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 АНАЛИЗ**

Студент В. П. Бычко

Преподаватель Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

[1 Постановка задачи 3](#_Toc157960226)

[2 Краткие теоритические сведения 4](#_Toc157960235)

[3 Результаты выполнения лабораторной работы](#_Toc157960235) 6

[Выводы](#_Toc157960236) 9

[Список использованных источников](#_Toc157960237) 10

[Приложение А (обязательное) Листинг програмного кода](#_Toc157960238) 11

## 1 ПОСТАНОВКА ЗАДАЧИ

Целью данной лабораторной работы является создание синтаксического анализатора для потока токенов, полученного с помощью лексического анализатора из лабораторной работы 2. Целью также является обнаружение некорректных ситуаций, возникающих при обработке неверной последовательности токенов.

# 2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Синтаксический анализ представляет собой критически важный второй шаг в многоступенчатом процессе разработки компилятора для программных языков. На этой стадии происходит детальная проверка входной последовательности символов (строки кода), которая осуществляется с целью установить, насколько корректно она соответствует заранее установленным правилам и структурам, прописанным в формальной грамматике программного языка. Данный этап предусматривает тщательный разбор синтаксической структуры входной строки, а также оценку соответствия этой строки синтаксическим нормам выбранного языка программирования.

После завершения первоначального этапа, известного как лексический анализ, начинается процесс синтаксического анализа. В результате успешного синтаксического анализа формируется так называемое дерево разбора или синтаксическое дерево, которое строится на базе грамматики, определенной для данного языка программирования. Синтаксический анализатор выполняет функцию проверки кода программы на соответствие правилам, которые предполагаются в контекстно-свободной грамматике. Если входные данные удовлетворяют заданным требованиям, то синтаксический анализатор приступает к созданию соответствующего дерева разбора для исходного кода программы. В случае обнаружения несоответствий или ошибок, анализатор обязан сообщить о них разработчику.

Существуют различные методики и подходы к синтаксическому анализу, которые можно классифицировать в две основные категории: методы нисходящего анализа и методы восходящего анализа. Подходы нисходящего анализа начинают процесс построения дерева разбора с верхушки дерева (корня) и двигаются вниз к листьям. Среди методов нисходящего анализа выделяют два основных типа: прогнозирующий анализ и рекурсивный анализ спуска.

Прогнозирующий анализ обладает способностью предугадывать, какое правило грамматики (продукцию) следует применить для обработки определенной входной строки в контексте синтаксического разбора. Этот тип анализа использует механизм, известный как "точка просмотра вперед", что позволяет анализатору учитывать следующие символы во входной строке, чтобы принимать решения без необходимости возврата. Прогнозирующий анализатор также называется парсером LL(1), где "LL" обозначает сканирование входной строки слева направо и построение вывода слева направо с использованием одного символа вперед для предсказания. Рекурсивный анализ спуска же представляет собой метод, при котором входная строка обрабатывается рекурсивно для построения дерева фраз. Он включает в себ применяется серия малых функций, каждая из которых отвечает за разбор определенного нетерминального символа грамматики. [1]

Методы восходящего анализа, напротив, начинают собирать дерево разбора с листьев, постепенно продвигаясь к корню. Эти методы часто используются в современных компиляторах и реализуются с помощью различных программных инструментов, таких как генераторы синтаксических анализаторов, которые автоматизируют и упрощают процесс создания анализаторов для разработчиков компиляторов. [2]

# 3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

Во время выполнения лабораторной работы был реализовал синтаксический анализатор языка С. Скриншот результатов представлен на рисунке 3.1.

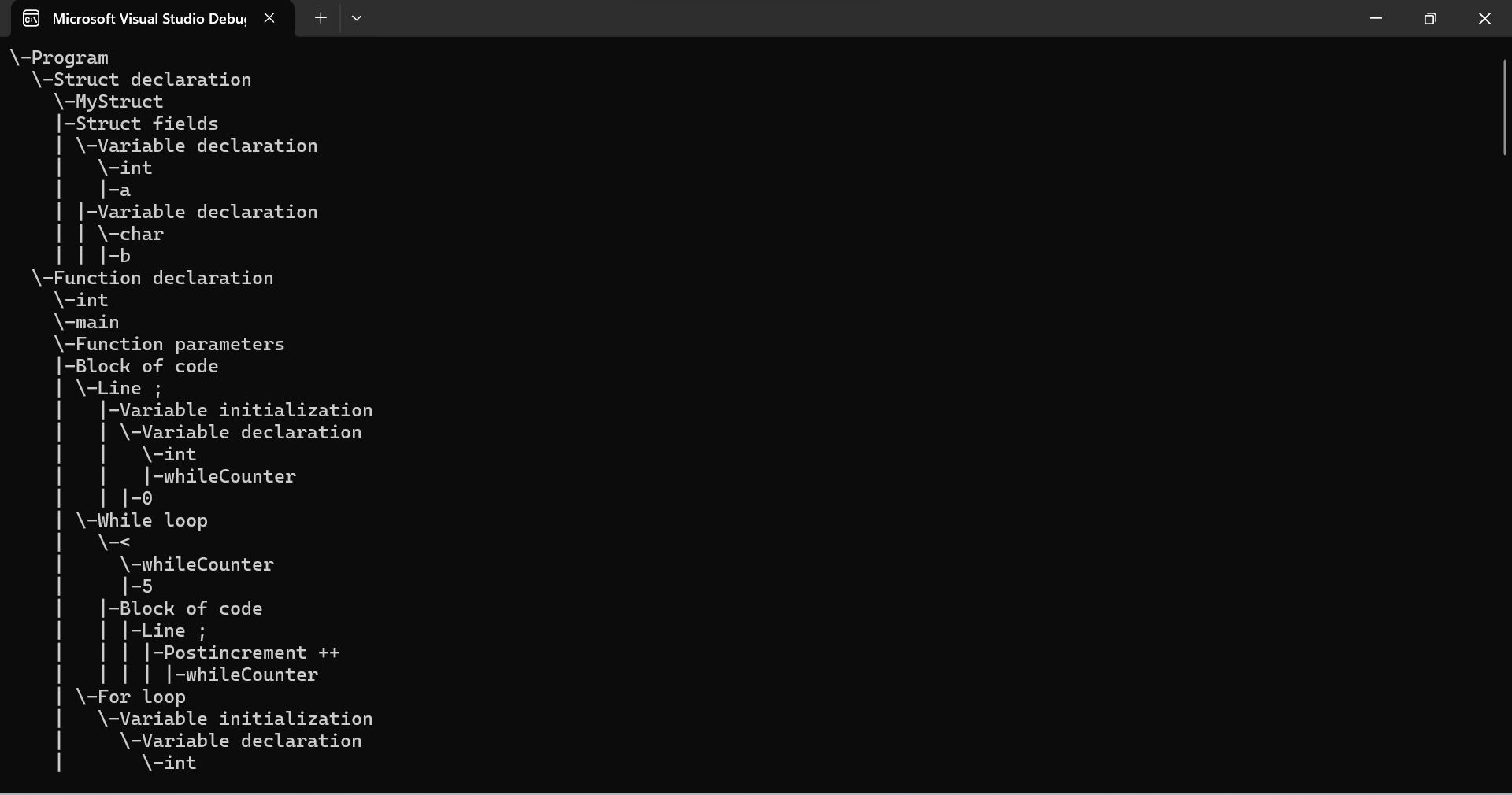


Рисунок 3.1 – Итог работы синтаксического анализатора

Изначальное содержание файла с анализируемым программным кодом представлено на рисунке 3.2.

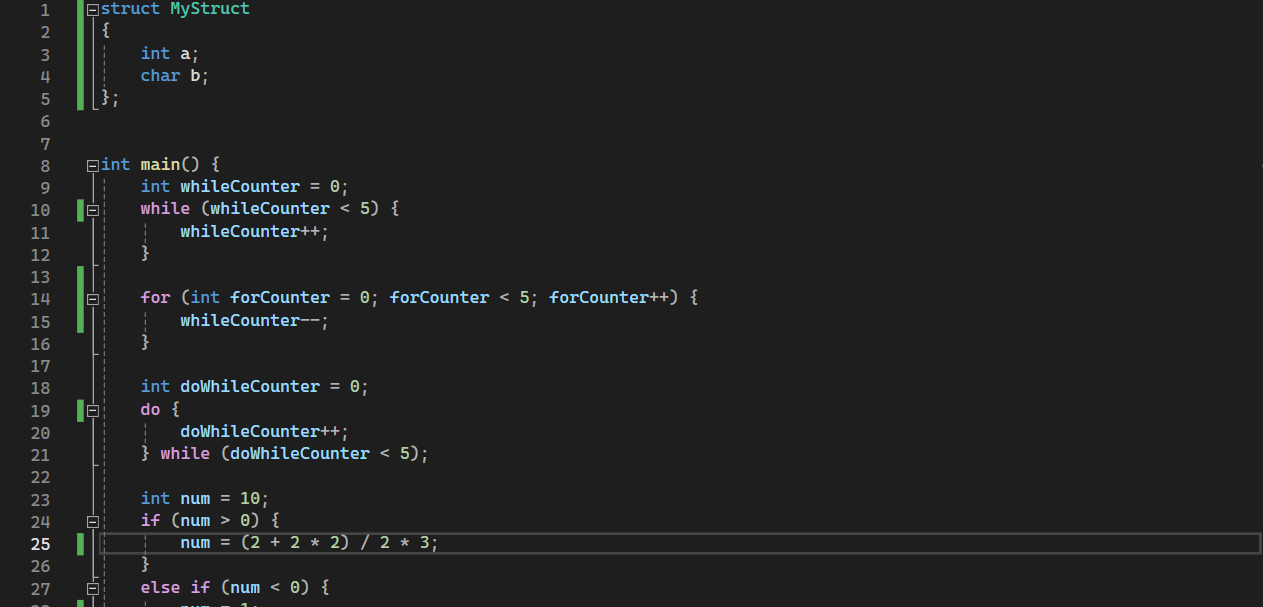


Рисунок 3.2 – Изначальное содержание текстового файла

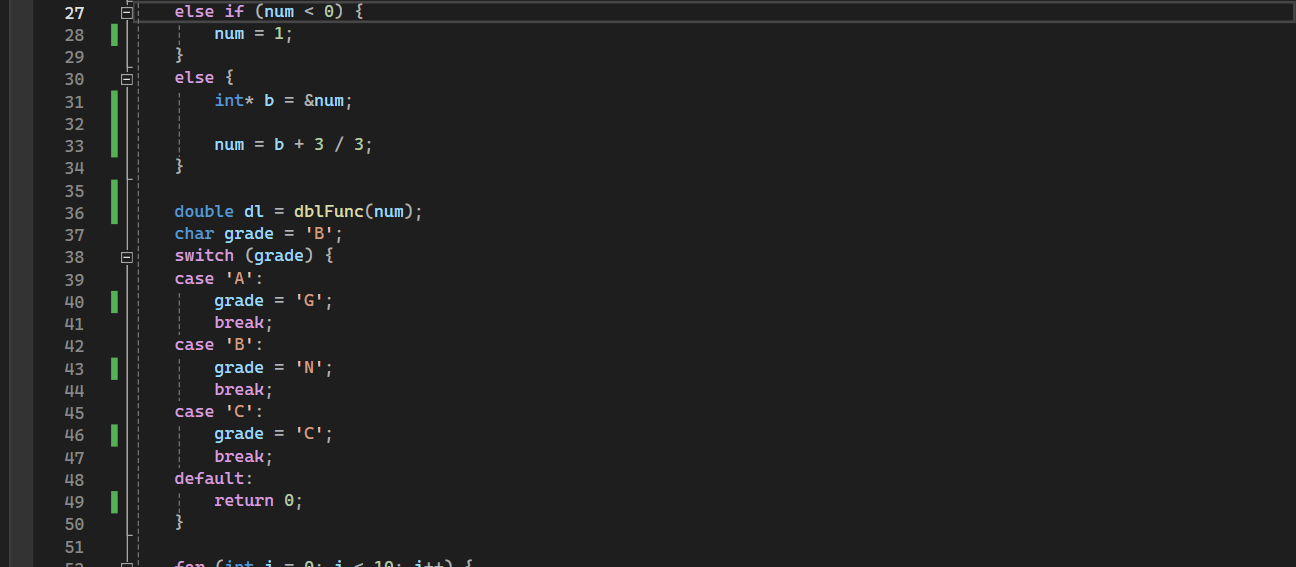


Рисунок 3.2 – Лист 2

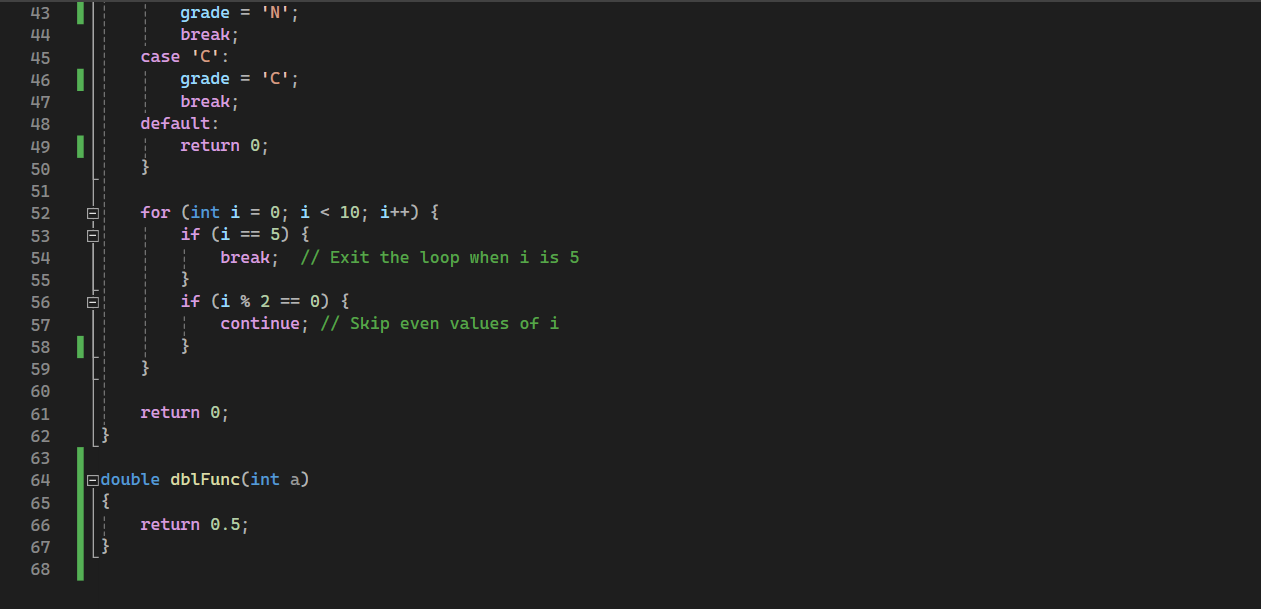


Рисунок 3.3 – Лист 3

## ВЫВОДЫ

В ходе выполнения лабораторной работы по написанию синтаксического анализатора были изучены основные принципы синтаксического анализа, алгоритмы разбора и структуры данных, необходимые для построения дерева разбора. Был реализован синтаксический анализатор, способный проверять правильность порядка лексем в соответствии с грамматикой языка.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Синтаксический анализ [Электронный ресурс]. – Режим доступа:

https://edu.vsu.ru/mod/resource/view.php?id=25354– Дата доступа: 05.03.2024

1. Введение в теорию компиляторов [Электронный ресурс]. – Режим доступа: https://habr.com/ru/articles/515420/ – Дата доступа: 05.03.2024

**ПРИЛОЖЕНИЕ А**

**(обязательное)**

**Листинг программного кода**

private abstract class Grammar

{

public abstract Node? Parse(List<Token> tokens, ref int i);

public Grammar Or(Grammar right)

{

if (this is null)

throw new ArgumentNullException("left");

if (right is null)

throw new ArgumentNullException("right");

return new DelegateGrammar(ParseOr);

Node? ParseOr(List<Token> tokens, ref int i)

{

int initial = i;

if (this.Parse(tokens, ref i) is Node leftNode)

return leftNode;

i = initial;

if (right.Parse(tokens, ref i) is Node rightNode)

return rightNode;

i = initial;

return null;

}

}

public GrammarRow Then(Grammar other, int count = 1)

{

ArgumentOutOfRangeException

.ThrowIfLessThanOrEqual(count, 0, nameof(count));

return new GrammarRow() { (this, count: 1, null), (other, count, null) };

}

public class GrammarRow

: List<(Grammar, int count, string? errorMessage)>

{

public GrammarRow Then(Grammar grammar, int count = 1)

{

ArgumentOutOfRangeException

.ThrowIfLessThanOrEqual(count, 0, nameof(count));

Add((grammar, count, null));

return this;

}

public GrammarRow WithError(string errorMessage)

{

this[^1] = this[^1] with { errorMessage = errorMessage };

return this;

}

public Grammar AsNode(Func<List<Node>, Node> merge)

{

return new DelegateGrammar(ParseAll);

Node? ParseAll(List<Token> tokens, ref int i)

{

int next = i;

List<Node> nodes = [];

foreach (var (grammar, count, errorMessage) in this)

{

for (int j = 0; j < count; j++)

{

if (grammar.Parse(tokens, ref next) is not Node node)

{

if (errorMessage is not null)

{

throw new SyntaxException(errorMessage);

}

else

{

return null;

}

}

else

{

nodes.Add(node);

}

}

}

i = next;

return merge(nodes);

}

}

}

public Grammar ThenNested(Grammar other, Func<Node, Node, Node> merge, bool canEmpty = false)

{

return new DelegateGrammar(ParseNested);

Node? ParseNested(List<Token> tokens, ref int i)

{

int initial = i;

LinkedList<Node> nodes = [];

if (this.Parse(tokens, ref i) is not Node startNode)

{

i = initial;

return null;

}

else

{

nodes.AddLast(startNode);

}

int next = i;

while (true)

{

if (other.Parse(tokens, ref next) is Node nextNode)

{

nodes.AddLast(nextNode);

i = next;

}

else

{

break;

}

}

int minNodesCount = canEmpty ? 1 : 2;

if (nodes.Count < minNodesCount)

{

i = initial;

return null;

}

if (nodes.Count == 1 && canEmpty)

return nodes.First!.Value;

return nodes.Aggregate(merge);

}

}

#region Language grammar

public static LazyGrammar ProgramGrammar => new(() =>

{

return

new ListGrammar(StructDeclarationGrammar

.Or(FunctionDeclarationGrammar)

.Or(FunctionPrototypeGrammar)

.Or(VariableCreatingGrammar

.Then(new TokenGrammar(TokenType.Punctuator, ";")).WithError("; missed after variable creating.")

.AsNode(nodes => nodes[0])),

"Program",

separator: null,

canEmpty: true);

});

public static LazyGrammar StructDeclarationGrammar => new(() =>

{

return

new TokenGrammar(TokenType.Type, "struct")

.Then(IdentifierGrammar)

.Then(new BlockGrammar("{", "}", new ListGrammar(VariableDeclarationGrammar

.Then(new TokenGrammar(TokenType.Punctuator, ";")).WithError("Invalid struct field declaration: ';' is missed.")

.AsNode(nodes => nodes[0]), "Struct fields", separator: null, canEmpty: true), "Struct fields"))

.Then(new TokenGrammar(TokenType.Punctuator, ";")).WithError("Invalid structure declaration: ';' is missed.")

.AsNode(nodes => new BinaryOperatorNode(nodes[1], ((OperatorNode)nodes[2]).Children.ToArray()[0], "Struct declaration"));

});

public static LazyGrammar FunctionDeclarationGrammar => new(() =>

{

return

TypeGrammar

.Then(IdentifierGrammar)

.Then(new TokenGrammar(TokenType.Punctuator, "("))

.Then(new ListGrammar(VariableDeclarationGrammar, "Function parameters", canEmpty: true))

.Then(new TokenGrammar(TokenType.Punctuator, ")")).WithError("Invalid function declaration: ')' is missed.")

.Then(ScopeGrammar)

.AsNode(nodes => new OperatorNode([nodes[0], nodes[1], nodes[3], nodes[5]], "Function declaration"));

});

public static LazyGrammar FunctionPrototypeGrammar => new(() =>

{

return

TypeGrammar

.Then(IdentifierGrammar)

.Then(new TokenGrammar(TokenType.Punctuator, "("))

.Then(new ListGrammar(VariableDeclarationGrammar, "Function parameters", canEmpty: true))

.Then(new TokenGrammar(TokenType.Punctuator, ")")).WithError("Invalid function declaration: ')' is missed.")

.Then(new TokenGrammar(TokenType.Punctuator, ";"))

.AsNode(nodes => new OperatorNode([nodes[0], nodes[1], nodes[3]], "Function prototype"));

});

}