

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 № 4  
на тему «Семантический анализатор»

Выполнил

Е. А. Киселёва

Проверил

Н. Ю. Гриценко

Минск 2024

## СОДЕРЖАНИЕ

1 Постановка задачи.....	3
2 Краткие теоретические сведения.....	4
3 Результаты выполнения лабораторной работы.....	5
Выводы .....	6
Список использованных источников .....	9
Приложение А (обязательное) Листинг исходного кода .....	10

## **1 ПОСТАНОВКА ЗАДАЧИ**

Целью выполнения данной лабораторной работы является разработка собственного семантического анализатора для языка программирования C++. Необходимо вывести результат анализа и обработать возможные семантические ошибки.

## 2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

К этапам трансляции относятся 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]

Обычно семантический анализ означает проверку правильности типов и форматов всех идентификаторов и данных, использованных в программе. Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки программы на соответствие определению языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице идентификаторов для последующего использования при генерации промежуточного кода. Кроме того, на этом этапе компилятор должен проверить, соблюдаются ли определенные контекстные условия входного языка. Например, в современных языках программирования одним из таких условий является необходимость описания переменных, то есть для каждого использования идентификатора должно быть единственное определение. Также важно, чтобы число и атрибуты фактических параметров при вызове процедуры соответствовали ее определению.

Абстрактное синтаксическое дерево конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы.

Синтаксические деревья используются в синтаксических анализаторах для промежуточного представления программы между деревом разбора (деревом с конкретным синтаксисом) и структурой данных, которая затем используется в качестве внутреннего представления в компиляторе или интерпретаторе программы для оптимизации и генерации кода. Возможные варианты подобных структур описываются абстрактным синтаксисом.

Абстрактное синтаксическое дерево отличается от дерева разбора тем, что в нём отсутствуют узлы и рёбра для тех синтаксических правил, которые не влияют на семантику программы. Классическим примером такого отсутствия являются группирующие скобки, так как в абстрактном синтаксическом дереве группировка операндов явно задаётся структурой дерева.

Для языка, который определяется контекстно-свободной грамматикой, создание дерева в синтаксическом анализаторе представляет собой простую задачу. Большинство правил в грамматике порождают новые узлы дерева, а символы в правиле становятся связями между узлами. Правила, которые не вносят ничего нового в структуру дерева, просто заменяются в вершине одним из своих символов. Кроме того, анализатор может сначала построить полное дерево разбора, а затем пройти по нему, удаляя узлы и связи, которые не используются в абстрактном синтаксисе для получения абстрактного синтаксического дерева.

### 3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В ходе лабораторной работы был реализован конечный вид анализатора кода, который включает в себя лексический, синтаксический и семантический анализы. Были выполнены проверки на такие типы ошибок как:

- объявление одноименных переменных или функций в одной области видимости;

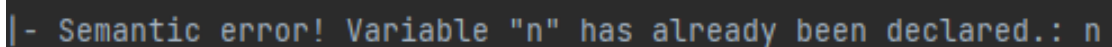
- неверное преобразование типов данных;

- несовпадение параметров и аргументов при вызове функции;

- неверное указание размера массива

- неверное применение закрывающихся одинарных и двойных кавычек.

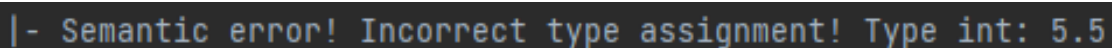
В случае объявления переменных или функций с одинаковыми именами в одной области видимости будет выведена ошибка. Пример ошибки при объявлении одноименных переменных или функций в одной области видимости представлен на рисунке 3.1.



```
|- Semantic error! Variable "n" has already been declared.: n
```

Рисунок 3.1 – Ошибка при объявлении одноименных переменных

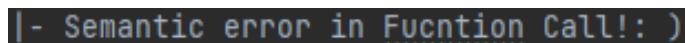
В случае неверного преобразования типов данных, когда целочисленной переменной присваивается, например, значение с плавающей точкой, будет выведена ошибка. Пример семантической ошибки при неверном преобразовании типов данных представлен на рисунке 3.2.



```
|- Semantic error! Incorrect type assignment! Type int: 5.5
```

Рисунок 3.2 – Ошибка при неверном преобразовании типов данных

В случае несовпадения количества параметров и аргументов при вызове функции с учетом того, что параметрам функции не присваивается значение, будет выведена соответствующая ошибка. Пример ошибки при несовпадении количества параметров и аргументов представлен на рисунке 3.3.



```
|- Semantic error in Fucntion Call!: )
```

Рисунок 3.3 – Ошибка при различном количестве параметров и аргументов

Ошибка при неверном указании количества элементов в массиве представлена на рисунке 3.4.

```
| - Semantic error! The number of elements in the array exceeds the declared parameter.: }
```

Рисунок 3.4 – Ошибка при неверном указании количества элементов в массиве

При помещении в одинарные кавычки более, чем одного символа, будет вызвана семантическая ошибка. Пример семантической ошибки при неверном использовании одинарных или двойных кавычек представлен на рисунке 3.5.

```
| - SEMANTIC ERROR!: 'sdfs'|
```

Рисунок 3.5 – Ошибка при неверном использовании одинарных или двойных кавычек

Таким образом в ходе данной лабораторной работы был организован полноценный анализатор кода, который включает в себя лексический, синтаксический и семантические анализы.

## **ВЫВОДЫ**

В ходе лабораторной работы был реализован семантический анализатор, основанный на результатах синтаксического анализатора. В итоге был получен полный анализатор кода программ на языке C++, включающий в себя лексический, синтаксический и семантический анализы.



## 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 itertools

from function import write_output_to_file
from main import lexer
from constants import data_types, keywords, standart_libraries, operators
import re

pattern = r'\((.*?)\)'
numbers = r'\d+'
commas = r','
semicolon = r';'

variable_types = {}
variable_scope = []
function_params = []
function_param = 0
for_params = []

arr_params = []
arr_param = 0

def check_variable(token_type, token, data_type):
    if 'VARIABLE' in token_type:
        variable_name = token
        variable_node = Node(data_type, variable_name)
        data_type = None
        variable_name = None
        return variable_node

def check_comma(token, current_node):
    if token == ',':
        comma_node = Node(",", "Comma")
        current_node.add_child(comma_node)

    return comma_node

def check_chno(token, current_node):
    if token == ';':
        chno_node = Node(token, "Chto")
        # current_node.add_child(data_list_node) # Добавляем data_list_node
        # в текущий узел
        current_node.add_child(chno_node)

    return chno_node

def check_comparison(token, current_node):
    comparison_node = ComparisonNode(token, "Comparison")
    current_node.add_child(comparison_node)
    return comparison_node
```

```

class Node:
    def __init__(self, name, node_type, data_type=None, array_in=None,
parent=None, children=None):
        self.name = name
        self.type = node_type
        self.data_type = data_type
        self.array_in = array_in
        self.parent = parent
        self.children = children if children is not None else []

    def add_child(self, node):
        node.parent = self
        self.children.append(node)

    def get_last_child(self):
        if self.children:
            return self.children[-1]
        else:
            return None

    def display(self, level=0):
        indent = "    " * level
        tree_structure = ""
        if self.data_type is not None and self.array_in is not None:
            tree_structure += f"{indent}|- {self.type}: {self.data_type} {self.name} [{self.array_in}]\n"
        if self.data_type is None and self.array_in is None:
            tree_structure += f"{indent}|- {self.type}: {self.name}\n"
        elif self.array_in is None:
            tree_structure += f"{indent}|- {self.type}: {self.data_type} {self.name}\n"
        elif self.data_type is None:
            tree_structure += f"{indent}|- {self.type}: {self.name} [{self.array_in}]\n"

        for child in self.children:
            tree_structure += child.display(level + 1)

        return tree_structure

class PreprocessorDirectiveNode(Node):
    def __init__(self, name, node_type):
        super().__init__(name, node_type)

class StatementNode(Node):
    def __init__(self, name, node_type):
        super().__init__(name, node_type)

class ClassNode(Node):
    def __init__(self, name, node_type):
        super().__init__(name, node_type)

class CommentNode(Node):
    def __init__(self, name, node_type):
        super().__init__(name, node_type)

```

```

class ForNode(Node):
    def __init__(self, name, node_type):
        super().__init__(name, node_type)

class IfNode(Node):
    def __init__(self, name, node_type):
        super().__init__(name, node_type)

class ElseNode(Node):
    def __init__(self, name, node_type):
        super().__init__(name, node_type)

class IfElseNode(Node):
    def __init__(self, name, node_type):
        super().__init__(name, node_type)

class WhileNode(Node):
    def __init__(self, name, node_type):
        super().__init__(name, node_type)

class ComparisonNode(Node):
    def __init__(self, name, node_type):
        super().__init__(name, node_type)

class AssignmentNode(Node):
    def __init__(self, name, node_type):
        super().__init__(name, node_type)

class ValueNode(Node):
    def __init__(self, name, node_type):
        super().__init__(name, node_type)

def find_chars_between(text, start_char, end_char):
    found_chars = []
    started = False

    for char in text:
        if char == start_char:
            started = True
            continue
        elif char == end_char:
            break

        if started:
            found_chars.append(char)

    return ' '.join(found_chars)

def build_syntax_tree(tokens):
    root = Node("Program", "ProgramType")
    current_node = root

    function_definitions = {}
    branch_stack = []
    square_stack = []
    param_stack = []

```

```

bracket_stack = []
include_stack = []
access_stack = []
data_stack = []
variable_stack = []
value_stack = []
io_stack = []
if_stack = []
return_stack = []
class_stack = []
struct_stack = []
object_stack = []
function_stack = []
std_stack = []
for_stack = []

is_string_declaration = False
is_value = False
inside_comment = False
is_array_declaration = False

array_name = None
data_type = None

current_comment = ""

for token, token_type, line in tokens:
    # print(variable_scope)
    if token in data_types:
        data_stack.append(token)

    if token == "//":
        continue

    if token == "/*":
        inside_comment = True
        current_comment += token[2:] + " "
        continue
    elif token == "*/":
        inside_comment = False
        current_comment = ""
        continue
    elif inside_comment:
        current_comment += token + " "
        continue

    if 'VARIABLE' in token_type or 'POINTER' in token_type:
        if len(data_stack) != 0:
            variable_already_exists = any(child.name == token for child
in current_node.children)
            if variable_already_exists:
                semantic_error_node = Node(token,
                                            f'Semantic error! Variable
"{token}" has already been declared.')
                current_node.add_child(semantic_error_node)
                break
            variable_types[token] = data_stack[-1]
            if data_stack[-1] == 'STRING':
                is_string_declaration = True
            if len(variable_scope) != 0:
                temp_scope = False
                for var, scope in variable_scope:

```

```

        if current_node.parent:
            temp_parent_node = current_node.parent
            if token == var and temp_parent_node.name ==
scope:
                semantic_error_node = Node(token,
                                            f'Semantic error!
Variable "{token}" has already been declared.')
                current_node.add_child(semantic_error_node)
                temp_scope = True
            else:
                if token == var and current_node.name == scope:
                    semantic_error_node = Node(token,
                                                f'Semantic error!
Variable "{token}" has already been declared.')
                    current_node.add_child(semantic_error_node)
                    temp_scope = True
                if temp_scope:
                    break
                variable_node = Node(token, 'Declare', data_stack[-
1].lower())
                data_stack.pop()
                is_value = True
            else:
                if token not in variable_types:
                    first_children = current_node.children[-1]
                    second_children = current_node.children[-2]

                    if first_children.type == 'Comma':
                        if second_children.type == 'Declare':
                            variable_node = Node(token, 'Declare',
second_children.data_type)
                            variable_types[token] = second_children.data_type
                        else:
                            is_string_declaration = False
                            variable_node = Node(token, 'Variable',
variable_types.get(token))
                            is_value = True

                if current_node.parent:
                    temp_parent_node = current_node.parent
                    variable_scope.append((token, temp_parent_node.name))
                else:
                    variable_scope.append((token, current_node.name))

                variable_stack.append(current_node)
                current_node.add_child(variable_node)
                current_node = variable_node
                parent_node = current_node.parent

                semicolon_present = False
                for tok, _, ln in tokens:
                    if ln == line and tok == ";" and parent_node.type in (
                        'ProgramType', 'Block', 'Declare', 'AccessModifier',
'ReturnStatement', 'Object') or parent_node.type in (
                        'Parameters', 'Function', 'Function Call', 'Colon',
'Out', 'Cin', 'StdNamespace', 'Variable', 'Operator Input', 'Array', 'Square
Block'):
                        semicolon_present = True
                        break

                if not semicolon_present:

```

```

        syntax_error_node = Node(f"Semicolon missing after variable
declaration.",
                                f'Syntax error!')
        current_node.add_child(syntax_error_node)
        break

    if 'ARRAY' in token_type:
        if len(data_stack) != 0:
            array_already_exists = any(child.name == token for child in
current_node.children)
            if array_already_exists:
                semantic_error_node = Node(token,
                                            f'Semantic error! Variable
"{token}" has already been declared.')
                current_node.add_child(semantic_error_node)
                break
            variable_types[token] = data_stack[-1]
            if data_stack[-1] == 'STRING':
                is_string_declaration = True
            if len(variable_scope) != 0:
                temp_scope = False
                for var, scope in variable_scope:
                    temp_parent_node = current_node.parent
                    if token == var and temp_parent_node.name == scope:
                        semantic_error_node = Node(token,
                                                    f'Semantic error!
Variable "{token}" has already been declared.')
                        current_node.add_child(semantic_error_node)
                        temp_scope = True
                if temp_scope:
                    break
            variable_node = Node(token, 'Declare Array', data_stack[-
1].lower())
            data_stack.pop()
            is_value = True
        else:
            is_string_declaration = False
            variable_node = Node(token, 'Array',
variable_types.get(token))
            is_value = True

        if current_node.parent:
            temp_parent_node = current_node.parent
            variable_scope.append((token, temp_parent_node.name))
        else:
            variable_scope.append((token, current_node.name))

        variable_stack.append(current_node)
        current_node.add_child(variable_node)
        current_node = variable_node
        parent_node = current_node.parent

        semicolon_present = False
        for tok, _, ln in tokens:
            if ln == line and tok == ";" and parent_node.type in (
                'ProgramType', 'Block', 'Declare', 'AccessModifier',
'ReturnStatement') or parent_node.type in (
                'Parameters', 'Function', 'Function Call', 'Colon',
'Cout', 'Cin', 'StdNamespace', 'Operator Input', 'Variable', 'Array'):
                semicolon_present = True
                break

```

```

        if not semicolon_present:
            syntax_error_node = Node(f"Semicolon missing after variable
declaration.",
                                    f'Syntax error!')
            current_node.add_child(syntax_error_node)
            break

    if token == '[':
        square_node = Node(current_node.name, 'Square Block')
        square_stack.append(current_node)
        current_node.add_child(square_node)
        current_node = square_node

    if token == ']':
        parent_node = current_node.parent
        if parent_node.type in ('Declare Array', 'Array'):
            pass
        if current_node.type == 'Square Block':
            temp_list = []
            temp_list.extend(current_node.children)
            semantic_error = False

            if len(temp_list) == 0:
                array_param = 0
            if len(temp_list) == 1:
                for i in temp_list:
                    if i.data_type != 'int':
                        semantic_error = True
                array_param = int(i.name)

            if semantic_error:
                semantic_error_node = Node(token, 'Semantic error! The
array parameter must be an integer.')
                current_node.add_child(semantic_error_node)
                break
            current_node = square_stack.pop()
            square_node = Node(token, 'End Square Block')
            current_node.add_child(square_node)

    if token == "#include":
        preprocessor_directive_node = PreprocessorDirectiveNode(token,
"PreprocessorDirective")
        include_stack.append(current_node)
        current_node.add_child(preprocessor_directive_node)
        current_node = preprocessor_directive_node

    if token in standart_libraries or token_type == 'HEADER FILE':
        header_file_node = Node(token, 'Header file')
        current_node.add_child(header_file_node)
        current_node = include_stack.pop()

    if token_type == "CLASS":
        class_node = ClassNode(token, "Class")
        class_stack.append(current_node)
        current_node.add_child(class_node)
        current_node = class_node

    if token_type == 'STRUCTURE':
        struct_node = Node(token, 'Structure')
        struct_stack.append(current_node)
        current_node.add_child(struct_node)
        current_node = struct_node

```



```

        if 'FUNCTION' in token_type:
            if len(data_stack) != 0:
                variable_types[token] = data_stack[-1]
                function_already_exists = any(child.name == token for child
in current_node.children)
                if function_already_exists:
                    semantic_error_node = Node(token,
                                                f'Semantic error! Function
"{token}" has already been declared.')
                    current_node.add_child(semantic_error_node)
                    break
                function_node = Node(token, 'Function', data_stack[-
1].lower())
                data_stack.pop()
            else:
                function_node = Node(token, 'Function Call',
variable_types.get(token))

                function_stack.append(current_node)
                current_node.add_child(function_node)
                current_node = function_node

    if token_type == 'CONSTRUCTURE':
        constructure_node = Node(token, 'Constructure')
        branch_stack.append(current_node)
        current_node.add_child(constructure_node)
        current_node = constructure_node
    if 'OBJECT OF' in token_type:
        object_node = Node(token, 'Object')
        # param_stack.append(current_node)
        object_stack.append(current_node)
        current_node.add_child(object_node)
        current_node = object_node

    if token_type == 'METHOD':
        method_node = Node(token, 'Method f')
        param_stack.append(current_node)
        current_node.add_child(method_node)
        current_node = method_node

    if token == "public" or token == "private" or token == 'protected':
        if len(access_stack) == 0:
            access_modifier_node = Node(token, "AccessModifier")
            access_stack.append(current_node)
            current_node.add_child(access_modifier_node)
            current_node = access_modifier_node
        else:
            current_node = access_stack.pop()
            access_modifier_node = Node(token, "AccessModifier")
            current_node.add_child(access_modifier_node)
            current_node = access_modifier_node

    if token == "{":
        if current_node.type in ('Declare', 'Variable'):
            semantic_error_node = Node(token, 'Semantic error! Block
after variable!')
            current_node.add_child(semantic_error_node)
            break

        if current_node.type == 'Function':

```

```

        branch_list_node = Node(current_node.data_type, "Block")
        branch_stack.append(current_node)
        current_node.add_child(branch_list_node)
        current_node = branch_list_node
    else:
        branch_list_node = Node(current_node.type, "Block")
        branch_stack.append(current_node)
        current_node.add_child(branch_list_node)
        current_node = branch_list_node

    if token == "}":
        temp_node = current_node.parent
        if temp_node.type == 'Declare Array' or temp_node.type ==
'Array':
            temp_list = []
            temp_list.extend(current_node.children)
            sum_comma = 0
            sum_values = 0

            for i in temp_list:
                if i.name == ',':
                    sum_comma += 1
                else:
                    sum_values += 1

            if array_param == 0:
                array_param = sum_values

            if sum_values > array_param:
                semantic_error_node = Node(token, 'Semantic error! The
number of elements in the array exceeds the declared parameter.')
                current_node.add_child(semantic_error_node)
                break

            if sum_comma >= sum_values or (sum_values - sum_comma) >= 2:
                syntax_error_node = Node('Missing comma', f'Syntax
error!')

                current_node.add_child(syntax_error_node)
                break
        current_node = branch_stack.pop()
        close_branch_node = Node(current_node.name, 'End Block')
        if current_node.type == 'ForLoop':
            current_node.add_child(close_branch_node)
            current_node = for_stack.pop()
        elif current_node.type == 'Constructure':
            current_node.add_child(close_branch_node)
            current_node = branch_stack.pop()
        elif current_node.type == 'IfStatement':
            current_node.add_child(close_branch_node)
            current_node = if_stack.pop()
        elif current_node.type == 'ElseStatement':
            current_node.add_child(close_branch_node)
            current_node = if_stack.pop()
        elif current_node.type == 'Function':
            current_node.add_child(close_branch_node)
            if len(function_stack) != 0:
                current_node = function_stack.pop()
            sum_func = 0
            for i in function_stack:
                sum_func += 1
            if sum_func > 0:
                while sum_func != 0:
                    current_node = function_stack.pop()

```

```

        sum_func -= 1
    else:

        current_node.add_child(close_branch_node)

        if current_node.type == 'Class':
            current_node = class_stack.pop()

        if token == "(":
            if current_node.type == "Function" or current_node.type ==
'Function Call' or current_node.type == 'ForLoop' or current_node.type ==
'Method f' or current_node.type == 'Object' or current_node.type ==
'Constructure' or current_node.type == "ProgramType" or current_node.type ==
"WhileLoop" or current_node.type == "IfStatement":
                parameters_list_node = Node("Parameters", "Parameters")
                param_stack.append(current_node)
                current_node.add_child(parameters_list_node)
                current_node = parameters_list_node
            else:
                bracket_list_node = Node(token, "Bracket")
                bracket_stack.append(current_node)
                current_node.add_child(bracket_list_node)
                current_node = bracket_list_node

        if token == ")":
            sum = 0
            for i in variable_stack:
                if current_node.type in ('Variable', 'Declare', 'Declare
Array', 'Array'):
                    sum += 1
            if sum > 0:
                while sum != 0:
                    current_node = variable_stack.pop()
                    sum -= 1
            bracket_node = Node(token, 'Bracket')

            if current_node.type == 'Bracket':
                parent_node = bracket_stack.pop()
                current_node = parent_node
                current_node.add_child(bracket_node)
            elif current_node.type == "Parameters":
                parent_node = current_node.parent
                if parent_node.type == 'Function':
                    function_children = []
                    function_children.extend(current_node.children)
                    function_param = 1
                    for i in function_children:
                        if i.type in ('Declare', 'Declare Array'):
                            if i.children:
                                children_temp = []
                                children_temp.extend(i.children)
                                for j in children_temp:
                                    if j.type == 'Value':
                                        function_params.append(
                                            (function_param, i.name,
i.data_type, j.name, parent_node.name))
                                        function_param += 1
                                    else:
                                        function_params.append((function_param,
i.name, i.data_type, None, parent_node.name))
                                        function_param += 1
                            if parent_node.type == 'Function Call':

```

```

function_call_params = []
function_call_childrens = []

function_call_childrens.extend(current_node.children)
function_param = 1

for i in function_call_childrens:
    if i.data_type != None:
        function_call_params.append((function_param,
i.name, i.data_type, parent_node.name))
        function_param += 1

semantic_error = False
num_func = 0
num_params = 0

for num, tok, dt, val, fn in function_params:
    if parent_node.name == fn:
        num_func += 1

for num, tok, dt, fn in function_call_params:
    if parent_node.name == fn:
        num_params += 1

if num_params < num_func:
    for num, tok, dt, val, fn in function_params:
        if fn == parent_node.name:
            if num > num_params:
                if val != None:
                    continue
            else:
                semantic_error = True
                break
elif num_params > num_func:
    semantic_error = True

for num, tok, dt, val, fn in function_params:
    for param_num, param_tok, param_dt, param_fn in
function_call_params:
        if param_fn == fn:
            if param_num == num:
                if param_dt != 'string' and dt ==
'string' or param_dt == 'string' and dt != 'string':
                    semantic_error = True
                    break
            if semantic_error:
                break

if semantic_error:
    semantic_error_node = Node(token, 'Semantic error in
Fucntion Call!')

    current_node.add_child(semantic_error_node)
    break

if parent_node.type == 'ForLoop':
    temp_list = []
    temp_list.extend(current_node.children)
    sum_semicolon = 0
    sum_etc = 0
    for i in temp_list:
        if i.name == ';':
            sum_semicolon += 1

```

```

        else:
            sum_etc += 1

    if sum_semicolon != 2:
        syntax_error_node = Node(token, f'Syntax error!
ForLoop')

        current_node.add_child(syntax_error_node)
        break

    current_node = param_stack.pop()

    if current_node.type == 'Function Call':
        current_node = function_stack.pop()

    if current_node.type == 'ForLoop':
        for var, scope in variable_scope:
            if scope == 'for':
                variable_scope.remove((var, scope))

    if current_node.type == 'Method f':
        if len(param_stack) != 0:
            current_node = param_stack.pop()

    if token_type in ('FLOAT', 'STRING', 'INTEGER', 'BOOLEAN'):
        if current_node.data_type in ('int', 'long long', 'long',
'short', 'unsigned short', 'unsigned int', \
'unsigned long long', 'unsigned
long'):

            if token_type in ('FLOAT', 'BOOLEAN'):
                semantic_error_node = Node(token, f'Semantic error!
Incorrect type assignment! Type {current_node.data_type}')
                current_node.add_child(semantic_error_node)
                break

            if token_type in ('STRING') and token.startswith('"'):
                semantic_error_node = Node(token, f'Semantic error!
Incorrect type assignment! Type {current_node.data_type}')
                current_node.add_child(semantic_error_node)
                break

            if current_node.data_type in ('float', 'double', 'long double'):
                if token_type in ('BOOLEAN'):
                    semantic_error_node = Node(token, f'Semantic error!
Incorrect type assignment! Type {current_node.data_type}')
                    current_node.add_child(semantic_error_node)
                    break

                if token_type in ('STRING') and token.startswith('"'):
                    semantic_error_node = Node(token, f'Semantic error!
Incorrect type assignment! Type {current_node.data_type}')
                    current_node.add_child(semantic_error_node)
                    break

            if current_node.data_type in ('signed char', 'char', 'unsigned
char', 'wchar_t', 'char8_t', 'char16_t', 'char32_t'):
                if token_type in ('FLOAT', 'BOOLEAN'):
                    semantic_error_node = Node(token, f'Semantic error!
Incorrect type assignment! Type {current_node.data_type}')
                    current_node.add_child(semantic_error_node)
                    break

                if token.startswith('"'):

```

```

        semantic_error_node = Node(token, f'Semantic error!
Incorrect type assignment! Type {current_node.data_type}')
        current_node.add_child(semantic_error_node)
        break

    if current_node.data_type == 'string':
        if token_type in ('FLOAT', 'INTEGER', 'BOOLEAN'):
            semantic_error_node = Node(token, f'Semantic error!
Incorrect type assignment! Type {current_node.data_type}')
            current_node.add_child(semantic_error_node)
            break

        if token.startswith('"'):
            semantic_error_node = Node(token, f'Semantic error!
Incorrect type assignment! Type {current_node.data_type}')
            current_node.add_child(semantic_error_node)
            break

    if current_node.data_type == 'bool':
        if token_type in ('FLOAT', 'INTEGER', 'STRING'):
            semantic_error_node = Node(token, f'Semantic error!
Incorrect type assignment! Type {current_node.data_type}')
            current_node.add_child(semantic_error_node)
            break

    if token_type == 'INTEGER':
        var_node = Node(token, 'Value', 'int')
    elif token_type == 'FLOAT':
        var_node = Node(token, 'Value', 'float')
    elif token_type == 'STRING':
        var_node = Node(token, 'Value', 'str')
    elif token_type == 'BOOLEAN':
        var_node = Node(token, 'Value', 'bool')

    current_node.add_child(var_node)

    if token in {"<", ">", "==", "!=", "<=", ">="}:
        comparison_node = check_comparison(token, current_node)

    if token == ',':
        if current_node.type in ('Variable', 'Declare', 'Square Block'):
            current_node = variable_stack.pop()
            comma_node = Node(token, 'Comma')
            current_node.add_child(comma_node)

    if token == ";":
        if current_node.type == 'Declare':
            temp_children = []
            temp_children.extend(current_node.children)

            temp_check = []

            syntax_error = False
            for i in temp_children:
                if i.name == '=':
                    break
                temp_check.append(i)

            syntax_error = False
            for i in temp_check:
                if i.type == 'Operator':
                    syntax_error = True

```

```

        break

    if syntax_error:
        syntax_error_node = Node(token, 'Syntax error! Error
Symbols')
        current_node.add_child(syntax_error_node)
        break

    if len(variable_stack) != 0:
        sum = 0
        for i in variable_stack:
            if current_node.type in ('Variable', 'Declare',
'ReturnStatement', 'Declare Array', 'Array'):
                sum += 1
        if sum > 0:
            while sum != 0:
                current_node = variable_stack.pop()
                sum -= 1

    if len(std_stack) != 0:
        current_node = std_stack.pop()
        sum_std = 0
        for i in std_stack:
            sum_std += 1
        if sum_std > 0:
            while sum_std != 0:
                current_node = std_stack.pop()
                sum_std -= 1

    if current_node.type == 'Object':
        if len(object_stack) != 0:
            current_node = object_stack.pop()
        sum = 0
        for i in object_stack:
            if current_node.type in ('Object'):
                sum += 1
        if sum > 0:
            while sum != 0:
                current_node = object_stack.pop()
                sum -= 1

    if current_node.type == 'Class':
        if len(class_stack) != 0:
            current_node = class_stack.pop()
        sum_class = 0
        for i in class_stack:
            sum_class += 1
        if sum_class > 0:
            while sum_class != 0:
                current_node = class_stack.pop()
                sum_class -= 1

    if current_node.type == 'Structure':
        if len(struct_stack) != 0:
            current_node = struct_stack.pop()
        sum_struct = 0
        for i in struct_stack:
            sum_struct += 1
        if sum_struct > 0:
            while sum_struct != 0:
                current_node = struct_stack.pop()
                sum_struct -= 1

```

```

if current_node.type == 'Function':
    if len(function_stack) != 0:
        current_node = function_stack.pop()
    sum_func = 0
    for i in function_stack:
        sum_func += 1
    if sum_func > 0:
        while sum_func != 0:
            current_node = function_stack.pop()
            sum_func -= 1

if current_node.type == 'Method f':
    if len(param_stack) != 0:
        current_node = param_stack.pop()
    sum_param = 0
    for i in param_stack:
        sum_param += 1
    if sum_param > 0:
        while sum_param != 0:
            current_node = param_stack.pop()
            sum_param -= 1

statement_node = StatementNode(token, "Statement")
current_node.add_child(statement_node)

if len(data_stack) != 0:
    data_stack.pop()
else:
    continue

if token == "=":
    assignment_node = Node(token, "Assignment")
    current_node.add_child(assignment_node)

if token == ".":
    dot_node = Node(token, "DotOperator")
    current_node.add_child(dot_node)

if token == "->":
    array_node = Node(token, "Array")
    current_node.add_child(array_node)

if token == "const":
    const_node = Node(token, "ConstModifier")
    current_node.add_child(const_node)

if token == "return":
    semicolon_present = False
    for tok, _, ln in tokens:
        if ln == line and tok == ";":
            semicolon_present = True
            break
    if not semicolon_present:
        syntax_error_node = Node("Syntax error: !!!Semicolon missing
after return statement",
                                f'Syntax error! {line}')
        current_node.add_child(syntax_error_node)
        break

parent_node = current_node
return_node = Node(token, "ReturnStatement")

```



```

        return_stack.append(current_node)
        current_node.add_child(return_node)
        current_node = return_node

    if token == "std":
        std_node = Node(token, "StdNamespace")
        # std_stack.append(current_node)
        current_node.add_child(std_node)
        parent_node = current_node
        # current_node = std_node

        semicolon_present = False
        for tok, _, ln in tokens:
            if ln == line and tok == ";":
                semicolon_present = True
                break

        if not semicolon_present:
            syntax_error_node = Node(f"123Syntax error: Semicolon missing
after variable declaration.",
                                   f'Syntax error! {line}')
            current_node.add_child(syntax_error_node)
            break

    if token == '::':
        colon_node = Node(token, 'Colon')
        # std_stack.append(current_node)
        current_node.add_child(colon_node)
        # current_node = colon_node

    if token == ':':
        if current_node.type == 'StdNamespace':
            syntax_error_node = Node(token, 'Syntax error! After std')
            current_node.add_child(syntax_error_node)
            break

    if token in ('cout', 'cin'):
        if token == 'cout':
            method_node = Node(token, 'Cout')
        if token == 'cin':
            method_node = Node(token, 'Cin')

        std_stack.append(current_node)
        current_node.add_child(method_node)
        current_node = method_node

    if token in ('endl'):
        method_node = Node(token, 'Endl')
        if len(std_stack) != 0:
            current_node = std_stack.pop()
            current_node.add_child(method_node)
    if token == "<<" or token == ">>":
        arithmetic_operator_node = Node(token, "Operator")
        current_node.add_child(arithmetic_operator_node)

    if token == "for" and token_type == 'KEYWORD':
        for_node = ForNode(token, "ForLoop")
        for_stack.append(current_node)
        current_node.add_child(for_node)
        current_node = for_node

    if token == "if" and token_type == 'KEYWORD':

```

```

        if_node = IfNode(token, "IfStatement")
        if_stack.append(current_node)
        current_node.add_child(if_node)
        current_node = if_node
    elif token == 'if' and token_type != 'KEYWORD':
        syntax_error_node = Node(token, f'Syntax error! In line {line}')
        current_node.add_child(syntax_error_node)
        break

    if token == 'else' and token_type == 'KEYWORD':
        else_node = IfNode(token, 'ElseStatement')
        if_stack.append(current_node)
        current_node.add_child(else_node)
        current_node = else_node
    if token == "while" and token_type == 'KEYWORD':
        while_node = WhileNode(token, "WhileLoop")
        current_node.add_child(while_node)
        current_node = while_node
    elif token == 'while' and token_type != 'KEYWORD':
        syntax_error_node = Node(token, f'Syntax error! In line {line}')
        current_node.add_child(syntax_error_node)
        break
    if token == "new" and token_type == 'KEYWORD':
        new_node = Node(token, "NewOperator")
        current_node.add_child(new_node)
    elif token == 'new' and token_type != 'KEYWORD':
        syntax_error_node = Node(token, f'Syntax error! In line {line}')
        current_node.add_child(syntax_error_node)
        break
    if token == "delete":
        delete_node = Node(token, "DeleteOperator")
        current_node.add_child(delete_node)
    elif token == 'delete' and token_type != 'KEYWORD':
        syntax_error_node = Node(token, f'Syntax error! In line {line}')
        current_node.add_child(syntax_error_node)
        break
    if token == "break":
        delete_node = Node(token, "Break")
        current_node.add_child(delete_node)
    elif token == 'break' and token_type != 'KEYWORD':
        syntax_error_node = Node(token, f'Syntax error! In line {line}')
        current_node.add_child(syntax_error_node)
        break
    if token == "continue":
        delete_node = Node(token, "Continue")
        current_node.add_child(delete_node)
    elif token == 'continue' and token_type != 'KEYWORD':
        syntax_error_node = Node(token, f'Syntax error! In line {line}')
        current_node.add_child(syntax_error_node)
        break
    if 'LEXICAL ERROR' in token_type:
        lexical_error_node = Node(token, f'{token_type} In line {line}')
        current_node.add_child(lexical_error_node)
        break
    if 'SYNTAX ERROR' in token_type:
        syntax_error_node = Node(token, token_type)
        current_node.add_child(syntax_error_node)
        break
    if 'SEMANTIC ERROR' in token_type:
        semantic_error_node = Node(token, token_type)
        current_node.add_child(semantic_error_node)
        break

```

```

    return root
def parser():
    tokens = lexer()
    tokens_iter = tokens
    syntax_tree = build_syntax_tree(tokens)
    semantic_error = False
    has_main = False
    for index, i in enumerate(syntax_tree.children):
        if i.type in ('Structure', 'Class'):
            if index + 1 < len(syntax_tree.children) and
syntax_tree.children[index + 1].type == 'Statement':
                continue
            else:
                semantic_error = True
                break
    file_path_output = 'output_parser.txt'
    if semantic_error:
        write_output_to_file(f'Syntax error! No ; after statement',
file_path_output)
    else:
        write_output_to_file(syntax_tree.display(), file_path_output)
    return syntax_tree

```