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             А. К. Хрищанович

Проверил                          Н. Ю. Гриценко

Минск 2024

**СОДЕРЖАНИЕ**

[1 Постановка задачи 3](#_Toc157722973)

[2 Краткие теоретические сведения 4](#_Toc157722974)

[3 Результаты выполнения лабораторной работы 5](#_Toc157722975)

[Выводы 6](#_Toc157722976)

[Список использованных источников 7](#_Toc157722977)

[Приложение А (обязательное) Листинг исходного кода 8](#_Toc157722978)

# **1 ПОСТАНОВКА ЗАДАЧИ**

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

# **2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

К этапам трансляции относятся следующие этапы:

– 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]

Семантический анализ обычно заключается в проверке правильности типа и вида всех идентификаторов и данных, используемых в программе.

Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице идентификаторов для последующего использования в процессе генерации промежуточного кода.

Кроме того, на этом этапе компилятор должен также проверить, соблюдаются ли определенные контекстные условия входного языка.

В современных языках программирования одним из примеров контекстных условий может служить обязательность описания переменных, то есть для каждого использующего вхождения идентификатора должно существовать единственное определяющее вхождение.

Число и атрибуты фактических параметров вызова процедуры должны быть согласованы с определением этой процедуры.

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 описывается контекстно-свободной грамматикой создание дерева в синтаксическом анализаторе является тривиальной задачей. Большинство правил в грамматике создают новую вершину, а символы в правиле становятся рёбрами. Правила, которые ничего не привносят в дерево, просто заменяются в вершине одним из своих символов. Кроме того, анализатор может создать полное дерево разбора и затем пройти по нему, удаляя узлы и рёбра, 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 совершены проверки на такие типы ошибок как:

– объявление одноименных переменных или функций в одной области видимости;

– несовпадение параметров и аргументов при вызове функции;

– неверное преобразование типов данных;

– неверное применение закрывающихся одинарных и двойных кавычек;

– неверное указание размера массива.

При неверном преобразовании типов данных, когда целочисленной переменной присваивается, например, значение с плавающей точкой, будет выведена ошибка об этом. Пример семантической ошибки при неверном преобразовании типов данных представлен на рисунке 3.1.



Рисунок 3.1 – Ошибка при неверном преобразовании типов данных

При несовпадении количества параметров и аргументов при вызове функции с учетом того, что параметрам функции не присваивается значение, также будет выведена семантическая ошибка. Пример тестового кода с ошибкой данного типа представлен на рисунке 3.2.

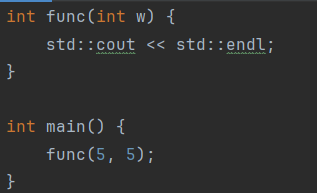


Рисунок 3.2 – Пример тестового кода

Пример семантической ошибки при несовпадении количества параметров и аргументов представлен на рисунке 3.3.



Рисунок 3.3 – Ошибка при различном количестве параметров и аргументов

При помещении в одинарные кавычки более, чем одного символа, будет вызвана также семантическая ошибка. Пример тестового кода представлен на рисунке 3.4.



Рисунок 3.4 – Пример тестового кода

Пример семантической ошибки при неверном использовании одинарных или двойных кавычек представлен на рисунке 3.5.



Рисунок 3.5 – Ошибка при неверном использовании одинарных или двойных кавычек

Ошибка при неверном указании количества элементов в массиве представлена на рисунке 3.6.



Рисунок 3.6 – Ошибка при неверном указании количества элементов в массиве

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 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 ошибок.

# **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 С++ [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 С++ [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/cpp/c-operators>. – Дата доступа: 27.02.2024.
6. Функции С++ [Электронный ресурс]. – Режим доступа: https://metanit.com/cpp/tutorial/3.1.php. – Дата доступа: 27.02.2024.
7. Классы С++ [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\_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, 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', 'Cout', '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 == 'CONSTUCTURE':

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 Bloсk'):

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