Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Разработка компиляторов

Домашнее задание №4

Проектирование интерпретатора.

Выполнил:

Маликов Глеб Игоревич

Группа № P3324

Преподаватель:

Лаздин Артур Вячеславович

Санкт-Петербург

2025

Оглавление

Задание

Целью работы является разработка интерпретатора для Тьюринг полного императивного языка программирования.

Язык должен включать в себя:

- Переменные (хранение переменных моделируется с использованием, например, хэш-таблиц, допустимы другие способы хранения), базовая часть задания предполагает единственный тип: целые со знаком;
- Присваивания (необходимо учитывать появление идентификаторов в правой — арифметической) части —, которые не были проинициализированы выше в области видимости).
- Реализация арифметических операций и операций сравнения (булевый тип вводить не обязательно, можно считать, что всё, что не нуль – истина, а нуль — ложь;
- Оператор ветвления с факультативной ветвью else;
- Оператор цикла (любой на ваш выбор).

Программа состоит из операторов присваивания, ветвления, циклов и осуществляет вывод значений переменных или выражений. Программа хранится в текстовом файле. (Примечание: не усложняйте структуру программы, понятно, что нет функций, массивов, структур, классов и пр.).

Файл с текстом программы должен подаваться как исходные данные интерпретатора. Желательно чтобы программы имели осмысленный вид (поиск n-го элемента ряда Фибоначчи, поиск НОД и пр.).

В качестве инструмента разработки интерпретатора предполагается ANTLR4.

Допустимо использовать Flex Bison.

Выбор языка программирования — на ваш выбор.

Дополнительные расширения:

- Строковый и вещественный типы данных, что предполагает минимальный контроль семантики выражений. Вы можете запретить сложение строковых литералов с целыми значениями, либо допустить, например: 4 + ‘abc’ будет преобразовано в ‘4abc’. Для операций вычитания, умножения, деления опишите семантику, принятую вами для операндов разных типов в отчете, продемонстрируйте корректное поведение интерпретатора на примерах.

Выполнение

Грамматика

```
grammar Minilang;

program: statement+ ;

statement
    : assignment ';'
    | ifStatement
    | whileStatement
    | printStatement ';'
    ;

assignment  : Identifier '=' expression ;
ifStatement : 'if' '(' expression ')' block ( 'else' block )? ;
whileStatement : 'while' '(' expression ')' block ;
printStatement : 'print' '(' expression ')' ;

block       : '{' statement+ '}' ;

expression
    : expression op=('*'|'/') expression
    | expression op=('+'|'-') expression
    | expression '%' expression
    | expression compOp expression
    | '(' expression ')'
    | Number
    | StringLiteral
    | Identifier
    ;

compOp
    : '<'
    | '>'
    | '<='
    | '>='
    | '=='
    ;

Identifier  : [a-zA-Z_][a-zA-Z_0-9]* ;
Number      : [0-9]+ ;
StringLiteral : '\'' (~['\r\n])* '\'' | '"' (~["\r\n])* '"' ;
WS          : [ \t\r\n]+ -> skip ;
```

Код

Main.kt

```
package co.glebmavi

import org.antlr.v4.runtime.CharStreams
import org.antlr.v4.runtime.CommonTokenStream

fun main(args: Array<String>) {
    if (args.size != 1) {
```

```
        println("Usage: <program> <filename>")
        return
    }

    val fileName = args[0]
    val input = CharStreams.fromFileName(fileName)
    val lexer = MinilangLexer(input)
    val tokens = CommonTokenStream(lexer)
    val parser = MinilangParser(tokens)
    val tree = parser.program()

    val visitor = MinilangEvalVisitor()
    visitor.visit(tree)
}
```

MinilangEvalVisitor.kt

```
package co.glebmavi

import org.antlr.v4.runtime.tree.TerminalNode

typealias intVal = Value.IntValue
typealias strVal = Value.StringValue
typealias doubleVal = Value.DoubleValue

class MinilangEvalVisitor : MinilangBaseVisitor<Value>() {

    // Таблица символов для хранения переменных.
    private val symbolTable = mutableMapOf<String, Value>()

    override fun visitProgram(ctx: MinilangParser.ProgramContext): Value? {
        ctx.statement().forEach { visit(it) }
        return null
    }

    override fun visitAssignment(ctx: MinilangParser.AssignmentContext):
Value {
        val id = ctx.Identifier().text
        val value = visit(ctx.expression())
        symbolTable[id] = value ?: throw RuntimeException("Ошибка вычисления
выражения для переменной: $id")
        return value
    }

    override fun visitPrintStatement(ctx:
MinilangParser.PrintStatementContext): Value? {
        val value = visit(ctx.expression())
        println(value?.asString())
        return value
    }

    override fun visitIfStatement(ctx: MinilangParser.IfStatementContext):
Value? {
        val condition = visit(ctx.expression())
        if (condition!!.asNumber() != 0.0) {
            return visit(ctx.block(0))
        } else if (ctx.block().size > 1) {
            return visit(ctx.block(1))
        }
        return null
    }
```

```
    override fun visitWhileStatement(ctx:
MinilangParser.WhileStatementContext): Value? {
        while (visit(ctx.expression())!!.asNumber() != 0.0) {
            visit(ctx.block())
        }
        return null
    }

    override fun visitBlock(ctx: MinilangParser.BlockContext): Value? {
        var last: Value? = null
        ctx.statement().forEach { last = visit(it) }
        return last
    }

    override fun visitExpression(ctx: MinilangParser.ExpressionContext):
Value? {
        when (ctx.childCount) {
            1 -> {
                // Одиночный токен: число, строковый литерал или
идентификатор.
                val child = ctx.getChild(0)
                if (child is TerminalNode) {
                    return when (child.symbol.type) {
                        MinilangParser.Number -> intVal(child.text.toInt())
                        MinilangParser.StringLiteral -> {
                            // Удаляем кавычки.
                            strVal(child.text.substring(1, child.text.length
- 1))
                        }
                        MinilangParser.Identifier -> {
                            val id = child.text
                            symbolTable[id] ?: throw
RuntimeException("Неопределённая переменная: $id")
                        }
                        else -> throw RuntimeException("Неизвестный токен:
${child.text}")
                    }
                }
            }
            3 -> {
                // Либо скобки, либо бинарное выражение.
                if (ctx.getChild(0).text == "(" && ctx.getChild(2).text ==
")") {
                    return visit(ctx.getChild(1))
                } else {
                    val left = visit(ctx.getChild(0))
                        ?: throw RuntimeException("Отсутствует левый операнд
в выражении: ${ctx.text}")
                    val op = ctx.getChild(1).text
                    val right = visit(ctx.getChild(2))
                        ?: throw RuntimeException("Отсутствует правый операнд
в выражении: ${ctx.text}")
                    return evaluateBinary(left, op, right)
                }
            }
            else -> {
                throw RuntimeException("Неподдерживаемое выражение:
${ctx.text}")
            }
        }
        return null
    }
```

```
    private fun evaluateBinary(left: Value, op: String, right: Value): Value
{
        return when (op) {
            "+" -> {
                if (left is strVal || right is strVal) {
                    strVal(left.asString() + right.asString())
                } else if (left is doubleVal || right is doubleVal) {
                    doubleVal(left.asNumber() + right.asNumber())
                } else {
                    intVal((left.asNumber() + right.asNumber()).toInt())
                }
            }
            "-" -> {
                if (left is strVal || right is strVal)
                    throw RuntimeException("Вычитание не поддерживается для
строк")
                else if (left is doubleVal || right is doubleVal)
                    doubleVal(left.asNumber() - right.asNumber())
                else
                    intVal((left.asNumber() - right.asNumber()).toInt())
            }
            "*" -> {
                if (left is strVal && right is intVal)
                    strVal(left.asString().repeat(right.value))
                else if (right is strVal && left is intVal)
                    strVal(right.asString().repeat(left.value))
                else if (left is strVal || right is strVal)
                    throw RuntimeException("Умножение не поддерживается для
строк")
                else if (left is doubleVal || right is doubleVal)
                    doubleVal(left.asNumber() * right.asNumber())
                else
                    intVal((left.asNumber() * right.asNumber()).toInt())
            }
            "/" -> {
                if (left is strVal || right is strVal)
                    throw RuntimeException("Деление не поддерживается для
строк")
                if (right.asNumber() == 0.0)
                    throw RuntimeException("Деление на ноль")
                else if (left is doubleVal || right is doubleVal)
                    doubleVal(left.asNumber() / right.asNumber())
                else
                    intVal((left.asNumber() / right.asNumber()).toInt())
            }
            "%" -> {
                if (left is strVal || right is strVal)
                    throw RuntimeException("Остаток от деления не
поддерживается для строк")
                if (right.asNumber() == 0.0)
                    throw RuntimeException("Деление по модулю на ноль")
                else
                    intVal((left.asNumber() % right.asNumber()).toInt())
            }
            // Операторы сравнения.
            "<" -> intVal(if (left.asNumber() < right.asNumber()) 1 else 0)
            ">" -> intVal(if (left.asNumber() > right.asNumber()) 1 else 0)
            "<=" -> intVal(if (left.asNumber() <= right.asNumber()) 1 else 0)
            ">=" -> intVal(if (left.asNumber() >= right.asNumber()) 1 else 0)
            "==" -> {
                if (left is strVal && right is strVal) {
                    intVal(if (left.asString() == right.asString()) 1 else 0)
```

```
                } else {
                    intVal(if (left.asNumber() == right.asNumber()) 1 else 0)
                }
            }
            else -> throw RuntimeException("Неизвестный оператор: $op")
        }
    }
}
```

Примеры работы

Fibonacci

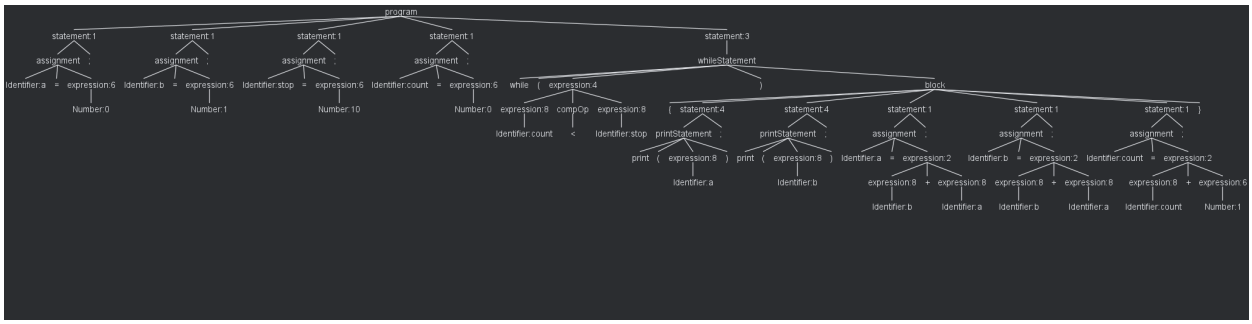
```
a = 0;
b = 1;
stop = 10;
count = 0;
while (count < stop) {
    print(a);
    print(b);
    a = b + a;
    b = b + a;
    count = count + 1;
}
```

Результат:

```
0

1

1

2

3

5

8

13

21

34

55

89

144

233

377

610

987

1597

2584
```

```
4181
```

Дерево:



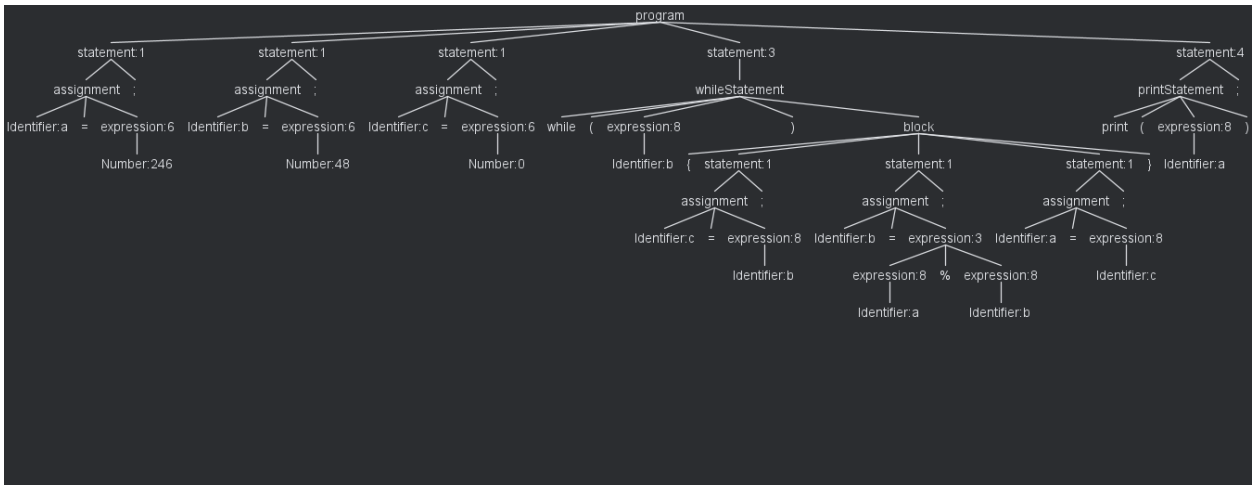
Greatest common divisor

```
a = 246;
b = 48;
c = 0;
while (b) {
    c = b;
    b = a % b;
    a = c;
}
print(a);
```

Результат:

```
6
```

Дерево:



String operations

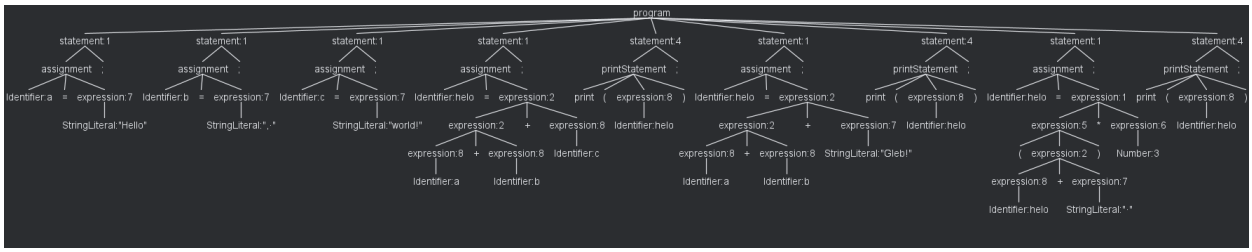
```
a = "Hello";
b = ", ";
c = "world!";

helo = a + b + c;
print(helo);

helo = a + b + "Gleb!";
print(helo);
```

```
helo = (helo + " ")* 3;
print(helo);
```

Результат:

```
Hello, world!

Hello, Gleb!

Hello, Gleb! Hello, Gleb! Hello, Gleb!
```

Дерево:



Заключение

В ходе выполнения данной работы была реализована интерпретация Тьюринг-полного императивного языка программирования. С использованием ANTLR4 была составлена грамматика, охватывающая основные элементы языка: работу с переменными, операции присваивания, арифметические и сравнительные операции, а также операторы ветвления и циклов.

Практическая проверка работы интерпретатора на примерах (вычисление ряда Фибоначчи, нахождение НОД, операции со строками) продемонстрировала его корректность и эффективность.

Таким образом, выполненная работа позволила глубже изучить принципы построения интерпретаторов и применения синтаксического анализа, а также получить ценный практический опыт разработки программных инструментов для анализа и исполнения кода.