Grammars

A **grammar** is a set of rules (**production rules**) that defines the valid structure of a particular language. It is used in **compiler construction** to describe the syntax of programming languages.

Components of Grammar

A grammar consists of four components, represented

as: G(V,T,P,S)G(V, T, P, S)G(V,T,P,S) where:

- **V** → Set of **variables** (also called non-terminals), represented by **capital letters**.
- **T** → Set of **terminals**, represented by **small letters** or symbols.
- **P** → **Production rules**, which define how variables and terminals can be combined.
- **S** → **Start symbol**, from which derivations begin.

Types of Grammar (According to Chomsky's Hierarchy)

1. **Type-0 Grammar (Unrestricted Grammar)** ○ No restrictions on production rules.
   ○ Can generate **any** computable language.

2. **Type-1 Grammar (Context-Sensitive Grammar)** ○ The length of the **left-hand side** of a production rule must not be greater than the **right-hand side**.

   ○ Used in **some complex language constructs**.
3. **Type-2 Grammar (Context-Free Grammar - CFG)**
   ○ Each production rule has a **single non-terminal** on the left-hand side. ○ Used in **syntax analysis (parsing)** of programming languages.

4. **Type-3 Grammar (Regular Grammar)**

○ Each production rule follows a strict pattern where terminals appear in a specific order. ○ Used in **lexical analysis (token recognition)**.

Context-Free Grammar (CFG) Definition

A **context-free grammar (CFG)** is a type of grammar where **each production rule has a single non-terminal on the left-hand side** and a combination of terminals and/or non-terminals on the right-hand side. It is widely used in **syntax analysis (parsing)** in compiler construction.

Why is CFG Important in a Compiler?

1. **Defines the Syntax of a Programming Language**
   - CFG provides formal rules that describe how statements, expressions, and structures should be written.
   - Example: How `if-else` statements or arithmetic expressions are formed.
2. **Helps in Syntax Analysis (Parsing)**
   - The **parser** in a compiler uses CFG to verify that the code is written correctly.
   - If the code does not follow the grammar rules, the compiler generates a **syntax error**.
3. **Used for Parse Tree Generation**
   - The compiler converts the input code into a **parse tree** using CFG, which helps in further processing like **semantic analysis and optimization**.

CFG in Compiler Phases

Compiler Phase	**Role of CFG**
Lexical Analysis	Breaks code into tokens (e.g., `if`, `(`, `x`, `>`, `0`, `)`, `{`, `y = 5;`, `}`)
Syntax Analysis (Parsing)	Uses CFG to check if tokens form a valid structure (e.g., correct `if-else` syntax)
Semantic Analysis	Ensures logical correctness (e.g., `x > 0` must be a valid condition)
Code Generation & Optimization	Converts valid syntax into machine code and optimizes performance

Conclusion

- **CFG is essential in compiler construction** because it defines valid syntax and helps the parser check for errors.
- The **syntax analyzer (parser)** of a compiler **uses CFG rules** to ensure that a program follows the correct structure.
- If a program **does not match the CFG rules**, the compiler throws a **syntax error**.