Phases of Program Execution:

- **Preprocessing**
  - **Purpose**: Handles directives for the preprocessor, such as #include and #define in C/C++ programs. It modifies the source code before the actual compilation starts.
- **Compilation**
  - **Purpose**: Translates the preprocessed source code into assembly code. This phase checks for syntax errors and generates a low-level representation of the program.
- **Assembly**
  - **Purpose**: Converts the assembly code produced by the compiler into machine code (binary instructions specific to the computer's architecture).
- **Static Linking**
  - **Purpose**: Combines multiple object files into a single executable file. It resolves references

between object files and includes libraries that are needed.

- **Example**: Combining main.o and utils.o into a single program.exe.

Dynamic Linking

- **Purpose**: Links libraries at run time rather than compile time. This allows programs to use shared libraries, reducing memory usage and disk space.

Compiler Phases

1. **Lexical Analysis (Scanning):**

   Scanner: reads the source code character by character and converts it into tokens.

2. **Syntax Analysis (Parsing):**

   Parser: Takes tokens generated by the scanner and assembles them into a hierarchical structure called an Abstract Syntax Tree (AST), which represents the grammatical structure of the source code.

3. **Semantic Analysis:**
   Semantic Analyzer: Ensures that the syntax tree follows the rules of the language's semantics. It typically generates an Intermediate Representation (IR) of the code.

4. **Intermediate Code Generation:** Converts the syntax tree into an intermediate code, which is not machine-specific.

5. **Code Optimization:**

- Optimizes the intermediate code to make it more efficient by removing unnecessary code and combining operations.

6. **Target Code Generation:** Translates the optimized intermediate code into machine code or assembly code.

Symbol Table Management

- **Storage:** Stores names of all entities (variables, functions, objects) in a structured format.
- **Verification:** Ensures variables are declared before use.
- **Type Checking:** Verifies assignments and expressions for type correctness.
- **Scope Resolution:** Determines the scope of a name to manage variable visibility.

Error Handling

- **Lexical Errors:** Mistakes in the spelling of identifiers, keywords, or operators.
- **Syntax Errors:** Missing punctuation, such as semicolons or parentheses.
- **Semantic Errors:** Incompatible value assignments or type mismatches.
- **Logical Errors:** Issues like unreachable code or infinite loops.

Error Recovery Techniques:

- **Panic Mode:** Skips sections of code until a safe point is found.
- **Phase Level:** Corrects errors by inserting missing characters.
- **Error Productions:** Uses specific rules for common errors.
- **Global Correction:** Attempts to correct errors by finding the closest matching valid code.

Formal and Informal Languages

- **Informal Languages:** These are the everyday languages we use to communicate with each other. They are flexible, with few rules, making them easy to speak and understand.

- **Formal Languages:** These are used in professional or academic contexts. They have strict rules and require careful choice of words and tone to ensure clarity and precision.

Chomsky Hierarchy

- **Type 0 (Unrestricted Grammars):** These have no restrictions on their production rules, allowing for complex and powerful expressions. They use Turing machines for processing.
  - Example: A → xxB
- **Type 1 (Context-Sensitive Grammars):** The number of symbols on the left side of a production rule must not increase on the right side. They use linear bounded automata.
  - Example: S → AT | ε, Tb → xy, A → a
- **Type 2 (Context-Free Grammars):** Each production rule has a single non-terminal on the left side. They use pushdown automata.
  - Example: A → aBb, Ab, Ba
- **Type 3 (Regular Grammars):** These have rules that produce regular languages and are processed by finite automata like NFA or DFA.
  - Example: A → xy, R → ajbc

Regular Expressions (R.E.)

- **Definition:** Regular expressions are sequences of strings from set of alphabets that are used to define patterns for matching strings. They are essential for defining tokens in lexical analysis.

1. Tokens

Tokens are sequences of characters that are treated as single logical entities in the context of programming languages. They are the building blocks of source code and can represent various elements, such as:

Pattern

Definition: A pattern is a rule that defines what a token should look like. It tells the lexical analyzer (Scanner) how to recognize and categorize different tokens.

Example: The pattern for an identifier might be "starts with a letter, followed by letters or numbers".

Lexemes

Definition: A lexeme is a sequence of characters in the source code that matches a pattern for a token. It represents the actual text matched by the scanner.

Example: In the line int x = 10;, the lexeme int matches the pattern for a keyword, x matches the pattern for an identifier, and 10 matches the pattern for a number.