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

ИНТЕРПРЕТАТОР ИСХОДНОГО КОДА

Выполнил студент гр.153502 Толстой Д.В.

Проверил ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Формулировка задачи 3](#_Toc158758016)

[2 Описание функций программы 4](#_Toc158758017)

[Приложение А (обязательное) Листинг исходного кода 5](#_Toc158758018)

1 ФОРМУЛИРОВКА ЗАДАЧИ

Целью выполнения лабораторной работы является разработка интерпретатора подмножества языка программирования, определенного в лабораторной работе 1. В процессе семантического анализа проверяется наличие семантических ошибок в исходной программе и накапливается информация о типах для следующей стадии – генерации кода. При семантическом анализе используются иерархические структуры, полученные во время синтаксического анализа для идентификации операторов и операндов выражений и инструкций. Далее интерпретатор начинает выполнение кода и выводит результат.

2 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Программа принимает на вход текст программы и с помощью иерархической структуры полученной при синтаксическом анализе проверяет семантические ошибки и выполняет операции. Код программы предоставлен на рисунке 1.

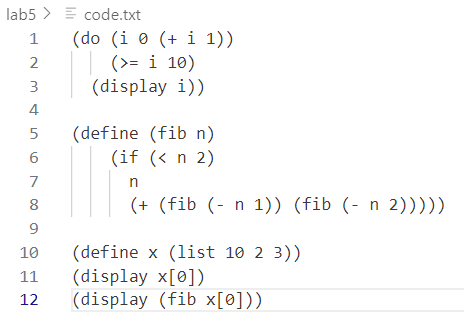


Рисунок 1 – Код программы

Вывод программы при запуске программы представлен на рисунке 2.

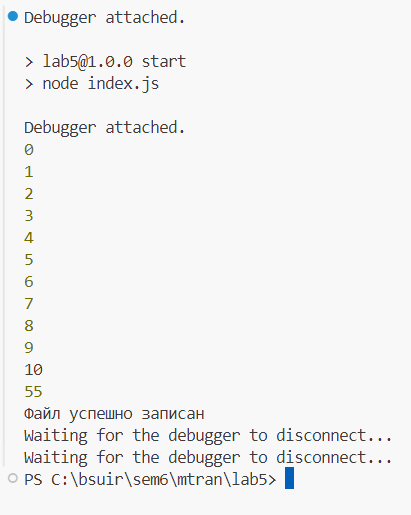


Рисунок 2 – Вывод программы

ПРИЛОЖЕНИЕ А

**(обязательное)**

**Листинг исходного кода**

Файл index.js

import tokenize from '../lab2/analyzer.js';

import buildSyntaxTree from '../lab3/syntaxAnalyzer.js';

import generateTables from '../lab4/generateTables.js';

import interpret from './interpret.js';

import fs from 'fs';

fs.readFile('code.txt', 'utf8', (err, data) => {

    if (err) {

      console.error(err);

      return;

    }

    let [answer, normalAnswer] = tokenize(data);

    let ast = buildSyntaxTree(normalAnswer);

    generateTables(ast);

    interpret(ast);

    fs.writeFile('tree.txt', JSON.stringify(ast, null, '\t'), function(error){

        if(error){

          return console.log(error);

        }

        console.log("Файл успешно записан");

    });

});

Файл interpret.js

import performCalculations from "./performCalculations.js";

import findFirstVariable from '../lab4/services/findFirstVariable.js';

import findVariableInStack from "./findVariableInStack.js";

import findFirstLiteral from '../lab4/services/findFirstLiteral.js';

import executeFunction from './executeFunction.js'

import getType from "./getType.js";

function interpret(node, stack=[]) {

    let stackOfTables = stack;

    function main(node) {

        stackOfTables.push([]);

        if (node.type === 'Operator') {

            let result = performCalculations(node, stackOfTables).value;

            if (stackOfTables.length !== 0) {

                if (stackOfTables[stackOfTables.length - 1].length !== 0) {

                    node.tableOfVariables = stackOfTables.pop();

                }

                else {

                    stackOfTables.pop();

                }

            }

            return result;

        } else if (node.type === 'Variable') {

            if (stackOfTables.length !== 0) {

                if (stackOfTables[stackOfTables.length - 1].length !== 0) {

                    node.tableOfVariables = stackOfTables.pop();

                }

                else {

                    stackOfTables.pop();

                }

            }

            let [newVariable, newIndex] = findVariableInStack(stackOfTables, node.name);

            return newVariable.value;

        } else if (node.type === 'CallExpression') {

            let args = node.params.map(param => {

                if (param.type.includes('Literal')) {

                    return {name: param.name, value: param.value, type: param.type};

                }

                if (param.type === 'Operator' || param.type === 'CallExpression') {

                    let result = interpret(param, stackOfTables);

                    return {value: result, type: getType(result)};

                }

                let [arg, argIndex] = findVariableInStack(stackOfTables, param.name);

                return {name: arg.name, value: arg.value, type: arg.type};

            });

            let [funcVar, funcIndex] = findVariableInStack(stackOfTables, node.name);

            let argsNames = funcVar.argsNames;

            let result = executeFunction(funcVar.body, args, stackOfTables, argsNames);

            if (stackOfTables.length !== 0) {

                if (stackOfTables[stackOfTables.length - 1].length !== 0) {

                    node.tableOfVariables = stackOfTables.pop();

                }

                else {

                    stackOfTables.pop();

                }

            }

            return result;

        } else if (node.type === 'Sys\_func') {

            if (node.name === 'do')

            {

                stackOfTables[stackOfTables.length - 1].push({name: node.params[0].name, type: 'Number', value: Number(node.params[0].params[0].value)});

                while (!performCalculations(node.params[1], stackOfTables).value) {

                    main(node.params[2]);

                    main({type: 'Sys\_func', name: 'set!', params: [{type: 'Variable', name: node.params[0].name, type: 'Number', value: Number(node.params[0].params[0].value)},

                node.params[0].params[1]]});

                }

            } else if (node.name === 'begin') {

                for (let param of node.params) {

                    main(param);

                }

                if (stackOfTables.length !== 0) {

                    if (stackOfTables[stackOfTables.length - 1].length !== 0) {

                        node.tableOfVariables = stackOfTables.pop();

                    }

                    else {

                        stackOfTables.pop();

                    }

                }

                return

            } else if (node.name === 'set!') {

                let [variable, index] = findVariableInStack(stackOfTables, node.params[0].name);

                if (node.params[1].type === 'Operator') {

                    variable.value = performCalculations(node.params[1], stackOfTables).value;

                } else if (node.params[1].type === 'Variable') {

                    let [newVariable, newIndex] = findVariableInStack(stackOfTables, node.params[1].name);

                    variable.value = newVariable.value;

                    variable.type = newVariable.type;

                } else if (node.params[1].type === 'CallExpression') {

                    // if (stackOfTables.length !== 0) {

                    //     if (stackOfTables[stackOfTables.length - 1].length !== 0) {

                    //         node.tableOfVariables = stackOfTables.pop();

                    //     }

                    //     else {

                    //         stackOfTables.pop();

                    //     }

                    // }

                    variable.value = main(node.params[1].type);

                }

            } else if (node.name === 'display') {

                if (node.params[0].type === 'Variable') {

                    let [newVariable, newIndex] = findVariableInStack(stackOfTables, node.params[0].name);

                    console.log(newVariable.value);

                } else if (node.params[0].type.includes('Literal')) {

                    console.log(node.params[0].value);

                } else {

                    console.log(main(node.params[0]));

                }

            } else if (node.name === 'if') {

                let condition = performCalculations(node.params[0], stackOfTables).value;

                if (stackOfTables.length !== 0) {

                    if (stackOfTables[stackOfTables.length - 1].length !== 0) {

                        node.tableOfVariables = stackOfTables.pop();

                    }

                    else {

                        stackOfTables.pop();

                    }

                }

                if (condition) {

                    return main(node.params[1]);

                } else if (node.params[2] !== undefined){

                    return main(node.params[2]);

                }

            } else if (node.name === 'define') {

                let variable = {name: node.params[0].name};

                if (node.params[0].type === 'CallExpression') {

                    variable.type = 'Function';

                    variable.body = node.params[1];

                    variable.argsNames = node.params[0].params;

                    variable.numberOfArgs = node.params[0].params.length;

                    for (let param of node.params[0].params) {

                        stackOfTables[stackOfTables.length - 1].push({name: param.name, type: 'Any'});

                    }

                } else if (node.params[1].type === 'CallExpression') {

                    let result = main(node.params[1]);

                    variable.value = result;

                    variable.type = getType(result);

                } else if (node.params[1].type === 'Operator') {

                    variable.value = performCalculations(node.params[1], stackOfTables).value;

                    let variableOrNull = findFirstVariable(node.params[1]);

                    if (variableOrNull !== null) {

                        let [newVariable, newIndex] = findVariableInStack(stackOfTables, variableOrNull)

                        variable.type = newVariable.type;

                    } else {

                        let literal = findFirstLiteral(node.params[1]);

                        if (!isNaN(Number(literal))) {

                            variable.type = 'Number';

                            variable.value = node.params[1].value;

                        } else if ((literal[0] === '\'' || literal[0] === '"') && (literal[literal.length - 1] === '\'' || literal[literal.length - 1] === '"')) {

                            variable.type = 'String';

                        } else {

                            variable.type = 'Boolean';

                        }

                    }

                } else if (node.params[1].name === 'list') {

                    variable.type = 'List';

                    variable.value = node.params[1].params;

                } else if (node.params[1].value === 't' || node.params[1].value === 'nil' || node.params[1].value === '#t' || node.params[1].value === '#f') {

                    variable.type = 'Boolean';

                    variable.value = node.params[1].value;

                } else if (!isNaN(Number(node.params[1].value))) {

                    variable.type = 'Number';

                    variable.value = node.params[1].value;

                } else if ((node.params[1].value[0] === '\'' || node.params[1].value[0] === '"') && (node.params[1].value[node.params[1].value.length - 1] === '\'' || node.params[1].value[node.params[1].value.length - 1] === '"')) {

                    variable.type = 'String';

                    variable.value = node.params[1].value;

                }

                stackOfTables[stackOfTables.length - 2].push(variable);

                if (stackOfTables.length !== 0) {

                    if (stackOfTables[stackOfTables.length - 1].length !== 0) {

                        node.tableOfVariables = stackOfTables.pop();

                    }

                    else {

                        stackOfTables.pop();

                    }

                }

                return;

            }

        } else if (node.type === 'Program') {

            for (let param of node.body) {

                main(param);

            }

            if (stackOfTables.length !== 0) {

                if (stackOfTables[stackOfTables.length - 1].length !== 0) {

                    node.tableOfVariables = stackOfTables.pop();

                }

                else {

                    stackOfTables.pop();

                }

            }

            return

        }

    }

    return main(node);

}

export default interpret;