Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

ОТЧЕТ

к лабораторной работе №3

на тему

**СИНТАКСИЧЕСКИЙ 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=) 8

[Список использованных источников](https://docs.google.com/document/d/17oy0HJ6nAz99n80cSTAL3yLtqgruM1Hfs6vvzZcnsOo/edit?pli=1&tab=t.0#heading=h.1t3h5sf) 9

[Приложение А (обязательное) Листинг программного кода](https://docs.google.com/document/d/17oy0HJ6nAz99n80cSTAL3yLtqgruM1Hfs6vvzZcnsOo/edit?pli=1&tab=t.0#heading=h.2s8eyo1) 10

**1 ЦЕЛЬ РАБОТЫ**

Целью работы является создание синтаксического анализатора, предназначенного для проверки соответствия исходного кода программы синтаксическим правилам выбранного языка программирования. Анализатор должен не только находить ошибки в структуре кода, но и строить синтаксическое дерево, которое отображает иерархию грамматических конструкций программы. Это дерево представляет собой удобную для дальнейшей обработки структуру, применимую на последующих этапах компиляции или интерпретации. Кроме того, анализатор обязан точно выявлять синтаксические ошибки и предоставлять детальные сообщения о их расположении и сути, что значительно упрощает процесс отладки и исправления кода.

В процессе реализации планируется применение одного из табличных методов синтаксического анализа, таких как *LL*-метод, *LR*-метод, метод предшествования и другие. Эти методы позволяют эффективно группировать лексемы программы согласно грамматическим правилам и определять корректность их последовательности на ранних этапах разбора. Особое внимание уделяется построению дерева разбора, которое может быть представлено в виде дерева зависимостей, дерева составляющих или их комбинации. Такое дерево не только отражает структуру программы, но и служит основой для дальнейшего анализа, оптимизации и генерации кода. Также анализатор должен быть гибким и масштабируемым, чтобы поддерживать расширение грамматики языка и адаптироваться к новым синтаксическим конструкциям.

Для реализации анализатора может быть использован либо нисходящий подход (например, рекурсивный спуск или *LL*-анализ), при котором разбор начинается со стартового символа и завершается последовательностью лексем, либо восходящий подход (например, *LR*-анализ), при котором правила грамматики восстанавливаются от лексем до стартового символа. Выбор подхода зависит от сложности грамматики и требований к производительности анализатора. В результате будет создан полноценный синтаксический анализатор, способный точно определять структуру программы, своевременно выявлять синтаксические ошибки и предоставлять подробную информацию об их характере. Это делает анализатор важным инструментом для разработки компиляторов, интерпретаторов и других систем обработки программного кода.

**2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Синтаксический анализ является одним из ключевых этапов в процессе компиляции или интерпретации программного кода. На этом этапе проверяется, насколько исходный текст программы соответствует синтаксическим правилам, определенным грамматикой языка. В результате анализа формируется структурированное представление программы, обычно в виде синтаксического дерева, которое отображает иерархию конструкций кода. Это дерево играет важную роль на последующих этапах обработки, таких как семантический анализ или генерация машинного кода.

Одним из распространенных подходов к реализации синтаксических анализаторов являются *LL*-парсеры (*left-to-right*, *leftmost derivation parsers*). Эти парсеры используют нисходящий метод анализа, начиная разбор с левого символа входной строки и выполняя левостороннее разворачивание грамматических правил [1]. Главная особенность *LL*-парсеров заключается в том, что они принимают решения о применении конкретных правил грамматики на основе текущего символа входной строки и состояния стека. Для этого строится таблица предопределенных решений, которая создается на основе анализа грамматики языка. Однако для корректной работы *LL*-парсеров грамматика должна удовлетворять ряду строгих требований. Прежде всего, она должна быть свободна от левой рекурсии, так как это может привести к бесконечным циклам при разборе. Левая рекурсия возникает, когда правило ссылается само на себя через первый символ своей правой части. устранить эту проблему, грамматику преобразуют в эквивалентную форму без левой рекурсии. Кроме того, для разрешения конфликтов выбора между правилами применяется факторизация грамматики, которая позволяет объединить общие префиксы альтернативных правил.

Для работы *LL*-парсера необходимы два ключевых понятия: множества *FIRST* и *FOLLOW*. Множество *FIRST* содержит все возможные терминалы, которые могут появиться первыми в строке, порождаемой данным нетерминалом. Множество *FOLLOW* определяет терминалы, которые могут следовать за данным нетерминалом в любой допустимой строке языка. Эти множества используются для заполнения таблицы решений парсера, где каждая ячейка указывает, какое правило следует применить при встрече конкретного терминала в данном контексте. Благодаря своей простоте реализации и высокой производительности, *LL*-парсеры широко применяются в практике разработки компиляторов и интерпретаторов. Они обеспечивают быстрый и надежный разбор входных данных, если грамматика языка соответствует необходимым требованиям.

**3 РЕЗУЛЬТАТЫ РАБОТЫ**

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

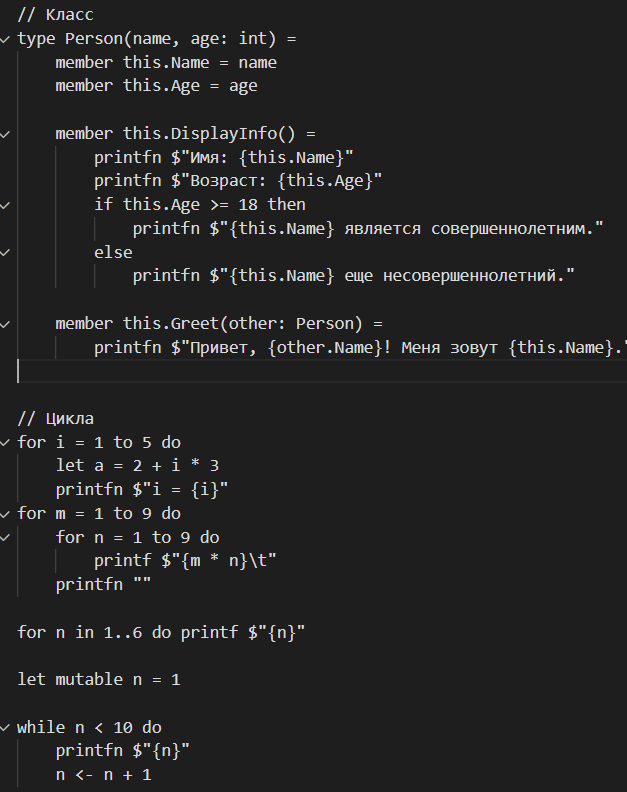
****

Рисунок 3.1 – Часть Анализируемого кода

Результатом работы разработанной программы является синтаксическое дерево в текстовом и графическом виде.

На рисунке 3.2 представлена часть построенного дерева.

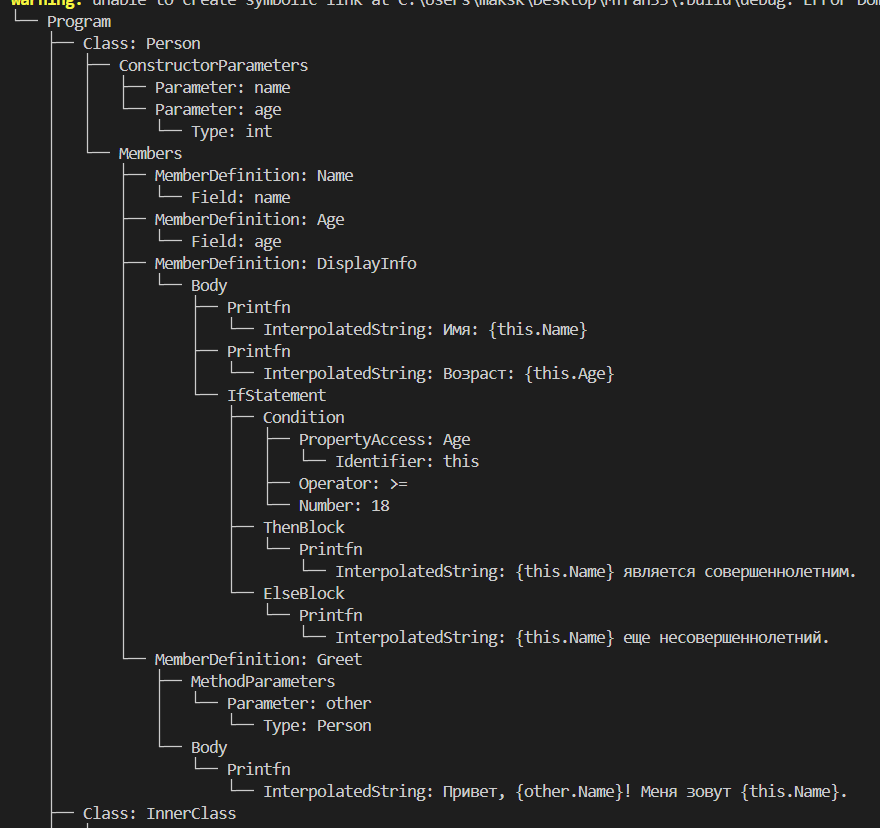


Рисунок 3.2 – Часть вывода дерева

В ходе выполнения работы был успешно создан и протестирован синтаксический анализатор. Парсер показал высокую производительность при обработке разнообразных синтаксических конструкций, включая выражения, функции и подключение модулей. Он корректно формирует синтаксическое дерево, которое наглядно демонстрирует структуру программы. Это является ключевым этапом для последующей обработки кода, например, при компиляции или интерпретации. Особого внимания заслуживает способность анализатора работать с вложенными конструкциями, что подтверждает его универсальность и надежность.

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 рамках выполнения лабораторной работы была разработана и реализована программа синтаксического анализа, основанная на применении одного из табличных методов, а именно *LL*-метода. Созданный анализатор успешно проверяет соответствие исходного текста программы формальным синтаксическим правилам языка и строит дерево разбора, которое наглядно отображает структуру входной последовательности токенов. Данное дерево представляет собой иерархическую организацию грамматических конструкций программы, что является важным этапом для дальнейшей обработки кода, например, при компиляции или интерпретации. В процессе работы были изучены и применены ключевые принципы построения синтаксических анализаторов, включая метод рекурсивного спуска и механизмы обработки ошибок.

Одним из значимых достижений работы стало успешное применение *LL*-метода для анализа контекстно-свободных грамматик. Этот метод, основанный на предсказании следующего шага разбора с использованием таблицы переходов, продемонстрировал высокую эффективность как для простых, так и для сложных конструкций языка. В частности, были реализованы механизмы для обработки условных операторов, циклов, функций и выражений, что подтвердило универсальность подхода. Кроме того, программа была дополнена функцией визуализации дерева разбора, что значительно упрощает анализ и отладку синтаксической структуры программы.

Таким образом, выполненная работа позволила углубить понимание принципов работы синтаксических анализаторов и методов их реализации, а также продемонстрировать практическое применение теоретических знаний о формальных грамматиках и алгоритмах парсинга. 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 счет поддержки дополнительных языковых конструкций или интеграции с семантическим анализатором для проверки типов данных и других аспектов программы.

В заключение стоит отметить, что выполненная работа не только закрепила теоретические знания, но и предоставила ценный практический опыт в разработке синтаксических 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] How to Make an LL(1) Parser [Электронный ресурс]. – Режим доступа: https://honeythecodewitch.com/blog/how-to-make-an-ll1-parser-lesson-1.

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

class Parser {

private let tokens: [Token]

private var current = 0

init(tokens: [Token]) {

self.tokens = tokens

}

private func isNumber(\_ value: String) -> Bool {

return Int(value) != nil

}

private func peek() -> Token {

return tokens[current]

}

private func next(offset: Int = 1) -> Token {

if current + offset < tokens.count {

return tokens[current + offset]

}

fatalError("Неожиданный конец токенов")

}

private func consume() -> Token {

guard current < tokens.count else {

fatalError("Попытка чтения за пределами массива токенов")

}

let token = tokens[current]

current += 1

return token

}

private func match(\_ type: TokenType, value: String? = nil) -> Bool {

if current < tokens.count && tokens[current].type == type {

if let value = value {

return tokens[current].value == value

}

return true

}

return false

}

func parseProgram() -> ASTNode {

let root = ASTNode(type: "Program")

while current < tokens.count {

if match(.error) {

fatalError("Ошибка в строке \(peek().line)")

} else if match(.keyword, value: "if") {

root.addChild(parseIfStatement())

} else if match(.keyword, value: "let") {

// Проверяем, является ли это объявлением функции

let nextToken = next()

if nextToken.type == .identifier && tokens[current + 2].value == "(" {

Logger.log("Нашли функцию \(nextToken.value)")

root.addChild(parseFunctionDeclaration())

} else {

root.addChild(parseVariableDeclaration())

}

} else if match(.keyword, value: "type") {

root.addChild(parseClassDeclaration())

} else if match(.keyword, value: "printfn") || match(.keyword, value: "print") {

root.addChild(parsePrintfn())

} else if match(.keyword, value: "while") {

root.addChild(parseWhileStatement())

} else if match(.keyword, value: "for") {

// Проверяем, какой тип цикла for

Logger.log("Токен ГЛАВНЫЙ \(peek().value)")

let nextToken = next(offset: 2)

if nextToken.value == "=" {

Logger.log("FOR TO \(peek().value)")

root.addChild(parseForToStatement())

} else if nextToken.value == "in" {

Logger.log("FOR IN \(peek().value)")

root.addChild(parseForInStatement())

} else {

fatalError("Неожиданный токен после 'for': \(nextToken.value)")

}

} else if match(.identifier) {

if match(.identifier) && next(offset: 1).value == "(" {

root.addChild(parseFunctionCall())

} else {

Logger.log("+++++ \(peek().value)")

root.addChild(parseAssignment())

}

} else {

fatalError("Неизвестный токен")

}

}

return root

}

private func parseFunctionCall() -> ASTNode {

let functionName = consume() // Имя функции

let functionCallNode = ASTNode(type: "FunctionCall", value: functionName.value)

consume() // Пропускаем "("

let argumentsNode = ASTNode(type: "Arguments")

while !match(.operatorToken, value: ")") {

let argument = consume()

let argumentNode = ASTNode(type: "Argument", value: argument.value)

argumentsNode.addChild(argumentNode)

if match(.operatorToken, value: ",") {

consume() // Пропускаем ","

}

}

consume() // Пропускаем ")"

// Добавляем узел Arguments только если есть аргументы

if argumentsNode.children.count > 0 {

functionCallNode.addChild(argumentsNode)

}

return functionCallNode

}