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             К.А. Тимофеев

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](#_Toc157722976) 8

[Список использованных источников 9](#_Toc157722977)

[Приложение А (обязательное) Листинг исходного кода 10](#_Toc157722978)

# **1 ПОСТАНОВКА ЗАДАЧИ**

Целью выполнения данной лабораторной работы является разработка собственного 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**

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]

Существует несколько видов деревьев разбора, к которым относятся:

– дерево зависимостей;

– дерево составляющих.

Дерево составляющих и дерево синтаксического разбора – это два термина, которые обозначают одно и тоже. Дерево составляющих описывает структура программы на уровне ее синтаксиса, разбивая ее на отдельные синтаксические единицы, например функции, циклы.

Дерево зависимостей в свою очередь помогает понять, какие части программы зависят от других. Дерево зависимостей описывает зависимостимежду компонентами программы и сфокусировано на отношениях между этими компонентами.

Грамматика – набор правил, описывающих, как необходимо формировать из алфавита языка строки, соответствующие синтаксису языка.

Существует множество методов синтаксического анализа, к которым относятся:

– LL(1), LR(1);

– LL(k), LR(k);

– LALR;

– GLK;

– SLK;

– метод рекурсивного спуска.

Метод LL(1), как сокращение от «Left-to-Right, left-hand, 1 symbol lookahead», сканирует входной текст слева направо, используя один символ предварительного просмотра, и строит наиболее левую выводимую последовательность.

Метод LR(1), как сокращение от «Left-to-Right, right-hand, 1 symbol lookahead», сканирует входной текст слева направо, используя один символ предварительного просмотра, и строит наиболее правую выводимую последовательность.

Метод LL(k), как сокращение от «Left-to-Right, left-hand, k symbols lookahead», сканирует входной текст слева направо, используя последовательность символов предварительного просмотра, и строит наиболее левую выводимую последовательность.

Метод LR(k), как сокращение от «Left-to-Right, right-hand, k symbols lookahead», сканирует входной текст слева направо, используя последовательность символов предварительного просмотра, и строит наиболее правую выводимую последовательность.

Глобальным отличие методов LL от LR является то, что метод синтаксического анализа LL начинается проходит с корня до листьев или сверху вниз (top-down), строя выводимую последовательно, в то время как метод LR проходит от листьев к корню или снизу вверх (bottom-up).

Метод синтаксического анализа LALR является вариантом алгоритма синтаксического анализа LR с использованием сокращенных или сжатых таблиц разбора. Сокращенная таблица создается путем объединения состояний, которые имеют одинаковые наборы пунктов.

Метод синтаксического анализа GLK является расширенным вариантом алгоритма LR, позволяя обрабатывать неоднозначные грамматики.

К неоднозначным грамматикам относятся те грамматики, в которых существует более одного способа произвести синтаксический анализ некоторой строки с помощью правил грамматики. Это означает что для некоторых цепочек существует несколько путей разбора.

Метод синтаксического разбора SLK является упрощенным вариантом алгоритма LR. Метод SLK более ограничен в возможностях разбора строк по правилам грамматики и более прост в реализации алгоритма

Метод рекурсивного спуска использует рекурсивную функцию для разбора и анализа грамматических правил.

# **3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ             РАБОТЫ**

В ходе лабораторной работы был реализован собственный синтаксический анализатор при помощи метода рекурсивного спуска, который на вход принимает результат лексического анализатора, а в результате отображает синтаксическое дерево разбора.

Листинг тестового кода представлен на рисунке 3.1.

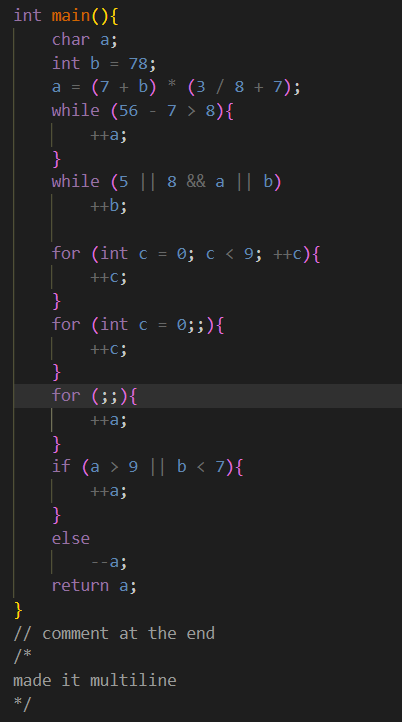


Рисунок 3.1 – Листинг первого тестового кода

Результат части обработки первого тестового кода представлен на рисунке 3.2.

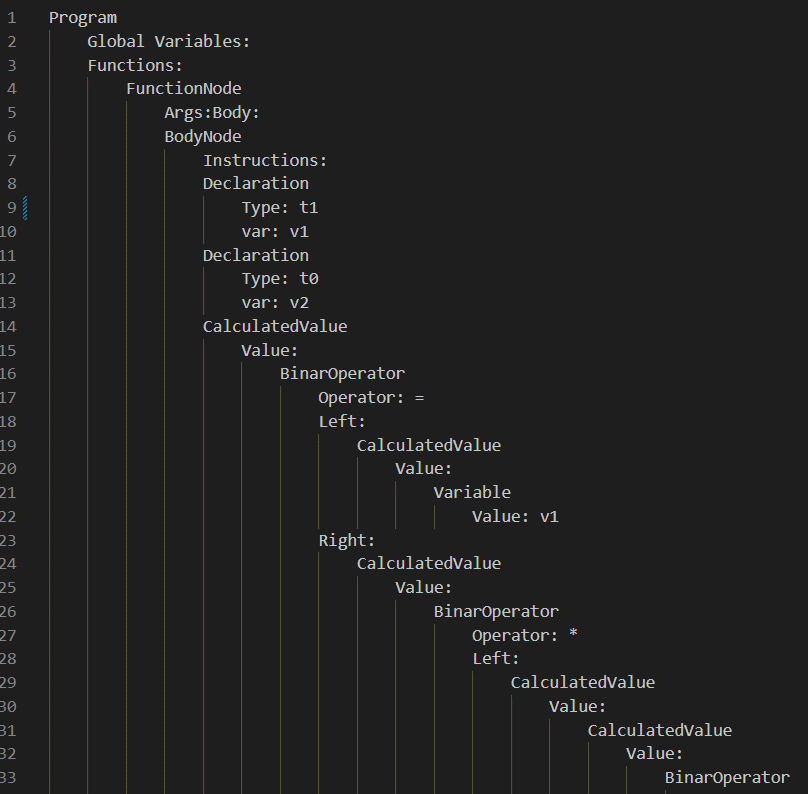


Рисунок 3.2 – Фрагмент результата обработки первого текстового кода

При написании кода на языке программирования С может возникнуть проблема, когда было забыто инициализирующее значение. Пример данной ошибки представлен на рисунке 3.3.



Рисунок 3.3 – Отсутствие символа «;» на конце строки

При отсутствии инициализирующего значения при имеющемся операторе присваивания анализатор выдаст ошибку. Сообщение ошибки представлено на рисунке 3.4.

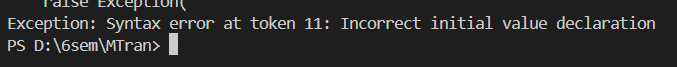


Рисунок 3.4 – Предупреждение об некорректном присваивании значения

В таком случае дерево не будет выведено.

# **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: 27.03.2024.
2. Синтаксический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/. – Дата доступа: 27.03.2024.

## **ПРИЛОЖЕНИЕ А**

## **(обязательное)**

## **Листинг исходного кода**

Листинг 1 – Программный код SyntaxAnalyzer.py

from LexicalAnalyzer import \*

from LexicalAnalyzer import IStringable, MyLiteral, MyType, VariableTableCell

keywordsDict = {kw.word: kw for kw in keywordsList}

bracketsdict = {br.symbols: br for br in brackets}

separatorsDict = {s.symbol: s for s in separators}

operatorsDict = {op.symbol: op for op in operators}

def find(iterable, func):

for it in iterable:

if func(it):

return it

class INode(IStringable):

def \_\_init\_\_(self, parent):

self.parent: INode = parent

def execute(self):

pass

# returns index of last scanned token

def buildTree(

self,

tokens: list[IStringable],

index: int,

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

) -> int:

pass

def getVariable(self, name: str):

pass

def toString(self, i: int):

pass

class UnarOperatorNode(INode):

def \_\_init\_\_(self, parent, operator: MyOperator):

super().\_\_init\_\_(parent)

self.operator = operator

self.variable = None

def buildTree(

self,

tokens: list[IStringable],

index: int,

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

) -> int:

i = index

op = self.operator

if (

tokens[i + 1] in vars\_table

and op.direction == "both"

or op.direction == "right"

):

self.variable = tokens[i + 1]

i += 1

elif (

tokens[i - 1] in vars\_table

and op.direction == "left"

or op.direction == "both"

):

self.variable = tokens[i - 1]

else:

raise Exception(

f"Syntax error at token {i}: unar operator require identifier as argument"

)

return i

def toString(self, i: int):

tab: str = ""

for ind in range(i):

tab += " "

return f"{tab}UnarOperator\n{tab} Operator: {self.operator.toString(i)}\n{tab} Var: {self.variable.toString(i)}"

class BinarOperatorNode(INode):

def \_\_init\_\_(self, parent: INode, operator: MyOperator, begin: int, end: int):

super().\_\_init\_\_(parent)

self.operator = operator

self.left = None

self.right = None

self.begin = begin

self.end = end

def buildTree(

self,

tokens: list[IStringable],

index: int,

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

) -> int:

lcn = CalculatedValueNode(self, self.begin, index)

lcn.buildTree(tokens, self.begin, type\_table, vars\_table, literals\_table)

self.left = lcn

rcn = CalculatedValueNode(self, index + 1, self.end)

rcn.buildTree(tokens, index + 1, type\_table, vars\_table, literals\_table)

self.right = rcn

return self.end

def toString(self, i: int):

tab: str = ""

for ind in range(i):

tab += " "

return f"{tab}BinarOperator\n{tab} Operator: {self.operator.symbol}\n{tab} Left: \n{self.left.toString(i+2)}\n{tab} Right: \n{self.right.toString(i+2)}"

class GetLiteralValueNode(INode):

def \_\_init\_\_(self, parent):

super().\_\_init\_\_(parent)

def buildTree(

self,

tokens: list[IStringable],

index: int,

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

) -> int:

if not (tokens[index] in literals\_table):

raise Exception(f"Syntax error at token {i}: Unkown literal")

self.literal: MyLiteral = tokens[index]

return index

def toString(self, i: int):

tab: str = ""

for ind in range(i):

tab += " "

return f"{tab}Literal\n{tab} Value: {self.literal.toString(i)}"

class ReturnNode(INode):

def \_\_init\_\_(self, parent):

super().\_\_init\_\_(parent)

self.retVal: CalculatedValueNode = None

def buildTree(

self,

tokens: list[IStringable],

index: int,

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

) -> int:

i = index + 1

while i < len(tokens) and tokens[i] != separatorsDict[";"]:

i += 1

cv = CalculatedValueNode(self, index + 1, i)

cv.buildTree(tokens, index + 1, type\_table, vars\_table, literals\_table)

self.retVal = cv

return i

def toString(self, i: int):

tab: str = ""

for ind in range(i):

tab += " "

return f"{tab}Return:\n{tab} Return value:\n{self.retVal.toString(i+2)}"

class WhileNode(INode):

def \_\_init\_\_(self, parent):

super().\_\_init\_\_(parent)

self.body: BodyNode = None

self.condition: CalculatedValueNode = None

def buildTree(

self,

tokens: list[IStringable],

index: int,

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

) -> int:

if (

tokens[index + 1] is not bracketsdict["()"]

or index + 1 not in bracketsdict["()"].openClose.keys()

):

raise Exception(

f"Syntax error at token {index+1}: Loop brackets are incorrect"

)

cn = CalculatedValueNode(

self, index + 2, bracketsdict["()"].openClose[index + 1]

)

cn.buildTree(tokens, index + 2, type\_table, vars\_table, literals\_table)

self.condition = cn

ind = bracketsdict["()"].openClose[index + 1] + 1

if tokens[ind] is bracketsdict["{}"]:

body = BodyNode(self, ind, bracketsdict["{}"].openClose[ind])

body.buildTree(tokens, ind, type\_table, vars\_table, literals\_table)

i = bracketsdict["{}"].openClose[ind]

self.body = body

else:

ti = ind

while ti < len(tokens) and tokens[ti] is not separatorsDict[";"]:

ti += 1

body = BodyNode(self, ind - 1, ti + 1)

body.buildTree(tokens, ind - 1, type\_table, vars\_table, literals\_table)

i = ti

self.body = body

return i

def toString(self, i: int):

tab: str = ""

for ind in range(i):

tab += " "

return f"{tab}While Cycle\n{tab} Condition:\n{self.condition.toString(i + 2)}\n{tab} Body:\n{self.body.toString(i+2)}"

class ForNode(INode):

def \_\_init\_\_(self, parent):

super().\_\_init\_\_(parent)

self.declaration: DeclarationNode = None

self.condition: CalculatedValueNode = None

self.action: CalculatedValueNode = None

self.body: BodyNode = None

def buildTree(

self,

tokens: list[IStringable],

index: int,

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

) -> int:

if (

tokens[index + 1] is not bracketsdict["()"]

or index + 1 not in bracketsdict["()"].openClose.keys()

):

raise Exception(

f"Syntax error at token {index+1}: Loop brackets are incorrect"

)

i = index + 2

if tokens[i] is not separatorsDict[";"]:

dn = DeclarationNode(self)

i = dn.buildTree(tokens, i, type\_table, vars\_table, literals\_table)

self.declaration = dn

i += 1

if tokens[i] is not separatorsDict[";"]:

oldI = i

while i < len(tokens) and tokens[i] is not separatorsDict[";"]:

i += 1

cond = CalculatedValueNode(self, oldI, i)

cond.buildTree(tokens, oldI, type\_table, vars\_table, literals\_table)

self.condition = cond

i += 1

if tokens[i] is not bracketsdict["()"]:

oldI = i

i = bracketsdict["()"].openClose[index + 1]

act = CalculatedValueNode(self, oldI, i)

act.buildTree(tokens, oldI, type\_table, vars\_table, literals\_table)

self.action = act

ind = bracketsdict["()"].openClose[index + 1] + 1

if tokens[ind] is bracketsdict["{}"]:

body = BodyNode(self, ind, bracketsdict["{}"].openClose[ind])

body.buildTree(tokens, ind, type\_table, vars\_table, literals\_table)

i = bracketsdict["{}"].openClose[ind]

self.body = body

else:

ti = ind

while ti < len(tokens) and tokens[ti] is not separatorsDict[";"]:

ti += 1

body = BodyNode(self, ind - 1, ti + 1)

body.buildTree(tokens, ind - 1, type\_table, vars\_table, literals\_table)

i = ti

self.body = body

return i

def toString(self, i: int):

tab: str = ""

for ind in range(i):

tab += " "

res = f"{tab}For Cycle\n"

if self.declaration:

res += f"\n{tab} Declaration:\n{self.declaration.toString(i + 2)}"

if self.condition:

res += f"\n{tab} Condition:\n{self.condition.toString(i + 2)}"

if self.action:

res += f"\n{tab} Action:\n{self.action.toString(i + 2)}"

res += f"\n{tab} Body:\n{self.body.toString(i+2)}"

return res

class IfNode(INode):

def \_\_init\_\_(self, parent):

super().\_\_init\_\_(parent)

self.thenBody: BodyNode = None

self.elseBody: BodyNode = None

self.condition: CalculatedValueNode = None

def buildTree(

self,

tokens: list[IStringable],

index: int,

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

) -> int:

print("in if node")

if (

tokens[index + 1] is not bracketsdict["()"]

or index + 1 not in bracketsdict["()"].openClose.keys()

):

raise Exception(

f"Syntax error at token {index+1}: Loop brackets are incorrect"

)

cn = CalculatedValueNode(

self, index + 2, bracketsdict["()"].openClose[index + 1]

)

cn.buildTree(tokens, index + 2, type\_table, vars\_table, literals\_table)

self.condition = cn

ind = bracketsdict["()"].openClose[index + 1] + 1

if tokens[ind] is bracketsdict["{}"]:

body = BodyNode(self, ind, bracketsdict["{}"].openClose[ind])

body.buildTree(tokens, ind, type\_table, vars\_table, literals\_table)

i = bracketsdict["{}"].openClose[ind]

self.thenBody = body

else:

ti = ind

while ti < len(tokens) and tokens[ti] is not separatorsDict[";"]:

ti += 1

body = BodyNode(self, ind - 1, ti + 1)

body.buildTree(tokens, ind - 1, type\_table, vars\_table, literals\_table)

i = ti

self.thenBody = body

if tokens[i + 1] is keywordsDict["else"]:

ind = i + 1

if tokens[ind] is bracketsdict["{}"]:

body = BodyNode(self, ind, bracketsdict["{}"].openClose[ind])

body.buildTree(tokens, ind, type\_table, vars\_table, literals\_table)

i = bracketsdict["{}"].openClose[ind]

self.elseBody = body

else:

ti = ind

while ti < len(tokens) and tokens[ti] is not separatorsDict[";"]:

ti += 1

body = BodyNode(self, ind - 1, ti + 1)

body.buildTree(tokens, ind - 1, type\_table, vars\_table, literals\_table)

i = ti

self.elseBody = body

return i

def toString(self, i: int):

tab: str = ""

for ind in range(i):

tab += " "

res = f"{tab}If statement\n{tab} Condition:\n{self.condition.toString(i + 2)}\n{tab} THEN:\n{self.thenBody.toString(i+2)}"

if self.elseBody is not None:

res += f"\n{tab} ELSE:\n{self.thenBody.toString(i+2)}"

return res

class GetVariableValueNode(INode):

def \_\_init\_\_(self, parent):

super().\_\_init\_\_(parent)

def buildTree(

self,

tokens: list[IStringable],

index: int,

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

) -> int:

scopeStart = tokens[index].scopeStart

if not (

scopeStart < index

and bracketsdict["{}"].openClose.get(scopeStart, -1) > index

or (

bracketsdict["()"].openClose[scopeStart + 1] > index

and tokens[scopeStart] is keywordsDict["for"]

)

):

raise Exception(f"Syntax error at token {index}: Unknown variable")

self.variable = tokens[index]

return index

def toString(self, i: int):

tab: str = ""

for ind in range(i):

tab += " "

return f"{tab}Variable\n{tab} Value: {self.variable.toString(i)}"

# NEED to know its boundaries

class CalculatedValueNode(INode):

def \_\_init\_\_(self, parent, begin: int, end: int):

super().\_\_init\_\_(parent)

self.begin = begin

self.end = end

self.value: INode = None

def buildTree(

self,

tokens: list[IStringable],

index: int,

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

) -> int:

print(f"from calculated begin = {self.begin}")

i = self.begin

currentPriorOperatorIndex = -1

currentPrior = 0

### CHANGE TO BEG END ###

while i < self.end:

if tokens[i] in operators:

print(

f" operating at {self.begin} {self.end} token is operator: {tokens[i].toString(i)} prior: {tokens[i].prior} currentPrior: {currentPrior}"

)

if tokens[i].prior > currentPrior:

currentPriorOperatorIndex = i

currentPrior = tokens[i].prior

elif tokens[i] in vars\_table:

if (

tokens[i + 1] in brackets

and i + 1 in tokens[i + 1].openClose.keys()

):

if tokens[i + 1] is bracketsdict["{}"]:

raise Exception(

f"Syntax error at token {i+1}: incorrect operator"

)

if 2 > currentPrior:

currentPrior = 2

currentPriorOperatorIndex = i

i = tokens[i + 1].openClose[i + 1]

elif tokens[i] in brackets:

print(f"i = {i} index = {index} begin = {self.begin} end = {self.end}")

print(

f"parsing brackets {tokens[i].toString(i)} new i will be {tokens[i].openClose[i]}"

)

if 2 > currentPrior:

currentPrior = 2

currentPriorOperatorIndex = i

i = tokens[i].openClose[i]

i += 1

if currentPrior == 0:

if index + 1 != self.end:

print(

f"index = {index}, end = {self.end} tokens[index] = {tokens[index].toString(index)}"

)

raise Exception(

f"Syntax error at token {i}: Incorrect initial value declaration"

)

if tokens[index] in literals\_table:

vn = GetLiteralValueNode(self)

vn.buildTree(tokens, index, type\_table, vars\_table, literals\_table)

self.value = vn

elif tokens[index] in vars\_table:

vn = GetVariableValueNode(self)

vn.buildTree(tokens, index, type\_table, vars\_table, literals\_table)

self.value = vn

elif tokens[currentPriorOperatorIndex] in operators:

op = find(operators, lambda op: op == tokens[currentPriorOperatorIndex])

if op.argCount == "unar":

uo = UnarOperatorNode(self, op)

i = uo.buildTree(

tokens,

currentPriorOperatorIndex,

type\_table,

vars\_table,

literals\_table,

)

self.value = uo

elif op.argCount == "binar":

bo = BinarOperatorNode(self, op, self.begin, self.end)

i = bo.buildTree(

tokens,

currentPriorOperatorIndex,

type\_table,

vars\_table,

literals\_table,

)

self.value = bo

elif tokens[currentPriorOperatorIndex] in brackets:

br: MyBrackets = tokens[currentPriorOperatorIndex]

cn = CalculatedValueNode(

self,

currentPriorOperatorIndex + 1,

br.openClose[currentPriorOperatorIndex],

)

cn.buildTree(

tokens,

currentPriorOperatorIndex + 1,

type\_table,

vars\_table,

literals\_table,

)

self.value = cn

i = br.openClose[currentPriorOperatorIndex]

else:

print(f"index is {index} currentPriorOpInd is {currentPriorOperatorIndex}")

raise Exception(

f"Syntax error at token {i}: Unkown type to identify operator"

)

return i

def getVariable(self, name: str):

self.parent.getVariable(name)

def toString(self, i: int):

tab: str = ""

for ind in range(i):

tab += " "

print(self)

return f"{tab}CalculatedValue\n{tab} Value: \n{self.value.toString(i+2)}"

class DeclarationNode(INode):

def \_\_init\_\_(self, parent: INode):

super().\_\_init\_\_(parent)

def buildTree(

self,

tokens: list[IStringable],

index: int,

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

) -> int:

i = index

self.varType = tokens[i]

i += 1

if not isinstance(tokens[i], VariableTableCell):

raise Exception(f"Incorrect var declaration syntax at token {i}")

self.var = tokens[i]

self.initialValue: CalculatedValueNode | None = None

i += 1

if tokens[i] is not separatorsDict[";"] and tokens[i] is not operatorsDict["="]:

raise Exception(

f"Syntax Error: incorrect var declaration syntax at token {i}"

)

if tokens[i] is operatorsDict["="]:

oldI = i

while i < len(tokens) and tokens[i] is not separatorsDict[";"]:

i += 1

if i == len(tokens):

raise Exception(f"Syntax arror at token {oldI}: There is no ; operator")

cn = CalculatedValueNode(self, oldI + 1, i)

cn.buildTree(tokens, oldI + 1, type\_table, vars\_table, literals\_table)

return i

if tokens[i] is separatorsDict[";"]:

return i

def getVariable(self, name: str):

return self.parent.getVariable(name)

def execute(self):

pass

def toString(self, i: int):

tab: str = ""

for ind in range(i):

tab += " "

res = f"{tab}Declaration\n{tab} Type: {self.varType.toString(i+1)}\n{tab} var: {self.var.toString(i+1)}"

if self.initialValue:

res += f"\n{tab} InitialValue: {self.initialValue.toString(i + 1)}"

return res

class BodyNode(INode):

def \_\_init\_\_(self, parent: INode, begin: int, end: int):

super().\_\_init\_\_(parent)

self.instructions: list[IStringable] = list()

self.begin = begin

self.end = end

def buildTree(

self,

tokens: list[IStringable],

index: int,

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

) -> int:

i = self.begin + 1

while i < self.end - 1:

if tokens[i] in type\_table:

if tokens[i + 1] not in vars\_table:

raise Exception(f"Expected identifier at token {i+1}")

name = tokens[i + 1]

if tokens[i + 2] == bracketsdict["()"]:

raise Exception(

f"Syntax error at token {i + 2}: you can't define function that is not global"

)

dn = DeclarationNode(self)

i = dn.buildTree(tokens, i, type\_table, vars\_table, literals\_table)

self.instructions.append(dn)

elif tokens[i] is keywordsDict["return"]:

rn = ReturnNode(self)

i = rn.buildTree(tokens, i, type\_table, vars\_table, literals\_table)

self.instructions.append(rn)

elif tokens[i] is keywordsDict["while"]:

wn = WhileNode(self)

i = wn.buildTree(tokens, i, type\_table, vars\_table, literals\_table)

self.instructions.append(wn)

elif tokens[i] is keywordsDict["for"]:

fn = ForNode(self)

i = fn.buildTree(tokens, i, type\_table, vars\_table, literals\_table)

self.instructions.append(fn)

elif tokens[i] is keywordsDict["if"]:

ifn = IfNode(self)

i = ifn.buildTree(tokens, i, type\_table, vars\_table, literals\_table)

self.instructions.append(ifn)

else:

oldI: int = i

while i < len(tokens) and tokens[i] is not separatorsDict[";"]:

i += 1

cn = CalculatedValueNode(self, oldI, i)

cn.buildTree(tokens, oldI, type\_table, vars\_table, literals\_table)

self.instructions.append(cn)

i += 1

return self.end

def toString(self, i: int):

tab: str = ""

for ind in range(i):

tab += " "

res = f"{tab}BodyNode\n{tab} Instructions:"

for instr in self.instructions:

res += f"\n{instr.toString(i + 1)}"

return res

class ArgumentDeclaration(INode):

def \_\_init\_\_(self, parent: INode):

super().\_\_init\_\_(parent)

self.type: MyType = None

self.var: VariableTableCell = Nonе

def buildTree(

self,

tokens: list[IStringable],

index: int,

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

) -> int:

if tokens[index] not in type\_table:

raise Exception(

f"Syntax error at token {index}: declaration of function argument without type")

self.type = tokens[index]

if tokens[index + 1] not in vars\_table:

raise Exception(

f"Syntax error at token {index + 1}: declaration of function argument without name")

self.var = tokens[index + 1]

return index + 1

def toString(self, i: int):

tab: str = ""

for ind in range(i):

tab += " "

return f"{tab}ArgumentDeclaration\n{tab} Type: {self.type.toString(i+1)}\n{tab} Var: {self.var.toString(i+1)}"

class FunctionNode(INode):

def \_\_init\_\_(self, parent: INode):

super().\_\_init\_\_(parent)

self.argsDeclarations: list[ArgumentDeclaration] = list()

self.body: BodyNode = None

self.name: VariableTableCell = None

def buildTree(

self,

tokens: list[IStringable],

index: int,

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

) -> int:

var: VariableTableCell = tokens[index + 1]

self.name = var.name

opArgBracket: MyBrackets = tokens[index + 2]

argEndIndex = opArgBracket.openClose[index + 2]

i = index + 2

while i < argEndIndex and i < argEndIndex - 1:

i += 1

adn = ArgumentDeclaration(self)

i = adn.buildTree(tokens, i, type\_table, vars\_table, literals\_table)

self.argsDeclarations.append(adn)

if (

tokens[i + 1] not in brackets

or tokens[i + 1] is not separatorsDict[","]):

raise Exception(

f"Syntax error at token {i + 1}: incorrect function argument separation")

i += 1

if tokens[argEndIndex + 1] is not bracketsdict["{}"]:

raise Exception(

f"Syntax error at token {argEndIndex + 1}: incorrect function block declaration start"

)

br: MyBrackets = tokens[argEndIndex + 1]

bn = BodyNode(self, argEndIndex + 1, br.openClose[argEndIndex + 1])

functionEnd = bn.buildTree(

tokens, argEndIndex + 1, type\_table, vars\_table, literals\_table

)

self.body = bn

return functionEnd

def toString(self, i: int):

tab: str = ""

for ind in range(i):

tab += " "

res = f"{tab}FunctionNode\n{tab} Args:"

for arg in self.argsDeclarations:

res += f"\n{arg.toString(i + 1)}"

res += f"Body:\n{self.body.toString(i+1)}"

return res

class Program(INode):

def \_\_init\_\_(

self,

tokens: list[IStringable],

type\_table: list[MyType],

vars\_table: list[VariableTableCell],

literals\_table: list[MyLiteral],

):

super().\_\_init\_\_(None)

self.functions = dict[str, FunctionNode]()

self.variables = dict[str, DeclarationNode]()

i: int = 0

while i < len(tokens):

if tokens[i] == keywordsDict["#include"]:

pass

elif tokens[i] in type\_table:

if tokens[i + 1] not in vars\_table:

raise Exception(f"Expected identifier at token {i+1}")

name = tokens[i + 1]

if tokens[i + 2] == bracketsdict["()"]:

fn = FunctionNode(self)

i = fn.buildTree(tokens, i, type\_table, vars\_table, literals\_table)

self.functions[name] = fn

else:

dn = DeclarationNode(self)

i = dn.buildTree(tokens, i, type\_table, vars\_table, literals\_table)

self.variables[name] = dn

else:

raise Exception(

"Error at token i: On;y function declarations, global variables and include directives can be in global scope"

)

i += 1

def toString(self, i: int):

tab: str = ""

for ind in range(i):

tab += " "

res = f"Program\n Global Variables:"

for var in self.variables.items():

res += f"\n{var[1].toString(2)}"

res += f"\n Functions:"

for f in self.functions.items():

res += f"\n{f[1].toString(2)}"

return res