кода**

class SemanticAnalyzer {

private var symbolTable = [String: SymbolInfo]()

private var currentScope = "global"

private var errors: [String] = []

private var reportedErrors = Set<String>()

struct SymbolInfo {

let name: String

let type: String

let isMutable: Bool

let scope: String

let line: Int

var isFunction: Bool

var parameters: [ParameterInfo]?

struct ParameterInfo {

let name: String

var type: String

}

}

func analyze(node: ASTNode) -> [String] {

errors.removeAll()

visit(node: node)

return errors

}

private func visit(node: ASTNode) {

switch node.type {

case "Program":

visitProgram(node)

case "FunctionDeclaration":

visitFunctionDeclaration(node)

case "VariableDeclaration":

visitVariableDeclaration(node)

case "Assignment":

visitAssignment(node)

case "FunctionCall":

visitFunctionCall(node)

case "IfStatement":

visitIfStatement(node)

case "WhileStatement":

visitWhileStatement(node)

case "ForInStatement", "ForToStatement":

visitForStatement(node)

case "Class":

visitClassDeclaration(node)

case "BinaryOp":

visitBinaryOperation(node)

case "Return":

visitReturn(node)

default:

// Всегда посещаем дочерние узлы

for child in node.children {

visit(node: child)

* }

}