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             Е. А. Киселёва

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 8](#_Toc157722977)

[Приложение А (обязательное) Листинг исходного кода 9](#_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 первой части тестового кода, представляющего собой быструю сортировку, представлен на рисунке 3.1.

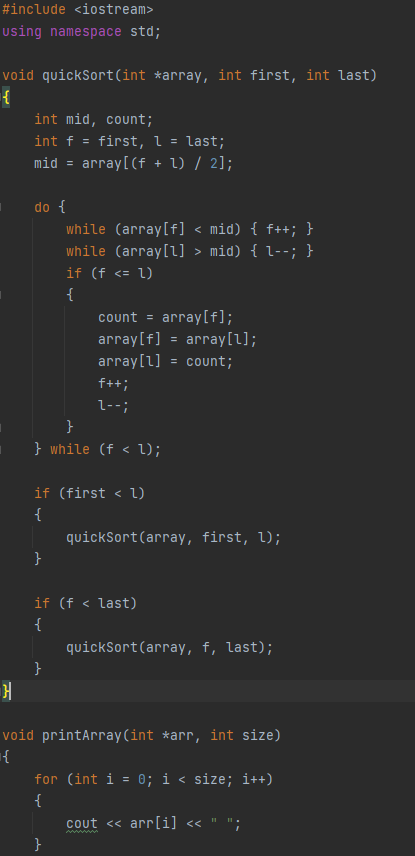


Рисунок 3.1 – Листинг первой части тестового кода

Листинг второй части тестового кода представлен на рисунке 3.2.

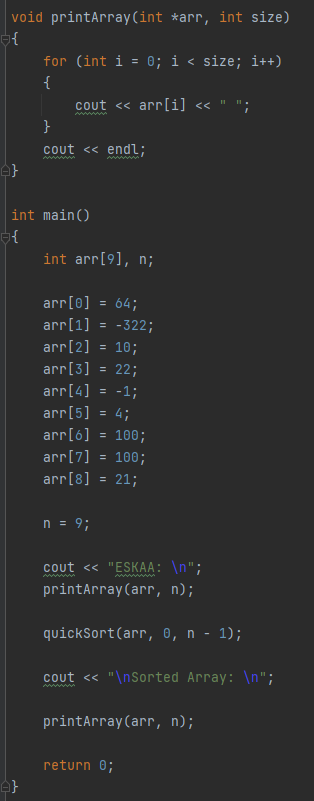


Рисунок 3.2 – Листинг второй части тестового кода

Результат интерпретации исходного кода представлен на рисунке 3.3.

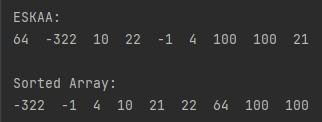


Рисунок 3.3 – Результат интерпретации исходного кода

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

import types

from core.checks import \*

from core.tree import \*

def CreateFunctionObj(name, args, code):

"""

Creates a new function object from the given data.

"""

# Build function signature string from arg\_types

args\_str = ", ".join(args)

function\_str = f"def {name}({args\_str}):\n{code}"

print(function\_str)

# Compile function

compiled\_func = compile(function\_str, "<string>", "exec")

func\_code = next((c for c in compiled\_func.co\_consts if isinstance(c, types.CodeType)), None)

# Creation error handler

if func\_code is None:

raise ValueError("Unable to find function code object.")

return types.FunctionType(func\_code, globals(), name)

class Translator:

"""

CPP to Python translator.

"""

def \_\_init\_\_(self, tree\_root, literal\_table, variable\_table):

"""

Initializes the translator object which can parse CPP AST and translate it to Python.

"""

# Tree provided to translation

self.Tree = tree\_root

# Environment, constants and variables

self.LiteralTable = literal\_table

self.VariableTable = variable\_table

self.Translate()

def Translate(self):

"""

Core function of the translator: parses the function from AST and translates them to python in order to use.

"""

if self.Tree is None:

return

children = self.Tree.GetChildren()

for child in children:

if child.Type == SyntaxTreNodeTypes.FUNCTION\_DECLARATION:

func\_nodes = child.GetChildren()

# Get function name

func\_name = self.GetVariable(func\_nodes[1].GetLexeme().itemValue).itemName

# Get function arguments

func\_args = self.GetFunctionArguments(func\_nodes[2])

# Get function body code

body\_instructions\_nodes = func\_nodes[3].GetChildren()

instructions = []

for instruction\_node in body\_instructions\_nodes:

if not instruction\_node:

continue

instruction = self.ParseInstruction(instruction\_node, 1)

if instruction is not None:

instructions.append(instruction)

func\_code = "\n".join(instructions)

# Get function object

function\_obj = CreateFunctionObj(func\_name, func\_args, func\_code)

# Assign function object to global scope

globals()[func\_name] = function\_obj

# Call main function

globals()["main"]()

def GetFunctionArguments(self, node):

"""

Parses function arguments node to get arguments in 'Python' form.

"""

arguments = []

for arg in node.GetChildren():

arg\_children = arg.GetChildren()

variable = self.GetVariable(arg\_children[1].GetLexeme().itemValue)

arg\_type = None

if isinstance(variable.itemType, list):

arg\_type = "list"

else:

arg\_type = self.GetArgType(variable.itemType)

if arg\_type is not None:

arg\_name = variable.itemName

arguments.append(f"{arg\_name}: {arg\_type}")

else:

raise ValueError("Function argument type error")

return arguments

def GetArgType(self, arg\_type):

"""

Parses arguments CPP type to Python type.

"""

if arg\_type == Language.VariableTypes.INT:

return "int"

elif arg\_type == Language.VariableTypes.STRING:

return "str"

elif arg\_type == Language.VariableTypes.BOOL:

return "bool"

elif arg\_type == Language.VariableTypes.DOUBLE:

return "float"

def GetVariable(self, variable\_id):

"""

Gets variable item from the table.

"""

variable = None

try:

variable = [v for v in self.VariableTable if v.itemId == variable\_id][0]

except:

raise ValueError("Bad variable id")

return variable

def GetLiteral(self, literal\_id):

"""

Gets literal item from the table.

"""

literal = None

try:

literal = [v for v in self.LiteralTable.Literals if v.itemId == literal\_id][0]

except:

raise ValueError("Bad variable id")

return literal

def ParseInstruction(self, instruction\_node, level):

"""

Parses given instruction node.

"""

instruction = None

lexeme = None

cur\_level = '\t' \* level

if instruction\_node.Type == SyntaxTreNodeTypes.COMMON:

lexeme = instruction\_node.GetLexeme()

if instruction\_node.Type == SyntaxTreNodeTypes.DECLARATION:

instruction = str(self.ParseVariableDeclarationStatement(instruction\_node, level))

elif instruction\_node.Type == SyntaxTreNodeTypes.FUNCTION\_CALL:

instruction = str(self.ParseFunctionCallStatement(instruction\_node, level))

elif lexeme and lexeme.itemValue == Language.Operators.EQUAL:

instruction = str(self.ParseOperator(instruction\_node, level))

elif lexeme and lexeme.itemValue == Language.KeyWords.IF:

instruction = str(self.ParseIfStatement(instruction\_node, level))

elif lexeme and lexeme.itemValue == Language.KeyWords.WHILE:

instruction = str(self.ParseWhileStatement(instruction\_node, level))

elif lexeme and lexeme.itemValue == Language.KeyWords.FOR:

instruction = str(self.ParseForStatement(instruction\_node, level))

elif lexeme and lexeme.itemValue == Language.KeyWords.DO:

instruction = str(self.ParseDoWhileStatement(instruction\_node, level))

elif lexeme and lexeme.itemValue in [Language.Operators.INCREMENT, Language.Operators.DECREMENT]:

instruction = str(self.ParseUnaryOperatorStatement(instruction\_node, level))

elif lexeme and lexeme.itemValue == Language.KeyWords.CIN:

instruction = str(self.ParseCinStatement(instruction\_node, level))

elif lexeme and lexeme.itemValue == Language.KeyWords.COUT:

instruction = str(self.ParseCoutStatement(instruction\_node, level))

elif lexeme and lexeme.itemValue in [Language.KeyWords.RETURN, Language.KeyWords.EXIT]:

instruction = str(self.ParseReturnExitStatement(instruction\_node, level))

elif lexeme and lexeme.itemValue == Language.KeyWords.BREAK:

instruction = f"{cur\_level}break"

elif lexeme and lexeme.itemValue == Language.KeyWords.CONTINUE:

instruction = f"{cur\_level}continue"

if instruction is not None:

return instruction

def ParseOperator(self, operator\_node, level):

"""

Parses the operator expression to Python.

"""

parts = []

operator = None

current\_level = level \* '\t'

if operator\_node.GetLexeme().itemType == Language.LexemeTypes.IDENTIFIER:

return str(self.GetVariable(operator\_node.GetLexeme().itemValue).itemName)

elif operator\_node.GetLexeme().itemType == Language.LexemeTypes.INT\_NUM:

return str(self.GetLiteral(operator\_node.GetLexeme().itemValue).itemValue)

try:

operator = inverted\_operators[operator\_node.GetLexeme().itemValue]

except:

raise ValueError("Unknown operator")

if operator == "-" and len(operator\_node.GetChildren()) == 1:

node = operator\_node.GetChildren()[0]

lexeme = operator\_node.GetChildren()[0].GetLexeme()

if not node.GetChildren():

if lexeme.itemType == Language.LexemeTypes.IDENTIFIER:

return "-" + str(self.GetVariable(lexeme.itemValue).itemName)

elif lexeme.itemType in [Language.LexemeTypes.INT\_NUM, Language.LexemeTypes.DOUBLE\_NUM]:

return "-" + str(self.GetLiteral(lexeme.itemValue).itemValue)

elif lexeme.itemType == Language.LexemeTypes.IDENTIFIER:

var\_name = str(self.GetVariable(lexeme.itemValue).itemName)

return "-" + f"{var\_name}[int({self.ParseOperator(node.GetChildren()[0], 0)})]"

for node in operator\_node.GetChildren():

lexeme = node.GetLexeme()

part = None

if not node.GetChildren():

if lexeme.itemType == Language.LexemeTypes.IDENTIFIER:

part = str(self.GetVariable(lexeme.itemValue).itemName)

elif lexeme.itemType in [Language.LexemeTypes.INT\_NUM, Language.LexemeTypes.DOUBLE\_NUM,

Language.LexemeTypes.STRING]:

part = str(self.GetLiteral(lexeme.itemValue).itemValue)

elif node.Type == SyntaxTreNodeTypes.FUNCTION\_CALL:

part = self.ParseFunctionCallStatement(node, 0)

elif lexeme.itemType == Language.LexemeTypes.IDENTIFIER:

var\_name = str(self.GetVariable(lexeme.itemValue).itemName)

part = f"{var\_name}[int({self.ParseOperator(node.GetChildren()[0], 0)})]"

elif IsOperator(operator):

if lexeme.itemValue == Language.Operators.MINUS and len(node.GetChildren()) == 1:

part = f"{self.ParseOperator(node, 0)}"

else:

part = f"({self.ParseOperator(node, 0)})"

if part is not None:

parts.append(part)

else:

raise ValueError("Bad operating part")

return current\_level + str(operator).join(parts)

def ParseVariableDeclarationStatement(self, declaration\_node, level):

"""

Parses the variable declaration statement to Python.

"""

declarations = []

current\_level = level \* '\t'

for node in declaration\_node.GetChildren()[1:]:

lexeme = node.GetLexeme()

declaration = None

if not node.GetChildren():

if lexeme.itemType == Language.LexemeTypes.IDENTIFIER:

variable = str(self.GetVariable(lexeme.itemValue).itemName)

declaration = f"{current\_level}{variable}=None"

elif lexeme.itemType == Language.LexemeTypes.IDENTIFIER:

var\_name = str(self.GetVariable(lexeme.itemValue).itemName)

var\_len = self.ParseOperator(node.GetChildren()[0], 0)

declaration = f"{current\_level}{var\_name}=list(None for \_ in range({var\_len}))"

elif lexeme.itemValue == Language.Operators.EQUAL:

declaration = f"{self.ParseOperator(node, level)}"

if declaration is not None:

declarations.append(declaration)

else:

raise ValueError("Unknown declaration")

return "\n".join(declarations)

def ParseIfStatement(self, if\_node, level):

"""

Parses the if statement to Python.

"""

current\_level = level \* '\t'

condition\_node = if\_node.GetChildren()[0]

code\_node = if\_node.GetChildren()[1]

instructions = [self.ParseInstruction(instruction, level + 1) for instruction in code\_node.GetChildren()]

code\_block = '\n'.join(instructions)

return f"{current\_level}if {self.ParseOperator(condition\_node, 0)}:\n{code\_block}"

def ParseFunctionCallStatement(self, call\_node, level):

"""

Parses the function call statement to Python.

"""

current\_level = level \* '\t'

function\_name = self.GetVariable(call\_node.GetChildren()[0].GetLexeme().itemValue).itemName

arguments = [self.ParseFunctionArgument(arg)

for arg in call\_node.GetChildren()[1].GetChildren()]

try:

return f"{current\_level}globals()['{function\_name}']({','.join(arguments)})"

except:

raise ValueError("Bad function call")

def ParseFunctionArgument(self, argument\_node):

"""

Gets the name of the function call argument.

"""

lexeme = argument\_node.GetLexeme()

if not argument\_node.GetChildren():

if lexeme.itemType == Language.LexemeTypes.IDENTIFIER:

return str(self.GetVariable(lexeme.itemValue).itemName)

elif lexeme.itemType in [Language.LexemeTypes.INT\_NUM, Language.LexemeTypes.DOUBLE\_NUM]:

return str(self.GetLiteral(lexeme.itemValue).itemValue)

elif lexeme.itemType == Language.LexemeTypes.STRING:

return repr(self.GetLiteral(lexeme.itemValue).itemValue)

elif lexeme.itemType == Language.LexemeTypes.IDENTIFIER:

var\_name = str(self.GetVariable(lexeme.itemValue).itemName)

var\_len = self.ParseOperator(argument\_node.GetChildren()[0], 0)

return f"{var\_name}[{var\_len}]"

elif lexeme.itemType == Language.LexemeTypes.OPERATOR:

return self.ParseOperator(argument\_node, 0)

def ParseWhileStatement(self, while\_node, level):

"""

Parses the while cycle statement to Python.

"""

current\_level = level \* '\t'

condition\_node = while\_node.GetChildren()[0]

code\_node = while\_node.GetChildren()[1]

instructions = [self.ParseInstruction(instruction, level + 1) for instruction in code\_node.GetChildren()]

code\_block = '\n'.join(instructions)

return f"{current\_level}while {self.ParseOperator(condition\_node, 0)}:\n{code\_block}"

def ParseForStatement(self, for\_node, level):

"""

Parses the for cycle statement to Python.

"""

current\_level = level \* '\t'

cycle\_level = (level + 1) \* '\t'

condition\_nodes = for\_node.GetChildren()

code\_node = condition\_nodes[len(for\_node.GetChildren()) - 1]

instructions = [self.ParseInstruction(instruction, level + 1) for instruction in code\_node.GetChildren()]

code\_block = '\n'.join(instructions)

if len(condition\_nodes) == 1 and condition\_nodes[0].Type == SyntaxTreNodeTypes.CODE\_BLOCK:

return f"{current\_level}while 1:\n{code\_block}"

elif len(condition\_nodes) == 4:

first\_condition = None

if condition\_nodes[0].Type == SyntaxTreNodeTypes.DECLARATION:

first\_condition = self.ParseVariableDeclarationStatement(condition\_nodes[0], level)

elif condition\_nodes[0].GetLexeme().itemType == Language.LexemeTypes.OPERATOR:

first\_condition = self.ParseOperator(condition\_nodes[0], level)

second\_condition = self.ParseOperator(condition\_nodes[1], 0)

third\_condition = self.ParseInstruction(condition\_nodes[2], level)

return f"{first\_condition}\n" \

f"{current\_level}while 1:\n" \

f"{cycle\_level}if not {second\_condition}: break\n" \

f"{code\_block}\n" \

f"{current\_level}{third\_condition}"

# return f"{current\_level}while {self.ParseOperator(condition\_node, 0)}:\n{code\_block}"

def ParseDoWhileStatement(self, doWhile\_node, level):

"""

Parses the do-while cycle statement to Python.

"""

current\_level = level \* '\t'

condition\_node = doWhile\_node.GetChildren()[1].GetChildren()[0]

code\_node = doWhile\_node.GetChildren()[0]

instructions\_out = [self.ParseInstruction(instruction, level) for instruction in code\_node.GetChildren()]

instructions\_in = [self.ParseInstruction(instruction, level + 1) for instruction in code\_node.GetChildren()]

code\_block\_out = '\n'.join(instructions\_out)

code\_block\_in = '\n'.join(instructions\_in)

return f"{code\_block\_out}" \

f"\n{current\_level}while {self.ParseOperator(condition\_node, 0)}:\n{code\_block\_in}"

def ParseUnaryOperatorStatement(self, unary\_node, level):

"""

Parses unary operator statements to Python.

"""

current\_level = level \* '\t'

var\_name = str(self.GetVariable(unary\_node.GetChildren()[0].GetLexeme().itemValue).itemName)

if unary\_node.GetLexeme().itemValue == Language.Operators.INCREMENT:

return f"{current\_level}{var\_name}+=1"

elif unary\_node.GetLexeme().itemValue == Language.Operators.DECREMENT:

return f"{current\_level}{var\_name}-=1"

def ParseCinStatement(self, cin\_node, level):

"""

Parses the input statement to Python.

"""

current\_level = level \* '\t'

children = cin\_node.GetChildren()

instructions = []

for node in children:

lexeme = node.GetLexeme()

instruction = None

if not node.GetChildren():

if lexeme.itemType == Language.LexemeTypes.IDENTIFIER:

arg\_type = str(self.GetArgType(self.GetVariable(lexeme.itemValue).itemType))

instruction = f"{current\_level}" \

f"{str(self.GetVariable(lexeme.itemValue).itemName)}={arg\_type}(input())"

elif lexeme.itemType == Language.LexemeTypes.IDENTIFIER:

var\_name = str(self.GetVariable(lexeme.itemValue).itemName)

var\_len = self.ParseOperator(node.GetChildren()[0], 0)

item\_type = str(self.GetArgType(self.GetVariable(lexeme.itemValue).itemType[1]))

instruction = f"{current\_level}" \

f"{var\_name}[{var\_len}]={item\_type}(input())"

if instruction is not None:

instructions.append(instruction)

else:

raise ValueError("Bad arg to input")

return '\n'.join(instructions)

def ParseCoutStatement(self, cout\_node, level):

"""

Parses the output statement to Python.

"""

current\_level = level \* '\t'

children = cout\_node.GetChildren()

messages = []

for node in children:

lexeme = node.GetLexeme()

message = None

if not node.GetChildren():

if lexeme.itemType == Language.LexemeTypes.IDENTIFIER:

message = str(self.GetVariable(lexeme.itemValue).itemName)

elif lexeme.itemType in [Language.LexemeTypes.INT\_NUM, Language.LexemeTypes.DOUBLE\_NUM]:

message = str(self.GetLiteral(lexeme.itemValue).itemValue)

elif lexeme.itemType == Language.LexemeTypes.STRING:

message = repr(self.GetLiteral(lexeme.itemValue).itemValue)

elif lexeme.itemValue == Language.KeyWords.ENDL:

message = repr("\n")

elif lexeme.itemType == Language.LexemeTypes.IDENTIFIER:

var\_name = str(self.GetVariable(lexeme.itemValue).itemName)

var\_len = self.ParseOperator(node.GetChildren()[0], 0)

message = f"{var\_name}[{var\_len}]"

if message is not None:

messages.append(message)

else:

raise ValueError("Bad arg to print")

return f"{current\_level}print({','.join(messages)}, end={repr('')})"

def ParseReturnExitStatement(self, return\_exit\_node, level):

"""

Parses the return and exit statements to Python.

"""

current\_level = level \* '\t'

children = return\_exit\_node.GetChildren()

return\_arg = None

if children:

node = children[0]

lexeme = node.GetLexeme()

if not node.GetChildren():

if lexeme.itemType == Language.LexemeTypes.IDENTIFIER:

return\_arg = str(self.GetVariable(lexeme.itemValue).itemName)

elif lexeme.itemType in [Language.LexemeTypes.INT\_NUM, Language.LexemeTypes.DOUBLE\_NUM]:

return\_arg = str(self.GetLiteral(lexeme.itemValue).itemValue)

elif lexeme.itemType == Language.LexemeTypes.STRING:

return\_arg = repr(self.GetLiteral(lexeme.itemValue).itemValue)

elif lexeme.itemType == Language.LexemeTypes.IDENTIFIER:

var\_name = str(self.GetVariable(lexeme.itemValue).itemName)

var\_len = self.ParseOperator(node.GetChildren()[0], 0)

return\_arg = f"{var\_name}[{var\_len}]"

if return\_arg is None:

raise ValueError("Bad return argument")

else:

if return\_exit\_node.GetLexeme().itemValue == Language.KeyWords.RETURN:

return f"{current\_level}return {return\_arg}"

elif return\_exit\_node.GetLexeme().itemValue == Language.KeyWords.EXIT:

return f"{current\_level}quit({return\_arg})"

else:

if return\_exit\_node.GetLexeme().itemValue == Language.KeyWords.RETURN:

return f"{current\_level}return"

elif return\_exit\_node.GetLexeme().itemValue == Language.KeyWords.EXIT:

return f"{current\_level}quit()"