Compiler Construction
Lab Manual # 03

**Department of Computer Science & Engineering,
University of Engineering & Technology Lahore,
Narowal Campus.**

Submitted by

2022-CS-529

Syed Subtain Ali

Section-A

Submitted to

Ma'am Iqra Muneer

Date: October 6, 2025

Table of Contents

What is Parser

A parser is a component of a compiler that takes input in the form of a sequence of tokens (from the lexical analyser) and produces a parse tree (or abstract syntax tree).

It checks whether the given program is syntactically correct according to the grammar of the programming language.

Usage in Compiler Construction

Purpose	**Explanation (Simple)**	**Example**
1. Syntax Checking	Ensures that the source code follows the correct grammatical rules.	Detects errors like missing semicolons, wrong nesting, etc.
2. Converts Tokens into Structure	Takes tokens (from lexical analyzer) and organizes them into a structured form.	Tokens like int, a, =, 5, ; become a tree structure.
3. Builds Parse Tree / Syntax Tree	Creates a tree that represents how statements are structured.	Useful for later stages like semantic analysis or code generation.
4. Error Reporting & Recovery	Reports where syntax errors occur and sometimes tries to continue parsing.	Example: "Syntax Error at line 3: unexpected 'else'"
5. Guides Semantic Analysis	Helps the next phase check meaning (types, scope, etc.) by providing structure.	Example: verifies if a variable was declared before use.

What is Parser Tree?

A Parse Tree (also called Syntax Tree) is a tree-shaped structure that shows how the grammar rules of a language are applied to form a valid program.

Example

For expression:

a + b * c

Parse Tree:

```
     E

   / | \

  E  +  T
```

```
|   / \

a   T * F

    |   |

    b   c
```

It shows the **hierarchical structure** — that b * c happens before adding a.

Usage of Parser Tree?

Use	Explanation
1. Represents program structure	Shows how program elements fit together according to grammar rules.
2. Used for semantic checking	The compiler uses this structure to check meaning (type, scope, etc.).
3. Helps in intermediate code generation	Provides the base to generate intermediate or machine code.
4. Debugging and visualization	Helpful to understand or debug how a compiler interprets code.

Real-Life Applications of Parser & Parse Tree

1. Compiler Design

- Converts programming code into machine-understandable form.
- Used in languages like C, Java, and Python compilers.
- Helps in syntax checking and code generation.

2. Web Browsers

- HTML, CSS, and JavaScript parsers interpret and render web pages.
- Parse trees help browsers understand page structure and layout.

3. Natural Language Processing (NLP)

- Parses human language sentences to understand grammar and meaning.
- Used in chatbots, translation systems, and voice assistants.

4. Data Processing

- Parses structured data formats like XML, JSON, and CSV.
- Enables data extraction, validation, and transformation in software systems.

5. Game Development

- Used in scripting engines to parse in-game scripts or rules.
- Helps interpret custom logic written by game designers.

6. Security & Threat Detection

- Log parsers analyze system or network logs.
- Helps detect suspicious patterns or intrusion attempts.

7. Software Configuration

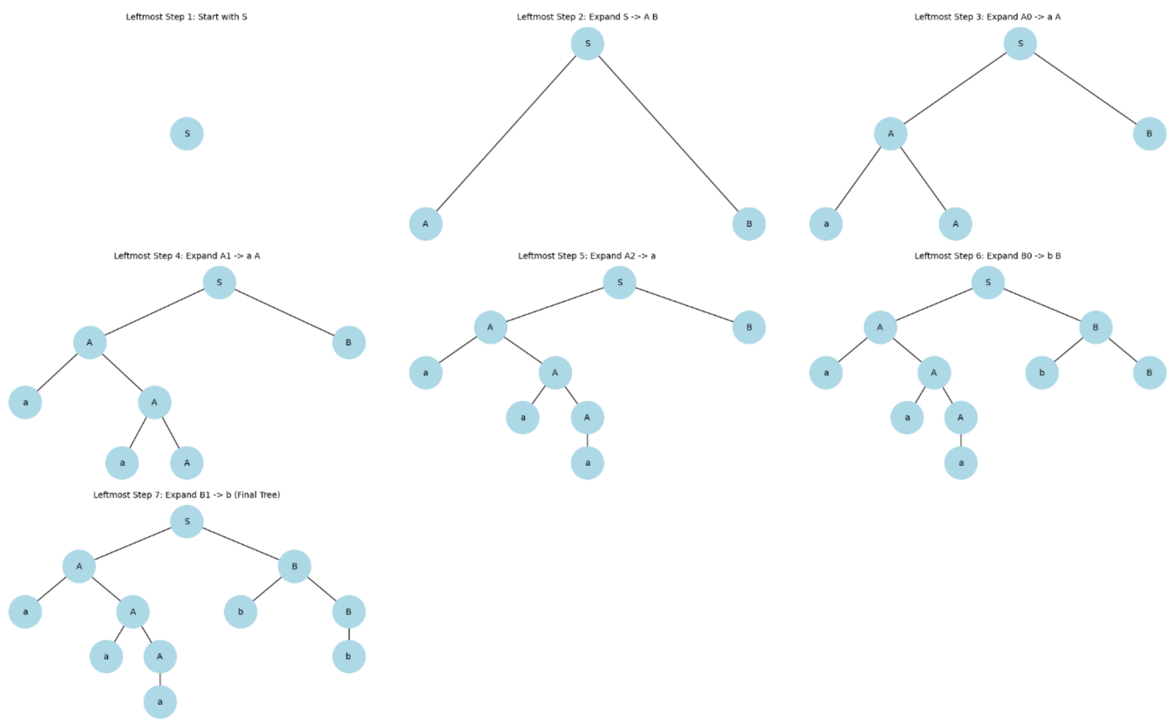
- Parses configuration files (e.g., .ini, .yaml, .json) to load settings automatically.

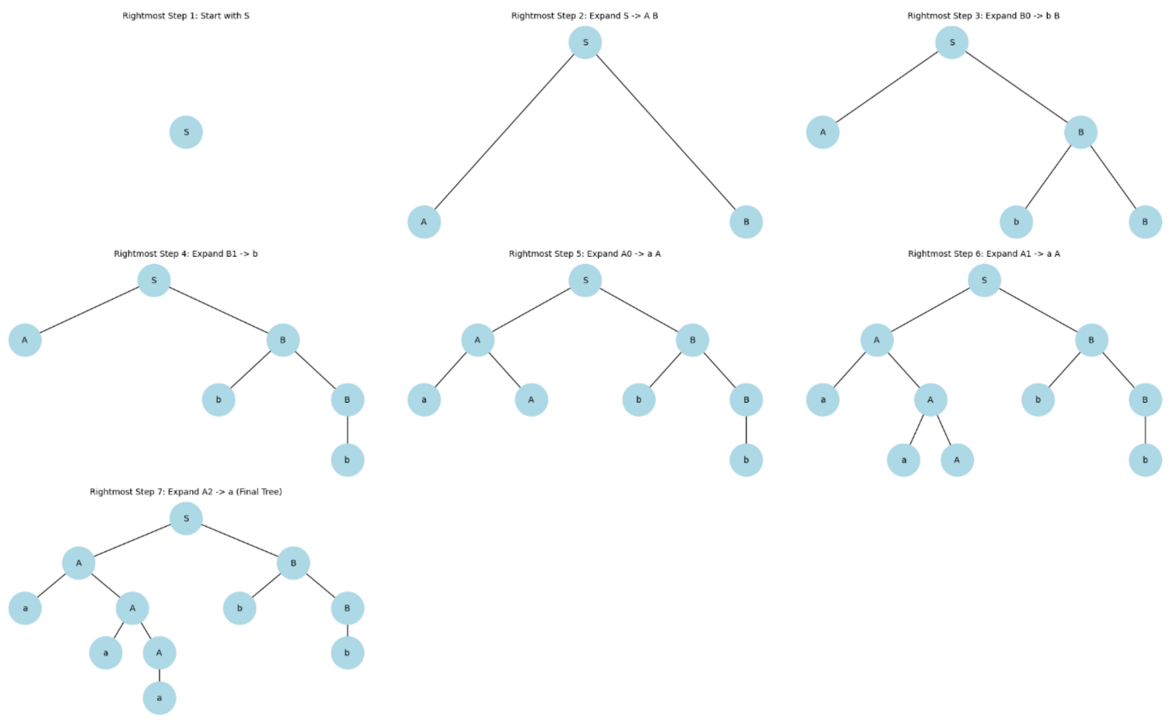
Lab Task Implementation

```
import networkx as nx

import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

def assign_positions(G, root, pos={}, x=0, y=0, width=1, height=1,
level=0):
    pos[root] = (x, y)
    children = list(G.successors(root))
    if children:
        dx = width / len(children)
        for i, child in enumerate(children):
            assign_positions(G, child, pos, x + i * dx - width / 2 + dx
/ 2, y - height, width / len(children), height, level + 1)
    return pos

def draw_subtree(ax, G, title):
    pos = assign_positions(G, "S", width=8, height=1)
    labels = nx.get_node_attributes(G, 'label')

    nx.draw(G, pos, labels=labels, with_labels=True, node_size=1500,
node_color="lightblue",
            font_size=10, arrows=False, ax=ax)
    ax.set_title(title, fontsize=10)
    ax.axis('off')

# Function to draw all leftmost steps in a grid
def draw_leftmost_steps():
    fig = plt.figure(figsize=(20, 12))
    gs = GridSpec(3, 3, figure=fig)

    G_left = nx.DiGraph()

    # Step 1: Start with S
    G_left.add_node("S", label="S")
    draw_subtree(fig.add_subplot(gs[0, 0]), G_left, "Leftmost Step 1:
Start with S")

    # Step 2: Expand S -> A B
    G_left.add_node("A0", label="A")
    G_left.add_node("B0", label="B")
    G_left.add_edge("S", "A0")
    G_left.add_edge("S", "B0")
    draw_subtree(fig.add_subplot(gs[0, 1]), G_left, "Leftmost Step 2:
Expand S -> A B")

    # Step 3: Expand leftmost A (A0) -> a A
    G_left.add_node("a0", label="a")
    G_left.add_node("A1", label="A")
    G_left.add_edge("A0", "a0")
    G_left.add_edge("A0", "A1")
    draw_subtree(fig.add_subplot(gs[0, 2]), G_left, "Leftmost Step 3:
Expand A0 -> a A")

    # Step 4: Expand leftmost A (A1) -> a A
    G_left.add_node("a1", label="a")
    G_left.add_node("A2", label="A")
    G_left.add_edge("A1", "a1")
    G_left.add_edge("A1", "A2")
    draw_subtree(fig.add_subplot(gs[1, 0]), G_left, "Leftmost Step 4:
Expand A1 -> a A")

    # Step 5: Expand leftmost A (A2) -> a
    G_left.add_node("a2", label="a")
    G_left.add_edge("A2", "a2")
    draw_subtree(fig.add_subplot(gs[1, 1]), G_left, "Leftmost Step 5:
Expand A2 -> a")

    # Step 6: Expand leftmost B (B0) -> b B
    G_left.add_node("b0", label="b")
    G_left.add_node("B1", label="B")
    G_left.add_edge("B0", "b0")
    G_left.add_edge("B0", "B1")
    draw_subtree(fig.add_subplot(gs[1, 2]), G_left, "Leftmost Step 6:
Expand B0 -> b B")
```

```
    # Step 7: Expand leftmost B (B1) -> b
    G_left.add_node("b1", label="b")
    G_left.add_edge("B1", "b1")
    draw_subtree(fig.add_subplot(gs[2, 0]), G_left, "Leftmost Step 7:
Expand B1 -> b (Final Tree)")

    plt.tight_layout()
    plt.show()

# Function to draw all rightmost steps in a grid
def draw_rightmost_steps():
    fig = plt.figure(figsize=(20, 12))
    gs = GridSpec(3, 3, figure=fig)

    G_right = nx.DiGraph()

    # Step 1: Start with S
    G_right.add_node("S", label="S")
    draw_subtree(fig.add_subplot(gs[0, 0]), G_right, "Rightmost Step 1:
Start with S")

    # Step 2: Expand S -> A B
    G_right.add_node("A0", label="A")
    G_right.add_node("B0", label="B")
    G_right.add_edge("S", "A0")
    G_right.add_edge("S", "B0")
    draw_subtree(fig.add_subplot(gs[0, 1]), G_right, "Rightmost Step 2:
Expand S -> A B")

    # Step 3: Expand rightmost B (B0) -> b B
    G_right.add_node("b0", label="b")
    G_right.add_node("B1", label="B")
    G_right.add_edge("B0", "b0")
    G_right.add_edge("B0", "B1")
    draw_subtree(fig.add_subplot(gs[0, 2]), G_right, "Rightmost Step 3:
Expand B0 -> b B")

    # Step 4: Expand rightmost B (B1) -> b
    G_right.add_node("b1", label="b")
    G_right.add_edge("B1", "b1")
    draw_subtree(fig.add_subplot(gs[1, 0]), G_right, "Rightmost Step 4:
Expand B1 -> b")

    # Step 5: Expand rightmost A (A0) -> a A
    G_right.add_node("a0", label="a")
    G_right.add_node("A1", label="A")
    G_right.add_edge("A0", "a0")
    G_right.add_edge("A0", "A1")
    draw_subtree(fig.add_subplot(gs[1, 1]), G_right, "Rightmost Step 5:
Expand A0 -> a A")

    # Step 6: Expand rightmost A (A1) -> a A
    G_right.add_node("a1", label="a")
    G_right.add_node("A2", label="A")
    G_right.add_edge("A1", "a1")
    G_right.add_edge("A1", "A2")
    draw_subtree(fig.add_subplot(gs[1, 2]), G_right, "Rightmost Step 6:
Expand A1 -> a A")

    # Step 7: Expand rightmost A (A2) -> a
    G_right.add_node("a2", label="a")
    G_right.add_edge("A2", "a2")
    draw_subtree(fig.add_subplot(gs[2, 0]), G_right, "Rightmost Step 7:
Expand A2 -> a (Final Tree)")

    plt.tight_layout()
    plt.show()

# Run the visualizations
draw_leftmost_steps()
draw_rightmost_steps()
```

Output


Leftmost Step 1: Start with S
Leftmost Step 2: Expand S -> A B
Leftmost Step 3: Expand A0 -> a A
Leftmost Step 4: Expand A1 -> a A
Leftmost Step 5: Expand A2 -> a
Leftmost Step 6: Expand B0 -> b B
Leftmost Step 7: Expand B1 -> b (Final Tree)


Rightmost Step 1: Start with S
Rightmost Step 2: Expand S -> A B
Rightmost Step 3: Expand B0 -> b B
Rightmost Step 4: Expand B1 -> b
Rightmost Step 5: Expand A0 -> a A
Rightmost Step 6: Expand A1 -> a A
Rightmost Step 7: Expand A2 -> a (Final Tree)

Conclusion

In this lab, we explored how parsers play a vital role in compiler design by analyzing program syntax according to defined grammar rules.

The construction of a parse tree helped us visualize the hierarchical structure of code and understand how statements are derived step by step.

Through practical implementation, we learned how parsing ensures correct program structure before further compilation stages.

This process not only detects syntax errors early but also prepares the foundation for semantic analysis and code generation.

Overall, the lab strengthened our understanding of how compilers interpret and organize source code efficiently.