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.253505 Снежко М.А.

Проверил: ассистент кафедры информатики
Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

1 ЦЕЛЬ РАБОТЫ

Целью данной лабораторной работы является разработка интерпретатора, который будет осуществлять преобразование кода на языке *Swift* в эквивалентный код на языке *F#*. Это включает в себя не только синтаксический и семантический анализ исходного кода, но и его интерпретацию для выполнения в среде *F#*[1].

В рамках работы необходимо:

1 Использовать существующие компоненты. На предыдущих этапах были реализованы лексический анализатор, синтаксический анализатор и семантический анализатор. Эти компоненты будут служить основой для интерпретации, обеспечивая корректное распознавание и анализ исходного кода Java.

2 Перевести синтаксическую структуру: В процессе работы будет разработан алгоритм, который обеспечит преобразование синтаксиса *Swift* в соответствующий синтаксис *F#*[2]. Это включает в себя такие конструкции, как классы, методы, условия и циклы.

Данная работа позволит углубить понимание механизмов работы языков программирования, а также развить навыки в области компиляторов и интерпретаторов.

2 ХОД РАБОТЫ

Во ходе выполнения лабораторной работы была выполнена интерпретация языка *F#*. Листинг программного кода приведен в приложении А. Пример кода на *F#* приведен рисунке 2.1.

```
let num = 42
if num < 0 then
    printfn "Число отрицательное"
elif num > 100 then
    printfn "Число положительное и больше 100"
else
    printfn "Число в промежутке (0, 100)"

// === Циклы ===
// Простая арифметика с циклом
for i = 1 to 5 do
    let a = 2 + i * 3
    printfn $"i = {i}, a = {a}"

// Таблица умножения 9x9
printfn "\nТаблица умножения:"
for m = 1 to 9 do
    for n = 1 to 9 do
        printf $"{m * n}\t"
    printfn ""
```

Рисунок 2.1 – Пример ввода на интерпретируемом языке

Результат работы интерпретатора представлен на рисунке 2.2.

```
Число в промежутке (0, 100)
i = 1, a = 5
i = 2, a = 8
i = 3, a = 11
i = 4, a = 14
i = 5, a = 17

Таблица умножения:
1       2       3       4       5       6       7       8       9
2       4       6       8       10      12      14      16      18
3       6       9       12      15      18      21      24      27
4       8       12      16      20      24      28      32      36
5       10      15      20      25      30      35      40      45
6       12      18      24      30      36      42      48      54
7       14      21      28      35      42      49      56      63
8       16      24      32      40      48      56      64      72
9       18      27      36      45      54      63      72      81
```

Рисунок 2.2 – Результат работы интерпретатора

Таким образом интерпретатор успешно справляется с генерацией и выполнением кода.

ЗАКЛЮЧЕНИЕ

В процессе выполнения лабораторной работы по созданию интерпретатора были изучены ключевые аспекты интерпретации программного кода, включая построчное выполнение, динамический анализ и обработку ошибок. Были разработаны алгоритмы для разбора исходного кода, проверки синтаксической и семантической корректности, а также для выполнения операций и вызова функций. Особое внимание было уделено обработке ошибок, чтобы предоставлять пользователю понятные и информативные сообщения о проблемах в коде, что способствует более эффективной отладке и улучшению качества программного обеспечения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Medium [Электронный ресурс]. – Режим доступа: https://medium.com/@bhagyesh.patil20/code-generation-in-compiler-design-93591d62efb6. – Дата доступа: 17.04.2025.

[2] GeeksForGeeks [Электронный ресурс]. – Режим доступа: https://www.geeksforgeeks.org/phases-of-a-compiler/. – Дата доступа: 17.04.2025.

ПРИЛОЖЕНИЕ А
(обязательное)
Листинг программного кода

```
class CodeGenerator {
    private var output = ""
    private var indentLevel = 0
    private var currentIndent: String {
        return String(repeating: " ", count: indentLevel * 4)
    }
    private let typeMap: [String: String] = [
        "int": "Int",
        "float": "Float",
        "double": "Double",
        "char": "Character",
        "bool": "Bool",
        "string": "String",
        "unit": "Void",
        "list": "Array",
        "array": "Array",
        "seq": "Array",
        "Map": "Dictionary",
        "Set": "Set",
    ]
    func generate(from node: ASTNode) -> String {
        output = ""
        generateNode(node)
        //return output
        return cleanGeneratedCode(output)
    }
     private func generateNode(_ node: ASTNode) {
        switch node.type {
        case "Program":
            generateProgram(node)
        case "FunctionDeclaration":
            generateFunction(node)
        case "VariableDeclaration":
            generateVariableDeclaration(node)
        case "IfStatement":
            generateIfStatement(node)
        case "WhileStatement":
            generateWhileStatement(node)
        case "ForInStatement":
            generateForInStatement(node)
        case "ForToStatement":
            generateForToStatement(node)
        case "FunctionCall":
            generateFunctionCall(node)
        case "BinaryOp":
            generateBinaryOp(node)
        case "Argument":
            generateNode(node.children.first ?? node)
        case "Assignment":
            generateAssignment(node)
        case "Return":
            generateReturn(node)
        case "Printfn":
            generatePrint(node, newline: true)
        case "Printf":
            generatePrint(node, newline: false)
        case "Class":
            generateClass(node)
        case "ThenBlock", "ElseBlock":
```

```
            // Обрабатываем все дочерние узлы блока
            indentLevel += 1
            for child in node.children {
                generateNode(child)
            }
            indentLevel -= 1
        case "Condition":
            generateCondition(node)
        case "MemberDefinition":
            generateMember(node)
        case "Number", "Float_number":
            let value = node.value.replacingOccurrences(of: "L", with: "")
            output += value
        case "String_literal":
            var cleanedValue = node.value
            if cleanedValue.hasPrefix("\"") && cleanedValue.hasSuffix("\"") {
                cleanedValue = String(cleanedValue.dropFirst().dropLast())}
            output += "\"\(cleanedValue)\""
        case "Boolean_literal":
            output += node.value.lowercased()
        case "Identifier":
            output += node.value
        case "ArrayAccess":
            generateArrayAccess(node)
        case "PropertyAccess":
            generatePropertyAccess(node)
        case "ArrayOrList":
            generateArrayOrList(node)
        case "StartValue", "EndValue", "Range":
            for child in node.children {
                generateNode(child)
            }
        case "Type":
            // Тип обычно обрабатывается в родительском узле
            break
        case "Value":
            output += node.value
        case "Body":
            // Обрабатываем все дочерние узлы тела
            indentLevel += 1
            for child in node.children {
                generateNode(child)
                if !child.type.hasPrefix("If") && !child.type.hasPrefix("For")
                    && !child.type.hasPrefix("While") && child.type != "Body"
                {
                    output += "\n"}}
            indentLevel -= 1
        case "SetDeclaration":
            generateSetDeclaration(node)
        case "SequenceDeclaration":
            generateSequenceDeclaration(node)
        case "ListDeclaration":
            generateListDeclaration(node)
        case "MapDeclaration":
            generateMapDeclaration(node)
        case "ArrayDeclaration":
            generateArrayDeclaration(node)
        case "KeyValuePair":
            generateKeyValuePair(node)
        default:
            print("Warning: Unknown node type \(node.type)")
            for child in node.children {
                generateNode(child)
            }
        }
}
```