Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе № 5

По теме «Интерпретация исходного кода»

Выполнил:

студент гр. 053502

Герчик А. В.

Проверил:

Гриценко Н. Ю.

Минск 2023

**Содержание**

1 Цель работы 3

2 Результаты 4

Приложение А (обязательное) Код программы 7

# 1 Цель работы

На основе результатов анализа лабораторных работ 1-4 выполнить интерпретацию программы.

# 2 Результаты

В данном разделе будет демонстрация работы интерпретатора кода программ, представленных в лабораторной работе 1.

На рисунках ниже представлен код программ(см. рисунок 2.1, рисунок 2.2).

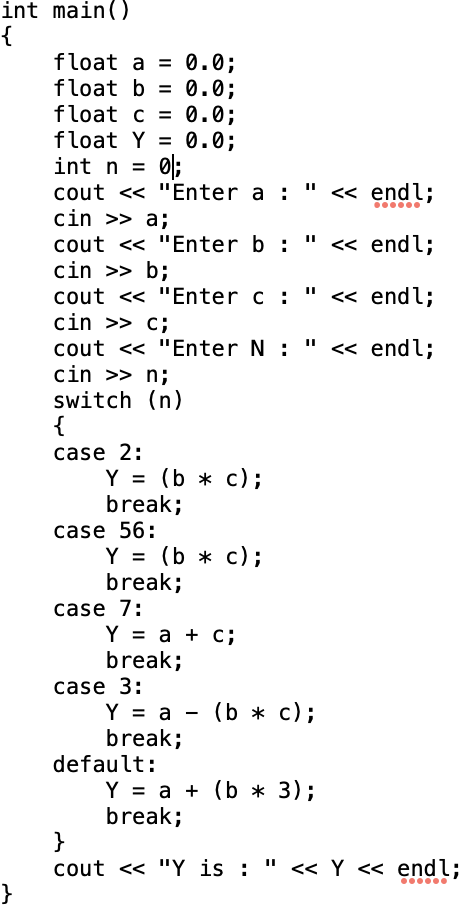


Рисунок 2.1 - Код первой тестовой программы

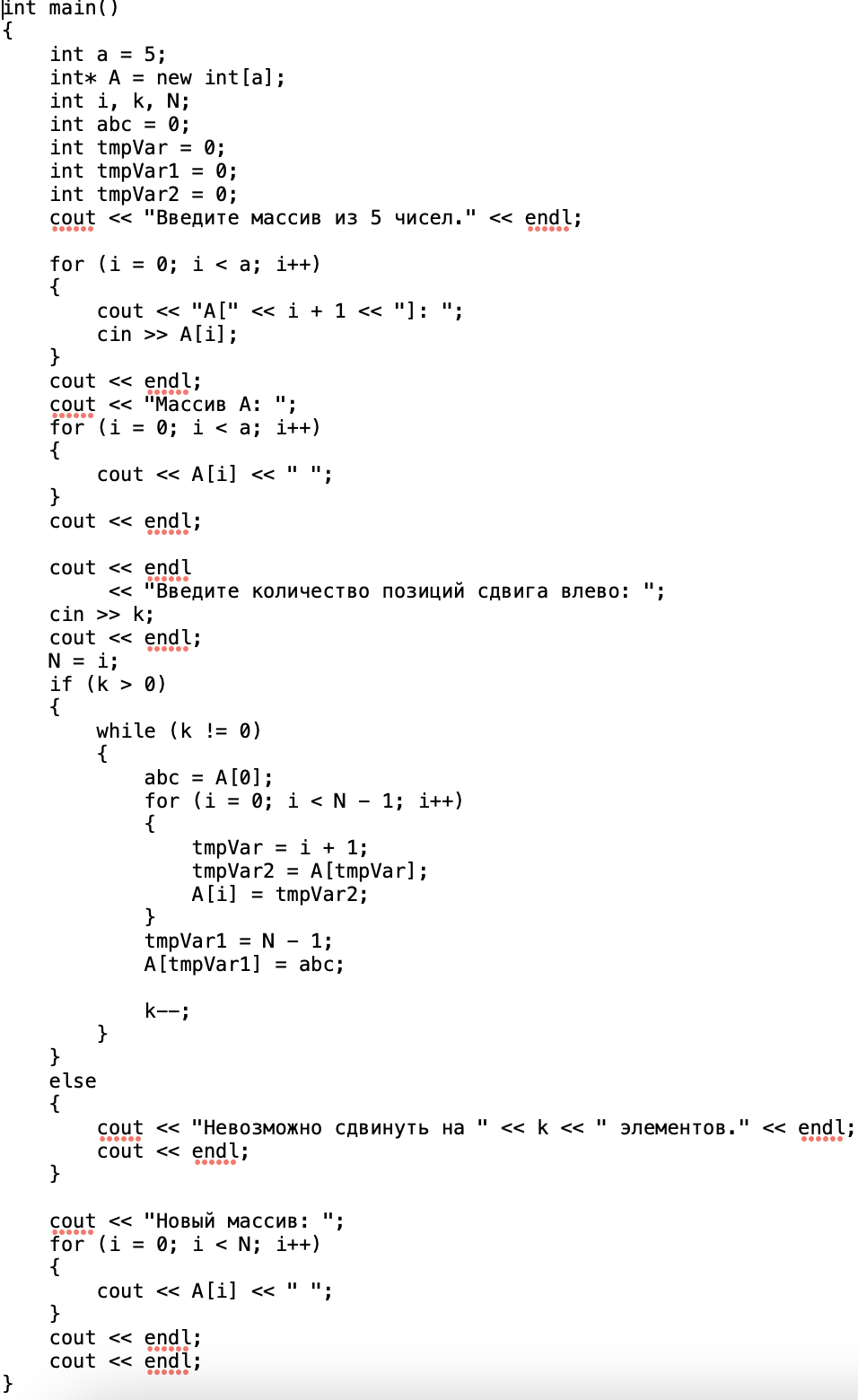


Рисунок 2.2 - Код второй тестовой программы

Результат интерпретации первой тестовой программы, после лексического, синтаксического, а также семантического анализа представлен ниже(см. рисунок 2.3). Программа принимает на вход 4 числа, и в зависимости от введеных чисел, выдает результат арифметических операций над этими числами.

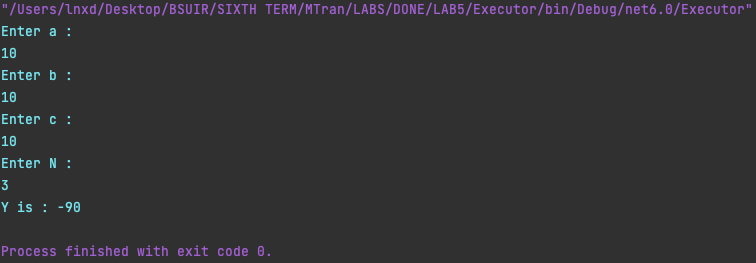


Рисунок 2.3 - Результат интерпретации первой тестовой программы

Результат интерпретации второй тестовой программы , после лексического, синтаксического, а также семантического анализа представлен ниже(см. рисунок 2.4). Программа принимает на вход 5 элементов массива, после чего принимает число, на которое массив будет сдвинут влево, после происходит сдвиг массива.

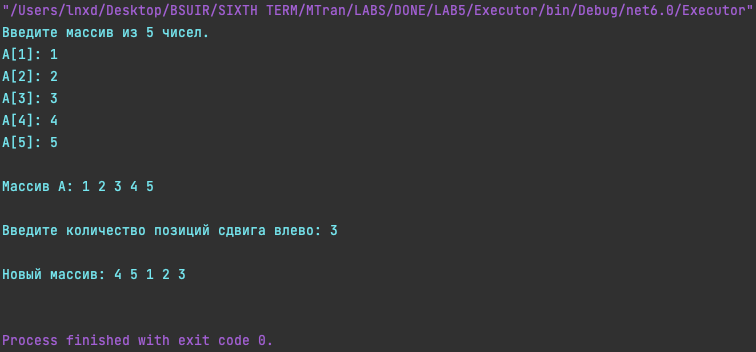


Рисунок 2.4 - Результат интерпретации второй тестовой программы

# Приложение А (обязательное) Код программы

using SyntaxAnalyzer.Nodes;

using SemanticAnalyzer.Functional;

namespace LAB5.Functional;

public class Executor

{

private AbstractNode Root { get; set; }

private Dictionary<string, Dictionary<string, object?>> VariableTables { get; set; } = new();

private Semantic Semantic { get; set; }

private string CodeBlock { get; set; }

private int CodeDepthLevel { get; set; }

private int CodeDepthParent { get; set; }

private int CodeBlockIndex { get; set; }

private bool NeedToExecute { get; set; }

private bool FoundBreak { get; set; }

private bool FoundDefault { get; set; }

private bool InSwitch { get; set; }

private object? SwitchValue { get; set; }

private bool InFor { get; set; }

public bool TEST { get; set; }

public string TEST1 { get; set; }

public bool TMPBOOL { get; set; }

private Dictionary<string, Dictionary<string, AbstractNode>> Functions { get; set; } = new();

public Executor(AbstractNode root, Dictionary<string, Dictionary<string, string>> variableTables, Semantic semantic,

bool test)

{

Root = root;

foreach (var codeBlock in variableTables.Keys)

{

VariableTables.Add(codeBlock, new());

foreach (var key in variableTables[codeBlock].Keys)

{

VariableTables[codeBlock].Add(key, new());

}

}

Semantic = semantic;

CodeBlock = "-1"; // would be needed for code expanding

CodeDepthLevel = -1;

CodeDepthParent = -1;

CodeBlockIndex = -1;

NeedToExecute = true;

FoundBreak = false;

FoundDefault = false;

InSwitch = false;

SwitchValue = null;

InFor = false;

TEST = test;

TEST1 = "";

TMPBOOL = false;

}

public void Execute()

{

WorkOnNode(Root);

}

public object? WorkOnNode(AbstractNode? abstractNode)

{

if (TMPBOOL)

{

CodeBlock = "1:0:1";

}

if (abstractNode == null)

{

return null;

}

if (abstractNode is StatementsNode statementsNode)

{

if (!InFor)

{

IncreaseDepth();

}

else

{

InFor = false;

}

foreach (var node in statementsNode.Nodes)

{

if (!FoundBreak)

{

WorkOnNode(node);

if (FoundBreak)

{

continue;

}

}

if (FoundBreak && !InSwitch)

{

WorkOnNode(node);

}

}

DecreaseDepthOnlyForLevel();

}

if (abstractNode is KeyWordNode keyWordNode)

{

switch (keyWordNode.KeyWord.Identifier)

{

case "endl":

return "\n";

case "break":

if (InSwitch)

{

FoundBreak = true;

}

break;

case "default":

if (InSwitch)

{

NeedToExecute = false;

}

else

{

FoundDefault = true;

}

return null;

}

}

if (abstractNode is CoutNode coutNode && (NeedToExecute || FoundDefault))

{

foreach (var param in coutNode.Parameters)

{

var readyParam = WorkOnNode(param);

if (readyParam is string paramAsSTR)

{

readyParam = paramAsSTR.Replace("\"", "").Replace("\\n", "\n");

}

Console.Write(readyParam);

}

}

if (abstractNode is CinNode cinNode && (NeedToExecute || FoundDefault))

{

foreach (var param in cinNode.Parameters)

{

var codeBlock = GetCodeBlock();

var paramType = Semantic.GetReturnType(param);

if (param is VariableNode variableNode)

{

while (codeBlock != "-1")

{

if (VariableTables[codeBlock].ContainsKey(variableNode.Variable.Identifier))

{

VariableTables[codeBlock][variableNode.Variable.Identifier] = paramType switch

{

"int" => int.Parse(Console.ReadLine()!),

"float" => double.Parse(Console.ReadLine()!),

"char" => char.Parse(Console.ReadLine()!),

"bool" => bool.Parse(Console.ReadLine()!),

\_ => Console.ReadLine()!,

};

break;

// switch (VariableTables[codeBlock][variableNode.Variable.Identifier])

// {

// case "int":

// return int.Parse(Console.ReadLine()!);

// case "float":

// return double.Parse(Console.ReadLine()!);

// case "char":

// return char.Parse(Console.ReadLine()!);

// case "bool":

// return bool.Parse(Console.ReadLine()!);

// default:

// return Console.ReadLine();

// }

//

// break;

}

else

{

codeBlock = ModifyLocalCodeBlock(codeBlock);

}

}

}

if (param is BinaryOperationNode binaryOperationNode)

{

var leftNode = binaryOperationNode.LeftNode as VariableNode;

var indexRightNodeToInsert = WorkOnNode(binaryOperationNode.RightNode) as int?;

while (codeBlock != "-1")

{

if (VariableTables[codeBlock].ContainsKey(leftNode!.Variable.Identifier))

{

break;

}

else

{

codeBlock = ModifyLocalCodeBlock(codeBlock);

}

}

switch (paramType)

{

case "int":

(VariableTables[codeBlock][leftNode!.Variable.Identifier] as List<int>)![

int.Parse(indexRightNodeToInsert.ToString()!)] = int.Parse(Console.ReadLine()!);

break;

case "float":

(VariableTables[codeBlock][leftNode!.Variable.Identifier] as List<double>)![

int.Parse(indexRightNodeToInsert.ToString()!)] = double.Parse(Console.ReadLine()!);

break;

case "char":

(VariableTables[codeBlock][leftNode!.Variable.Identifier] as List<char>)![

int.Parse(indexRightNodeToInsert.ToString()!)] = char.Parse(Console.ReadLine()!);

break;

case "bool":

(VariableTables[codeBlock][leftNode!.Variable.Identifier] as List<bool>)![

int.Parse(indexRightNodeToInsert.ToString()!)] = bool.Parse(Console.ReadLine()!);

break;

default:

(VariableTables[codeBlock][leftNode!.Variable.Identifier] as List<string>)![

int.Parse(indexRightNodeToInsert.ToString()!)] = Console.ReadLine()!;

break;

}

}

}

return null;

}

if (abstractNode is LiteralNode literalNode)

{

switch (literalNode.Literal.Type)

{

case "int literal":

return int.Parse(literalNode.Literal.Identifier);

case "float literal":

return double.Parse(literalNode.Literal.Identifier);

case "char literal":

return char.Parse(literalNode.Literal.Identifier.Replace("\'", ""));

case "bool literal":

return bool.Parse(literalNode.Literal.Identifier);

default:

return literalNode.Literal.Identifier;

}

}

if (abstractNode is FunctionNode functionNode)

{

if (functionNode.Function.Identifier == "main")

{

WorkOnNode(functionNode.Body);

}

else

{

IncreaseDepth();

Functions.Add(functionNode.Function.Identifier,

new Dictionary<string, AbstractNode> { { CodeBlock, functionNode.Body } });

DecreaseDepth();

ExecuteNode(functionNode.Body);

}

return null;

}

if (abstractNode is FunctionExecutionNode functionExecutionNode)

{

var codeLevel = CodeDepthLevel;

var codeParent = CodeDepthParent;

var codeBlock = CodeBlock;

foreach (var key in Functions[functionExecutionNode.Function.Identifier].Keys)

{

foreach (var body in Functions[functionExecutionNode.Function.Identifier].Values)

{

var paramtrs = VariableTables[key].Keys.ToList();

for (int index = 1; index < functionExecutionNode.Parameters.Count; index++)

{

VariableTables[key][paramtrs[index]] = WorkOnNode(functionExecutionNode.Parameters[index]);

}

CodeBlock = key;

CodeDepthLevel = int.Parse(key.Split(":")[0]);

CodeDepthParent = int.Parse(key.Split(":")[^1]);

InFor = true;

WorkOnNode(body);

}

}

CodeDepthLevel = codeLevel;

CodeDepthParent = codeParent;

CodeBlock = codeBlock;

return null;

}

if (abstractNode is VariableNode varNode)

{

var codeBlock = GetCodeBlock();

while (codeBlock != "-1")

{

if (VariableTables[codeBlock].ContainsKey(varNode.Variable.Identifier))

{

return VariableTables[codeBlock][varNode.Variable.Identifier];

}

else

{

codeBlock = ModifyLocalCodeBlock(codeBlock);

}

}

}

if (abstractNode is SwitchNode switchNode)

{

var codeBlock = GetCodeBlock();

while (codeBlock != "-1")

{

if (VariableTables[codeBlock].ContainsKey(switchNode.Variable.Identifier))

{

break;

}

else

{

codeBlock = ModifyLocalCodeBlock(codeBlock);

}

}

SwitchValue = VariableTables[codeBlock][switchNode.Variable.Identifier];

WorkOnNode(switchNode.Body);

NeedToExecute = true;

FoundDefault = false;

InSwitch = false;

return null;

}

if (abstractNode is CaseNode caseNode)

{

NeedToExecute = false;

if (caseNode.Literal.Identifier.Replace("\'", "") == SwitchValue?.ToString())

{

NeedToExecute = true;

InSwitch = true;

}

return null;

}

if (abstractNode is WhileNode whileNode && (NeedToExecute || FoundDefault))

{

while (true)

{

var whileCondition = WorkOnNode(whileNode.Condition) as bool?;

if (whileCondition != null)

{

if (whileCondition == false)

{

break;

}

}

InFor = true;

var saveCodeBlock = CodeBlock;

var saveCodeLevel = CodeDepthLevel;

var saveCodeParent = CodeDepthParent;

WorkOnNode(whileNode.Body);

CodeBlock = saveCodeBlock;

CodeDepthLevel = saveCodeLevel;

CodeDepthParent = saveCodeParent;

}

return null;

}

if (abstractNode is IfNode ifNode && (NeedToExecute || FoundDefault))

{

var ifCondition = WorkOnNode(ifNode.Condition) as bool?;

object? ifResult;

if (ifCondition == true)

{

ifResult = WorkOnNode(ifNode.Body);

}

else

{

ifResult = WorkOnNode(ifNode.ElseBody);

}

return ifResult;

}

if (abstractNode is ForNode forNode && (NeedToExecute || FoundDefault))

{

IncreaseDepth();

WorkOnNode(forNode.First);

while (true)

{

var forCondition = WorkOnNode(forNode.Second) as bool?;

if (forCondition != null)

{

if (forCondition == false || FoundBreak)

{

DecreaseDepthOnlyForLevel();

FoundBreak = false;

break;

}

}

InFor = true;

var saveCodeBlock = CodeBlock;

var saveCodeLevel = CodeDepthLevel;

var saveCodeParent = CodeDepthParent;

WorkOnNode(forNode.Body);

WorkOnNode(forNode.Third);

CodeBlock = saveCodeBlock;

CodeDepthLevel = saveCodeLevel;

CodeDepthParent = saveCodeParent;

}

CodeDepthParent -= 1;

ExecuteNode(forNode);

return null;

}

if (abstractNode is UnaryOperationNode unaryOperationNode && (NeedToExecute || FoundDefault))

{

if (unaryOperationNode.Operator.Identifier == "++")

{

var variable = unaryOperationNode.Operand as VariableNode;

var codeBlock = GetCodeBlock();

if (variable.Variable.Type == "int")

{

VariableTables[codeBlock][variable.Variable.Identifier] =

(VariableTables[codeBlock][variable.Variable.Identifier] as int?)! + 1;

}

if (variable.Variable.Type == "float")

{

VariableTables[codeBlock][variable.Variable.Identifier] =

(VariableTables[codeBlock][variable.Variable.Identifier] as double?)! + 1;

}

if (variable.Variable.Type == "char")

{

VariableTables[codeBlock][variable.Variable.Identifier] =

(VariableTables[codeBlock][variable.Variable.Identifier] as char?)! + 1;

}

}

if (unaryOperationNode.Operator.Identifier == "--")

{

var variable = unaryOperationNode.Operand as VariableNode;

var codeBlock = GetCodeBlock();

if (variable.Variable.Type == "int")

{

VariableTables[codeBlock][variable.Variable.Identifier] =

(VariableTables[codeBlock][variable.Variable.Identifier] as int?)! - 1;

}

if (variable.Variable.Type == "float")

{

VariableTables[codeBlock][variable.Variable.Identifier] =

(VariableTables[codeBlock][variable.Variable.Identifier] as double?)! - 1;

}

if (variable.Variable.Type == "char")

{

VariableTables[codeBlock][variable.Variable.Identifier] =

(VariableTables[codeBlock][variable.Variable.Identifier] as char?)! - 1;

}

}

return null;

}

if (abstractNode is BinaryOperationNode binaryOperationNod && (NeedToExecute || FoundDefault))

{

switch (binaryOperationNod.Operator.Identifier)

{

case "=":

var codeBlock = GetCodeBlock();

if (binaryOperationNod.LeftNode is VariableNode variableNode)

{

while (codeBlock != "-1")

{

if (VariableTables[codeBlock].ContainsKey(variableNode.Variable.Identifier))

{

break;

}

else

{

codeBlock = ModifyLocalCodeBlock(codeBlock);

}

}

VariableTables[codeBlock][variableNode.Variable.Identifier] =

WorkOnNode(binaryOperationNod.RightNode);

}

if (binaryOperationNod.LeftNode is BinaryOperationNode binaryOperationNoed)

{

var leftNode = binaryOperationNoed.LeftNode as VariableNode;

var indexNode = WorkOnNode(binaryOperationNoed.RightNode) as int?;

while (codeBlock != "-1")

{

if (VariableTables[codeBlock].ContainsKey(leftNode?.Variable.Identifier))

{

break;

}

else

{

codeBlock = ModifyLocalCodeBlock(codeBlock);

}

}

var returnType3 = Semantic.GetReturnType(binaryOperationNoed.LeftNode).Replace("\*", "");

if (returnType3 == "int")

{

(VariableTables[codeBlock][leftNode.Variable.Identifier] as List<int>)

[int.Parse(indexNode.ToString())] =

int.Parse((WorkOnNode(binaryOperationNod.RightNode) as int?).ToString());

if (TEST)

{

TEST1 = CodeBlock;

CodeBlock = codeBlock;

TMPBOOL = true;

}

break;

}

if (returnType3 == "float")

{

(VariableTables[codeBlock][leftNode.Variable.Identifier] as List<double>)

[int.Parse(indexNode.ToString())] =

double.Parse((WorkOnNode(binaryOperationNod.RightNode) as double?).ToString());

break;

}

if (returnType3 == "char")

{

(VariableTables[codeBlock][leftNode.Variable.Identifier] as List<char>)

[int.Parse(indexNode.ToString())] =

char.Parse((WorkOnNode(binaryOperationNod.RightNode) as char?).ToString());

break;

}

if (returnType3 == "bool")

{

(VariableTables[codeBlock][leftNode.Variable.Identifier] as List<bool>)

[int.Parse(indexNode.ToString())] =

bool.Parse((WorkOnNode(binaryOperationNod.RightNode) as bool?).ToString());

break;

}

else

{

(VariableTables[codeBlock][leftNode.Variable.Identifier] as List<string>)[

int.Parse(indexNode.ToString())] = (WorkOnNode(binaryOperationNod.RightNode) as string);

break;

}

}

break;

case "==":

case "!=":

case "<":

case ">":

case "+":

case "-":

case "\*":

case "/":

var leftNodeReturnType = Semantic.GetReturnType(binaryOperationNod.LeftNode);

var rightNodeReturnType = Semantic.GetReturnType(binaryOperationNod.RightNode);

switch (leftNodeReturnType)

{

case "int":

var leftAsINT = WorkOnNode(binaryOperationNod.LeftNode) as int?;

switch (rightNodeReturnType)

{

case "int":

var rightAsINT = WorkOnNode(binaryOperationNod.RightNode) as int?;

switch (binaryOperationNod.Operator.Identifier)

{

case "==":

return leftAsINT == rightAsINT;

case "!=":

return leftAsINT != rightAsINT;

case "<":

return leftAsINT < rightAsINT;

case ">":

return leftAsINT > rightAsINT;

case "+":

return leftAsINT + rightAsINT;

case "-":

return leftAsINT - rightAsINT;

case "\*":

return leftAsINT \* rightAsINT;

case "/":

return leftAsINT / rightAsINT;

}

break;

case "float":

var rightAsFLOAT = WorkOnNode(binaryOperationNod.RightNode) as double?;

switch (binaryOperationNod.Operator.Identifier)

{

case "==":

return leftAsINT == rightAsFLOAT;

case "!=":

return leftAsINT != rightAsFLOAT;

case "<":

return leftAsINT < rightAsFLOAT;

case ">":

return leftAsINT > rightAsFLOAT;

case "+":

return leftAsINT + rightAsFLOAT;

case "-":

return leftAsINT - rightAsFLOAT;

case "\*":

return leftAsINT \* rightAsFLOAT;

case "/":

return leftAsINT / rightAsFLOAT;

}

break;

case "char":

var rightAsCHAR = WorkOnNode(binaryOperationNod.RightNode) as char?;

switch (binaryOperationNod.Operator.Identifier)

{

case "==":

return leftAsINT == rightAsCHAR;

case "!=":

return leftAsINT != rightAsCHAR;

case "<":

return leftAsINT < rightAsCHAR;

case ">":

return leftAsINT > rightAsCHAR;

case "+":

return leftAsINT + rightAsCHAR;

case "-":

return leftAsINT - rightAsCHAR;

case "\*":

return leftAsINT \* rightAsCHAR;

case "/":

return leftAsINT / rightAsCHAR;

}

break;

}

break;

case "float":

var leftAsFLOAT = WorkOnNode(binaryOperationNod.LeftNode) as double?;

switch (rightNodeReturnType)

{

case "int":

var rightAsINT = WorkOnNode(binaryOperationNod.RightNode) as int?;

switch (binaryOperationNod.Operator.Identifier)

{

case "==":

return leftAsFLOAT == rightAsINT;

case "!=":

return leftAsFLOAT != rightAsINT;

case "<":

return leftAsFLOAT < rightAsINT;

case ">":

return leftAsFLOAT > rightAsINT;

case "+":

return leftAsFLOAT + rightAsINT;

case "-":

return leftAsFLOAT - rightAsINT;

case "\*":

return leftAsFLOAT \* rightAsINT;

case "/":

return leftAsFLOAT / rightAsINT;

}

break;

case "float":

var rightAsFLOAT = WorkOnNode(binaryOperationNod.RightNode) as double?;

switch (binaryOperationNod.Operator.Identifier)

{

case "==":

return leftAsFLOAT == rightAsFLOAT;

case "!=":

return leftAsFLOAT != rightAsFLOAT;

case "<":

return leftAsFLOAT < rightAsFLOAT;

case ">":

return leftAsFLOAT > rightAsFLOAT;

case "+":

return leftAsFLOAT + rightAsFLOAT;

case "-":

return leftAsFLOAT - rightAsFLOAT;

case "\*":

return leftAsFLOAT \* rightAsFLOAT;

case "/":

return leftAsFLOAT / rightAsFLOAT;

}

break;

case "char":

var rightAsCHAR = WorkOnNode(binaryOperationNod.RightNode) as char?;

switch (binaryOperationNod.Operator.Identifier)

{

case "==":

return leftAsFLOAT == rightAsCHAR;

case "!=":

return leftAsFLOAT != rightAsCHAR;

case "<":

return leftAsFLOAT < rightAsCHAR;

case ">":

return leftAsFLOAT > rightAsCHAR;

case "+":

return leftAsFLOAT + rightAsCHAR;

case "-":

return leftAsFLOAT - rightAsCHAR;

case "\*":

return leftAsFLOAT \* rightAsCHAR;

case "/":

return leftAsFLOAT / rightAsCHAR;

}

break;

}

break;

case "char":

var leftAsCHAR = WorkOnNode(binaryOperationNod.LeftNode) as char?;

switch (rightNodeReturnType)

{

case "int":

var rightAsINT = WorkOnNode(binaryOperationNod.RightNode) as int?;

switch (binaryOperationNod.Operator.Identifier)

{

case "==":

return leftAsCHAR == rightAsINT;

case "!=":

return leftAsCHAR != rightAsINT;

case "<":

return leftAsCHAR < rightAsINT;

case ">":

return leftAsCHAR > rightAsINT;

case "+":

return leftAsCHAR + rightAsINT;

case "-":

return leftAsCHAR - rightAsINT;

case "\*":

return leftAsCHAR \* rightAsINT;

case "/":

return leftAsCHAR / rightAsINT;

}

break;

case "float":

var rightAsFLOAT = WorkOnNode(binaryOperationNod.RightNode) as double?;

switch (binaryOperationNod.Operator.Identifier)

{

case "==":

return leftAsCHAR == rightAsFLOAT;

case "!=":

return leftAsCHAR != rightAsFLOAT;

case "<":

return leftAsCHAR < rightAsFLOAT;

case ">":

return leftAsCHAR > rightAsFLOAT;

case "+":

return leftAsCHAR + rightAsFLOAT;

case "-":

return leftAsCHAR - rightAsFLOAT;

case "\*":

return leftAsCHAR \* rightAsFLOAT;

case "/":

return leftAsCHAR / rightAsFLOAT;

}

break;

case "char":

var rightAsCHAR = WorkOnNode(binaryOperationNod.RightNode) as char?;

switch (binaryOperationNod.Operator.Identifier)

{

case "==":

return leftAsCHAR == rightAsCHAR;

case "!=":

return leftAsCHAR != rightAsCHAR;

case "<":

return leftAsCHAR < rightAsCHAR;

case ">":

return leftAsCHAR > rightAsCHAR;

case "+":

return leftAsCHAR + rightAsCHAR;

case "-":

return leftAsCHAR - rightAsCHAR;

case "\*":

return leftAsCHAR \* rightAsCHAR;

case "/":

return leftAsCHAR / rightAsCHAR;

}

break;

}

break;

case "bool":

var leftAsBOOL = WorkOnNode(binaryOperationNod.LeftNode) as bool?;

var leftAsBOOLAsINT = 0;

if (leftAsBOOL == true)

{

leftAsBOOLAsINT = 1;

}

var rightAsBOOL = WorkOnNode(binaryOperationNod.RightNode) as bool?;

var rightAsBOOLAsINT = 0;

if (rightAsBOOL == true)

{

rightAsBOOLAsINT = 1;

}

switch (binaryOperationNod.Operator.Identifier)

{

case "==":

return leftAsBOOLAsINT == rightAsBOOLAsINT;

case "!=":

return leftAsBOOLAsINT != rightAsBOOLAsINT;

case "<":

return leftAsBOOLAsINT < rightAsBOOLAsINT;

case ">":

return leftAsBOOLAsINT > rightAsBOOLAsINT;

case "+":

return leftAsBOOLAsINT + rightAsBOOLAsINT;

case "-":

return leftAsBOOLAsINT - rightAsBOOLAsINT;

case "\*":

return leftAsBOOLAsINT \* rightAsBOOLAsINT;

case "/":

return leftAsBOOLAsINT / rightAsBOOLAsINT;

}

break;

case "string":

var leftAsSTRING = WorkOnNode(binaryOperationNod.LeftNode) as string;

var rightAsSTRING = WorkOnNode(binaryOperationNod.RightNode) as string;

switch (binaryOperationNod.Operator.Identifier)

{

case "==":

return leftAsSTRING == rightAsSTRING;

case "!=":

return leftAsSTRING != rightAsSTRING;

case "<":

return leftAsSTRING.CompareTo(rightAsSTRING) < 0 ? true : false;

case ">":

return leftAsSTRING.CompareTo(rightAsSTRING) > 0 ? true : false;

case "+":

return leftAsSTRING + rightAsSTRING;

case "-":

throw new Exception("Not Real to do - with 2 strings");

case "\*":

throw new Exception("Not Real to do \* with 2 strings");

case "/":

throw new Exception("Not Real to do / with 2 strings");

}

break;

}

break;

case "new":

var returnType = Semantic.GetReturnType(binaryOperationNod.LeftNode).Replace("\*", "");

switch (returnType)

{

case "int":

var intListCount = WorkOnNode(binaryOperationNod.RightNode) as int?;

var intList = new List<int>(int.Parse(intListCount.ToString()!));

for (var index = 0; index < intListCount; index++)

{

intList.Add(0);

}

return intList;

case "float":

var floatListCount = WorkOnNode(binaryOperationNod.RightNode) as int?;

var floatList = new List<double>(int.Parse(floatListCount.ToString()!));

for (var index = 0; index < floatListCount; index++)

{

floatList.Add(0);

}

return floatList;

case "char":

var charListCount = WorkOnNode(binaryOperationNod.RightNode) as int?;

var charList = new List<char>(int.Parse(charListCount.ToString()!));

for (var index = 0; index < charListCount; index++)

{

charList.Add('0');

}

return charList;

case "string":

var stringListCount = WorkOnNode(binaryOperationNod.RightNode) as int?;

var stringList = new List<string>(int.Parse(stringListCount.ToString()!));

for (var index = 0; index < stringListCount; index++)

{

stringList.Add("");

}

return stringList;

}

break;

case "[]":

var returnTypeForBracets = Semantic.GetReturnType(binaryOperationNod.LeftNode).Replace("\*", "");

switch (returnTypeForBracets)

{

case "int":

var intList = WorkOnNode(binaryOperationNod.LeftNode) as List<int>;

var intIndex = WorkOnNode(binaryOperationNod.RightNode) as int?;

return intList![int.Parse((intIndex.ToString()!))];

case "float":

var floatList = WorkOnNode(binaryOperationNod.LeftNode) as List<double>;

var floatIndex = WorkOnNode(binaryOperationNod.RightNode) as int?;

return floatList![int.Parse((floatIndex.ToString()!))];

case "char":

var charList = WorkOnNode(binaryOperationNod.LeftNode) as List<char>;

var charIndex = WorkOnNode(binaryOperationNod.RightNode) as int?;

return charList![int.Parse((charIndex.ToString()!))];

case "string":

var stringList = WorkOnNode(binaryOperationNod.LeftNode) as List<string>;

var stringIndex = WorkOnNode(binaryOperationNod.RightNode) as int?;

return stringList![int.Parse((stringIndex.ToString()!))];

}

break;

}

return null;

}

if (abstractNode is VariableTypeNode)

{

return null;

}

return null;

}

public string GetCodeBlock()

{

return CodeBlock;

}

public void ExecuteNode(AbstractNode? abstractNode)

{

if (abstractNode == null)

{

return;

}

if (abstractNode is StatementsNode statementsNode)

{

IncreaseDepth();

foreach (var node in statementsNode.Nodes)

{

ExecuteNode(node);

}

DecreaseDepthOnlyForLevel();

}

if (abstractNode is ForNode forNode)

{

ExecuteNode(forNode.Body);

}

if (abstractNode is IfNode ifNode)

{

ExecuteNode(ifNode.Body);

ExecuteNode(ifNode.ElseBody);

}

if (abstractNode is WhileNode whileNode)

{

ExecuteNode(whileNode.Body);

}

}

private void IncreaseDepth()

{

CodeDepthLevel += 1;

CodeDepthParent += 1;

var block = CodeBlock.Split(":");

block[0] = CodeDepthLevel.ToString();

CodeBlock = "";

CodeBlock += block[0];

for (int index = 1; index < block.Length; index++)

{

CodeBlock += $":{block[index]}";

}

CodeBlock += $":{CodeDepthParent}";

}

private void DecreaseDepth()

{

CodeDepthLevel -= 1;

CodeDepthParent -= 1;

CodeBlock = CodeBlock.Remove(CodeBlock.Length - 2);

var block = CodeBlock.Split(":");

block[0] = CodeDepthLevel.ToString();

CodeBlock = "";

CodeBlock += block[0];

for (int index = 1; index < block.Length; index++)

{

CodeBlock += $":{block[index]}";

}

}

private void DecreaseDepthOnlyForLevel()

{

CodeDepthLevel -= 1;

CodeBlock = CodeBlock.Remove(CodeBlock.Length - 2);

var block = CodeBlock.Split(":");

block[0] = CodeDepthLevel.ToString();

CodeBlock = "";

CodeBlock += block[0];

for (int index = 1; index < block.Length; index++)

{

CodeBlock += $":{block[index]}";

}

}

private string ModifyLocalCodeBlock(string codeBlockToModify)

{

codeBlockToModify = codeBlockToModify.Remove(codeBlockToModify.Length - 2);

var block = codeBlockToModify.Split(":");

block[0] = (int.Parse(block[0]) - 1).ToString();

codeBlockToModify = "";

codeBlockToModify += block[0];

for (int index = 1; index < block.Length; index++)

{

codeBlockToModify += $":{block[index]}";

}

return codeBlockToModify;

}

}