Assignment 01: Syntax Analysis

Course: Principles of Compiler Design

Name: Abraham Addisu

ID: 1504850

1. Theory: Context-Free Grammar (CFG)

Definition

A Context-Free Grammar (CFG) is a formal way to describe the syntax of a programming language. It consists of four parts:

1. **Non-terminals:** Placeholders that can be replaced (e.g., E, T).
2. **Terminals:** The actual symbols/tokens (e.g., +, *, id).
3. **Productions:** The rules that define how symbols can be substituted.
4. **Start Symbol:** The initial non-terminal where derivation begins.

Example

A simple CFG for an assignment A simple CFG for basic addition:

- $S \rightarrow S + T \mid T$
- $T \rightarrow id$

2. Implementation: C++ Scanner/Tokenizer

Objectives

To implement a lexical scanner that identifies identifiers and numbers in C++.

C++ Source Code

This program reads an input string and categorizes each component.

```
#include <iostream>
#include <cctype>
#include <string>
```

```
using namespace std;

// This function identifies Identifiers and Numbers
void simpleScanner(string input) {
    int i = 0;
    while (i < input.length()) {
        if (isspace(input[i])) { i++; continue; }

        // Identifying Identifiers (Starting with a letter)
        if (isalpha(input[i])) {
            string result = "";
            while (i < input.length() && isalnum(input[i])) {
                result += input[i++];
            }
            cout << "[TOKEN]: Identifier | [VALUE]: " << result << endl;
        }
        // Identifying Numbers (Including decimals)
        else if (isdigit(input[i])) {
            string result = "";
            while (i < input.length() && (isdigit(input[i]) || input[i] == '.')) {
                result += input[i++];
            }
            cout << "[TOKEN]: Number     | [VALUE]: " << result << endl;
        }
        else {
            cout << "[TOKEN]: Operator   | [VALUE]: " << input[i++] << endl;
        }
    }
}
```

```
int main() {
    string sourceCode = "x = 5 + 2.5";
    cout << "Scanning: " << sourceCode << "\n" << endl;
    simpleScanner(sourceCode);
    return 0;
}
```

---

3. Problem Solving: Left Recursion Elimination

Grammar Provided

1. E → E + T | T
2. T → T * F | F
3. F → (E) | id

The Elimination Process

The grammar is left-recursive because the leftmost symbol on the right side matches the non-terminal on the left. We eliminate this to prevent infinite loops in top-down parsers.

Transformed Grammar

Non-terminal	Production Rule
E	E → T E'
E'	E' → T E' | ε
T	T → FT'
T'	T'→ *FT' | ε
F	F → (E) | id

4. Conclusion

This assignment demonstrates the foundational steps of Syntax Analysis, from defining CFGs to implementing lexical scanners and transforming grammars for parser compatibility. This work serves as the prerequisite for Assignment 01, which focuses on Type Conversion and Semantic Analysis.