Bahir Dar University Department of software Engineering

Course title: principle of compiler design

Name	**Id**
Hermela Tesfaye	**1506530**

Submitted to Lec. Wendmu

Submission date 27/04/2018

1. Difference between Parse Tree and Abstract Syntax Tree (AST)

1.1 Parse Tree

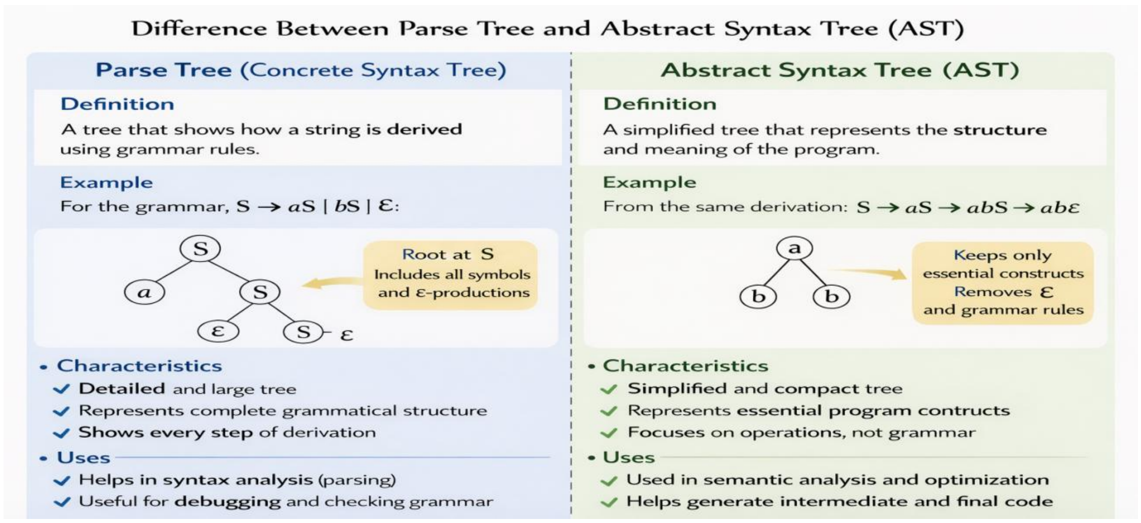
A parse tree is a tree representation that illustrates the syntactic structure of a string according to a given context-free grammar. It shows how the start symbol of the grammar derives the input string step by step using production rules. Used to verify whether the input string conforms to the grammar rules of the language.Characteristics:

- ✓ Includes all grammar symbols (both terminals and non-terminals)
- ✓ Closely follows the grammar productions
- ✓ Represents the complete syntactic structure of the input
- ✓ Mainly used during the syntax analysis phase of a compiler

1.2 Abstract Syntax Tree (AST)

An Abstract Syntax Tree (AST) is a condensed and simplified version of the parse tree. It represents the logical structure of a program while ignoring unnecessary syntactic details such as punctuation symbols and intermediate non-terminals.it focuses on Program semantics rather than grammar.Characteristics:

- ✓ Contains only essential language constructs
- ✓ Removes redundant grammar symbols
- ✓ More compact and easier to process
- ✓ Used in later compiler phases such as semantic analysis and code generation


Difference Between Parse Tree and Abstract Syntax Tree (AST)

Parse Tree (Concrete Syntax Tree)

Definition

A tree that shows how a string **is derived** using grammar rules.

Example

For the grammar, S → *a*S | *b*S | ε:

S
a
S
ε
S
ε
Root at S
Includes all symbols
and ε-productions

- **Characteristics**
  - ✓ **Detailed** and large tree
  - ✓ Represents complete grammatical structure
  - ✓ **Shows every step** of derivation
- **Uses**
  - ✓ Helps in **syntax analysis** (parsing)
  - ✓ Useful for **debugging** and checking grammar

Abstract Syntax Tree (AST)

Definition

A simplified tree that represents the **structure** and meaning of the program.

Example

From the same derivation: S → *a*S → *ab*S → *ab*ε

- **Characteristics**
  - ✓ **Simplified** and **compact tree**
  - ✓ Represents **essential program contructs**
  - ✓ Focuses on operations, not grammar
- **Uses**
  - ✓ **Used in semantic analysis and optimization**
  - ✓ **Helps generate intermediate and final code**

2.Checking Balanced Curly Braces in C++

In programming, balanced braces are crucial for proper execution of code blocks. This program checks whether a given string has balanced curly braces '{}' using the stack data structure.Balanced braces mean every opening brace '{' has a corresponding closing brace '}' in correct order.A stack is used because it follows LIFO (Last-1In, First-Out), which is perfect for tracking nested structures

Code

```
#include <iostream>
#include <stack>
using namespace std;

bool isBalanced(string s) {
    stack<char> st;

    for (char c : s) {
        if (c == '{') {
            st.push(c);
        } else if (c == '}') {
            if (st.empty())
                return false;
            st.pop();
        }
    }
    return st.empty();
}

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

    if (isBalanced(s))
        cout << "Balanced Curly Braces";
    else
        cout << "Not Balanced Curly Braces";

    return 0;
}
```

How the Code Works

1. The program takes a string input from the user.
2. Each character is checked:
   - If '{' → push it onto the stack.
   - If '}' → check if the stack is empty:
     - Empty → braces are not balanced → return false.
     - Not empty → pop the top element (matches with '{').
3. After iterating through the string:
   - If stack is empty → Balanced
   - If stack is not empty → Not Balanced

Example Outputs

Input	Output
{}	Balanced
{{}}	Balanced

{{}	Not Balanced
}{	Not Balanced

This program efficiently checks balanced curly braces using a stack, which is helpful in compilers, parsers, and code validation tools.

3.Given the grammar:

S → aS | bS | ε

This grammar generates **all strings over {a, b}**, including the empty string.

All strings of length 3

All possible strings of length 3 using **a** and **b** are:

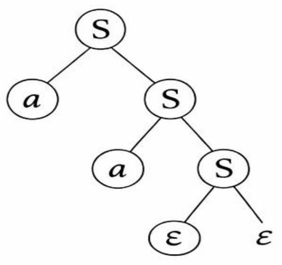
1. aaa
2. aab
3. aba
4. abb

5.baa
6.bab
7.bba
8.bbb

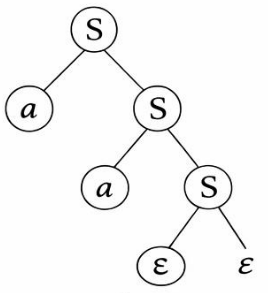
General idea of the parse trees

✓ Each **a** or **b** comes from applying S → aS or S → bS
✓ The derivation **must end with** S → ε
✓ For length 3, the tree will have **3 terminals** and **1 ε at the end**

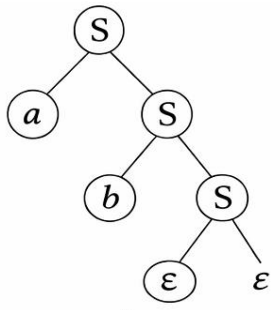
Parse Trees (exam-style) Below are **clear and simple parse trees**.
All trees have the same structure; only terminals change.



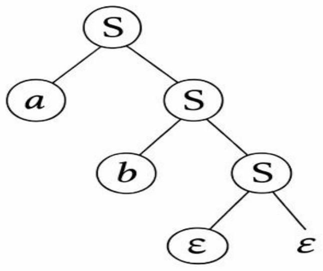
aaa



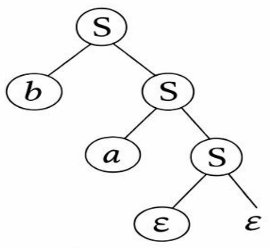
aab



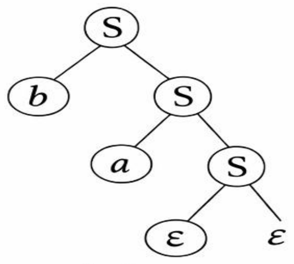
aba



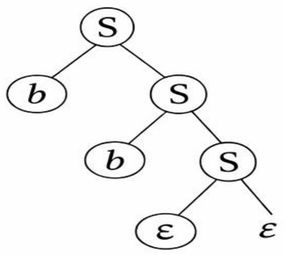
abb



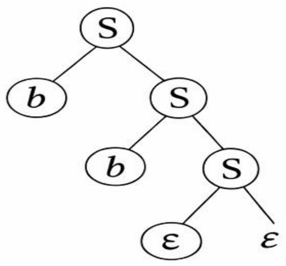
baa



bab



bba



bbb