

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 |
| 2. Краткие теоретические сведения..... | 4 |
| 3. Примеры работы парсера | 6 |
| 4. Выводы | 10 |
| 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 C++.

2 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.

3 ПРИМЕРЫ РАБОТЫ ПАРСЕРА

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

```
int main() {  
    int n = 10;  
    int array[5] = {1, 2, 3, 4, 5};  
    int array2[5] = {1, 2, 3, 4, 5};  
    int array3[10];  
    int j = 0;  
    for (int i = 0; i < 5; i++) {  
        array3[j] = array[i];  
        j++;  
        array3[j] = array2[i];  
        j++;  
    }  
  
    for (int i = 0; i < 10; i++) {  
        cout << array3[i] << " ";  
    }  
  
    return 0;  
}
```

Рисунок 1 – Пример программы

Результат программы в консоли (см. рисунок 2).

```
C:\Users\olegv\Desktop\Labs\Mtran\venv
1 1 2 2 3 3 4 4 5 5

Process finished with exit code 0
```

Рисунок 2 – Вывод в консоли тестовой программы

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

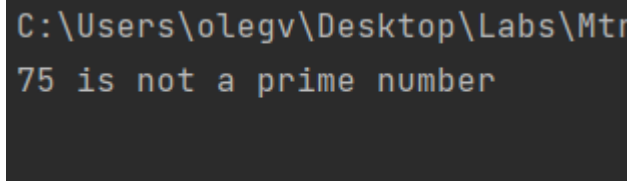
```
void check(int n)
{
    int prime[n+1];
    for (int i = 0; i <= n; i++) {
        prime[i] = 0;
    }
    prime[0] = 1;
    prime[1] = 1;
    for (int i = 2; i <= n; i++) {
        if (prime[i] == 0) {
            if (i * i <= n) {
                for (int j = i * i; j <= n; j += i) {
                    prime[j] = 1;
                }
            }
        }
    }
    if (prime[n] == 0) {
        cout << n << " is a prime number" << endl;
    }
    else {
        cout << n << " is not a prime number" << endl;
    }
}

int main()
{
    int n = rand() % 100;
    check(n);

    return 0;
}
```

Рисунок 3 – Код первой программы из первой лабораторной работы

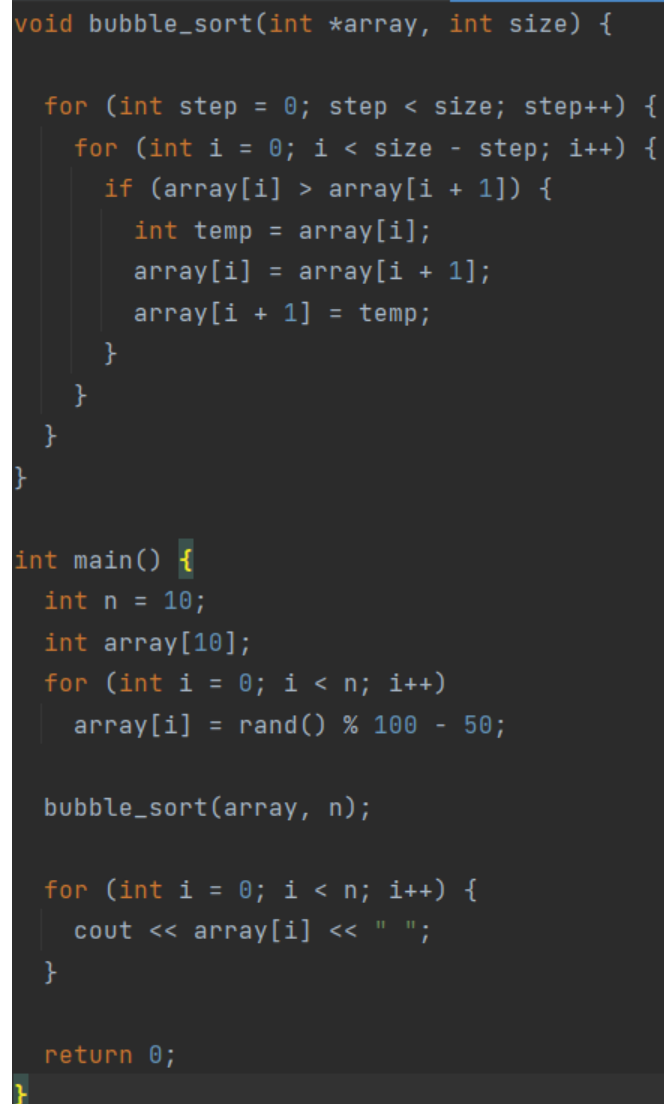
Результат программы в консоли (см. рисунок 4).



```
C:\Users\olegv\Desktop\Labs\Mtr
75 is not a prime number
```

Рисунок 4 – Вывод в консоль первой программы

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



```
void bubble_sort(int *array, int size) {
    for (int step = 0; step < size; step++) {
        for (int i = 0; i < size - step; i++) {
            if (array[i] > array[i + 1]) {
                int temp = array[i];
                array[i] = array[i + 1];
                array[i + 1] = temp;
            }
        }
    }
}

int main() {
    int n = 10;
    int array[10];
    for (int i = 0; i < n; i++)
        array[i] = rand() % 100 - 50;

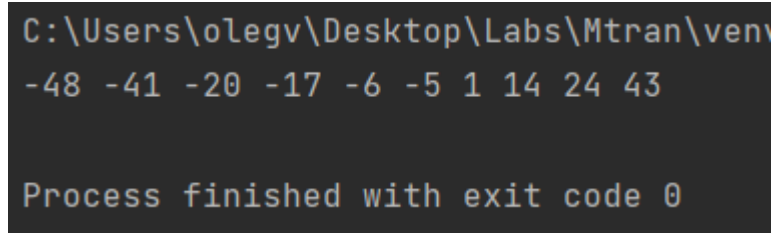
    bubble_sort(array, n);

    for (int i = 0; i < n; i++) {
        cout << array[i] << " ";
    }

    return 0;
}
```

Рисунок 5 – Код второй программы из первой лабораторной работы

Результат программы в консоли (см. рисунок 6).



```
C:\Users\olegv\Desktop\Labs\Mtran\venv
-48 -41 -20 -17 -6 -5 1 14 24 43

Process finished with exit code 0
```

Рисунок 6 – Вывод в консоль второй программы

4 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)

```