

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 № 3  
на тему «Синтаксический 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.....	6
Выводы .....	6
Список использованных источников .....	10
Приложение А (обязательное) Листинг исходного кода .....	11

## **1 ПОСТАНОВКА ЗАДАЧИ**

Целью выполнения данной лабораторной работы является разработка синтаксического анализатора для языка программирования C++. Необходимо вывести результат синтаксического анализа в виде дерева составляющих, а также обработать возможные синтаксические ошибки.

## 2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Этапами трансляции являются лексический анализ, синтаксический анализ, семантический анализ, оптимизация, генерация кода.

На этапе генерации компилятор создаёт код, который состоит из инструкций, понятных для целевой аппаратной платформы. Итоговый файл компилируется в исполняемый, который может быть запущен на целевой платформе без необходимости наличия исходного кода.

Фаза эмуляции интерпретатора происходит во время выполнения программы. В отличие от компилятора, интерпретатор работает с кодом напрямую, без предварительной генерации машинного кода.

Первым этапом трансляции является лексический анализатор. Он сканирует последовательность символов, составляющих исходную программу, и группирует их в лексемы, представляющие собой значимые последовательности символов. Лексема представляет собой базовую единицу, которая может быть ключевым словом, идентификатором или константным значением. Для каждой лексемы анализатор создает токен, который является сущностью, содержащей имя и значение. [1]

Синтаксический анализатор проверяет, соответствуют ли выражения, сформированные в исходной программе, правилам грамматики языка программирования. Он использует результаты лексического анализа в качестве входных данных и разбирает их в соответствии с грамматическими правилами. Обычно результат синтаксического анализа представляется в виде синтаксического дерева разбора. [2]

Существует несколько видов деревьев разбора, а именно дерево зависимостей и дерево составляющих.

Дерево составляющих и дерево синтаксического разбора – это два термина, которые описывают одно и то же. Дерево составляющих описывает структуру программы на уровне синтаксиса, разделяя её на отдельные синтаксические единицы, такие как функции и циклы. Дерево зависимостей, в свою очередь, помогает понять, какие части программы зависят от других, описывая зависимости между компонентами программы и фокусируясь на отношениях между этими компонентами.

Грамматика представляет собой набор правил, которые определяют, как нужно формировать строки из алфавита языка в соответствии с синтаксисом языка.

Существует множество методов синтаксического анализа, включая:

- LL(1), LR(1);
- LL(k), LR(k);
- LALR;
- GLK;
- SLK;
- метод рекурсивного спуска.

Метод LL(1), который расшифровывается как «Слева-направо, слева-рука, 1 символ предварительного просмотра», сканирует входной текст слева направо, используя один символ предварительного просмотра, и строит наиболее левую выводимую последовательность. Метод LR(1), обозначаемый как «Слева-направо, справа-рука, 1 символ предварительного просмотра», также сканирует входной текст слева направо, но использует один символ предварительного просмотра, и строит наиболее правую выводимую последовательность.

Методы LL(k) и LR(k) аналогичны, за исключением использования последовательности символов предварительного просмотра вместо одного символа.

Главное различие между методами LL и LR заключается в порядке прохода: метод синтаксического анализа LL начинает проход с корня к листьям (сверху вниз), в то время как метод LR начинает с листьев к корню (снизу вверх).

Метод LALR представляет собой вариацию метода LR с использованием сокращенных таблиц разбора, объединяя состояния с одинаковыми наборами пунктов.

Метод GLK является расширенным вариантом метода LR и способен обрабатывать неоднозначные грамматики, где одной строке соответствуют несколько путей разбора.

Метод SLK представляет собой упрощенную версию метода LR и менее гибок в разборе строк по правилам грамматики.

Метод рекурсивного спуска использует рекурсивные функции для разбора и анализа грамматических правил.

## 3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В процессе выполнения лабораторной работы был разработан синтаксический анализатор, использующий метод рекурсивного спуска. Он принимает на вход результаты лексического анализатора и возвращает синтаксическое дерево разбора в качестве выходных данных.

Для проверки работы анализатора использовались примеры кода, взятые из отчёта по первой лабораторной работе по данной дисциплине. Листинг первого тестового кода, реализующего поиск подстроки в строке, представлен на рисунке 3.1.

```
#include <iostream>
#include <string>
int main() {
    std::string text = "Hello, world!";
    std::string pattern = "world";
    int n = text.length();
    int m = pattern.length();
    int pos = -1;
    for (int i = 0; i <= n - m; ++i) {
        int j;
        for (j = 0; j < m; ++j) {
            if (text[i + j] != pattern[j]) {
                break;
            }
        }
        if (j == m) {
            pos = i;
            break;
        }
    }
    if (pos != -1) {
        std::cout << "Pattern found at position: " << pos << std::endl;
    } else {
        std::cout << "Pattern not found." << std::endl;
    }
    return 0;
}
```

Рисунок 3.1 – Листинг первого тестового кода

Результат части обработки первого тестового кода на рисунке 3.2.

```
| - PreprocessorDirective: #include
|   | - Header file: <iostream>
| - PreprocessorDirective: #include
|   | - Header file: <string>
| - Function Call: main
|   | - Parameters: Parameters
| - Block: Block
|   | - StdNamespace: std
|     | - Colon: ::
|       | - Declare: string text
|         | - Assignment: =
|         |   | - Value: "Hello, world!"
| - Statement: ;
|   | - StdNamespace: std
|     | - Colon: ::
|       | - Declare: string pattern
|         | - Assignment: =
|         |   | - Value: "world"
| - Statement: ;
| - Declare: int n
|   | - Assignment: =
|   |   | - Variable: text
|   |   |   | - DotOperator: .
|   |   |   | - Method f: length
|   |   |   | - Parameters: Parameters
| - Statement: ;
| - Declare: int m
|   | - Assignment: =
|   |   | - Variable: pattern
|   |   |   | - DotOperator: .
|   |   |   | - Method f: length
|   |   |   | - Parameters: Parameters
| - Statement: ;
| - Declare: int pos
|   | - Assignment: =
|   |   | - Operator: -
|   |   |   | - Value: 1
| - Statement: ;
| - ForLoop: for
|   | - Parameters: Parameters
|   |   | - Declare: int i
```

Рисунок 3.2 – Результат обработки первого текстового кода

Листинг второго тестового кода, реализующего класс `Rectangle`, представлен на рисунке 3.3.

```
#include <iostream>
class Rectangle {
private:
    int width;
    int height;
public:
    Rectangle(int w, int h){}
    void setWidth(int w) { width = w; }
    void setHeight(int h) { height = h; }
    int getWidth() const { return width; }
    int getHeight() const { return height; }
    int area() const { return width * height; }
};
int main() {
    int a;
    Rectangle rect(5, 3);
    std::cout << "Width: " << rect.getWidth() << std::endl;
    std::cout << "Height: " << rect.getHeight() << std::endl;
    std::cout << "Area of the rectangle: " << rect.area() << std::endl;
    rect.setWidth(7);
    rect.setHeight(4);
    std::cout << "Updated area of the rectangle: " << rect.area() << std::endl;
    return 0;
}
```

Рисунок 3.3 – Листинг второго тестового кода

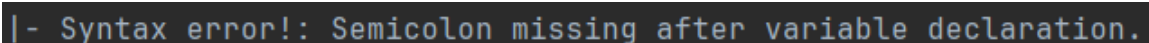
Результат части обработки второго тестового кода представлен на рисунке 3.4.

```
| - PreprocessorDirective: #include
| - Header file: <iostream>
| - Class: Rectangle
| - Block: Block
|   | - AccessModifier: private
|   |   | - Declare: int width
|   |   | - Statement: ;
|   |   | - Declare: int height
|   |   | - Statement: ;
|   | - AccessModifier: public
|   |   | - Constructure: Rectangle
|   |   |   | - Parameters: Parameters
|   |   |   |   | - Declare: int w
|   |   |   |   | - Comma: ,
|   |   |   |   | - Declare: int h
|   |   |   | - Block: Block
|   |   | - Function: void setWidth
|   |   |   | - Parameters: Parameters
|   |   |   |   | - Declare: int w
|   |   |   | - Block: Block
|   |   |   |   | - Variable: width
|   |   |   |   |   | - Assignment: =
|   |   |   |   |   | - Variable: w
|   |   |   |   | - Statement: ;
|   |   | - Function: void setHeight
|   |   |   | - Parameters: Parameters
|   |   |   |   | - Declare: int h
|   |   |   | - Block: Block
|   |   |   |   | - Variable: height
|   |   |   |   |   | - Assignment: =
|   |   |   |   |   | - Variable: h
|   |   |   |   | - Statement: ;
|   |   | - Function: int getWidth
|   |   |   | - Parameters: Parameters
|   |   |   |   | - ConstModifier: const
|   |   |   | - Block: Block
|   |   |   |   | - ReturnStatement: return
|   |   |   |   |   | - Variable: width
|   |   |   |   |   | - Statement: ;
```

Рисунок 3.4 – Результат обработки второго тестового кода

В ходе данной лабораторной работы также были обработаны возможные синтаксические ошибки. Это включало в себя проверку на наличие лишних или недостающих скобок, кавычек и символов многострочного комментария, а также наличие или отсутствие символов «,» в перечислениях и «;» после операций. Также проводилась проверка на неправильную последовательность символов и отсутствие знака одной косой черты у однострочного комментария.

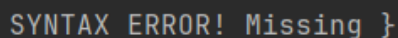
Пример ошибки, связанной с пропуском символа «;» в конце строки, представлен на рисунке 3.5.



```
|- Syntax error!: Semicolon missing after variable declaration.
```

Рисунок 3.5 – Предупреждение об отсутствии символа «;» на конце строки

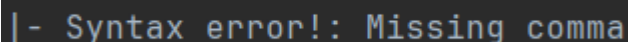
При написании кода существует возможность допустить ошибку в количестве скобок, кавычек или символов многострочного комментария. Предупреждение об ошибке такого характера представлено на рисунке 3.6.



```
SYNTAX ERROR! Missing }
```

Рисунок 3.6 – Предупреждение о недостающей скобке

Синтаксический анализатор предупредит об упущенном символе «,». Пример данной ошибки представлен на рисунке 3.7.



```
|- Syntax error!: Missing comma
```

Рисунок 3.7 – Предупреждение об отсутствии символа «,» в перечислении

Таким образом в ходе данной лабораторной работы был разработан синтаксический анализатор, который основан на алгоритме рекурсивного спуска, выводит дерево составляющих в конце обработки программы, а также обрабатывает возможные синтаксические ошибки.



## **ВЫВОДЫ**

В ходе лабораторной работы был реализован синтаксический анализатор, основанный на методе рекурсивного спуска. По итогу обработки результатов лексического анализатора синтаксическим, строится дерево составляющих или дерево синтаксического разбора программы. Также была реализована обработка возможных синтаксических ошибок.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Лексический анализатор [Электронный ресурс]. – Режим доступа: <https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/>. – Дата доступа: 10.03.2024.

[2] Синтаксический анализатор [Электронный ресурс]. – Режим доступа: <https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/>. – Дата доступа: 10.03.2024.

[3] Введение в C++ [Электронный ресурс]. – Режим доступа: <https://metanit.com/cpp/tutorial/2.5.php>. – Дата доступа: 11.03.2024.

[4] Типы данных [Электронный ресурс]. – Режим доступа: <https://metanit.com/cpp/tutorial/2.3.php>. – Дата доступа: 11.03.2024.

[5] Операторы в C++ [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/cpp/c-operators>. – Дата доступа: 13.03.2024.

[6] Функции C++ [Электронный ресурс]. – Режим доступа: <https://metanit.com/cpp/tutorial/3.1.php>. – Дата доступа: 13.03.2024.

[7] Классы C++ [Электронный ресурс]. – Режим доступа: <https://ravesli.com/urok-113-klassy-obekty-i-metody-klassov/>. – Дата доступа: 13.03.2024.

# ПРИЛОЖЕНИЕ А

## (обязательное)

### Листинг исходного кода

#### Листинг 1 – Программный код parser.py

```
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';'

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_cho(token, current_node):
    if token == ';':
        cho_node = Node(token, "Cho")
        # current_node.add_child(data_list_node) # Добавляем data_list_node
        # в текущий узел
        current_node.add_child(cho_node)
        return cho_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, date_type=None, array_in=None,
parent=None, children=None):
        self.name = name
        self.type = node_type
        self.date_type = date_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.date_type is not None and self.array_in is not None:
            tree_structure += f"{indent}|- {self.type}: {self.date_type} {self.name} [{self.array_in}]\n"
        else:
            tree_structure += f"{indent}|- {self.type}: {self.name}\n"
```

```

        if self.date_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.date_type}
{self.name}\n"
        elif self.date_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 ArrayNode(Node):
    def __init__(self, data_type: None, name, array_in: []):
        if data_type is not None:
            super().__init__(f"{data_type} {name}[{array_in}]", "Array")
        else:
            super().__init__(f"{name}[{array_in}]", "Array")
        self.data_type = data_type
        self.name = name
        self.size = array_in
    def display(self, level=0):
        indent = "    " * level
        if self.data_type is not None:
            print(f"{indent}|- Array: {self.data_type}
{self.name}[{self.size}]")
        else:
            print(f"{indent}|- Array: {self.name}[{self.size}]")
class VariableNode(Node):
    def __init__(self, data_type: None, name):
        if data_type is not None:
            super().__init__(f"{data_type} {name}", "Declare")
        else:
            super().__init__(f"{name}", "Variable")
        self.data_type = data_type
        self.name = name
    def display(self, level=0):
        indent = "    " * level
        if self.data_type:
            print(f"{indent}|- Declare: {self.data_type} {self.name}")
        else:
            print(f"{indent}|- Variable: {self.name}")
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 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 = {}
    help_tokens = []
    branch_stack = []
    param_stack = []
    bracket_stack = []
    include_stack = []
    access_stack = []
    data_stack = []
    variable_stack = []
    io_stack = []
    if_stack = []
    return_stack = []
    class_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:
        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
            comment_node = CommentNode(current_comment[:-1], "Comment")
            current_node.add_child(comment_node)

```

```

        current_comment = ""
        continue
    elif inside_comment:
        current_comment += token + " "
        continue
    if 'VARIABLE' in token_type or 'POINTER' in token_type or 'ARRAY' in
token_type:
        match = re.search(pattern, token_type)
        if match:
            if len(data_stack) != 0:
                data_type = match.group(1)
                if data_type == 'STRING':
                    is_string_declaration = True
                    variable_node = Node(token, 'Declare', data_type.lower())
                    data_stack.pop()
                    is_value = True
            else:
                is_string_declaration = False
                variable_node = Node(token, 'Variable', None)
                is_value = True
            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', 'Colon', 'Variable',
'Operator Input', '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
        else:
            variable_node = Node(token, 'Variable', None)
            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') or parent_node.type in (
                    'Parameters', 'Function', 'ForLoop', 'Variable',
'Operator Input'):
                    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:
        array_name = token

```

```

        is_array_declaration = True
        match = re.search(pattern, token_type)
        if match:
            if len(data_stack) != 0:
                data_type = match.group(1)
            else:
                data_type = None
        else:
            print('Error')
    if token == '[':
        tok_list = []
        for inner_token, _, line in tokens:
            tok_list.append(inner_token)
            array_in = find_chars_between(tok_list, '[', ']')
            if inner_token == ']':
                if is_array_declaration:
                    if len(data_stack) != 0:
                        array_node = Node(array_name, 'Declare array',
data_type.lower(), array_in)
                        data_stack.pop()
                    else:
                        array_node = Node(array_name, 'Array', None,
array_in)
                        array_name = None
                        variable_stack.append(current_node)
                        current_node.add_child(array_node)
                        current_node = array_node
                        parent_node = current_node.parent
Semicolon missing after variable declaration.",
                        is_array_declaration = False
                        tok_list.clear()
                        break
                if is_string_declaration:
                    array_node = Node(array_in, 'Inside array')
                    current_node.add_child(array_node)
                    is_string_declaration = False
                    tok_list.clear()
                    break
            break
    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":
        parent_node = current_node
        class_node = ClassNode(token, "Class")
        class_stack.append(current_node)
        current_node.add_child(class_node)
        current_node = class_node
    if 'FUNCTION DEC' in token_type:
        match = re.search(pattern, token_type)
        if match:
            if len(data_stack) != 0:
                data_type = match.group(1)
                function_node = Node(token, 'Function',
data_type.lower())

```

```

        data_stack.pop()
    else:
        function_node = Node(token, 'Function', None)
        branch_stack.append(current_node)
        current_node.add_child(function_node)
        current_node = function_node
    else:
        print('Error')
if token_type == 'FUNCTION CALL':
    function_call_node = Node(token, 'Function Call')
    param_stack.append(current_node)
    current_node.add_child(function_call_node)
    current_node = function_call_node

if 'OBJECT OF' in token_type:
    object_node = Node(token, 'Object')
    param_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_type == 'CONSTRUCTURE':
    constructure_node = Node(token, 'Constructure')
    branch_stack.append(current_node)
    current_node.add_child(constructure_node)
    current_node = constructure_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 == "{":
    sum = 0
    for i in variable_stack:
        if current_node.type in ('Variable', 'Declare'):
            sum += 1
    if sum > 0:
        while sum != 0:
            current_node = variable_stack.pop()
            sum -= 1
    branch_list_node = Node("Block", "Block")
    branch_stack.append(current_node)
    current_node.add_child(branch_list_node)
    current_node = branch_list_node
    if current_node.type == 'Block':
        parent_node = current_node.parent
        num_values = 0
        num_commas = 0
        tok_list = []
        if parent_node.type == 'Declare array':
            for inner_token, _, line in tokens:
                tok_list.append(inner_token)
            array_in = find_chars_between(tok_list, '{', '}')

```



```

        num_values = len(re.findall(numbers, array_in))
        num_commas = len(re.findall(commas, array_in))
        if num_commas >= num_values or (num_values - num_commas)
>= 2:
            syntax_error_node = Node('Missing comma', f'Syntax
error!')
            current_node.add_child(syntax_error_node)
            break
    elif token == "}":
        current_node = branch_stack.pop()
        if current_node.type == 'ForLoop':
            current_node = for_stack.pop()
        if current_node.type == 'Constructure' or current_node.type ==
'Function':
            current_node = branch_stack.pop()
            if current_node.type == 'IfStatement':
                current_node = if_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
                if current_node.type == 'Parameters':
                    parent_node = current_node.parent
                    num_semicolon = 0
                    tok_list = []
                    if parent_node.type == 'ForLoop':
                        for inner_token, _, line in tokens:
                            tok_list.append(inner_token)
                            array_in = find_chars_between(tok_list, '(', ')')
                            num_semicolon = len(re.findall(semicolon,
array_in))
                    if num_semicolon % 2 != 0:
                        syntax_error_node = Node(token, f'Syntax error!
In line {line}')
                        current_node.add_child(syntax_error_node)
                        break
            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'):
                    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":

```

```

        current_node = param_stack.pop()
        if current_node.type == 'Function Call' or current_node.type
== 'Method f' or current_node.type == 'Object':
            current_node = param_stack.pop()
    if token_type in ('FLOAT', 'STRING', 'INTEGER'):
        var_node = Node(token, 'Value')
        current_node.add_child(var_node)
        if len(io_stack) != 0:
            current_node = io_stack.pop()
        sum = 0
        for i in io_stack:
            sum += 1
        if sum > 0:
            while sum != 0:
                current_node = io_stack.pop()
                sum -= 1
    if token in {"<", ">", "==", "!=", "<=", ">="}:
        comparison_node = check_comparison(token, current_node)
    if token == ',':
        if current_node.type in ('Variable', 'Declare'):
            current_node = variable_stack.pop()
            comma_node = Node(token, 'Comma')
            current_node.add_child(comma_node)
    if token == ";":
        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 == '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 == '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 == "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 == ";" and parent_node.type in
('ProgramType', 'Block', 'Operator Input', 'Object'):
                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 in ('cout', 'endl', 'cin') and token_type == "METHOD":
        method_node = Node(token, "Method")
        current_node.add_child(method_node)
        current_node = std_stack.pop()
        if len(std_stack) != 0:
            current_node = std_stack.pop()

```

```

sum = 0
for i in std_stack:
    sum += 1
if sum > 0:
    while sum != 0:
        current_node = std_stack.pop()
        sum -= 1
if token in ('cout', 'endl', 'cin') and token_type == "KEYWORD":
    method_node = Node(token, "Method")
    current_node.add_child(method_node)
    # current_node = io_stack.pop()
    if len(io_stack) != 0:
        current_node = io_stack.pop()
sum = 0
for i in io_stack:
    sum += 1
if sum > 0:
    while sum != 0:
        current_node = io_stack.pop()
        sum -= 1
if token == "<<" or token == ">>":
    io_operator_node = Node(token, 'Operator Input')
    io_stack.append(current_node)
    current_node.add_child(io_operator_node)
    current_node = io_operator_node
    # current_node.add_child(io_operator_node)
if token in operators and token_type == 'ARITHMETIC OPERATOR':
    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
elif token == 'for' 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 == "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 = Node(token, "Else")
    parent_node = current_node.parent
    if isinstance(parent_node, IfNode):
        if_else_node = IfElseNode(token, "IfElseStatement")
        parent_node.add_child(if_else_node)
        current_node = if_else_node
        current_node = branch_stack.pop()
    else:
        current_node.add_child(else_node)
elif token == 'else' 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 == "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
    return root
tokens = lexer()
tokens_iter = tokens
syntax_tree = build_syntax_tree(tokens_iter)
file_path_output = 'output_parser.txt'
write_output_to_file(syntax_tree.display(), file_path_output)

```