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. 053501

Волковский О. А.

Проверил

Ассистент кафедры информатики

Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1. Цель работы 3](#_Toc128743288)

[2. Краткие теоретические сведения 4](#_Toc128743289)

[3. Примеры работы парсера 6](#_Toc128743290)

[4. Выводы 10](#_Toc128743291)

ПРИЛОЖЕНИЕ А. Листинг кода 11

1. **ЦЕЛЬ РАБОТЫ**

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

В данной лабораторной работе необходимо выполнить заключительную стадию, используя написанные ранее лексический, синтаксический и семантический анализаторы создать работающий интерпретатор подмножества языка С++.

1. **КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Интерпретатор — это программа, которая выполняет интерпретацию.

Интерпретация — это построчный анализ, обработка и выполнение исходного кода программы или запроса.

Транслятор — это программа, которая переводит входную программу на исход­ном языке в эквивалентную ей выходную программу на результирую­щем языке.

Итак, чтобы создать транслятор, необходимо прежде всего выбрать входной и выходной языки. С точки зрения преобразования предложений входного язы­ка в эквивалентные им предложения выходного языка транслятор выступает как переводчик. Например, трансляция программы с языка С в язык ассемблера, по сути, ничем не отличается от перевода, скажем, с русского языка на английский, с той только разницей, что сложность языков несколько иная.

Результатом работы транслятора будет результирующая программа, но только в том случае, если текст исходной программы является правильным.

Таким образом, компилятор отличается от транслятора лишь тем, что его ре­зультирующая программа всегда должна быть написана на языке машинных ко­дов или на языке ассемблера.

Компиляторы, безусловно, самый распространенный вид трансляторов. Они име­ют самое широкое практическое применение, которым обязаны широкому рас­пространению всевозможных языков программирования.

Сам по себе этот промежуточный язык не может непосредственно исполняться на компьюте­ре, а требует специального промежуточного интерпретатора для выполнения написан­ных на нем программ.

Интерпретаторы не очень сильно отличаются от компиляторов. Они также конвертируют высокоуровневые языки в читаемые машиной бинарные эквиваленты. Каждый раз, когда интерпретатор получает на выполнение код языка высокого уровня, то, прежде чем сконвертировать его в машинный код, он конвертирует этот код в промежуточный язык. Каждая часть кода интерпретируется и выполняется отдельно и последовательно, и, если в какой-то части будет найдена ошибка, она остановит интерпретацию кода без трансляции следующей части кода.

Интерпретатор берет одну инструкцию, транслирует и выполняет ее, а затем берет следующую инструкцию. Компилятор же транслирует всю программу сразу, а потом выполняет ее.

Компилятор генерирует отчет об ошибках после трансляции всего, в то время как интерпретатор прекратит трансляцию после первой найденной ошибки.

Компилятор по сравнению с интерпретатором требует больше времени для анализа и обработки языка высокого уровня.

Помимо времени на обработку и анализ, общее время выполнения кода компилятора быстрее в сравнении с интерпретатором.

Естественно, трансляторы и компиляторы, как и все прочие программы, разраба­тывает группа разработчиков.

Простой интерпретатор анализирует и тут же выполняет программу построчно, по мере поступления её исходного кода на вход интерпретатора. Достоинством такого подхода является мгновенная реакция. Недостаток — такой интерпретатор обнаруживает ошибки в тексте программы только при попытке выполнения команды с ошибкой.

1. **ПРИМЕРЫ РАБОТЫ ПАРСЕРА**

Рассмотрим следующую программу, приведенную в тестовом примере (см. рисунок 1).

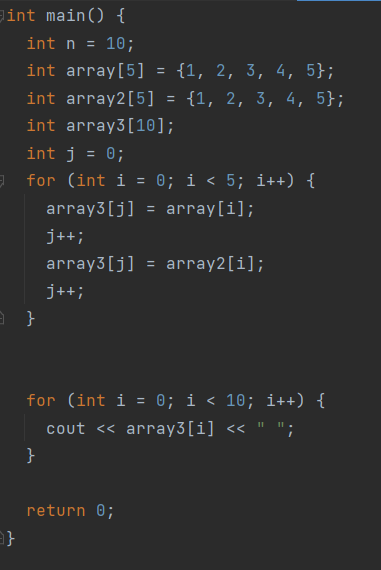


Рисунок 1 – Пример программы

Результат программы в консоли (см. рисунок 2).

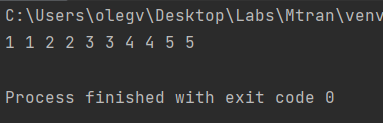


Рисунок 2 – Вывод в консоли тестовой программы

Теперь рассмотрим работу программы на первом примере первой лабораторной работы, которая проверят число на простоту (см. рисунок 3).

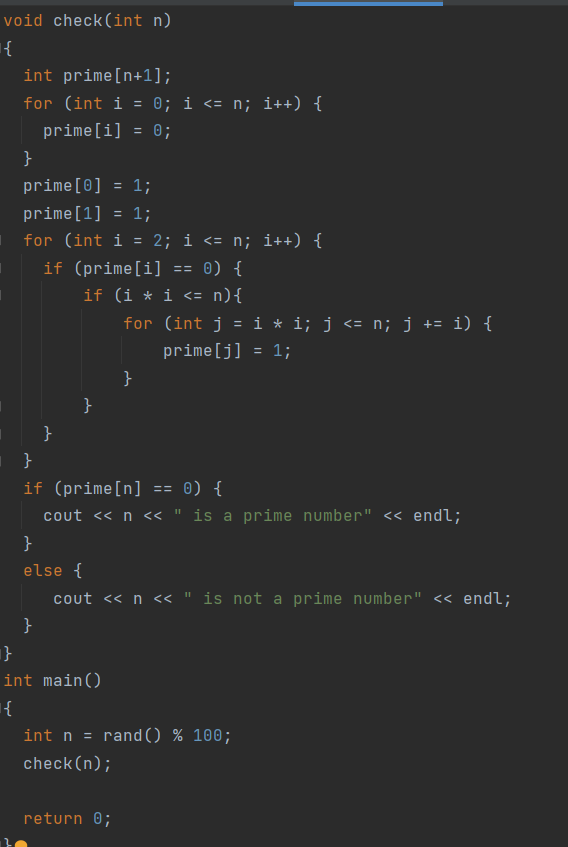


Рисунок 3 – Код первой программы из первой лабораторной работы

Результат программы в консоли (см. рисунок 4).

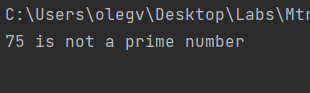


Рисунок 4 – Вывод в консоль первой программы

Теперь рассмотрим работу программы на втором примере из лабораторной работы (см. рисунок 5).

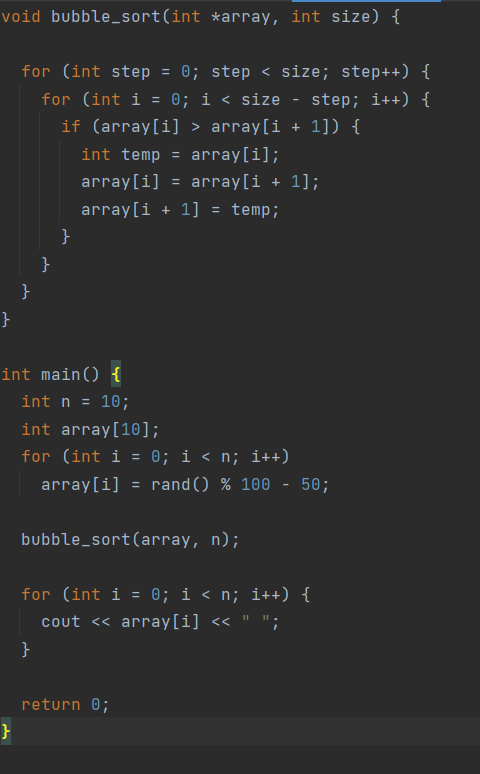


Рисунок 5 – Код второй программы из первой лабораторной работы

Результат программы в консоли (см. рисунок 6).

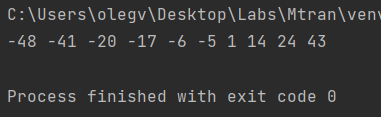


Рисунок 6 – Вывод в консоль второй программы

1. **ВЫВОДЫ**

В результате работы были получены знания об интерпретаторе и интерпретации. В итоге работы был разработан интерпретатор подмножества языка C++ на основе уже имеющегося лексического, синтаксического и семантического анализаторов.

**ПРИЛОЖЕНИЕ А**

**Листинг кода**

def execute\_program(node, function\_tree\_list, variable\_table):

global return\_flag

if node.node\_type == "Assign\_node":

if node.arithmetic\_sign == "=":

variable\_table[node.left\_value.name] = execute\_program(node.right\_value, function\_tree\_list, variable\_table)

elif node.arithmetic\_sign == "+=":

variable\_table[node.left\_value.name] += execute\_program(node.right\_value, function\_tree\_list,

variable\_table)

elif node.arithmetic\_sign == "/=":

variable\_table[node.left\_value.name] /= execute\_program(node.right\_value, function\_tree\_list,

variable\_table)

elif node.arithmetic\_sign == "\*=":

variable\_table[node.left\_value.name] \*= execute\_program(node.right\_value, function\_tree\_list,

variable\_table)

else:

variable\_table[node.left\_value.name] = None

if node.node\_type == "Build\_in\_node":

if node.function\_name == "return":

return node.arguments[0]

if node.function\_name == "continue":

return\_flag = 1

if node.function\_name == "break":

return\_flag = 2

if node.function\_name == "cout":

for arg in node.arguments:

print(arg, end='')

if node.node\_type == "Compare\_node":

compare\_left\_value = execute\_program(node.left\_value, function\_tree\_list, variable\_table)

compare\_right\_value = execute\_program(node.right\_value, function\_tree\_list, variable\_table)

if node.compare\_sign == "<":

return compare\_left\_value < compare\_right\_value

if node.compare\_sign == ">":

return compare\_left\_value > compare\_right\_value

if node.compare\_sign == "<=":

return compare\_left\_value <= compare\_right\_value

if node.compare\_sign == ">=":

return compare\_left\_value >= compare\_right\_value

if node.compare\_sign == "==":

return compare\_left\_value == compare\_right\_value

if node.compare\_sign == "!=":

return compare\_left\_value != compare\_right\_value

if node.node\_type == "Expression\_node":

expr\_left\_value = execute\_program(node.left\_value, function\_tree\_list, variable\_table)

expr\_right\_value = execute\_program(node.right\_value, function\_tree\_list, variable\_table)

if node.operator == "+":

return expr\_left\_value + expr\_right\_value

elif node.operator == "-":

return expr\_left\_value - expr\_right\_value

elif node.operator == "/":

return expr\_left\_value / expr\_right\_value

elif node.operator == "\*":

return expr\_left\_value \* expr\_right\_value

elif node.operator == "++":

return expr\_left\_value + 1

elif node.operator == "--":

return expr\_left\_value - 1

elif node.operator == "%":

return expr\_left\_value % expr\_right\_value

elif node.operator == "/":

return expr\_left\_value / expr\_right\_value

else:

return expr\_left\_value

if node.node\_type == "For\_node":

start\_value = execute\_program(node.start\_value\_node, function\_tree\_list, variable\_table)

variable\_table[node.variable\_node.name] = start\_value

while True:

if not execute\_program(node.compare\_node, function\_tree\_list, variable\_table):

break

for child in node.children:

execute\_program(child, function\_tree\_list, variable\_table)

if return\_flag == 1:

return\_flag = 0

break

elif return\_flag == 2 or return\_flag == 3:

break

if return\_flag == 2:

return\_flag = 0

break

variable\_table[node.variable\_node.name] = execute\_program(node.expression\_node, function\_tree\_list,

variable\_table)

if node.node\_type == "Func\_declaration\_node":

new\_variable\_table = {}

for val, arg2 in zip(variable\_table, node.arguments):

new\_variable\_table[arg2] = val

func\_result = None

for child in node.children:

func\_result = execute\_program(child, function\_tree\_list, new\_variable\_table)

return func\_result

if node.node\_type == "Func\_node":

for func in function\_tree\_list:

if func[0] == node.function\_name:

return execute\_program(func[1], function\_tree\_list, node.arguments)

if node.node\_type == "If\_node":

if node.condition is None or execute\_program(node.condition, function\_tree\_list, variable\_table):

for child in node.children:

execute\_program(child, function\_tree\_list, variable\_table)

if node.node\_type == "Variable\_node":

if node.is\_array:

return node.cur\_array

else:

return node.cur\_array

if node.node\_type == "Block\_node":

for child in node.children:

execute\_program(child, function\_tree\_list, variable\_table)