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

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

2 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.

3 ПРИМЕРЫ РАБОТЫ ПАРСЕРА

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

```
int main()
{
  int n = 10;
  int answer = n * n;
  cout << "n * n = " << answer << endl;
}
```

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).

```
Semantic error: undefined variable 'l' at line 4
```

Рисунок 2 – Пример вывода ошибки с неопределенной переменной

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

```
Semantic error: l_value could be constant '1' at line 4, column 4
```

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).

```
Semantic error: the types of l_value and r_value are not equal '[n, *, m]' at line 5 column 11
```

Рисунок 4 – Пример несовместимости типов

4 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
```