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 → Aα
```

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 → Aα | β
```

Concrete Example:

```
E → E + T | T
```

2. Indirect Left Recursion

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

Example:

```
A → Bα
B → Aβ
```

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 → E + T | T
```

Parser Behavior:

- To parse `E`, the parser tries `E → 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 → Aα | β
```

It can be transformed into:

```
A → βA'
A' → αA' | ε
```

Example: Removing Left Recursion

Original Grammar:

```
E → E + T | T
```

After Removing Left Recursion:

```
E  → T E'
E' → + T E' | ε
```

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 → AB

A → aA | ε

B → bB | b

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

Given Grammar

S → AB

A → aA | ε

B → bB | 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 | ε`).
- **B** generates **one or more b's** (because of `bB | b`).

So the grammar generates strings of the form:

a* b+

The string `"aab"` fits this pattern:

- `aa` → generated by **A**
- `b` → generated by **B**

Step 2: Derivation of the String

Leftmost Derivation

S ⇒ AB

⇒ aA B

⇒ aaA B

⇒ aaε B

⇒ aa B

⇒ aa b

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

Step 3: Constructing the Parse Tree

Parse Tree for "aab"

```
     S
     / \
    A   B
   /     \
  a       b
   \
    A
   /
  a
   \
    A
    |
    ε
```

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