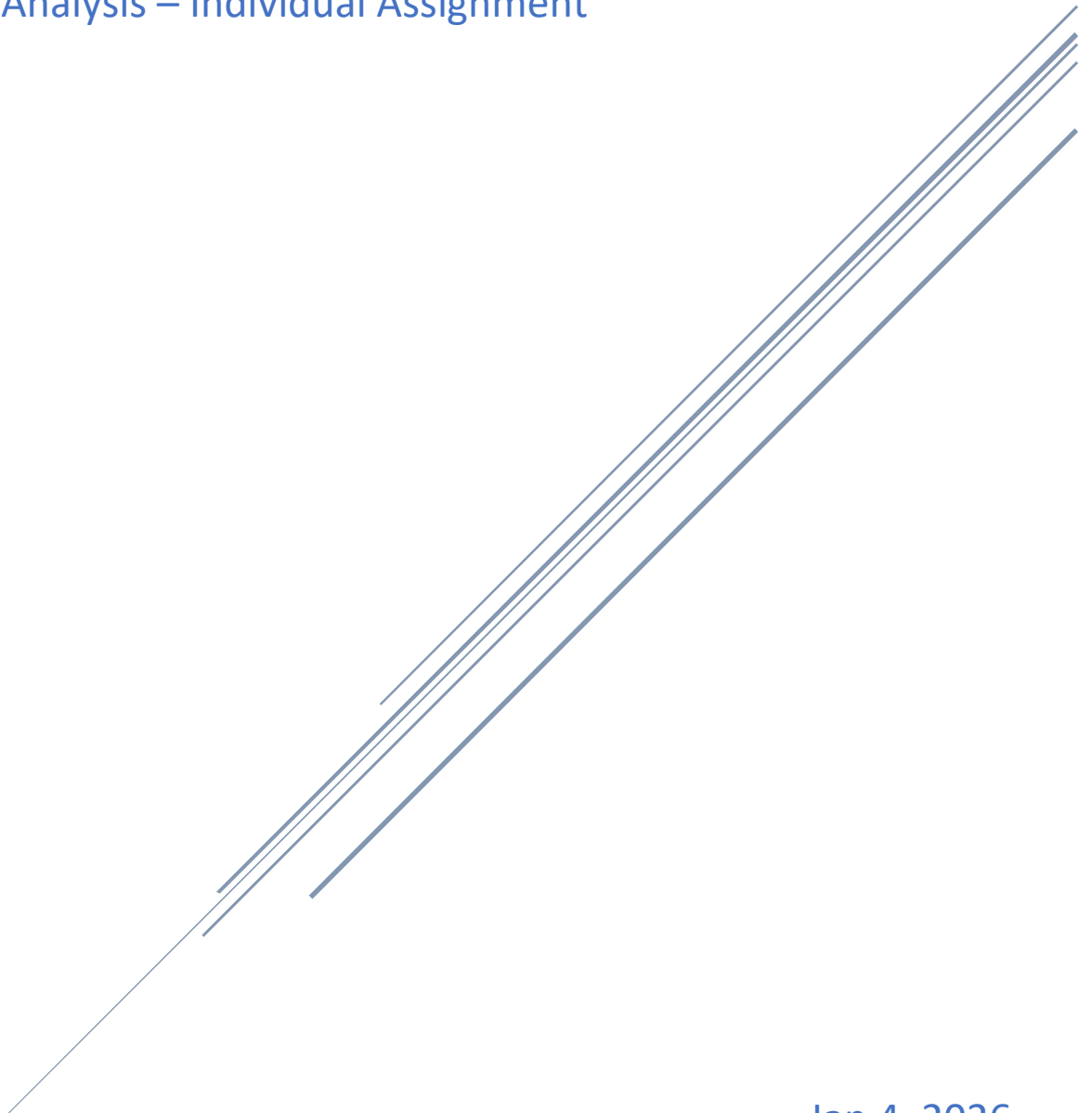




**Bahir Dar University**  
Institute of Technology /BiT

# Principles of Compiler Design

Syntax Analysis – Individual Assignment



Jan 4, 2026

Yonatan Ayisheshim [BDU1508377]

## Table of Contents

1. Introduction.....	2
2. Theory - Difference between Parse Tree and Abstract Syntax Tree (AST) .....	2
2.1 Parse Tree.....	2
2.2 Abstract Syntax Tree (AST).....	2
2.3 Key Differences .....	3
3. C++ Implementation - Checking Balanced Square Brackets [ ] .....	3
3.1 Problem Statement .....	3
3.2 Approach .....	3
3.3 C++ Program.....	3
3.4 Example Execution .....	5
4. Problem Solving.....	5
4.1 Grammar Analysis and Parse Trees .....	5
4.2 Parse Trees .....	5
4.3 Observation .....	6
5. Conclusion .....	7
6. References.....	7

# 1. Introduction

Syntax analysis is a critical phase of compiler design that verifies whether a sequence of tokens generated by the lexical analyzer conforms to the grammatical structure of the programming language. This assignment focuses on understanding syntactic structures through theory, practical C++ implementation, and grammar-based problem solving.

## 2. Theory - Difference between Parse Tree and Abstract Syntax Tree (AST)

### 2.1 Parse Tree

A **parse tree** (also known as a concrete syntax tree) is a hierarchical representation of the syntactic structure of a string derived according to the grammar rules of a language.

#### Characteristics:

- Represents **every grammar symbol** (terminals and non-terminals)
- Closely follows the **context-free grammar (CFG)**
- Useful for **syntax verification and teaching**
- Often **large and verbose**

**Example:** For grammar rule,

```
E → E + T
```

The parse tree explicitly shows  $E$ ,  $+$ , and  $T$  as nodes.

### 2.2 Abstract Syntax Tree (AST)

An **abstract syntax tree (AST)** is a condensed, semantic-oriented representation of the source code.

#### Characteristics:

- Omits unnecessary grammar symbols (parentheses, punctuation)
- Captures **essential hierarchical structure**
- More **compact and efficient**
- Used in **semantic analysis, optimization, and code generation**

**Example:** The expression,

a + b

In an AST is represented simply as:



## 2.3 Key Differences

Aspect	Parse Tree	Abstract Syntax Tree (AST)
Grammar dependency	Strictly grammar-based	Grammar-independent
Size	Large	Compact
Purpose	Syntax validation	Semantic analysis
Includes terminals	Yes	No
Used in	Parsing phase	Later compiler phases

## 3. C++ Implementation - Checking Balanced Square Brackets [ ]

### 3.1 Problem Statement

Write a C++ program that checks whether a given string contains **balanced square brackets** ( [ and ] ).

### 3.2 Approach

A **stack-based approach** is used:

- Push ' [ ' when encountered
- Pop when ' ] ' is encountered
- If a closing bracket appears with an empty stack → unbalanced
- Stack must be empty at the end for the string to be balanced

### 3.3 C++ Program

```

#include <iostream>
#include <stack>
#include <string>

using namespace std;

bool isBalanced(const string&
input) {
    stack<char> st;

    for (char ch : input) {
        if (ch == '[') {
            st.push(ch);
        } else if (ch == ']') {
            if (st.empty()) {
                return false;
            }
            st.pop();
        }
    }
    return st.empty();
}

```

```

int main() {
    string input;
    cout << "Enter a string: ";
    cin >> input;

    if (isBalanced(input)) {
        cout << "The string has balanced
square brackets." << endl;
    } else {
        cout << "The string does NOT have
balanced square brackets." << endl;
    }

    return 0;
}

```

### 3.4 Example Execution

Input	Output
[[]]	Balanced
[] [[]]	Balanced
[]]	Not Balanced
[[]]	Not Balanced

## 4. Problem Solving

### 4.1 Grammar Analysis and Parse Trees

Given Grammar:  $S \rightarrow aS \mid bS \mid \varepsilon$

Where:

- Terminals:  $\{a, b\}$
- Non-terminal:  $s$
- Start symbol:  $s$
- $\varepsilon$  denotes the empty string

**Derivations:**

1. **String "aa":**  $S \rightarrow aS \rightarrow aaS \rightarrow aa\varepsilon = aa$
2. **String "ab":**  $S \rightarrow aS \rightarrow abS \rightarrow ab\varepsilon = ab$
3. **String "ba":**  $S \rightarrow bS \rightarrow baS \rightarrow ba\varepsilon = ba$
4. **String "bb":**  $S \rightarrow bS \rightarrow bbS \rightarrow bb\varepsilon = bb$

*The list of strings of length 2 is: aa, ab, ba, bb.*

### 4.2 Parse Trees

Below are the textual representations of the parse trees for each generated string.

<p><b>A. Parse Tree for "aa"</b></p> <pre> S / \ a  S    / \   a  S            ε </pre>	<p><b>B. Parse Tree for "ab"</b></p> <pre> S / \ a  S    / \   b  S            ε </pre>
<p><b>C. Parse Tree for "ba"</b></p> <pre> S / \ b  S    / \   a  S            ε </pre>	<p><b>D. Parse Tree for "bb"</b></p> <pre> S / \ b  S    / \   b  S            ε </pre>

### 4.3 Observation

The grammar is **right-recursive** and generates **all possible strings** of a and b, including the empty string. Each character corresponds to a recursive expansion of s until termination via  $\epsilon$ .

## 5. Conclusion

This assignment demonstrated:

- The conceptual difference between **parse trees and ASTs**
- Practical application of **stack-based parsing** in C++
- Grammar-driven string generation and parse tree construction

Understanding these concepts is essential for mastering compiler front-end design and lays the foundation for advanced topics such as semantic analysis and code generation.

## 6. References

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. *Compilers: Principles, Techniques, and Tools* (2nd ed.)
- Compiler Design Lecture Notes