1. What is the purpose of a compiler?

Answer:
A compiler translates a program written in a high-level programming language into machine code that a computer can execute. It also checks for syntax and semantic errors during this translation.

2. What are language processors? Name a few.

Answer:
Language processors are tools that process source code written in programming languages. Examples include:

- **Compiler** – Converts source code to machine code.
- **Interpreter** – Executes code line-by-line.
- **Assembler** – Converts assembly language to machine code.
- **Preprocessor** – Performs tasks like macro expansion before compilation.

3. What are the main phases of a compiler?

Answer:
The main phases of a compiler are:

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization

6. Code Generation

4. What is the role of the lexical analyzer?

Answer:
The lexical analyzer reads the source code and converts it into a sequence of tokens, which are the meaningful elements (like keywords, identifiers, literals) for further analysis.

5. How does a compiler differ from an interpreter?

Answer:
A compiler translates the whole program into machine code before execution, while an interpreter executes the program line-by-line without producing intermediate machine code.

6. What is intermediate code, and why is it used?

Answer:
Intermediate code is a representation of the source program that is between high-level code and machine code. It is used to make the compiler more portable and enable platform-independent optimization.

7. What are the goals of code optimization?

Answer:
Code optimization aims to improve the performance and efficiency of the compiled code by reducing execution time, memory usage, or power consumption.

8. How has the evolution of programming languages influenced compiler design?

Answer:
As languages evolved from low-level to high-level and now to multi-paradigm languages (like object-oriented or functional), compilers have adapted to handle more complex syntax, semantics, and runtime features.

9. What is the difference between syntax analysis and semantic analysis?

Answer:

- **Syntax Analysis** checks the structure of code based on grammar rules.

- **Semantic Analysis** ensures the meaning is correct (e.g., type compatibility, variable declarations).

11. What is the front-end and back-end of a compiler?

Answer:

- **Front-End**: Includes lexical analysis, syntax analysis, and semantic analysis. It checks correctness and builds an intermediate representation.

- **Back-End**: Includes code optimization and code generation. It produces efficient machine code from the IR.

12. What is the symbol table in a compiler?

Answer:
A symbol table is a data structure used by the compiler to store information about identifiers (like variables, functions, classes) such as type, scope, and memory location.

14. Explain the difference between source code and object code.

Answer:

- **Source Code**: The high-level code written by the programmer.

- **Object Code**: The machine-level code generated by the compiler that can be executed by the CPU.

15. What is parsing? What are its types?

Answer:
Parsing is the process of analyzing the syntax of tokens to form a parse tree. The main types are:

- **Top-down Parsing** (e.g., recursive descent)
- **Bottom-up Parsing** (e.g., LR parsers)

16. What is the role of error handling in a compiler?

Answer:
Error handling detects and reports errors during compilation (syntax, semantic, etc.), helps in debugging, and often allows compilation to continue to find more errors.

20. Why is intermediate representation (IR) useful in compilers?

Answer:
IR serves as an abstraction that is easier to analyze and optimize, and makes the compiler more portable by separating machine-independent and machine-dependent parts.

Phase	Type of Error Detected	Example
Lexical Analysis	Invalid symbols or illegal character sequences	@ in int @var = 5;
Syntax Analysis	Syntax errors (violations of grammar rules)	Missing semicolon or unmatched braces
Semantic Analysis	Type mismatch, undeclared variables	Adding a string to an integer
Intermediate Code Generation	Logical inconsistencies	Improper expression translation

Code Optimization	Redundant operations	`x = x + 0`
Code Generation	Platform-specific code generation issues	Invalid register usage
Linking/Assembly	Missing symbols, unresolved references	Undefined external function calls

Phase	Error Type	Error Handling Technique
Lexical Analysis	Invalid characters, malformed tokens	- **Panic Mode**: Skip to next whitespace or delimiter- **Error Tokens**: Replace with valid ones
Syntax Analysis	Grammar violations (e.g., missing `;`)	- **Panic Mode Recovery**: Skip tokens until a synchronizing token is found- **Phrase-level Recovery**: Insert/delete tokens to match grammar- **Error Productions**: Extend grammar to handle common mistakes
Semantic Analysis	Type mismatches, undeclared variables	- **Symbol Table Lookup**: Report undefined identifiers- **Type Checking**: Report incompatible operations and suggest corrections
Intermediate Code Generation	Inconsistent intermediate code	- **Static Checking**: Use assertions and checks to detect errors before code generation
Code Optimization	Redundant or incorrect optimizations	- **Validation Checks**: Ensure that optimizations preserve program semantics

Code Generation	Invalid instructions or register issues	- **Target-specific Checks**: Ensure proper instruction set usage and valid register allocation
Linking and Assembly	Unresolved external references	- **Error Messages**: Report undefined symbols- **Fallback Linking**: Use stubs or default definitions