Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы трансляции

ОТЧЁТ
по лабораторной работе
на тему

Интерпретация исходного кода

Выполнил
Студент гр. 053502
Юрьев В.А.

Проверил
Ассистент кафедры информатики
Гриценко Н.Ю.

Минск 2023

СОДЕРЖАНИЕ

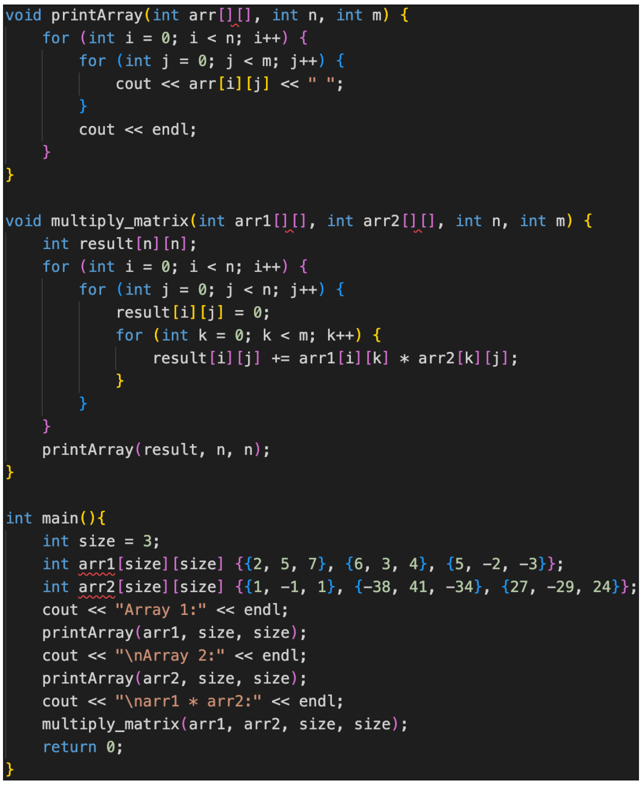
1 Цель работы... 3
2 Результаты 4
Приложение А Код программы 7

1 ЦЕЛЬ РАБОТЫ

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

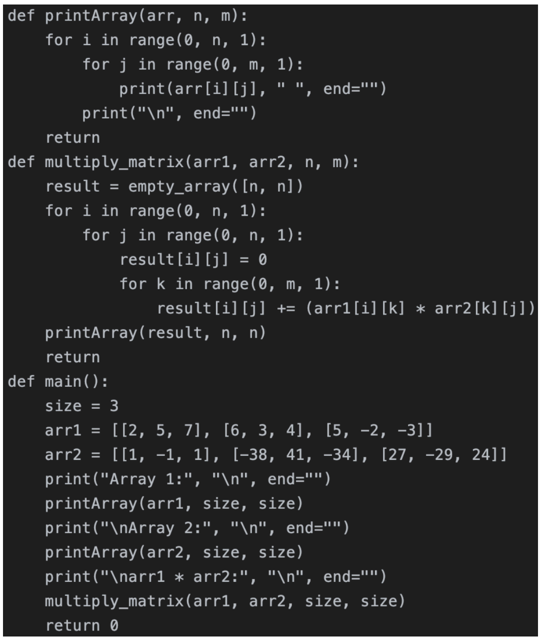
2 РЕЗУЛЬТАТЫ

Рассмотрим результат интерпретации программы, выполняющей умножение двух матриц (см. рисунки 2.1, 2.2, 2.3).



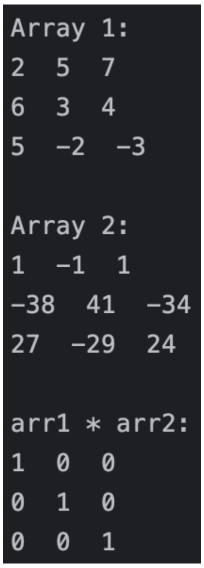
```
void printArray(int arr[][], int n, int m) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}

void multiply_matrix(int arr1[][], int arr2[][], int n, int m) {
    int result[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            result[i][j] = 0;
            for (int k = 0; k < m; k++) {
                result[i][j] += arr1[i][k] * arr2[k][j];
            }
        }
    }
    printArray(result, n, n);
}

int main(){
    int size = 3;
    int arr1[size][size] {{2, 5, 7}, {6, 3, 4}, {5, -2, -3}};
    int arr2[size][size] {{1, -1, 1}, {-38, 41, -34}, {27, -29, 24}};
    cout << "Array 1:" << endl;
    printArray(arr1, size, size);
    cout << "\nArray 2:" << endl;
    printArray(arr2, size, size);
    cout << "\narr1 * arr2:" << endl;
    multiply_matrix(arr1, arr2, size, size);
    return 0;
}
```

Рисунок 2.1 – Исходная программа, выполняющая умножение матриц



```
def printArray(arr, n, m):
    for i in range(0, n, 1):
        for j in range(0, m, 1):
            print(arr[i][j], " ", end="")
        print("\n", end="")
    return
def multiply_matrix(arr1, arr2, n, m):
    result = empty_array([n, n])
    for i in range(0, n, 1):
        for j in range(0, n, 1):
            result[i][j] = 0
            for k in range(0, m, 1):
                result[i][j] += (arr1[i][k] * arr2[k][j])
    printArray(result, n, n)
    return
def main():
    size = 3
    arr1 = [[2, 5, 7], [6, 3, 4], [5, -2, -3]]
    arr2 = [[1, -1, 1], [-38, 41, -34], [27, -29, 24]]
    print("Array 1:", "\n", end="")
    printArray(arr1, size, size)
    print("\nArray 2:", "\n", end="")
    printArray(arr2, size, size)
    print("\narr1 * arr2:", "\n", end="")
    multiply_matrix(arr1, arr2, size, size)
    return 0
```

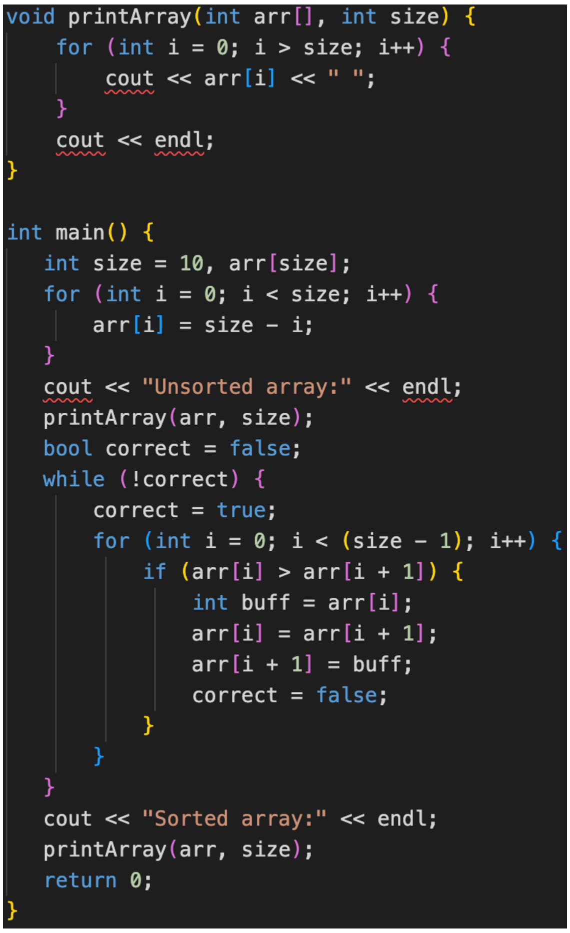
Рисунок 2.2 – Транслированный код программы, выполняющей умножение матриц



```
Array 1:
2  5  7
6  3  4
5  -2  -3

Array 2:
1  -1  1
-38  41  -34
27  -29  24

arr1 * arr2:
1  0  0
0  1  0
0  0  1
```

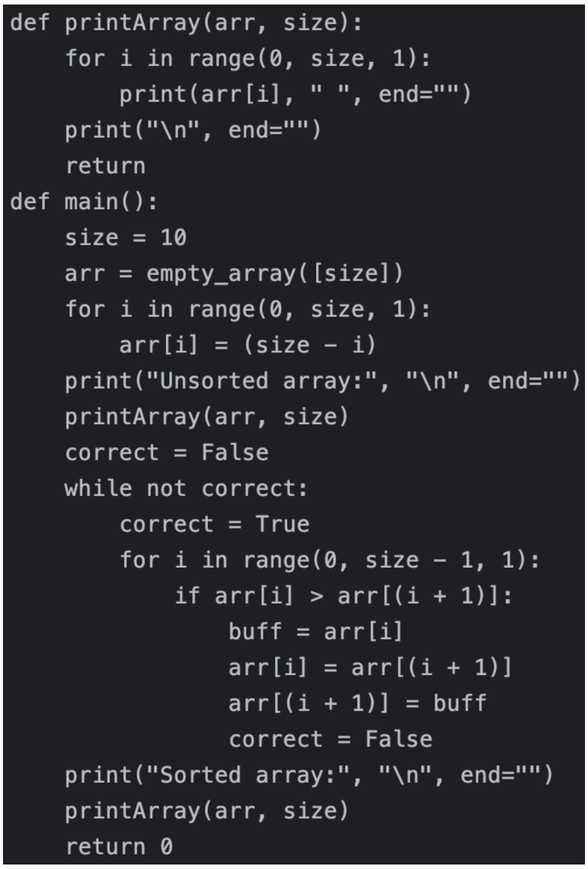
Рисунок 2.3 – Результат выполнения транслированной программы, выполняющей умножение матриц

Рассмотрим результат интерпретации программы, выполняющей сортировку пузырьком (см. рисунки 2.4, 2.5, 2.6).



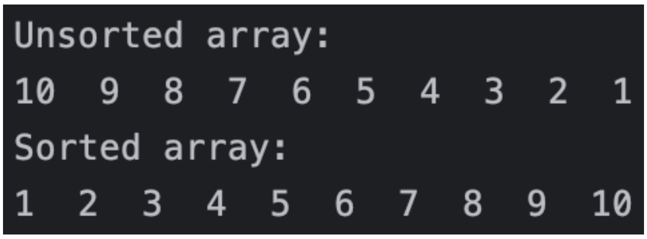
```
void printArray(int arr[], int size) {
    for (int i = 0; i > size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int size = 10, arr[size];
    for (int i = 0; i < size; i++) {
        arr[i] = size - i;
    }
    cout << "Unsorted array:" << endl;
    printArray(arr, size);
    bool correct = false;
    while (!correct) {
        correct = true;
        for (int i = 0; i < (size - 1); i++) {
            if (arr[i] > arr[i + 1]) {
                int buff = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = buff;
                correct = false;
            }
        }
    }
    cout << "Sorted array:" << endl;
    printArray(arr, size);
    return 0;
}
```

Рисунок 2.4 – Исходная программа, выполняющая сортировку пузырьком



```
def printArray(arr, size):
    for i in range(0, size, 1):
        print(arr[i], " ", end="")
    print("\n", end="")
    return
def main():
    size = 10
    arr = empty_array([size])
    for i in range(0, size, 1):
        arr[i] = (size - i)
    print("Unsorted array:", "\n", end="")
    printArray(arr, size)
    correct = False
    while not correct:
        correct = True
        for i in range(0, size - 1, 1):
            if arr[i] > arr[(i + 1)]:
                buff = arr[i]
                arr[i] = arr[(i + 1)]
                arr[(i + 1)] = buff
                correct = False
    print("Sorted array:", "\n", end="")
    printArray(arr, size)
    return 0
```

Рисунок 2.5 – Транслированный код программы, выполняющей сортировку пузырьком



```
Unsorted array:
10  9  8  7  6  5  4  3  2  1
Sorted array:
1  2  3  4  5  6  7  8  9  10
```

Рисунок 2.6 – Результат выполнения транслированной программы, выполняющей сортировку пузырьком

Проверка программ на ошибки выполняется интерпретатором.

ПРИЛОЖЕНИЕ А
Код программ

```
from functions.lexer import Lexer
from functions.parser import Parser
from functions.semantic import Semantic
from entities.print import PrintClass
from nodes.nodes_module import *
from entities.constants import execute_command, unary_operators, cast_var_types, empty_array


class Translator:
    def __init__(self, path):
        printer = PrintClass()

        lexer = Lexer()
        tokens = lexer.get_tokens(path)
        printer.print_tokens(tokens)

        parser = Parser(lexer)
        tree = parser.parse_block()
        printer.print_tree(tree)

        semantic = Semantic()
        semantic.analyze(tree)
        self.code = self._create_code(tree)
        printer.print_code(self.code)
        printer._print_name('CODE EXECUTE')
        self.execute()

    def _translate_statement(self, node, depth):
        result = ''
        for entity in node.nodes:
            result += depth * '\t' + f'{self._create_code(entity, depth)}\n'

        return result[:-1]

    def _translate_binary_operation(self, node):
        left = self._create_code(node.left_node)
        right = self._create_code(node.right_node)

        if isinstance(node.left_node, BinaryOperationNode) and node.left_node.operation.word !=
'[':
            left = '(' + left + ')'
        if isinstance(node.right_node, BinaryOperationNode) and node.right_node.operation.word
!= '[':
            right = '(' + right + ')'

        operation = node.operation.word
        if operation == '&&':
            operation = 'and'
        elif operation == '||':
            operation = 'or'
        elif operation == '[':
            return f'{left}[{right}]'

        return f'{left} {operation} {right}'

    def _translate_key_word(self, node):
        if node.word.word == 'endl':
            return '"\\n"'

    def _translate_cin(self, node):
        expression = f'{self._create_code(node.expression[0])}'
        data_type = node.expression[0].variable.token_type.split()[0].lower()
        inputs = 'input()' if data_type != 'int' and data_type != 'float' else f'{data_type}(input())'
        for var in node.expression[1:]:
            expression += f', {self._create_code(var)}'
            data_type = var.variable.token_type.split()[0].lower()
            inputs += ', input()' if data_type != 'int' and data_type != 'float' else f',
{data_type}(input())'

        return expression + ' = ' + inputs

    def _translate_cout(self, node):
        expression = f'print({self._create_code(node.expression[0])}'
        for val in node.expression[1:]:
            expression += f', {self._create_code(val)}'

        return expression + ', end="")'

    def _translate_while(self, node, depth):
        result = f'while {self._create_code(node.condition)}:\n'
        result += self._create_code(node.body, depth + 1)

        return result

    def _translate_for(self, node, depth):
        if not node.begin:
            result = f'while(True):\n'
            result += self._create_code(node.body, depth + 1)

            return result

        if isinstance(node.begin, BinaryOperationNode):
            variable = f'{node.begin.left_node.variable.word}'
            begin = node.begin.right_node.constant.word
        else:
            raise Exception('Invalid FOR variable define')

        if isinstance(node.condition, BinaryOperationNode):
            operation = node.condition.operation.word
            end = self._create_code(node.condition.right_node)
            if operation == '<':
                loop_range = f'{begin}, {end}'
            elif operation == '<=':
                loop_range = f'{begin}, {end} + 1'
            elif operation == '>':
                loop_range = f'{begin}, {end}'
            elif operation == '>=':
                loop_range = f'{begin}, {end} - 1'
            else:
                raise Exception('Invalid FOR condition define')
        else:
            raise Exception('Invalid FOR condition define')

        if isinstance(node.step, UnaryOperationNode):
            step = '-1' if node.step.operation.word == '--' else '1'
        elif isinstance(node.step, BinaryOperationNode) and node.step.operation == '+=':
            step = self._create_code(node.step.right_node)
        elif isinstance(node.step, BinaryOperationNode) and node.step.operation == '-=':
            step = f'-{self._create_code(node.step.right_node)}'
        else:
            raise Exception('Invalid FOR step define')

        return f'for {variable} in range({loop_range}, {step}):\n' + self._create_code(node.body,
depth + 1)

    def _translate_if_condition(self, node, depth):
        result = f'if {self._create_code(node.condition)}:\n'
        result += self._create_code(node.body, depth + 1)

        if node.else_condition:
            statement = self._create_code(node.else_condition, depth)

            word = ''
            for i in range(2):
                word += statement[i]
            if word == 'if':
                statement = '\t' * depth + 'elif' + statement[2:]
            else:
                new_statement = ''
                for s in statement.split('\n'):
                    new_statement += f'\t{s}\n'

                statement = '\t' * depth + 'else:\n' + new_statement

            result += f'\n{statement}'
        return result

    def _translate_function(self, node, depth):
        result = f'def {node.name.word}('
        if node.parameters:
            for p in node.parameters:
                result += f'{self._create_code(p)}, '
            result = result[:-2]
        result += '):\n' + self._create_code(node.body, depth + 1)
        return result

    def _translate_function_call(self, node):
        result = f'{node.name.word}('
        if node.parameters:
            for p in node.parameters:
                result += f'{self._create_code(p)}, '
            result = result[:-2]
        result += ')'
        return result

    def _translate_return(self, node):
        if isinstance(node.statement, Token) and node.statement.word == 'void':
            statement = ''
        else:
            statement = self._create_code(node.statement)
        return 'return ' + (statement if statement else '')

    def _translate_cast(self, node):
        return
f'{cast_var_types[node.cast_type.word.lower()]}({self._create_code(node.expression)})'

    def _empty_array(self, shape):
        if not shape:
            return None

        if not isinstance(shape, ConstantNode):
            return 'empty_array([' + ', '.join(self._create_code(s) for s in shape) + '])'

        if len(shape) == 1:
            return [0] * int(shape[0].constant.word)

        return [self._empty_array(shape[1:]) for _ in range(int(shape[0].constant.word))]

    def _string_array(self, item):
        if isinstance(item, list):
            result = ', '.join(self._string_array(i) for i in item)
            return f'[{result}]'
        else:
            return self._create_code(item)

    def _create_code(self, node, depth=0):
        if isinstance(node, Token):
            return node.word
        if isinstance(node, StatementsNode):
            return self._translate_statement(node, depth)
        elif isinstance(node, UnaryOperationNode):
            if node.operation.word == '!':
                return f'not {self._create_code(node.node)}'
            elif node.operation.word in unary_operators:
                node.operation.word = '+=' if node.operation.word == '++' else '-='
                return self._create_code(
                    BinaryOperationNode(node.operation, node.node, ConstantNode(Token('1', 'INT
CONSTANT'))))
        elif isinstance(node, BinaryOperationNode):
            return self._translate_binary_operation(node)
        elif isinstance(node, VariableNode):
            return node.variable.word
        elif isinstance(node, ConstantNode):
            constant = node.constant
            if constant.token_type == 'BOOL CONSTANT':
                return constant.word[0].upper() + constant.word[1:]
            return constant.word
        elif isinstance(node, KeyWordNode):
            return self._translate_key_word(node)
        elif isinstance(node, CinNode):
            return self._translate_cin(node)
        elif isinstance(node, CoutNode):
            return self._translate_cout(node)
        elif isinstance(node, WhileNode):
            return self._translate_while(node, depth)
        elif isinstance(node, ForNode):
            return self._translate_for(node, depth)
        elif isinstance(node, IfNode):
            return self._translate_if_condition(node, depth)
        elif isinstance(node, FunctionNode):
            return self._translate_function(node, depth)
        elif isinstance(node, FunctionCallNode):
            return self._translate_function_call(node)
        elif isinstance(node, SwitchNode):
            pass
        elif isinstance(node, CaseNode):
            pass
        elif isinstance(node, ArrayDefinition):
            return f'{node.variable.variable.word} = {self._empty_array(node.sizes)}'
        elif isinstance(node, Array):
            return self._string_array(node.elements)
        elif isinstance(node, ReturnNode):
            return self._translate_return(node)
        elif isinstance(node, CastNode):
            return self._translate_cast(node)

    def execute(self):
        locals()['empty_array'] = empty_array
        exec(self.code + execute_command, locals())
```