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

Семантический анализатор.

Выполнил

Студент гр. 053501

Волковский О.А.

Проверил

Ассистент кафедры информатики

Гриценко Н.Ю.

Минск 2023

СОДЕРЖАНИЕ

[1. Цель работы 3](#_Toc128743288)

[2. Краткие теоретические сведения 4](#_Toc128743289)

[3. Примеры работы парсера 5](#_Toc128743290)

[4. Выводы 6](#_Toc128743291)

ПРИЛОЖЕНИЕ А. Листинг кода 7

1. **ЦЕЛЬ РАБОТЫ**

Создать семантический анализатор для реализации возможности интерпретации программы на выбранном языке. Необходимо показать скриншоты нахождения 2-х семантических ошибок.

1. **КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Семантический анализ является одним из основных этапов теории трансляции. Он представляет собой процесс проверки исходного кода на наличие семантических ошибок, которые не могут быть обнаружены на уровне лексического и синтаксического анализа.

Фаза контроля типов проверяет, удовлетворяет ли программа контекстным условиям. Главной составляющей контекстных условия является правильное использование программой типов данных, предоставляемых входным языком, т.е. корректность выражений, встречающихся в программе, с точки зрения использования типов.

Идентификация идентификаторов – одна из задач, решение которой необходимо для проверки правильности использования типов. Понятно, что мы не можем убедиться в правильности использования типов в какой-нибудь конструкции до тех пор, пока не определим типы всех ее составных частей. Например, для того, чтобы выяснить правильность оператора присваивания мы должны знать типы его получателя (левой части) и источника (правой части). Для того, чтобы выяснить, каков тип идентификатора, являющегося, например, получателем присваивания, мы должны понять, каким образом этот идентификатор был объявлен в программе.

Каждое вхождение идентификатора в программу является либо определяющим, либо использующим. Под определяющим вхождением идентификатора понимается его вхождение в описание, например, int i. Все остальные вхождения являются использующими, например, i = 5 или i+13.

Цель идентификации идентификаторов – определить тип использующего вхождения идентификатора.

1. **ПРИМЕРЫ РАБОТЫ ПАРСЕРА**

Рассмотрим следующую программу, для которой было построено синтаксическое дерево в предыдущей лабораторной (см. рисунок 1).

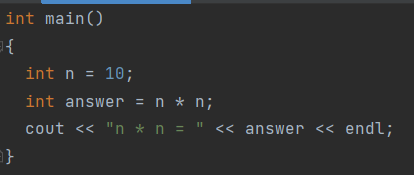


Рисунок 1 – Пример программы

Добавим, например, в выражение переменную, которая не определена (int answer = l \* n), в этом случае парсер отловит ошибку (см. рисунок 2).



Рисунок 2 – Пример вывода ошибки c неопределенной переменной

Добавим, например, определение переменной, но имя переменной будет является константным значением, в этом случае парсер отловит ошибку (см. рисунок 3).



Рисунок 3 – Пример ошибки с некорректным левым значением выражения

Если добавим определение новой переменной m(string m = “1”) и попробуем изменить логику answer(int answer = n \* m;), то парсер отловит ошибку несовместимости типов (см. рисунок 4).

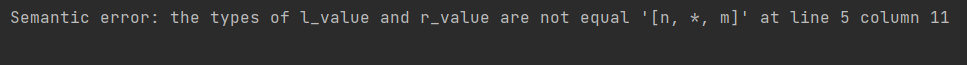


Рисунок 4 – Пример несовместимости типов

1. **ВЫВОДЫ**

Таким образом, реализация семантического анализатора из теории трансляции позволяет производить проверку исходного кода на наличие семантических ошибок. Это важный шаг в процессе компиляции, который помогает обнаружить ошибки, которые могут привести к неправильной работе программы или ее аварийному завершению.

Семантический анализатор выполняет несколько задач, включая проверку соответствия типов данных в операциях, обнаружение необъявленных переменных и функций в исходном коде и проверку правильного их использования.

Семантический анализатор реализован на основании синтаксического дерева (см. приложение А).

В процессе работы анализатор может выявлять различные ошибки, такие как неправильное использование операторов и функций, приведение несовместимых типов переменных. Использование семантического анализатора позволяет повысить качество и надежность программного обеспечения, ускорить процесс разработки и снизить затраты на отладку и исправление ошибок.

**ПРИЛОЖЕНИЕ А**

**Листинг кода**

def build\_tree(tokens):

root = Node("Block\_node")

cur\_node = root

variable\_table = {}

index = 0

while index < len(tokens):

cur\_line = tokens[index].line

if tokens[index].token\_type == "DATA\_TYPE":

if tokens[index + 1].token\_type == "FUNCTION":

new\_node = FuncDeclarationNode("Func\_declaration\_node")

new\_node.return\_value = tokens[index].value

new\_node.parent = cur\_node

index += 2

if tokens[index].value != "(":

print(

f"Syntax error: unexpected character '{tokens[index].value}' at line {tokens[index].line}, column {tokens[index].column}")

sys.exit()

index += 1

while tokens[index].value != ")":

if tokens[index].token\_type != "DATA\_TYPE":

print(

f"Syntax error: unexpected character '{tokens[index].value}' at line {tokens[index].line}, column {tokens[index].column}")

sys.exit()

if tokens[index + 1].token\_type != "VARIABLE":

print(

f"Semantic error: l\_value could be non constant '{tokens[index + 1].value}' at line {tokens[index + 1].line}, column {tokens[index + 1].column}")

sys.exit()

if tokens[index + 1].value in variable\_table:

print(

f"Syntax error: variable redeclaration '{tokens[index + 1].value}' at line {tokens[index + 1].line}, column {tokens[index + 1].column}")

sys.exit()

var\_node = VariableNode("Variable\_node")

var\_node.variable\_type = tokens[index].value

var\_node.variable\_name = tokens[index + 1].value

new\_node.arguments.append(var\_node)

variable\_table[tokens[index + 1].value] = var\_node

index += 2

if tokens[index].value == ')':

break

index += 1

cur\_node.children.append(new\_node)

index += 1

cur\_node = new\_node

elif tokens[index + 1].token\_type == "VARIABLE":

if tokens[index + 1].value in variable\_table:

print(

f"Syntax error: variable redeclaration '{tokens[index + 1].value}' at line {tokens[index + 1].line}, column {tokens[index + 1].column}")

sys.exit()

new\_node = VariableNode("Variable\_node")

new\_node.variable\_type = tokens[index].value

new\_node.variable\_name = tokens[index + 1].value

if tokens[index + 2].value == "[":

new\_node.is\_array = True

new\_node.parent = cur\_node

variable\_table[tokens[index + 1].value] = new\_node

cur\_node.children.append(new\_node)

index += 1

else:

if tokens[index + 1].token\_type == "ARITHMETIC\_OPERATION":

print(

f"Syntax error: expected variable but found '{tokens[index + 1].value}' at line {tokens[index + 1].line}, column {tokens[index + 1].column}")

sys.exit()

else:

print(

f"Semantic error: l\_value could be non constant '{tokens[index + 1].value}' at line {tokens[index + 1].line}, column {tokens[index + 1].column}")

sys.exit()

elif tokens[index].value == "for":

new\_node = ForNode("For\_node", VariableNode("Variable\_node"), ExpressionNode("Expression\_node", ""),

CompareNode("Compare\_node"), ExpressionNode("Expression\_node", ""))

index += 2

new\_node.token = tokens[index]

if tokens[index].token\_type != "DATA\_TYPE":

print(

f"Syntax error: unexpected character '{tokens[index].value}' at line {tokens[index].line}, column {tokens[index].column}")

sys.exit()

if tokens[index + 1].token\_type != "VARIABLE":

print(

f"Semantic error: l\_value could be non constant '{tokens[index + 1].value}' at line {tokens[index + 1].line}, column {tokens[index + 1].column}")

sys.exit()

new\_node.variable\_node.variable\_type = tokens[index].value

new\_node.variable\_node.variable\_name = tokens[index + 1].value

variable\_table[tokens[index + 1].value] = new\_node.variable\_node

index += 2

if tokens[index].value != "=":

print(

f"Syntax error: unexpected character '{tokens[index].value}' at line {tokens[index].line}, column {tokens[index].column}")

sys.exit()

index += 1

statement = []

while tokens[index].value != ";":

statement.append(tokens[index])

index += 1

index += 1

new\_node.start\_value\_node = parse\_statement(new\_node.start\_value\_node, statement, variable\_table, cur\_line)

statement = []

while tokens[index].value != ";":

statement.append(tokens[index])

index += 1

index += 1

new\_node.compare\_node = parse\_compare\_statement(new\_node.compare\_node, statement, variable\_table, cur\_line)

statement = []

while tokens[index].value != ")":

statement.append(tokens[index])

index += 1

new\_node.expression\_node = parse\_statement(new\_node.expression\_node, statement, variable\_table, cur\_line)

index += 1

new\_node.parent = new\_node

cur\_node.children.append(new\_node)

cur\_node = new\_node

elif tokens[index].value == "[":

index += 1

statement = []

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

statement.append(tokens[index])

index += 1

new\_node = parse\_statement(cur\_node, statement, variable\_table, cur\_line)

cur\_node.children[-1].array\_index = new\_node

new\_node.parent = cur\_node.children[-1]

index += 1

elif tokens[index].token\_type == "VARIABLE":

new\_node = AssignNode("Assign\_node")

if tokens[index].value not in variable\_table:

print(

f"Semantic error: undefined variable '{tokens[index].value}' at line {tokens[index].line}, column {tokens[index].column}")

sys.exit()

new\_node.left\_value = copy.deepcopy(variable\_table[tokens[index].value])

index += 1

if tokens[index].value == "[":

index += 1

statement = []

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

statement.append(tokens[index])

index += 1

new\_node.left\_value.array\_index = parse\_statement(new\_node.left\_value.array\_index, statement,

variable\_table, cur\_line)

index += 1

if tokens[index].token\_type == "ARITHMETIC\_OPERATION":

new\_node.arithmetic\_sign = tokens[index].value

index += 1

statement = []

while tokens[index - 1].line == tokens[index].line and tokens[index].value != ";":

statement.append(tokens[index])

index += 1

if tokens[index].value != ";":

print(

f"Syntax error: expected ; but found '{tokens[index].value}' at line {tokens[index].line}, column {tokens[index].column}")

sys.exit()

new\_node.right\_value = parse\_statement(new\_node, statement, variable\_table, cur\_line)

new\_node.parent = cur\_node

cur\_node.children.append(new\_node)

index += 1

elif tokens[index].value == "if":

new\_node = IfNode("If\_node")

index += 2

statement = []

while tokens[index].value != ")":

statement.append(tokens[index])

index += 1

new\_node.condition = parse\_compare\_statement(new\_node, statement, variable\_table, cur\_line)

index += 1

new\_node.parent = cur\_node

cur\_node.children.append(new\_node)

elif tokens[index].value == "else":

if len(cur\_node.children) == 0 or (cur\_node.children[-1].node\_type != "If\_node" and (len(cur\_node.children) > 1 and cur\_node.children[-2].node\_type != "If\_node")):

print(

f"Syntax error: expected if before but found '{tokens[index].value}' at line {tokens[index].line}, column {tokens[index].column}")

sys.exit()

new\_node = IfNode("If\_node")

index += 1

if tokens[index].value == "if":

statement = []

while tokens[index].value != ")":

statement.append(tokens[index])

index += 1

new\_node.condition = parse\_compare\_statement(new\_node, statement, variable\_table, cur\_line)

index += 1

new\_node.parent = cur\_node

cur\_node.children.append(new\_node)

elif tokens[index].value == "cout":

new\_node = BuildInNode("Build\_in\_node")

new\_node.function\_name = "cout"

index += 1

while tokens[index].value != ";":

index += 1

new\_node.arguments.append(tokens[index])

index += 1

index += 1

new\_node.parent = cur\_node

cur\_node.children.append(new\_node)

elif tokens[index].value == "break":

new\_node = BuildInNode("Build\_in\_node")

new\_node.function\_name = "break"

index += 2

new\_node.parent = cur\_node

cur\_node.children.append(new\_node)

elif tokens[index].value == "continue":

new\_node = BuildInNode("Build\_in\_node")

new\_node.function\_name = "continue"

index += 2

new\_node.parent = cur\_node

cur\_node.children.append(new\_node)

elif tokens[index].value == "return":

new\_node = BuildInNode("Build\_in\_node")

new\_node.function\_name = "return"

index += 1

new\_node.arguments.append(tokens[index])

index += 2

new\_node.parent = cur\_node

cur\_node.children.append(new\_node)

elif tokens[index].token\_type == "FUNCTION":

new\_node = FuncNode("Func\_node")

new\_node.function\_name = tokens[index].value

index += 2

while tokens[index].value != ")":

var\_node = copy.deepcopy(variable\_table[tokens[index].value])

var\_node.children = []

new\_node.arguments.append(var\_node)

index += 1

index += 2

new\_node.parent = cur\_node

cur\_node.children.append(new\_node)

elif tokens[index].value == "{":

new\_node = Node("Block\_node")

new\_node.parent = cur\_node

cur\_node.children.append(new\_node)

cur\_node = new\_node

index += 1

elif tokens[index].value == "}":

cur\_node = cur\_node.parent

index += 1

elif tokens[index].value == ";":

if cur\_node.node\_type == "Variable\_node" or cur\_node.node\_type == "Func\_node":

cur\_node = cur\_node.parent

else:

print(

f"Syntax error: unexpected character '{tokens[index].value}' at line {tokens[index].line}, column {tokens[index].column}")

sys.exit()

else:

print(

f"Syntax error: unexpected character '{tokens[index].value}' at line {tokens[index].line}, column {tokens[index].column}")

sys.exit()

return root