**CC LAB 01** | LEX and YACC Theory

**Aim:** Write answers of all questions:

1. What are compilers and assemblers?
2. Explain in detail phases of compilation process.
3. How lexical analysis and syntax analysis works? What are the tools for that purpose?
4. How Tokens, lexemes and pattern are different?
5. Explain in detail LEX and YACC specification.

**Answers:**

1. **What are compilers and assemblers?**

Compilers and assemblers are software tools used in the process of converting high-level programming languages into machine code that a computer's central processing unit (CPU) can understand and execute.

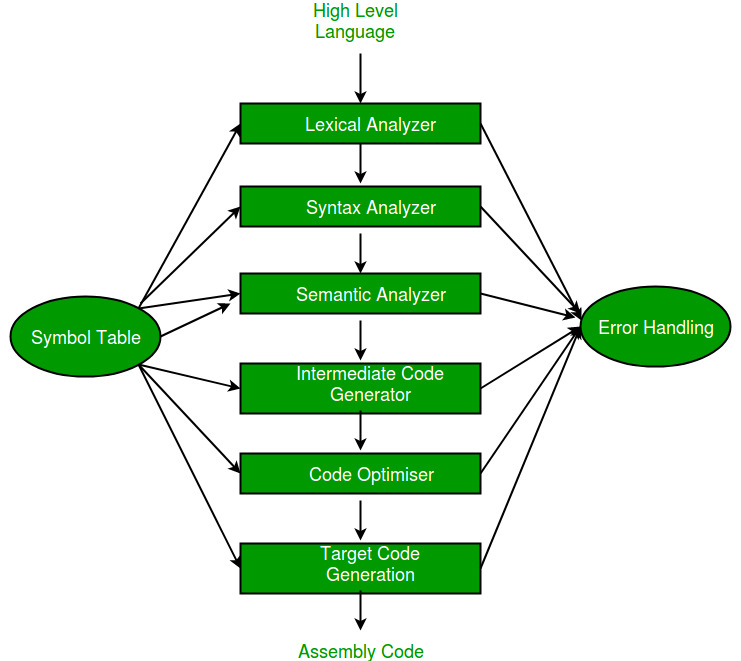
**Compilers**:

A compiler is a software program that takes the entire source code written in a high-level programming language (e.g., C, C++, Java, Python) and translates it into an equivalent program in a lower-level language, typically machine code or intermediate code. The compiled output is a standalone executable file that can be directly executed by the computer's CPU. The compilation process includes multiple stages, such as lexical analysis, syntax analysis, semantic analysis, code generation, and optimization. The main advantage of using a compiler is that it produces optimized code for the target machine, which can result in faster and more efficient programs.

**Assemblers**:

An assembler is a program that translates assembly language code into machine code. Assembly language is a low-level programming language that closely resembles the architecture of the target computer's CPU. Each assembly language instruction corresponds to a specific machine instruction. Assemblers convert these human-readable mnemonics and operands into binary machine code that the CPU can directly execute. Unlike high-level languages, assembly languages provide a more direct control over hardware resources and are often used for low-level programming, embedded systems, and performance-critical applications.

1. **Explain in detail various phases of compilation process.**
2. **Lexical Analysis**: The first phase of a compiler is lexical analysis, also known as scanning. This phase reads the source code and breaks it into a stream of tokens, which are the basic units of the programming language. The tokens are then passed on to the next phase for further processing.
3. **Syntax Analysis**: The second phase of a compiler is syntax analysis, also known as parsing. This phase takes the stream of tokens generated by the lexical analysis phase and checks whether they conform to the grammar of the programming language. The output of this phase is usually an Abstract Syntax Tree (AST).
4. **Semantic Analysis**: The third phase of a compiler is semantic analysis. This phase checks whether the code is semantically correct, i.e., whether it conforms to the language’s type system and other semantic rules.
5. **Intermediate Code Generation**: The fourth phase of a compiler is intermediate code generation. This phase generates an intermediate representation of the source code that can be easily translated into machine code.
6. **Optimization**: The fifth phase of a compiler is optimization. This phase applies various optimization techniques to the intermediate code to improve the performance of the generated machine code.
7. **Code Generation**: The final phase of a compiler is code generation. This phase takes the optimized intermediate code and generates the actual machine code that can be executed by the target hardware.



1. **How lexical analysis and syntax analysis works? What are the tools for that purpose?**

**Lexical Analysis**

Lexical analysis, also known as scanning, is the first phase of the compilation process. Its primary goal is to read the source code character by character and group them into meaningful tokens. Tokens are the smallest units of a programming language, such as keywords, identifiers, operators, and literals. The lexical analyzer uses regular expressions or finite automata to identify and categorize these tokens based on the language's lexical grammar.

The process of lexical analysis typically involves the following steps:

* 1. **Scanning**: The source code is read character by character from left to right.
  2. **Recognizing Lexemes**: The scanner identifies substrings of characters in the source code that form a valid token according to the language's lexical grammar. These substrings are known as lexemes.
  3. **Tokenization**: For each recognized lexeme, the lexical analyser generates a token that includes the lexeme's type and value (if applicable).
  4. **Skipping Whitespace and Comments**: The lexical analyser ignores whitespace characters (spaces, tabs, line breaks) and comments during tokenization, as they do not contribute to the program's structure.
  5. **Handling Errors**: If the lexical analyser encounters an invalid character sequence (e.g., an unknown symbol), it reports an error.

**Syntax Analysis**

Syntax analysis, also known as parsing, is the second phase of the compilation process. It takes the sequence of tokens generated by lexical analysis and arranges them into a hierarchical structure called the abstract syntax tree (AST) or parse tree. The parser verifies whether the arrangement of tokens adheres to the rules defined by the language's syntax.

The process of syntax analysis involves the following steps:

1. **Grammar Parsing**: The parser uses a formal grammar, such as context-free grammar (CFG), to define the language's syntax rules. These rules specify how valid sequences of tokens can be combined to form language constructs (statements, expressions, etc.).
2. **Constructing the Parse Tree**: The parser uses the grammar rules to construct the parse tree, which represents the program's syntactic structure. Each node in the parse tree corresponds to a language construct, and the tree's structure mirrors the hierarchical relationships between these constructs.
3. **Error Detection**: If the sequence of tokens does not conform to the language's syntax rules, the parser detects syntax errors and reports them to the user.
4. **Abstract Syntax Tree (AST) Generation (Optional):** In some cases, the parser may further simplify the parse tree and generate an abstract syntax tree (AST). The AST omits certain grammar-specific details and focuses on representing the program's logic and structure more succinctly.

Tools for Lexical Analysis and Syntax Analysis:

Several tools and libraries are available to perform lexical analysis and syntax analysis. Here are some popular ones:

1. **Lex and Flex**: Lex and Flex are tools for generating lexical analysers (scanners) based on regular expressions. They generate C or C++ code that can be integrated into the compiler.
2. **Yacc and Bison**: Yacc and Bison are tools for generating parsers based on a formal grammar, typically specified using BNF (Backus-Naur Form). They produce C or C++ code for constructing the parse tree or AST.
3. **How Tokens, lexemes and pattern are different?**