Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы трансляции

ОТЧЁТ

по лабораторной работе

на тему

Синтаксический анализатор

Выполнил

Студент гр. 053501

Волковский О.А.

Проверил

Ассистент кафедры информатики

Гриценко Н.Ю.

Минск 2023

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc131172917)

[2 Краткие теоритические сведения 4](#_Toc131172918)

[3 Примеры работы парсера 5](#_Toc131172919)

[4 Выводы 8](#_Toc131172920)

[ПРИЛОЖЕНИЕ А. Листинг кода 9](#_Toc131172921)

1. **ЦЕЛЬ РАБОТЫ**

Представление грамматических фраз исходной программы выполнить в виде дерева. Реализовать синтаксический анализатор с использованием одного из табличных методов (LL-, LR-метод, метод предшествования и пр.).

1. **КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Фаза синтаксического анализа и построения синтаксического дерева является важным этапом в теории трансляции. Ее задача - проверить синтаксическую правильность входного языка и создать структуру данных, которая представляет его в виде дерева.

Синтаксический анализатор применяет заданную грамматику и создает синтаксическое дерево, проходя по входному коду. Структура дерева отображает иерархию синтаксических конструкций и их подструктур входного языка. Это дает возможность использовать дерево для дальнейшей обработки, такой как оптимизация и генерация исполняемого кода.

Несмотря на то, что существуют и другие методы анализа входного языка, синтаксический анализ и построение синтаксического дерева являются важными этапами в теории трансляции, позволяя проверить синтаксическую корректность и создать структуру данных для дальнейшей обработки. Синтаксическое дерево является ключевым инструментом для анализа и оптимизации кода.

1. **ПРИМЕРЫ РАБОТЫ ПАРСЕРА**

Рассмотрим следующую программу и построенное на её основе синтаксическое дерево (см. рисунок 1):

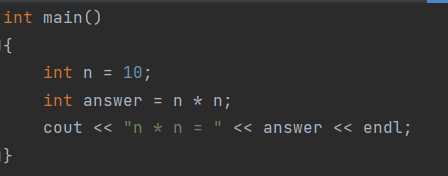


Рисунок 1 – Пример программы

Синтаксическое дерево выглядит следующим образом (см. рисунок 2-3):

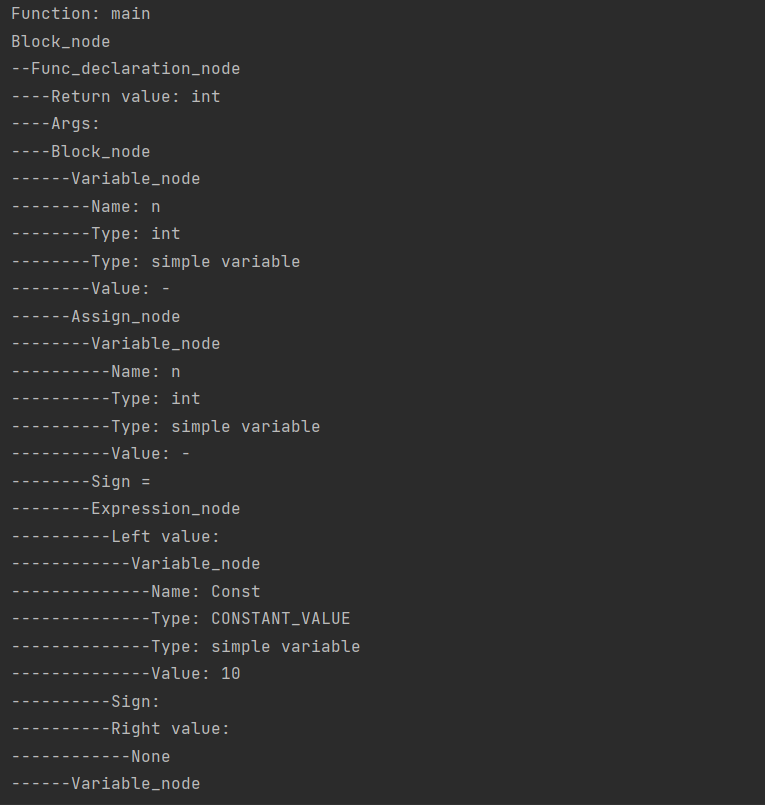


Рисунок 2 – Построенное синтаксическое дерево

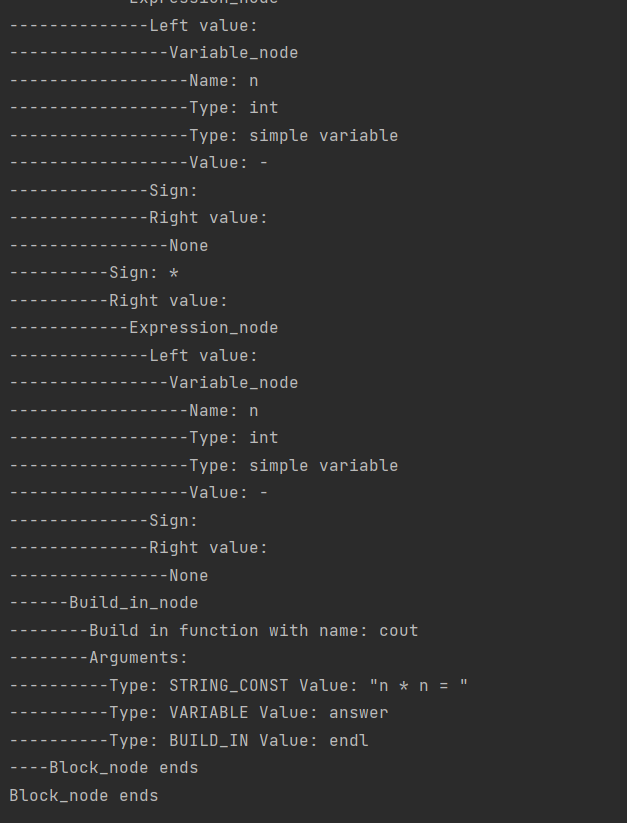


Рисунок 3 – Построенное синтаксическое дерево

Корнем дерева является узел «Block\_node». В нём содержится список утверждений. Утверждение – некоторая логически выполнимая единица. Существуют как специализированные утверждения, например, для ветвлений, циклов, так и утверждения, просто содержащие в себе выражения. Выражение – логическая единица языка, возвращающая некоторое значение при выполнении. Однако в ходе работы парсера необходимо обрабатывать синтаксические ошибки. Добавим, например, второе определение переменной n, в этом случае парсер отловит ошибку (см. рисунок 4).

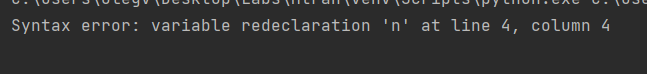


Рисунок 4 – Пример вывода ошибки со вторым определением переменной

Добавим, например, определение переменной, но без имени (double = 0.12) в этом случае парсер отловит ошибку (см. рисунок 5).



Рисунок 5 – Пример ошибки с некорректным определением переменной

Если строка не будет заканчиваться точкой с запятой (int n = 10), то парсер отловит ошибку (см. рисунок 6).



Рисунок 6 – Пример забытой точки с запятой

Обернем тело функции в примере в блок else. В этом случае парсер выдаст ошибку, т.к. нет предшествующего блока if (см. рисунок 7).



Рисунок 7 – Пример неверной конструкции if-else

Также парсер будет отлавливать ошибки при построении некорректных конструкций таких как: циклы, различные вычисления, сравнения, присвоения и др.

1. **ВЫВОДЫ**

Таким образом, в ходе лабораторной работы был реализован парсер подмножества языка С++, который позволяет анализировать написанный на этом языке код.

Было произведено исследование синтаксиса языка С++, чтобы определить основные конструкции и операторы, которые необходимо реализовать в парсере. В результате был разработан алгоритм, основанный на методе рекурсивного спуска, который позволяет обрабатывать и преобразовывать синтаксические конструкции языка в структуры данных.

Реализация парсера (см. приложение А) включала в себя создание синтаксического анализатора, который использует токены для построения дерева разбора. В процессе работы над парсером было выявлено множество ошибок и неточностей, которые требовали доработки и исправления кода.

Было проведено тестирование парсера на наборе тестовых данных, включающих различные конструкции языка С++. Результаты тестирования показали, что парсер корректно обрабатывает входные данные и выводит ожидаемые результаты.

**ПРИЛОЖЕНИЕ А**

**Листинг кода**

def parse\_statement(parent\_node, statement, variable\_table, cur\_line):

new\_node = build\_expression\_tree(statement, variable\_table, cur\_line)

if new\_node is not None:

new\_node.parent = parent\_node

return new\_node

def build\_expression\_tree(expression, variable\_table, cur\_line):

if not expression:

return None

min\_operator = None

parentheses\_count = 0

maximum\_parentheses = 100

for i in range(len(expression) - 1, -1, -1):

if expression[i].value == ')':

parentheses\_count += 1

elif expression[i].value == '(':

parentheses\_count -= 1

elif parentheses\_count < maximum\_parentheses:

if expression[i].token\_type == "ARITHMETIC\_OPERATION":

min\_precedence = get\_priority(expression[i].value)

min\_operator = i

maximum\_parentheses = parentheses\_count

elif parentheses\_count == maximum\_parentheses:

if expression[i].token\_type == "ARITHMETIC\_OPERATION" and get\_priority(expression[i].value) < min\_precedence:

min\_precedence = get\_priority(expression[i].value)

min\_operator = i

if min\_operator is not None:

root = ExpressionNode("Expression\_node", expression[min\_operator].value)

root.right\_value = build\_expression\_tree(expression[min\_operator + 1:], variable\_table, cur\_line)

root.left\_value = build\_expression\_tree(expression[:min\_operator], variable\_table, cur\_line)

if root.left\_value is not None and root.right\_value is not None:

if root.left\_value.expression\_result\_type != "Const" and root.right\_value.expression\_result\_type != "Const":

if root.left\_value.expression\_result\_type != root.right\_value.expression\_result\_type:

print(

f"Semantic error: the types of l\_value and r\_value are not equal '{expression}' at line {expression[0].line} column {expression[0].column}")

sys.exit()

root.expression\_result\_type = root.left\_value.expression\_result\_type

else:

root = ExpressionNode("Expression\_node", "")

root.left\_value = VariableNode("Variable\_node")

if expression[0].value == "(":

expression = expression[1:]

if expression[0].token\_type == "VARIABLE":

if expression[0].value not in variable\_table:

print(

f"Semantic error: undefined variable '{expression[0].value}' at line {expression[0].line} column {expression[0].column}")

sys.exit()

root.left\_value = copy.deepcopy(variable\_table[expression[0].value])

if len(expression) > 1 and expression[1].value == "[":

index = 1

statement = []

while expression[index].value != "]":

statement.append(expression[index])

index += 1

root.left\_value.array\_index = parse\_statement(root.left\_value.array\_index, statement, variable\_table, cur\_line)

root.expression\_result\_type = root.left\_value.variable\_type

else:

root.left\_value.variable\_type = expression[0].token\_type

root.left\_value.cur\_value = expression[0].value

root.left\_value.variable\_name = "Const"

root.expression\_result\_type = root.left\_value.variable\_name

return root

def get\_priority(operator):

if operator in ['+', '-', '+=', '-=']:

return 1

elif operator in ['\*', '/', '/=', '\*=']:

return 2

else:

return float('inf')

def parse\_compare\_statement(parent\_node, statement, variable\_table, cur\_line):

new\_node = CompareNode("Compare\_node")

operator\_pos = 0

for i in range(len(statement)):

if statement[i].token\_type == "COMPARISON\_SIGN":

operator\_pos = i

new\_node.compare\_sign = statement[operator\_pos].value

new\_node.left\_value = parse\_statement(new\_node, statement[:operator\_pos], variable\_table, cur\_line)

new\_node.right\_value = parse\_statement(new\_node, statement[operator\_pos + 1:], variable\_table, cur\_line)

new\_node.parent = parent\_node

return new\_node

def print\_ast(node, indent):

if node is None:

print(f"{indent \* '-'}None")

return

print(f"{indent \* '-'}{node.node\_type}")

if node.node\_type == "Assign\_node":

print\_ast(node.left\_value, indent + 2)

print(f"{(indent + 2) \* '-'}Sign {node.arithmetic\_sign}")

print\_ast(node.right\_value, indent + 2)

if node.node\_type == "Build\_in\_node":

print(f"{(indent + 2) \* '-'}Build in function with name: {node.function\_name}")

print(f"{(indent + 2) \* '-'}Arguments:")

for arg in node.arguments:

print(f"{(indent + 4) \* '-'}Type: {arg.token\_type} Value: {arg.value}")

if node.node\_type == "Compare\_node":

print(f"{(indent + 2) \* '-'}Left value:")

print\_ast(node.left\_value, indent + 4)

print(f"{(indent + 2) \* '-'}Sign: {node.compare\_sign}")

print(f"{(indent + 2) \* '-'}Right value:")

print\_ast(node.right\_value, indent + 4)

if node.node\_type == "Expression\_node":

print(f"{(indent + 2) \* '-'}Left value:")

print\_ast(node.left\_value, indent + 4)

print(f"{(indent + 2) \* '-'}Sign: {node.operator}")

print(f"{(indent + 2) \* '-'}Right value:")

print\_ast(node.right\_value, indent + 4)

if node.node\_type == "For\_node":

print(f"{(indent + 2) \* '-'}Variable args:")

print\_ast(node.variable\_node, indent + 4)

print(f"{(indent + 2) \* '-'}Starting value:")

print\_ast(node.start\_value\_node, indent + 4)

print(f"{(indent + 2) \* '-'}Exit condition:")

print\_ast(node.compare\_node, indent + 4)

print(f"{(indent + 2) \* '-'}Args changes:")

print\_ast(node.expression\_node, indent + 4)

if node.node\_type == "Func\_declaration\_node":

print(f"{(indent + 2) \* '-'}Return value: {node.return\_value}")

print(f"{(indent + 2) \* '-'}Args:")

for item in node.arguments:

print\_ast(item, indent + 4)

if node.node\_type == "Func\_node":

print(f"{(indent + 2) \* '-'}Function name: {node.function\_name}")

print(f"{(indent + 2) \* '-'}Input args:")

for item in node.arguments:

print\_ast(item, indent + 4)

if node.node\_type == "If\_node":

print(f"{(indent + 2) \* '-'}if condition:")

if node.condition is None:

print(f"{(indent + 2) \* '-'}None")

else:

print\_ast(node.condition, indent + 4)

if node.node\_type == "Variable\_node":

print(f"{(indent + 2) \* '-'}Name: {node.variable\_name}")

print(f"{(indent + 2) \* '-'}Type: {node.variable\_type}")

if node.is\_array:

print(f"{(indent + 2) \* '-'}Type: array ")

print(f"{(indent + 2) \* '-'}Values: {node.cur\_array}")

else:

print(f"{(indent + 2) \* '-'}Type: simple variable")

print(f"{(indent + 2) \* '-'}Value: {node.cur\_value}")

return node.variable\_type

for child in node.children:

print\_ast(child, indent + 2)

if node.node\_type == "Block\_node":

print(f"{indent \* '-'}{node.node\_type} ends")