

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
2 Краткие теоритические сведения	4
3 Примеры работы парсера	5
4 Выводы	8
ПРИЛОЖЕНИЕ А. Листинг кода	9

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.).

2 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.

3 ПРИМЕРЫ РАБОТЫ ПАРСЕРА

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

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

Рисунок 1 – Пример программы

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

```
Function: main
Block_node
--Func_declaration_node
----Return value: int
----Args:
----Block_node
-----Variable_node
-----Name: n
-----Type: int
-----Type: simple variable
-----Value: -
-----Assign_node
-----Variable_node
-----Name: n
-----Type: int
-----Type: simple variable
-----Value: -
-----Sign =
-----Expression_node
-----Left value:
-----Variable_node
-----Name: Const
-----Type: CONSTANT_VALUE
-----Type: simple variable
-----Value: 10
-----Sign:
-----Right value:
-----None
-----Variable_node
```

Рисунок 2 – Построенное синтаксическое дерево

```

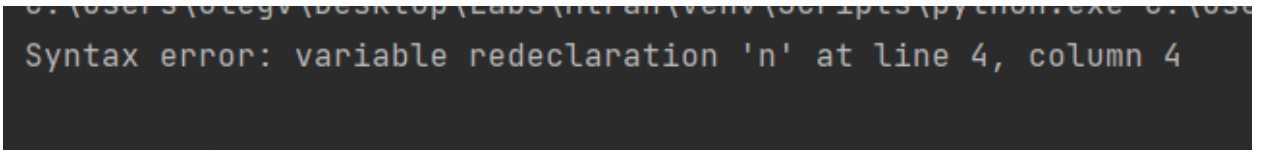
Expression_node
-----Left value:
-----Variable_node
-----Name: n
-----Type: int
-----Type: simple variable
-----Value: -
-----Sign:
-----Right value:
-----None
-----Sign: *
-----Right value:
-----Expression_node
-----Left value:
-----Variable_node
-----Name: n
-----Type: int
-----Type: simple variable
-----Value: -
-----Sign:
-----Right value:
-----None
-----Build_in_node
-----Build in function with name: cout
-----Arguments:
-----Type: STRING_CONST Value: "n * n = "
-----Type: VARIABLE Value: answer
-----Type: BUILD_IN Value: endl
----Block_node ends
Block_node ends

```

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

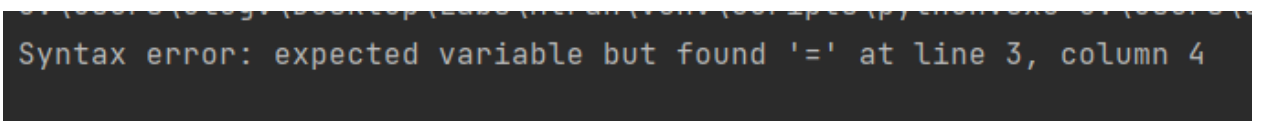
синтаксические ошибки. Добавим, например, второе определение переменной `n`, в этом случае парсер отловит ошибку (см. рисунок 4).



```
0. (0.0013 (0.0094 (desktop (2405 (int (int (0.0113 (python.exe 0. (0.0013  
Syntax error: variable redeclaration 'n' at line 4, column 4
```

Рисунок 4 – Пример вывода ошибки со вторым определением переменной

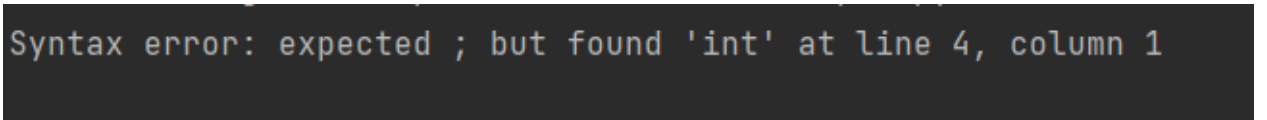
Добавим, например, определение переменной, но без имени (`double = 0.12`) в этом случае парсер отловит ошибку (см. рисунок 5).



```
0. (0.0013 (0.0094 (desktop (2405 (int (int (0.0113 (python.exe 0. (0.0013  
Syntax error: expected variable but found '=' at line 3, column 4
```

Рисунок 5 – Пример ошибки с некорректным определением переменной

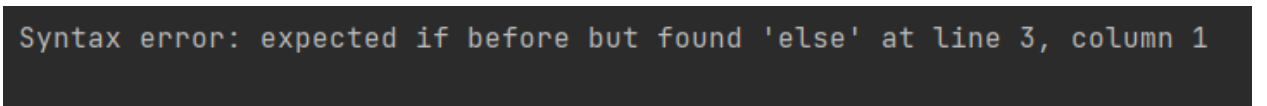
Если строка не будет заканчиваться точкой с запятой (`int n = 10`), то парсер отловит ошибку (см. рисунок 6).



```
Syntax error: expected ; but found 'int' at line 4, column 1
```

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).



```
Syntax error: expected if before but found 'else' at line 3, column 1
```

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.

4 ВЫВОДЫ

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

Было произведено исследование синтаксиса языка C++, 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 C++. 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")

```