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](#_Toc133418928)

[2 Результаты 4](#_Toc133418929)

[Приложение А Код программы 7](#_Toc133418930)

# **ЦЕЛЬ РАБОТЫ**

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

# **РЕЗУЛЬТАТЫ**

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

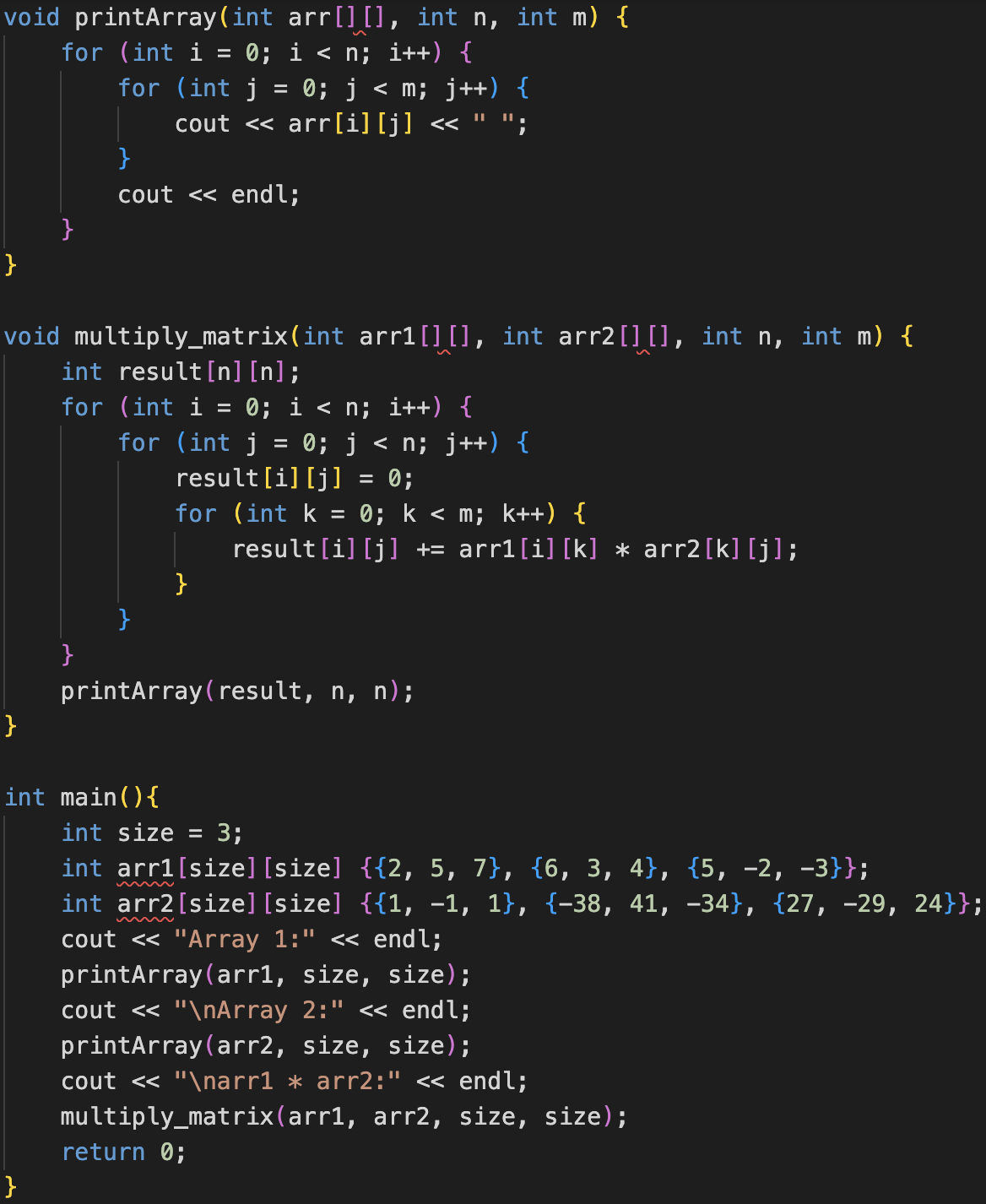


Рисунок 2.1 – Исходная программа, выполняющая умножение матриц

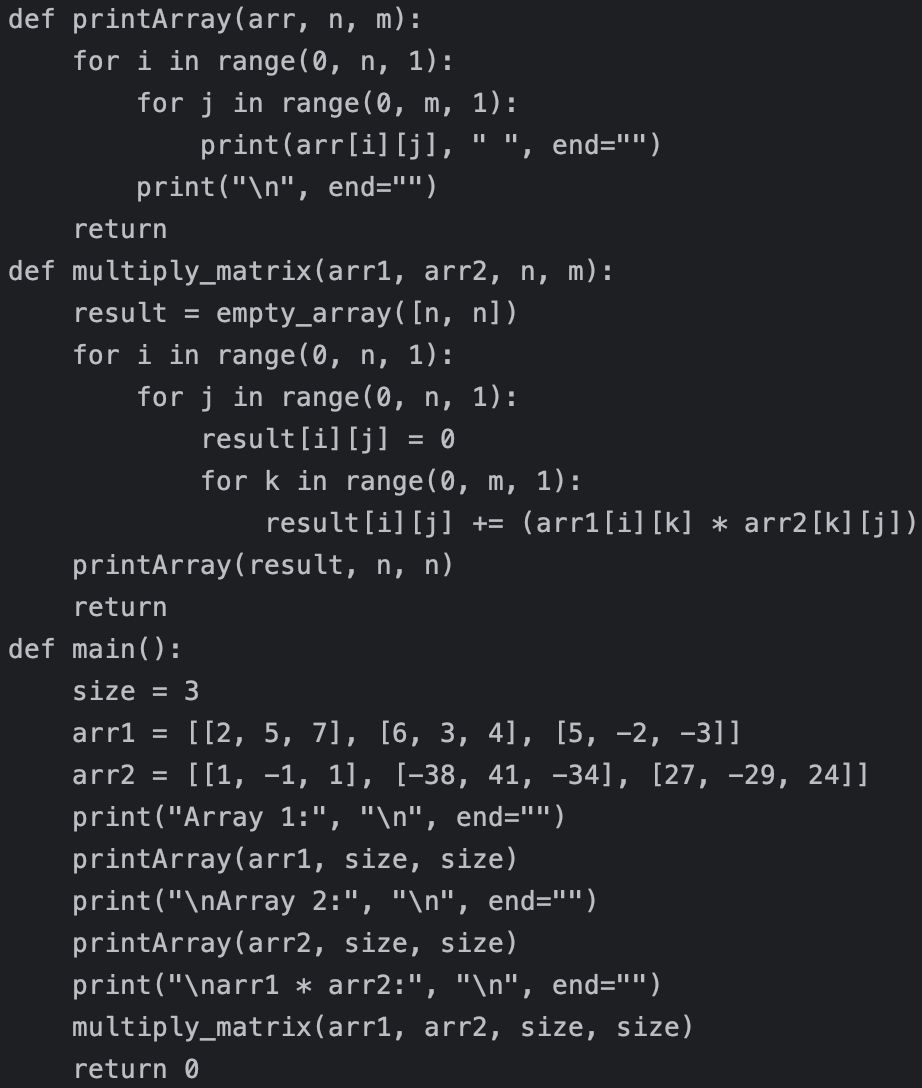


Рисунок 2.2 – Транслированный код программы, выполняющей умножение матриц

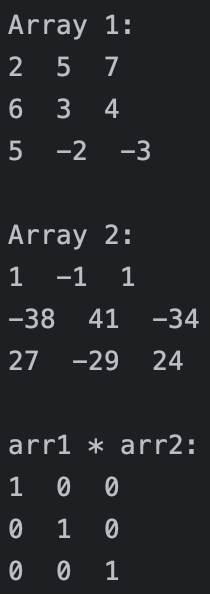


Рисунок 2.3 – Результат выполнения транслированной программы, выполняющей умножение матриц

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

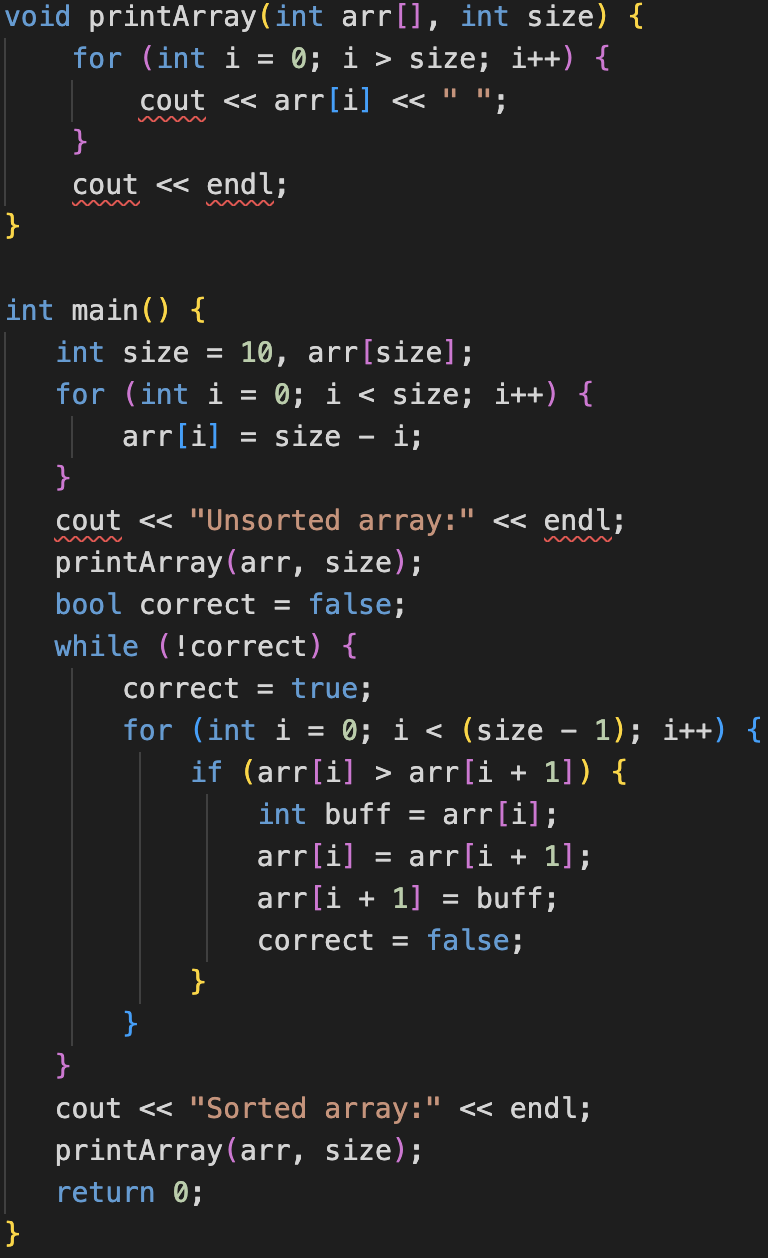


Рисунок 2.4 – Исходная программа, выполняющая сортировку пузырьком

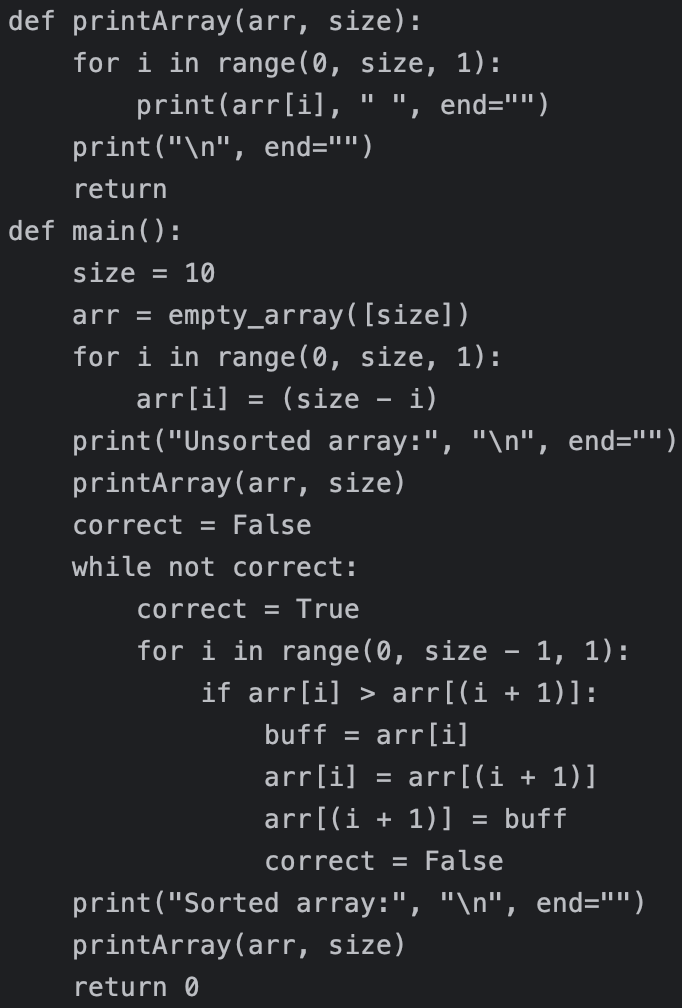


Рисунок 2.5 – Транслированный код программы, выполняющей сортировку пузырьком

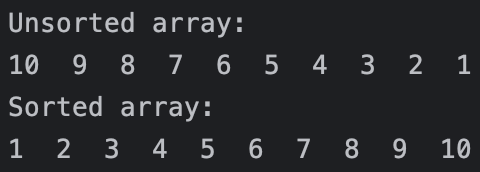


Рисунок 2.6 – Результат выполнения транслированной программы, выполняющей сортировку пузырьком

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())