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

Целью выполнения данной лабораторной работы является на основе результатов анализа лабораторных работы 1-4 выполнить трансляцию программы с языка программирования C++ 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 C++ 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 C++ на язык программирования Python с последующей интерпретацией кода.

Листинг первой части тестового кода, представляющего собой быструю сортировку, представлен на рисунке 3.1.

```cpp
#include <iostream>
using namespace std;

void quickSort(int *array, int first, int last)
{
    int mid, count;
    int f = first, l = last;
    mid = array[(f + l) / 2];

    do {
        while (array[f] < mid) { f++; }
        while (array[l] > mid) { l--; }
        if (f <= l)
        {
            count = array[f];
            array[f] = array[l];
            array[l] = count;
            f++;
            l--;
        }
    } while (f < l);

    if (first < l)
    {
        quickSort(array, first, l);
    }

    if (f < last)
    {
        quickSort(array, f, last);
    }
}

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

Рисунок 3.1 – Листинг первой части тестового кода

Листинг второй части тестового кода представлен на рисунке 3.2.

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

int main()
{
    int arr[9], n;

    arr[0] = 64;
    arr[1] = -322;
    arr[2] = 10;
    arr[3] = 22;
    arr[4] = -1;
    arr[5] = 4;
    arr[6] = 100;
    arr[7] = 100;
    arr[8] = 21;

    n = 9;

    cout << "ESKAA: \n";
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    cout << "\nSorted Array: \n";

    printArray(arr, n);

    return 0;
}
```

Рисунок 3.2 – Листинг второй части тестового кода

Результат интерпретации исходного кода представлен на рисунке 3.3.

```
ESKAA:
64  -322  10  22  -1  4  100  100  21

Sorted Array:
-322  -1  4  10  21  22  64  100  100
```

Рисунок 3.3 – Результат интерпретации исходного кода

Таким образом в ходе лабораторной работы был реализован интерпретатор для программ на языке C++, 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 C++, 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 C++ [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 C++ [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/cpp/c-operators. – Дата доступа: 27.02.2024.

[6] Функции C++ [Электронный ресурс]. – Режим доступа: https://metanit.com/cpp/tutorial/3.1.php. – Дата доступа: 27.02.2024.

[7] Классы C++ [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).itemN
ame)

        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()"
```