Principles of Compiler Design

Assignment 1 – Student 32 (Question 32)

Student Name: Tamer Getaw
ID: BDU1507859
Course: Principles of Compiler Design
Topic: Syntax Analysis

1. Theory: Context-Free Grammar (CFG)

Definition

A **Context-Free Grammar (CFG)** is a type of formal grammar used to define the syntax of programming languages in a **precise and structured way**. CFGs are essential in compiler design because they allow the compiler to recognize **valid programs** and generate **parse trees** or **abstract syntax trees (ASTs)**.

Formally, a CFG is defined as:

$G= (V, \Sigma, P, S)$

Where:

- **V (Non-terminals):** Symbols representing groups of strings (like expressions, terms, statements)
- **Σ (Terminals):** Actual symbols appearing in the language (like +, *, id)
- **P (Productions):** Rules describing how non-terminals can be expanded into terminals and/or other non-terminals
- **S (Start symbol):** The starting non-terminal from which derivations begin

Example: Consider a CFG for simple arithmetic expressions:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow id$

- **Non-terminals:** E (Expression), T (Term), F (Factor)
- **Terminals:** +, *, id
- **Start symbol:** E

How CFG Works

1. Begin with the **start symbol** E
2. Apply production rules step by step until only **terminals remain**
3. The result is a **valid string** in the language

Example Derivation:

To derive id + id * id:

E
→ E + T
→ T + T
→ F + T
→ id + T
→ id + T * F
→ id + F * F
→ id + id * F
→ id + id * id

This shows how CFGs can represent nested arithmetic expressions, which are essential for compilers to **parse and understand programs**.

2. C++: Scanner / Tokenizer for Identifiers and Numbers

Purpose

A **scanner** (also called a **lexer**) is the first stage of a compiler. It reads the **raw source code** and splits it into **tokens**, which are the smallest meaningful units:

- **Identifiers** (variable names, function names)
- **Numbers** (integers, decimals)
- **Keywords** (if, while, return, etc.)
- **Operators** (+, -, *, /)

The scanner helps the parser by providing a **structured input**, making syntax analysis easier.

C++ Implementation

```cpp
#include <iostream>
#include <cctype>
using namespace std;

void scan(const string& input) {
    int i = 0;
    while (i < input.length()) {
        // Identifier: starts with a letter, can include letters & digits
        if (isalpha(input[i])) {
            string id = "";
            while (isalnum(input[i])) {
                id += input[i];
                i++;
            }
            cout << "Identifier: " << id << endl;
        }
        // Number: sequence of digits
        else if (isdigit(input[i])) {
            string num = "";
            while (isdigit(input[i])) {
                num += input[i];
                i++;
            }
            cout << "Number: " << num << endl;
        }
        else {
            i++; // Ignore other characters (spaces, punctuation)
        }
    }
}

int main() {
    string input;
    cout << "Enter input string: ";
    cin >> input;
    scan(input);
    return 0;
}
```

Detailed Explanation

1. **Character Analysis:**
    - isalpha() checks for letters (A–Z, a–z)
    - isdigit() checks for digits (0–9)
2. **Building Tokens:**
    - Identifiers: start with a letter → include letters/digits
    - Numbers: only digits
3. **Output:**
    - The program prints each token on a separate line, which can be used by the **parser**.
4. **Example Input/Output:**

Input: x1 + 42
Output:
Identifier: x1
Number: 42

This small scanner can be **expanded** to handle operators, keywords, or comments.

3. Problem-Solving: Left Recursion Elimination

Given Grammar

E → E + T | T
T → T * F | F
F → (E) | id

Why Eliminate Left Recursion?

- Left recursion occurs when a non-terminal **refers to itself as the first symbol** on the right-hand side.
- Example: E → E + T is **left-recursive**
- **Problem:** Top-down parsers (like LL(1) or recursive descent) **cannot handle left recursion** → may enter infinite recursion
- **Solution:** Convert grammar to an equivalent **right-recursive** form

Step 1: Eliminate Left Recursion from E

Original:

E → E + T | T

Transform using **E → T E'**, where E' handles recursion:

E → T E'

E' → + T E' | ε

- ε means **empty string** (no further recursion)

Step 2: Eliminate Left Recursion from T

Original:

T → T * F | F

Transform similarly:

T → F T'
T' → * F T' | ε

Step 3: Final Grammar (Left Recursion Removed)

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Step 4: Explanation of the Transformed Grammar

- **E → T E'**: E starts with a term, followed by zero or more + T sequences
- **T → F T'**: T starts with a factor, followed by zero or more * F sequences
- **F → (E) | id**: Factor is either a parenthesized expression or an identifier
- This grammar is now **suitable for LL(1) parsing**
- It preserves the **original language** but avoids infinite recursion

Step 5: Example Derivation: To derive id + id * id using the new grammar:

E
→ T E'
→ F T' E'
→ id T' E'
→ id E' (T' → ε)
→ id + T E'
→ id + F T' E'
→ id + id T' E'
→ id + id * F T' E'
→ id + id * id T' E'
→ id + id * id E'
→ id + id * id (E' → ε)

 The same string as before, parsed**without left recursion**.

 Summary

- **CFGs** define programming language syntax clearly
- **Scanners** break input into meaningful tokens for parsing
- **Left recursion elimination** is essential for top-down parsers
- The transformed grammar preserves the language and allows **safe recursive descent parsing**