

Student 27

(Theory): Explain left recursion and why it should be removed for top-down parsers

Left Recursion in Compiler Design

Definition of Left Recursion

A grammar is said to be **left recursive** if a non-terminal symbol appears as the **leftmost symbol** on the right-hand side of its own production.

Formally, a grammar has **left recursion** if there exists a non-terminal **A** such that:

$$A \rightarrow A\alpha$$

where α is a string of terminals and/or non-terminals.

Types of Left Recursion

1. Direct Left Recursion

When a non-terminal directly calls itself as the first symbol.

Example:

$$A \rightarrow A\alpha \mid \beta$$

Concrete Example:

$$E \rightarrow E + T \mid T$$

2. Indirect Left Recursion

When a non-terminal derives another non-terminal that eventually leads back to itself.

Example:

$$A \rightarrow B\alpha$$

$$B \rightarrow A\beta$$

Here, **A** indirectly derives itself through **B**.

Why Left Recursion Should Be Removed for Top-Down Parsers

Top-Down Parsing Overview

Top-down parsers (such as **LL(1)** and **recursive descent parsers**) construct the parse tree from the **root to the leaves**.

They expand the **leftmost non-terminal first**.

Problem with Left Recursion

When a top-down parser encounters a left-recursive production, it causes **infinite recursion**.

Example Grammar:

$$E \rightarrow E + T \mid T$$

Parser Behavior:

- To parse E , the parser tries $E \rightarrow E + T$
- This requires parsing E again
- The process repeats infinitely without consuming input

Consequences

- Infinite recursion
- Stack overflow
- Parser never terminates
- Grammar becomes **unsuitable for LL parsers**

Therefore, **left recursion must be eliminated** to make the grammar compatible with top-down parsing techniques.

Removal of Left Recursion

General Technique

For a grammar of the form:

$$A \rightarrow A\alpha \mid \beta$$

It can be transformed into:

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \varepsilon \end{aligned}$$

Example: Removing Left Recursion

Original Grammar:

$$E \rightarrow E + T \mid T$$

After Removing Left Recursion:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \varepsilon \end{aligned}$$

This new grammar:

- Produces the same language
- Avoids infinite recursion
- Is suitable for **LL(1) and recursive descent parsers**

Importance of Removing Left Recursion

- Enables **top-down parsing**
- Prevents infinite loops in recursive descent parsers
- Simplifies grammar analysis
- Helps in constructing **predictive parse tables**
- Essential for compiler implementation

Conclusion

Left recursion is a property of grammars that causes serious problems for top-down parsers by leading to infinite recursion. Since LL parsers expand the leftmost symbol first, left-recursive grammars must be transformed into equivalent non-left-recursive forms. Removing left recursion is a fundamental step in syntax analysis and plays a crucial role in designing efficient and correct compilers.

2. (C++): Write a C++ program to **tokenize arithmetic expressions** with integers and +/.*.

```
#include <iostream>
#include <cctype>
using namespace std;
int main() {
    string expr;
    cout << "Enter an arithmetic expression: ";
    getline(cin, expr);
    cout << "\nTokens:\n";
    for (int i = 0; i < expr.length(); i++) {
        // If digit, read the full integer
        if (isdigit(expr[i])) {
            int num = 0;
            while (i < expr.length() && isdigit(expr[i])) {
                num = num * 10 + (expr[i] - '0');
                i++;
            }
            i--; // step back
            cout << "INTEGER: " << num << endl;
        }

        // If operator
        else if (expr[i] == '+' || expr[i] == '-' ||
            expr[i] == '*' || expr[i] == '/') {
            cout << "OPERATOR: " << expr[i] << endl;
        }

        // Ignore spaces
        else if (isspace(expr[i])) {
            continue;
        }
    }
}
```

```
    }  
    // Invalid character  
    else {  
        cout << "INVALID TOKEN: " << expr[i] << endl;  
    }  
}  
return 0;  
}
```

3. (Problem-solving): Grammar:

$$S \rightarrow AB$$
$$A \rightarrow aA \mid \varepsilon$$
$$B \rightarrow bB \mid b$$

Construct the **parse tree** for "aab".

Given Grammar

$$S \rightarrow AB$$
$$A \rightarrow aA \mid \varepsilon$$
$$B \rightarrow bB \mid b$$

Given Input String

"aab"

Step 1: Understanding the Grammar

- **S** is the start symbol.
- **A** generates **zero or more a's** (because of $aA \mid \varepsilon$).
- **B** generates **one or more b's** (because of $bB \mid b$).

So the grammar generates strings of the form:

$a^* b^+$

The string "aab" fits this pattern:

- $aa \rightarrow$ generated by **A**
- $b \rightarrow$ generated by **B**

Step 2: Derivation of the String

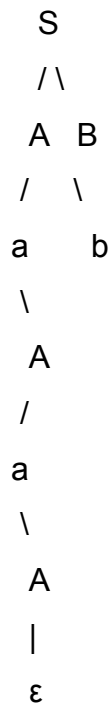
Leftmost Derivation

$S \Rightarrow AB$
 $\Rightarrow aA B$
 $\Rightarrow aaA B$
 $\Rightarrow aa\epsilon B$
 $\Rightarrow aa B$
 $\Rightarrow aa b$

Thus, the string "**aab**" is successfully derived.

Step 3: Constructing the Parse Tree

Parse Tree for "aab"



Step 4: Explanation of the Parse Tree

- **S** expands into **A B**
- **A** produces two **a**'s and then terminates with ϵ
- **B** produces a single **b**
- Leaf nodes (terminals) read from left to right give:

a a b

Final Output String

aab

Conclusion

The parse tree correctly represents how the grammar derives the string "**aab**".
This demonstrates:

- Recursive expansion of non-terminals
- Use of **ϵ -productions**
- Correct hierarchical structure of syntax analysis