Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина «Методы трансляции»

|  |  |
| --- | --- |
|  |  |

**ОТЧЕТ**

к лабораторной работе № 5

на тему «Интерпретация исходного кода»

Выполнил             А. К. Хрищанович

Проверил                          Н. Ю. Гриценко

Минск 2024

**СОДЕРЖАНИЕ**

[1 Постановка задачи 3](#_Toc157722973)

[2 Краткие теоретические сведения 4](#_Toc157722974)

[3 Результаты выполнения лабораторной работы 5](#_Toc157722975)

[Выводы 6](#_Toc157722976)

[Список использованных источников 11](#_Toc157722977)

[Приложение А (обязательное) Листинг исходного кода 12](#_Toc157722978)

# **1 ПОСТАНОВКА ЗАДАЧИ**

Целью выполнения данной лабораторной работы является на основе результатов анализа лабораторных работы 1-4 выполнить трансляцию программы с языка программирования С++ на язык программирования Python, после чего выполнить интерпретацию программы.

# **2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

К этапам трансляции относятся следующие этапы:

– лексический анализ;

– синтаксический анализ;

– семантический анализ;

– оптимизация;

– генерация кода.

На этапе генерации компилятор создает код, который представляет собой набор инструкций, понятных для целевой аппаратной платформы, итоговый файл компилируется в исполняемый файл, который может быть запущен на целевой платформе без необходимости наличия кода.

Фаза эмуляции интерпретатора происходит во время выполнения программы. В отличие от компилятора, интерпретатор работает с кодом напрямую, без предварительной генерации машинного кода.

Лексический анализатор – первый этап трансляции. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в лексемы или значащие последовательности. Лексема – это элементарная единица, которая может являться ключевым словом, идентификатором, константным значением. Для каждой лексемы анализатор строит токен, который по сути является кортежем, содержащим имя и значение.[1]

Синтаксический анализатор выясняет, удовлетворяют ли предложения, из которых состоит исходная программа, правилам грамматики языка программирования. Синтаксический анализатор получает на вход результат лексического анализатора и разбирает его в соответствии с грамматикой. Результат синтаксического анализа обычно представляется в виде синтаксического дерева разбора.[2]

Семантический анализ обычно заключается в проверке правильности типа и вида всех идентификаторов и данных, используемых в программе.

Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице идентификаторов для последующего использования в процессе генерации промежуточного кода.

В данной лабораторной работе были использованы результаты анализа лексического, синтаксического и семантического анализаторов, после чего каждый узел дерева разбора был переведен с языка программирования С++ на язык программирования Python. После чего была выполнена интерпретация программ. Программами называются тестовые исходные коды, представленные в лабораторной работе 1.

# **3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ             РАБОТЫ**

В ходе лабораторной работы был реализован транслятор программ с языка программирования С++ на язык программирования Python с последующей интерпретацией кода.

Листинг первого тестового кода представлен на рисунке 3.1.

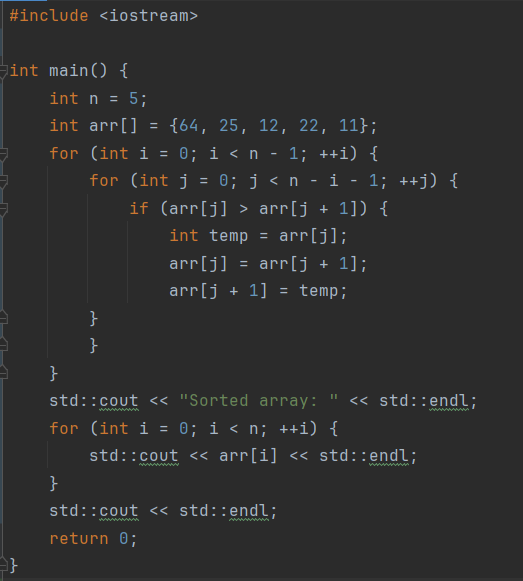


Рисунок 3.1 – Листинг первого тестового кода

Результат трансляции первого тестового кода представлен   
на рисунке 3.2.

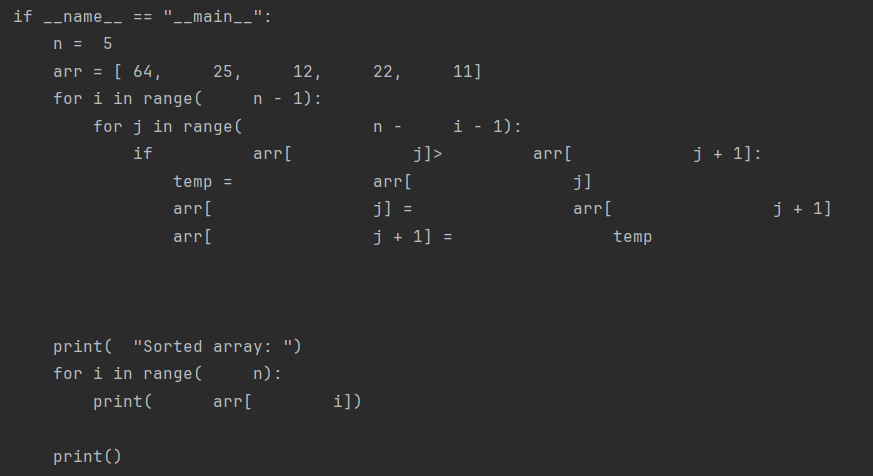


Рисунок 3.2 – Результат трансляции первого исходного кода

Результат интерпретации первого исходного кода представлен   
на рисунке 3.3.

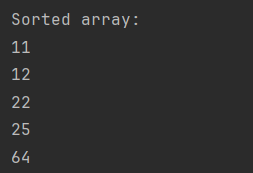


Рисунок 3.3 – Результат интерпретации первого исходного кода

Листинг второго исходного кода представлен на рисунке 3.4.

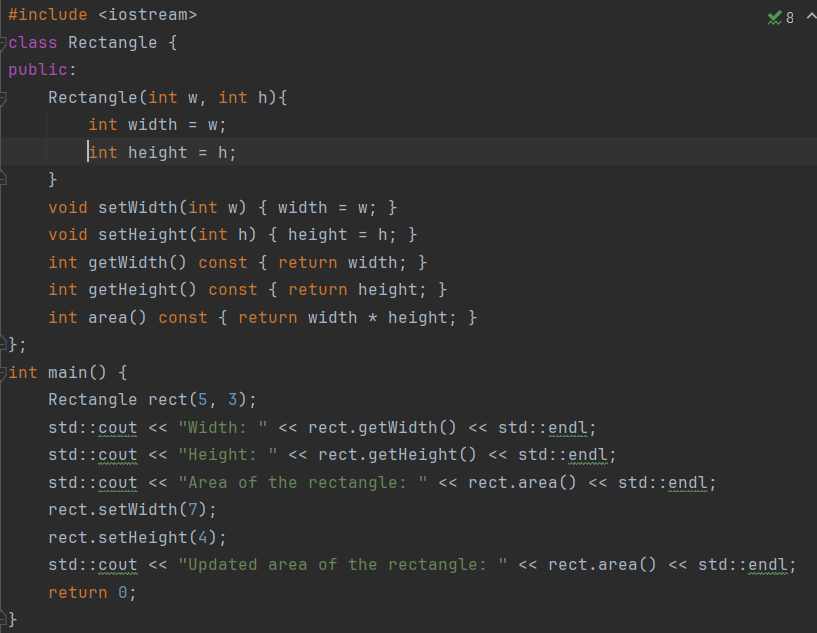


Рисунок 3.4 – Листинг второго тестового кода

Результат трансляции второго тестового кода представлен   
на рисунке 3.5.

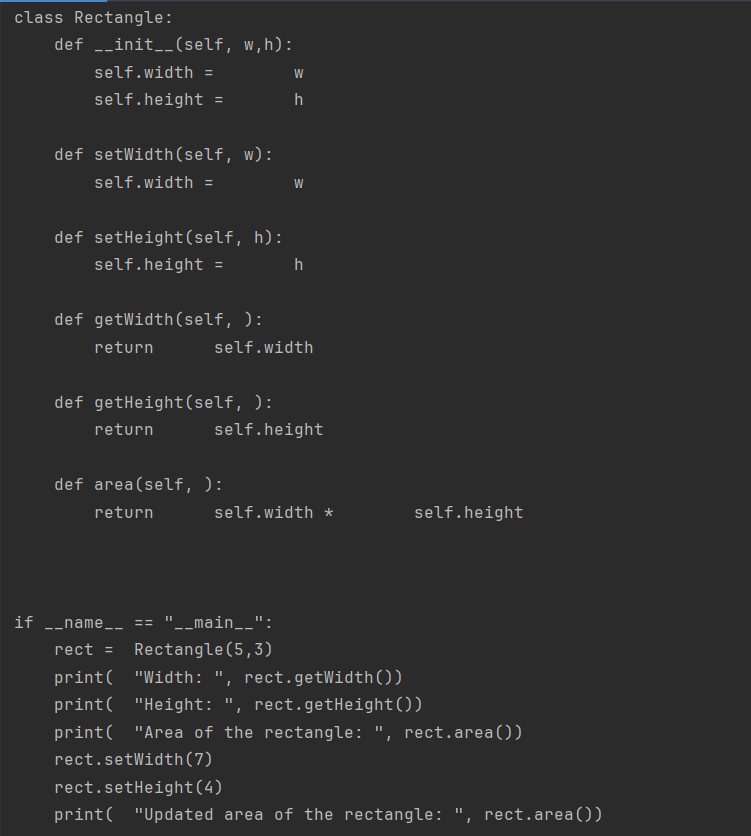


Рисунок 3.5 – Результат трансляции второго тестового кода

Результат интерпретации второго тестового кода представлен на рисунке 3.6.

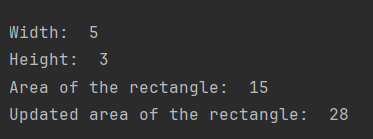


Рисунок 3.6 – Результат интерпретации второго тестового кода

Листинг третьего тестового кода представлен на рисунке 3.7.

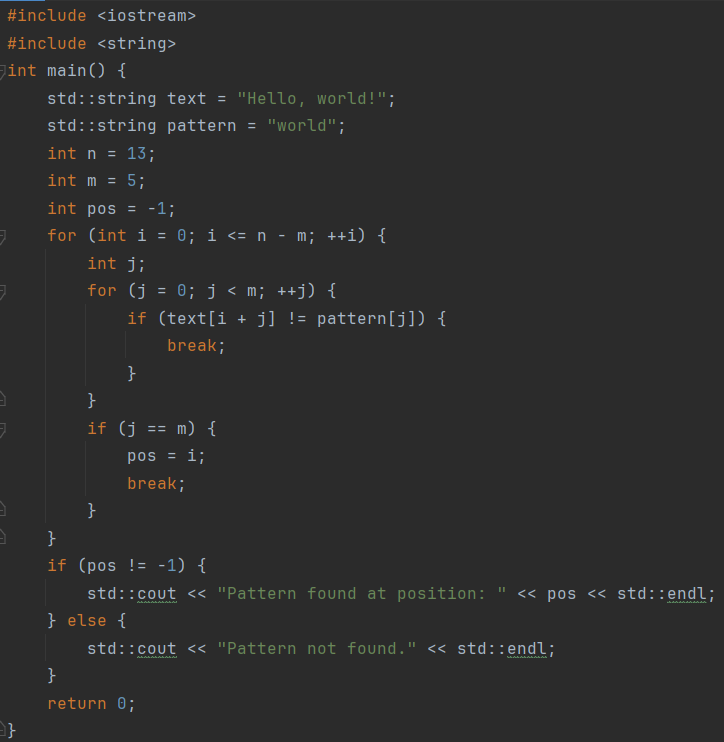


Рисунок 3.7 – Листинг третьего тестового кода

Результат трансляции третьего тестового кода представлен   
на рисунке 3.8.

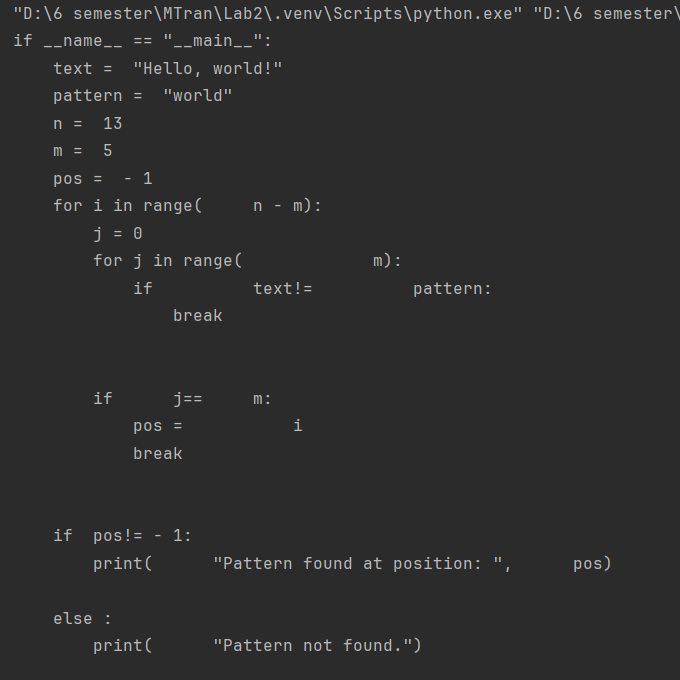


Рисунок 3.8 – Результат трансляции третьего тестового кода

Результат интерпретации третьего тестового кода представлен на рисунке 3.9.

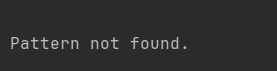


Рисунок 3.9 – Результат интерпретации третьего исходного кода

Таким образом в ходе лабораторной работы был реализован интерпретатор для программ на языке С++, который переводит их на язык программирования Python после чего проводит интерпретацию полученного при трансляции кода.

# **ВЫВОДЫ**

В ходе лабораторной работы был реализован был реализован интерпретатор для программ на языке С++, который переводит их на язык программирования Python после чего проводит интерпретацию полученного при трансляции кода.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Лексический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/. – Дата доступа: 27.02.2024.
2. Синтаксический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/. – Дата доступа: 27.02.2024.
3. Введение в С++ [Электронный ресурс]. – Режим доступа: <https://metanit.com/cpp/tutorial/2.5.php>. – Дата доступа: 28.02.2024.
4. Типы данных [Электронный ресурс]. – Режим доступа: <https://metanit.com/cpp/tutorial/2.3.php>. – Дата доступа: 28.02.2024.
5. Операторы в С++ [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/cpp/c-operators>. – Дата доступа: 27.02.2024.
6. Функции С++ [Электронный ресурс]. – Режим доступа: https://metanit.com/cpp/tutorial/3.1.php. – Дата доступа: 27.02.2024.
7. Классы С++ [Электронный ресурс]. – Режим доступа: <https://ravesli.com/urok-113-klassy-obekty-i-metody-klassov/>. – Дата доступа: 27.02.2024.

## **ПРИЛОЖЕНИЕ А**

## **(обязательное)**

## **Листинг исходного кода**

Листинг 1 – Программный код parser.py

from parser import parser

class Translator:

def \_\_init\_\_(self):

self.code\_python = ''

self.private\_member = []

self.variable\_list = []

self.temp\_list = ''

self.in\_class = False

self.in\_if = False

self.depth = 0

self.temp\_depth = 0

self.count\_compare = 0

def declare\_translator(self, node):

self.code\_python += '\t' \* self.depth

variable\_exists = False

for name, \_ in self.variable\_list:

if node.name == name:

variable\_exists = True

break

if not variable\_exists:

if self.in\_class:

self.variable\_list.append((node.name, True))

else:

self.variable\_list.append((node.name, False))

for name, check in self.variable\_list:

if node.name == name:

if check:

self.code\_python += f'self.{node.name}'

else:

self.code\_python += f'{node.name}'

break

for member\_declare in node.children:

if member\_declare.type == 'Assignment':

assignment\_node = member\_declare.name

self.code\_python += f' {assignment\_node} '

if member\_declare.type == 'Value':

value\_node = member\_declare.name

self.code\_python += f' {value\_node}'

if member\_declare.type == 'Variable':

self.declare\_translator(member\_declare)

if member\_declare.type == 'Operator':

operator\_node = member\_declare.name

self.code\_python += f' {operator\_node}'

if member\_declare.type == 'DotOperator':

self.code\_python += member\_declare.name

if member\_declare.type == 'Method f':

self.code\_python += member\_declare.name

self.function\_translator(member\_declare)

if member\_declare.type == 'Comparison':

if self.in\_for and not self.in\_if:

self.code\_python = self.code\_python[:-1]

else:

self.code\_python += member\_declare.name

if member\_declare.type == 'Block' and (member\_declare.name == 'Declare Array' or member\_declare.name == 'Array'):

self.code\_python += '['

self.block\_translator(member\_declare)

if member\_declare.type == 'End Block':

self.code\_python += ']'

if member\_declare.type == 'Square Block' and node.type == 'Array':

self.code\_python += '['

self.block\_translator(member\_declare)

if member\_declare.type == 'End Square Block' and node.type == 'Array':

self.code\_python += ']'

if member\_declare.type == 'Array':

self.declare\_translator(member\_declare)

if 'Semantic' in member\_declare.type:

self.code\_python += member\_declare.type

break

def parameters\_translator(self, node):

for member\_param in node.children:

if member\_param.type == 'Variable':

self.declare\_translator(member\_param)

if member\_param.type == 'Declare':

if self.in\_class:

self.in\_class = False

self.declare\_translator(member\_param)

self.in\_class = True

else:

self.declare\_translator(member\_param)

if member\_param.type == 'Array':

self.declare\_translator(member\_param)

if member\_param.type == 'Comma':

self.code\_python += member\_param.name

if member\_param.type == 'Value':

self.declare\_translator(member\_param)

def function\_translator(self, node):

for member\_function in node.children:

if member\_function.type == 'Parameters':

self.temp\_depth = self.depth

self.depth = 0

if self.in\_class:

self.code\_python += '(self, '

self.parameters\_translator(member\_function)

self.code\_python += ')'

else:

self.code\_python += '('

self.parameters\_translator(member\_function)

self.code\_python += ')'

self.depth = self.temp\_depth

if member\_function.type == 'Block':

self.depth += 1

self.code\_python += f':\n'

self.block\_translator(member\_function)

if member\_function.type == 'End Block':

self.depth -= 1

self.code\_python += '\n'

def main\_translator(self, node):

for member\_function in node.children:

if member\_function.type == 'Block':

self.depth += 1

self.code\_python += f':\n'

self.block\_translator(member\_function)

if member\_function.type == 'End Block':

self.depth -= 1

self.code\_python += '\n'

def access\_specifier\_translator(self, node):

for member\_access in node.children:

if member\_access.type == 'Declare':

self.declare\_translator(member\_access)

if member\_access.type == 'Statement':

self.code\_python += '\n'

if member\_access.type == 'Constructure':

self.code\_python += '\t' \* self.depth

self.code\_python += f'def \_\_init\_\_'

self.function\_translator(member\_access)

if member\_access.type == 'Function':

self.code\_python += '\t' \* self.depth

self.code\_python += f'def {member\_access.name}'

self.function\_translator(member\_access)

def return\_translator(self, node):

for member\_return in node.children:

if member\_return.type == 'Variable':

self.declare\_translator(member\_return)

if member\_return.type == 'Statement':

self.code\_python += '\n'

if member\_return.type == 'Value':

self.declare\_translator(member\_return)

def object\_translator(self, node):

for member\_object in node.children:

if member\_object.type == 'DotOperator':

self.code\_python += member\_object.name

if member\_object.type == 'Function Call':

self.code\_python += member\_object.name

self.function\_translator(member\_object)

def print\_translator(self, node):

for member\_print in node.children:

if member\_print.type == 'Value':

self.declare\_translator(member\_print)

self.code\_python += ', '

if member\_print.type == 'Object':

self.code\_python += member\_print.name

self.object\_translator(member\_print)

self.code\_python += ', '

if member\_print.type == 'Variable':

self.declare\_translator(member\_print)

self.code\_python += ', '

if member\_print.type == 'Array':

self.declare\_translator(member\_print)

self.code\_python += ', '

def for\_param\_translator(self, node):

count = 0

count\_comparison = 0

for member\_param in node.children:

if member\_param.type == 'Statement':

count += 1

if count == 0:

if member\_param.type in ('Declare', 'Variable'):

self.code\_python += member\_param.name

if count == 1:

if member\_param.type in ('Declare', 'Variable'):

self.code\_python += ' in range('

self.declare\_translator(member\_param)

self.code\_python += ')'

def for\_translator(self, node):

for member\_for in node.children:

if member\_for.type == 'Parameters':

self.for\_param\_translator(member\_for)

if member\_for.type == 'Block':

self.depth += 1

self.code\_python += f':\n'

self.block\_translator(member\_for)

if member\_for.type == 'End Block':

self.depth -= 1

self.in\_for = False

self.code\_python += '\n'

def if\_translator(self, node):

for member\_if in node.children:

if member\_if.type == 'Parameters':

self.parameters\_translator(member\_if)

if member\_if.type == 'Block':

self.depth += 1

self.code\_python += f':\n'

self.block\_translator(member\_if)

if member\_if.type == 'End Block':

self.in\_if = False

self.depth -= 1

self.code\_python += '\n'

def block\_translator(self, node):

parent\_node = node.parent

for member\_block in node.children:

if member\_block.name in ('public', 'protected', 'private'):

self.access\_specifier\_translator(member\_block)

if member\_block.type == 'Variable':

self.declare\_translator(member\_block)

if member\_block.type == 'Declare':

if member\_block.children:

self.declare\_translator(member\_block)

else:

self.code\_python += '\t' \* self.depth

variable\_exists = False

for name, \_ in self.variable\_list:

if member\_block.name == name:

variable\_exists = True

break

if not variable\_exists:

if self.in\_class:

self.variable\_list.append((member\_block.name, True))

else:

self.variable\_list.append((member\_block.name, False))

for name, check in self.variable\_list:

if member\_block.name == name:

if check:

self.code\_python += f'self.{member\_block.name}'

else:

self.code\_python += f'{member\_block.name}'

break

if member\_block.data\_type in ('int', 'long long', 'long', 'short', 'unsigned short', 'unsigned int', \

'unsigned long long', 'unsigned long', 'float', 'double', 'long double'):

self.code\_python += f' = 0'

if member\_block.data\_type in ('str', 'string', 'signed char', 'char', 'unsigned char', 'wchar\_t', 'char8\_t', 'char16\_t', 'char32\_t'):

self.code\_python += f"= ''"

if member\_block.type == 'Declare Array':

self.declare\_translator(member\_block)

if member\_block.type == 'Array':

self.declare\_translator(member\_block)

if member\_block.type == 'Statement':

self.code\_python += '\n'

if member\_block.type == 'ReturnStatement':

self.code\_python += '\t' \* self.depth

if parent\_node.name != 'main':

self.code\_python += f'{member\_block.name} '

self.return\_translator(member\_block)

else:

continue

if member\_block.type == 'Function Call':

self.code\_python += '\t' \* self.depth

self.code\_python += member\_block.name

self.function\_translator(member\_block)

if member\_block.type == 'Class':

self.temp\_list = member\_block

self.class\_translator(member\_block)

if member\_block.type == 'Cout':

self.code\_python += '\t' \* self.depth

self.code\_python += 'print('

self.print\_translator(member\_block)

if member\_block.type == 'Endl':

if self.code\_python[-2] == ',' and self.code\_python[-1] == ' ':

self.code\_python = self.code\_python[:-2] + ')'

else:

self.code\_python += ')'

if member\_block.type == 'Object':

self.code\_python += '\t' \* self.depth

self.code\_python += member\_block.name

self.object\_translator(member\_block)

if member\_block.type == 'ForLoop':

self.code\_python += '\t' \* self.depth

self.in\_for = True

self.code\_python += 'for '

self.for\_translator(member\_block)

if member\_block.type == 'IfStatement':

self.code\_python += '\t' \* self.depth

self.in\_if = True

self.code\_python += 'if '

self.if\_translator(member\_block)

if member\_block.type == 'ElseStatement':

self.code\_python += '\t' \* self.depth

self.code\_python += 'else '

self.if\_translator(member\_block)

if member\_block.type == 'Break':

self.code\_python += '\t' \* self.depth

self.code\_python += 'break'

if member\_block.type == 'Value':

self.declare\_translator(member\_block)

if member\_block.type == 'Comma':

self.code\_python += ', '

def class\_translator(self, node):

for member\_class in node.children:

if member\_class.type == 'Block':

self.depth += 1

self.code\_python += f':\n'

self.block\_translator(member\_class)

if member\_class.type == 'End Block':

self.depth -= 1

self.in\_class = False

self.code\_python += '\n'

if member\_class.type == 'Object':

self.code\_python += '\t' \* self.depth

self.code\_python += f'{member\_class.name} = {self.temp\_list.name}'

self.function\_translator(member\_class)

def translator(self, syntax\_tree):

root = syntax\_tree.children

for member\_root in root:

if member\_root.type == 'Class':

self.in\_class = True

self.code\_python += f'class {member\_root.name}'

self.class\_translator(member\_root)

if member\_root.type == 'Statement':

self.code\_python += '\n'

if member\_root.type == 'Function':

if member\_root.name == 'main':

self.code\_python += f'if \_\_name\_\_ == "\_\_main\_\_"'

self.main\_translator(member\_root)

else:

self.code\_python += '\t' \* self.depth

self.code\_python += f'def {member\_root.name}'

self.function\_translator(member\_root)

if member\_root.type == 'Declare':

self.code\_python += '\t' \* self.depth

self.declare\_translator(member\_root)

if member\_root.type == 'Declare array':

self.code\_python += '\t' \* self.depth

self.declare\_translator(member\_root)

if 'Semantic' in member\_root.type:

self.code\_python += member\_root.type

break

translator = Translator()

translator.translator(parser())

print(translator.code\_python)

exec(translator.code\_python)