Core Functions of a Mini Compiler: Lexer and Parser

In a mini compiler, two of the most essential components are the **Lexer** and the **Parser**. Together, they form the foundation of the compilation process by breaking down source code into manageable components and ensuring its grammatical correctness. Here is an explanation of their roles and significance:

---

1. Lexical Analyzer (Lexer)

The Lexer, also known as the Lexical Analyzer, is the first stage of the compilation process. Its main responsibility is to convert the raw source code into a sequence of meaningful tokens. Tokens are atomic units of the programming language, such as keywords, identifiers, literals, and operators.

Key Responsibilities:

- **Tokenization:**
  - Scans the source code character by character.
  - Groups characters into meaningful sequences called tokens.
  - Examples of tokens include `if`, `while`, `int`, `=`, `+`, etc.
- **Eliminating Whitespace and Comments:**
  - Ignores unnecessary elements like whitespace, tabs, and comments that do not affect the logical structure of the code.
- **Error Detection:**
  - Identifies lexical errors such as invalid characters or improperly formed tokens (e.g., unclosed string literals).

Output:

The Lexer outputs a stream of tokens to the next stage (the Parser), each token being a pair of:

1. **Token type** (e.g., `IDENTIFIER`, `KEYWORD`, `NUMBER`).
2. **Value** (e.g., the name of a variable, the specific keyword).

---

2. Syntax Analyzer (Parser)

The Parser, or Syntax Analyzer, is the second stage of the compilation process. Its main task is to analyze the sequence of tokens generated by the Lexer and determine if they conform to the grammatical rules (syntax) of the programming language.

Key Responsibilities:

- **Parsing the Tokens:**

    - Constructs a hierarchical structure, often represented as a **Parse Tree** or **Abstract Syntax Tree (AST)**.
    - Each node in the tree represents a syntactic construct, such as an expression, a statement, or a block of code.
- **Grammar Validation:**
    - Ensures that the arrangement of tokens adheres to the language's context-free grammar.
    - For example, in a C-like language, the statement `int x = 10;` would be valid, but `int = x 10;` would produce a syntax error.
- **Error Detection and Recovery:**
    - Detects syntax errors like missing semicolons, mismatched parentheses, or invalid expressions.
    - Attempts to recover from errors to continue parsing the rest of the code.

Output:

The Parser outputs a **Parse Tree** or an **Abstract Syntax Tree (AST)**, which is a structured representation of the code. This tree is passed to later stages like the Semantic Analyzer and Code Generator.

Collaboration Between Lexer and Parser

- **Sequential Operation:**
    - The Lexer processes the raw source code and produces tokens.
    - The Parser consumes these tokens to build the syntax tree.
- **Error Reporting:**
    - Lexical errors are identified by the Lexer, while grammatical errors are flagged by the Parser.
- **Dependency:**
    - The Parser relies on accurate tokenization by the Lexer for correct parsing. Conversely, the Lexer needs to align its token definitions with the language grammar defined for the Parser.

Importance of Lexer and Parser in a Mini Compiler

- The **Lexer** ensures that the source code is broken down into meaningful components, enabling efficient analysis and error detection.
- The **Parser** ensures that the code's structure is syntactically valid, laying the groundwork for subsequent stages like semantic analysis and code generation.

Together, these components streamline the compilation process, converting human-readable code into a form that can be further processed and eventually executed by a machine.