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](#_Toc157722973)

[2 Краткие теоретические сведения 4](#_Toc157722974)

[3 Результаты выполнения лабораторной работы 6](#_Toc157722975)

[Выводы 6](#_Toc157722976)

[Список использованных источников 10](#_Toc157722977)

[Приложение А (обязательное) Листинг исходного кода 11](#_Toc157722978)

# **1 ПОСТАНОВКА ЗАДАЧИ**

Целью выполнения данной лабораторной работы является разработка синтаксического анализатора для языка программирования С++. 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.

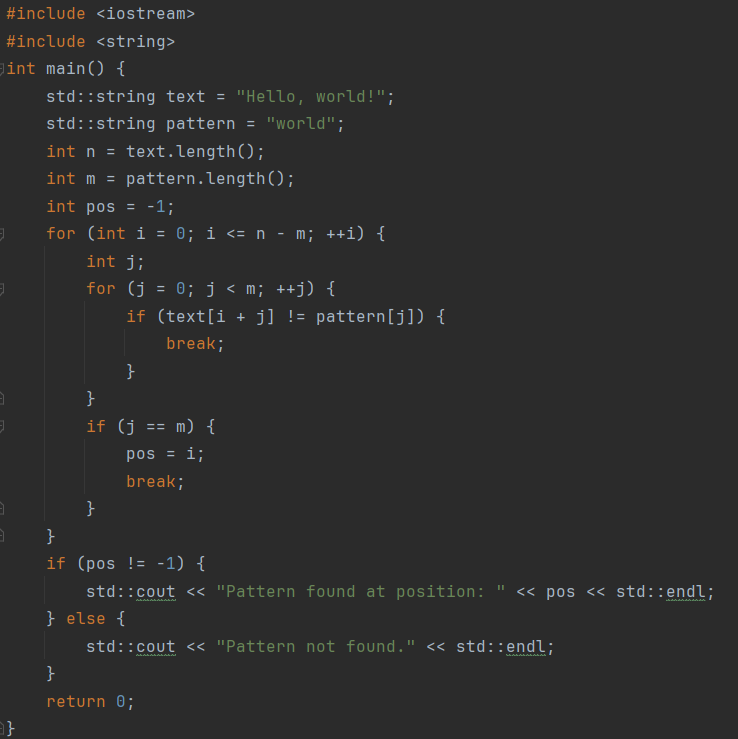


Рисунок 3.1 – Листинг первого тестового кода

Результат части обработки первого тестового кода на рисунке 3.2.

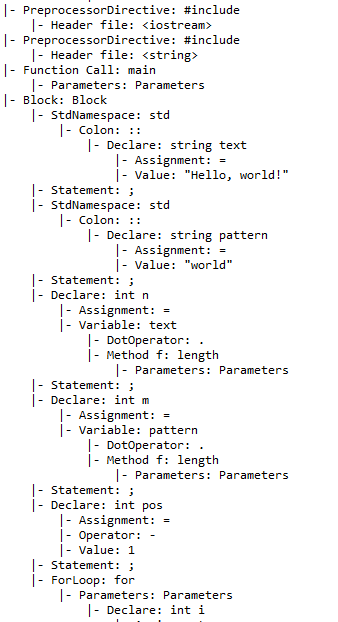


Рисунок 3.2 – Результат обработки первого текстового кода

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

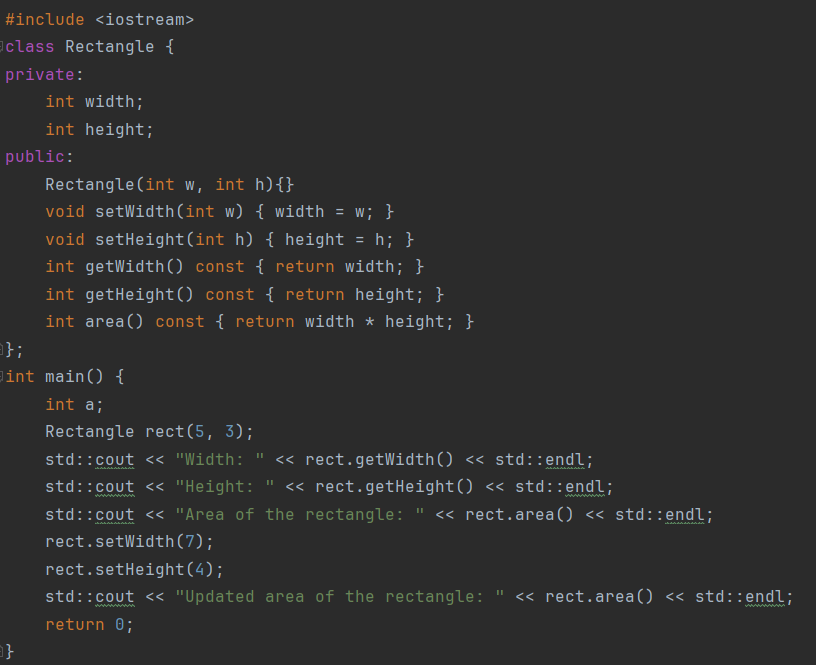


Рисунок 3.3 – Листинг второго тестового кода

Результат части обработки второго тестового кода представлен на рисунке 3.4.

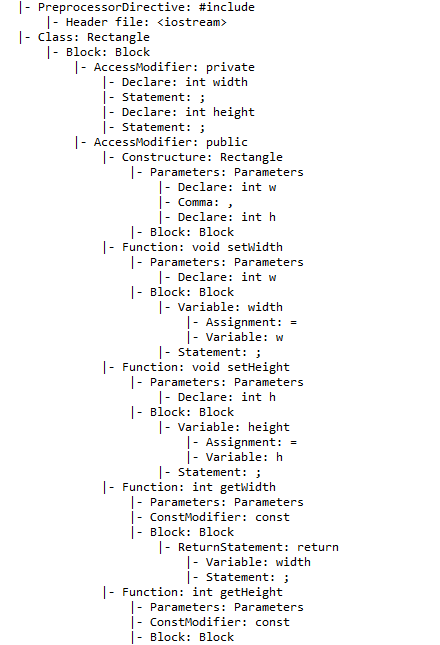


Рисунок 3.4 – Результат обработки второго тестового кода

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.



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.



Рисунок 3.6 – Предупреждение о недостающей скобке

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



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 С++ [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 С++ [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/cpp/c-operators>. – Дата доступа: 13.03.2024.
6. Функции С++ [Электронный ресурс]. – Режим доступа: https://metanit.com/cpp/tutorial/3.1.php. – Дата доступа: 13.03.2024.
7. Классы С++ [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\_chto(token, current\_node):

if token == ';':

chto\_node = Node(token, "Chto")

# current\_node.add\_child(data\_list\_node) # Добавляем data\_list\_node в текущий узел

current\_node.add\_child(chto\_node)

return chto\_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"

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 == 'CONSTUCTURE':

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)