Compiler Design Viva and Interview Questions

1. What is a Compiler?

Answer:
A compiler is a program that translates source code written in a high-level programming language into machine code or intermediate code for execution.

---

2. What are the Phases of a Compiler?

Answer:
The phases include:

1. **Lexical Analysis:** Converts source code into tokens.
2. **Syntax Analysis:** Checks the syntax using a parse tree.
3. **Semantic Analysis:** Validates logical consistency and data types.
4. **Intermediate Code Generation:** Produces intermediate representation (e.g., 3-address code).
5. **Code Optimization:** Improves code efficiency.
6. **Code Generation:** Produces target machine code.
7. **Code Linking and Loading:** Combines generated code with libraries for execution.

---

3. What is the Role of a Symbol Table in Compilation?

Answer:
The symbol table stores information about identifiers such as variables, functions, and their attributes (e.g., scope, type). It helps in semantic analysis and error detection.

---

4. What is Lexical Analysis?

Answer:
Lexical Analysis is the first phase of compilation that converts the source code into a sequence of tokens (smallest units of the language).

---

5. What is the Difference Between a Compiler and an Interpreter?

Answer:

Compiler	Interpreter
Translates the entire code before execution.	Translates and executes line by line.
Faster execution after compilation.	Slower due to runtime translation.

---

6. What is a Context-Free Grammar (CFG)?

Answer:
CFG is a formal grammar consisting of:

- Terminals (e.g., a, b, c).
- Non-terminals (e.g., S, A).
- Production rules (e.g., S → aSb).
- Start symbol.
  It is used in syntax analysis for parsing.

---

7. What is a Parse Tree?

Answer:
A parse tree visually represents the syntactic structure of a string derived from a grammar. Each node represents a grammar rule applied.

---

8. What is Ambiguity in a Grammar?

Answer:
A grammar is ambiguous if there exists more than one parse tree for a single string. Ambiguity leads to difficulties in syntax analysis.

---

9. Explain Left Recursion. How Can It Be Eliminated?

Answer:
Left recursion occurs when a non-terminal calls itself on the leftmost side of a production (e.g., $A \rightarrow A\alpha \mid \beta$).
Elimination: Rewrite as:
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$

---

10. What is the Difference Between Top-Down and Bottom-Up Parsing?

Answer:

Top-Down Parsing	Bottom-Up Parsing
Starts from the start symbol and derives input.	Builds parse tree from input to start symbol.
Example: Recursive Descent.	Example: Shift-Reduce Parsing.

---

11. What is the Role of Intermediate Code in Compilation?

Answer:
Intermediate code bridges source code and machine code, making it easier to perform optimization and portability (e.g., 3-address code, quadruples).

---

12. What is Peephole Optimization?

Answer:
Peephole Optimization is a local optimization technique that examines a small window of instructions to eliminate redundancies and improve performance.

---

13. What is a DFA, and How is it Used in Lexical Analysis?

Answer:
A Deterministic Finite Automaton (DFA) is used to recognize tokens. The lexical analyzer constructs DFA based on regular expressions for token patterns.

---

14. What is the Purpose of Code Optimization?

Answer:
Code optimization enhances performance by reducing execution time, memory usage, and resource consumption without altering the program's output.

---

15. What are the Types of Parsing Techniques?

Answer:

1. **Top-Down Parsing:** Recursive Descent, LL Parsing.
2. **Bottom-Up Parsing:** LR Parsing, SLR, LALR, Canonical LR.

---

16. What is a Left Factoring in Grammar?

Answer:
Left Factoring is a technique to rewrite grammar to remove common prefixes.
Example:
$A \rightarrow \alpha\beta1 \mid \alpha\beta2$
Becomes:
$A \rightarrow \alpha A'$
$A' \rightarrow \beta1 \mid \beta2$

---

17. What is Backpatching?

Answer:
Backpatching is used in intermediate code generation to handle forward jumps, resolving addresses later.

---

18. What are Different Types of Intermediate Representations?

Answer:

1. **Abstract Syntax Tree (AST).**
2. **3-Address Code (TAC).**
3. **Quadruples.**
4. **Triples.**

---

19. What are the Components of a Compiler?

Answer:

1. **Front-End:** Lexical analysis, syntax analysis, semantic analysis.

2. **Intermediate Code Generator.**
3. **Optimizer.**
4. **Back-End:** Code generation and linking.

---

20. What are LR Parsers?

Answer:
LR Parsers (Left-to-right scanning, Rightmost derivation) include SLR, LALR, and Canonical LR. They efficiently parse context-free grammars.

21. What is Bootstrapping in Compiler Design?

Answer:
Bootstrapping is the process of writing a compiler in the language it is intended to compile. For example, writing a C compiler in C.
Steps:

1. Write a minimal compiler in another language.
2. Use it to compile the full compiler in the target language.

---

22. What is the Difference Between Lexeme, Token, and Pattern?

Answer:

- **Lexeme:** Sequence of characters forming a valid unit (e.g., while).
- **Token:** The category or type of a lexeme (e.g., KEYWORD).
- **Pattern:** The rule or regular expression defining the lexeme.

---

23. What is Register Allocation in Code Generation?

Answer:
Register allocation assigns program variables to a limited number of CPU registers to optimize performance.
Methods:

1. Graph coloring.
2. Linear scan allocation.

---

24. What is an Abstract Syntax Tree (AST)?

Answer:
AST is a tree representation of the abstract syntactic structure of source code. It eliminates unnecessary syntactic details like parentheses.

---

25. What is the Difference Between Intermediate Code and Machine Code?

Answer:

- **Intermediate Code:** Abstract, independent of the machine (e.g., 3-address code).
- **Machine Code:** Binary instructions specific to a target CPU.

---

26. What is a Compiler Front-End?

Answer:
The front-end processes the source code for:

1. **Lexical Analysis:** Tokenizes code.
2. **Syntax Analysis:** Parses tokens into a tree.
3. **Semantic Analysis:** Validates meaning and data types.

---

27. Explain Shift-Reduce Parsing.

Answer:
Shift-Reduce Parsing is a bottom-up approach that uses a stack to build parse trees.
Steps:

1. **Shift:** Push tokens onto the stack.
2. **Reduce:** Replace symbols on the stack with a grammar rule.

---

28. What is the Role of the Error Handler in a Compiler?

Answer:
The error handler detects, reports, and recovers from errors during compilation.
Types:

- **Lexical Errors:** Invalid characters.
- **Syntax Errors:** Parsing issues.
- **Semantic Errors:** Type mismatches.

---

29. What is Three-Address Code (TAC)?

Answer:
TAC is a type of intermediate code with at most three operands per instruction.
Example:

```
t1 = a + b
t2 = t1 * c
```

---

30. What is Code Folding?

Answer:
Code Folding identifies and replaces repeated code patterns with a single instance to save memory and enhance performance.

---

31. Explain the Role of a Parser Generator Like YACC.

Answer:
YACC (Yet Another Compiler-Compiler) automates the creation of parsers by generating syntax analyzers from grammar definitions.

---

32. What is Canonical LR Parsing?

Answer:
Canonical LR is the most powerful LR parser that handles all context-free grammars. It constructs a canonical collection of LR(1) items.

---

33. What is the Purpose of the Back-End of a Compiler?

Answer:
The back-end generates efficient machine code, optimizes it, and handles register allocation and instruction scheduling.

---

34. What is Instruction Scheduling?

Answer:
Instruction scheduling arranges the order of instructions to minimize execution time while avoiding conflicts like pipeline stalls.

---

35. What is Static and Dynamic Scoping?

Answer:

- **Static Scoping:** Variable resolution based on code structure (e.g., most programming languages).
- **Dynamic Scoping:** Variable resolution based on call stack during runtime.

---

36. What is a Cross-Compiler?

Answer:
A cross-compiler generates code for a platform other than the one it runs on.
Example: Compiling code on Windows for execution on an ARM-based device.

---

37. What are the Types of Errors Handled in Compilation?

Answer:

1. **Lexical Errors:** Unknown symbols.
2. **Syntax Errors:** Missing semicolons or mismatched brackets.
3. **Semantic Errors:** Type mismatches or undeclared variables.
4. **Logical Errors:** Incorrect logic, identified by debugging.

---

38. What is Dead Code Elimination?

Answer:
Dead Code Elimination removes unreachable or unnecessary code to optimize programs.
Example: Removing unused variables or statements after a return.

---

39. What is Loop Unrolling in Optimization?

Answer:
Loop Unrolling replicates loop bodies to reduce iteration overhead.
Example:

```
for (i = 0; i < 4; i++)
    sum += arr[i];
```

Becomes:

```
sum += arr[0] + arr[1] + arr[2] + arr[3];
```

---

40. What are Inline Functions in Compilation?

Answer:
Inline functions replace function calls with the actual function code to reduce call overhead.
Limitation: Increases binary size.

41. What is Tail Call Optimization?

Answer:
Tail Call Optimization (TCO) is a technique where the compiler reuses the current function's stack frame for a tail-recursive call, reducing stack usage.
Example (Without TCO):
Each recursive call creates a new stack frame.
Example (With TCO):
The recursive call overwrites the existing frame.

---

42. What is Code Motion in Optimization?

Answer:
Code motion moves code outside a loop if it does not depend on the loop iteration.
Example:

```
for (i = 0; i < n; i++)
    x = 5 + 3; // Can be moved out of the loop
```

Becomes:

```
x = 5 + 3;
for (i = 0; i < n; i++)
```

---

43. What is a Predictive Parser?

Answer:
Predictive parsing is a top-down approach that uses lookahead tokens to decide which production rule to apply. It works for LL(1) grammars.

---

44. Explain the Concept of a Lookahead in Parsing.

Answer:
A lookahead refers to peeking at the next token(s) to guide parsing decisions.

- **LL(1):** One token lookahead.
- **LR(k):** k-token lookahead.

---

45. What are Compiler-Directed Optimizations?

Answer:
Optimizations guided by compiler analysis, such as:

1. **Loop Optimization:** Unrolling or fusion.
2. **Instruction Scheduling:** Avoiding pipeline stalls.
3. **Register Allocation:** Efficient use of CPU registers.

---

46. What is a Syntax-Directed Translation (SDT)?

Answer:
SDT associates semantic actions with grammar rules to build intermediate code or other representations during parsing.

---

47. What is Intermediate Code Representation (ICR)?

Answer:
ICR is an abstract representation of source code that bridges the front-end and back-end of a compiler. Examples include:

1. **Three-Address Code (TAC).**
2. **Quadruples.**
3. **Static Single Assignment (SSA).**

---

48. What is the Role of Semantic Analysis?

Answer:
Semantic Analysis ensures that the source code adheres to language rules, such as type checking and scope resolution. It uses the **symbol table** extensively.

---

49. What is Constant Folding in Compilation?

Answer:
Constant Folding computes constant expressions at compile time to reduce runtime computation.
Example:

```
x = 3 * 4; // Folded to x = 12
```

---

50. What is the Role of Machine-Independent Optimization?

Answer:
Machine-independent optimizations improve code efficiency regardless of the target architecture. Examples include:

1. **Dead Code Elimination.**
2. **Constant Propagation.**
3. **Loop Optimization.**

---

51. What are Machine-Dependent Optimizations?

Answer:
These optimizations target specific hardware features, such as:

1. **Register Allocation.**
2. **Instruction Scheduling.**
3. **Pipelining Optimization.**

---

52. What is the Function of a Macro Processor?

Answer:
A macro processor replaces macros with their corresponding code before compilation, enabling code reuse and simplification.

---

53. What is Instruction Pipelining?

Answer:
Instruction pipelining overlaps the execution of multiple instructions by dividing them into stages (e.g., fetch, decode, execute) to improve performance.

---

54. What is Register Spilling?

Answer:
Register spilling occurs when there are not enough CPU registers to hold all variables, forcing some to be stored in memory.

---

55. What is Aliasing in Compilation?

Answer:
Aliasing occurs when two or more variables reference the same memory location. It complicates optimization as changes to one variable affect the other.

---

56. Explain LR Parsing Conflicts.

Answer:

1. **Shift-Reduce Conflict:** The parser cannot decide whether to shift or reduce.
2. **Reduce-Reduce Conflict:** The parser cannot decide which rule to reduce.

---

57. What is the Principle of Bootstrapping in Compiler Development?

Answer:
Bootstrapping allows a compiler to compile itself using an earlier version or a simpler implementation of itself.

---

58. What are Inline Expansion and Its Advantages?

Answer:
Inline expansion replaces function calls with the function body, eliminating overhead.
Advantages:

1. Faster execution.
2. Potential for further optimization.

---

59. What is Static Single Assignment (SSA)?

Answer:
SSA is an intermediate representation where each variable is assigned exactly once, simplifying optimization and analysis.

---

60. What is a Dynamic Compiler?

Answer:
Dynamic compilers, such as JIT (Just-In-Time) compilers, translate code during execution for performance benefits.
Example: JVM for Java.

---

61. What is Loop Fusion?

Answer:
Loop Fusion combines two loops into one to reduce overhead and enhance cache performance.
Example:

```
for (i = 0; i < n; i++) { a[i] += 1; }
for (i = 0; i < n; i++) { b[i] *= 2; }
```

Becomes:

```
for (i = 0; i < n; i++) { a[i] += 1; b[i] *= 2; }
```

62. What is a SLR(1) Parser?

Answer:
SLR(1) (Simple LR) is a type of bottom-up parser that uses one lookahead token to make parsing decisions. It is simpler than Canonical LR but has limitations in handling more complex grammars.
Strengths: Simple to implement and faster than full LR parsing.
Weaknesses: It may not handle all LR(1) grammars.

---

63. What is an NFA and DFA in Lexical Analysis?

Answer:

- **NFA (Non-deterministic Finite Automaton):** Allows multiple transitions from a state for the same input symbol, making it flexible but harder to implement.
- **DFA (Deterministic Finite Automaton):** Each state has exactly one transition for each input symbol, which simplifies implementation but may require more states.

---

64. What is the Difference Between Syntactic Sugar and a Syntax Sugar in Programming Languages?

Answer:

- **Syntactic Sugar:** Refers to syntax that is designed to be easier for humans to read or write.
- **Syntax Sugar:** Generally refers to shorthand notations that make the syntax more expressive, like operator overloading or lambda functions.

---

65. Explain the Role of the Symbol Table in a Compiler.

Answer:
The **symbol table** stores information about variables, functions, objects, and other identifiers in the program. It is used for semantic analysis and type checking.
Fields in a Symbol Table:

- **Name** of the identifier.
- **Type** (int, char, etc.).
- **Scope** (local/global).
- **Memory location**.

---

66. What is Syntax Tree in Compilation?

Answer:
A syntax tree is a tree-like structure that represents the syntactic structure of the source code according to the grammar of the programming language. It helps in parsing the source code and is used by the compiler to generate machine code.

---

67. What is a Parse Tree?

Answer:
A **Parse Tree** represents the hierarchical syntactic structure of a string derived from the grammar. It is used by parsers to analyze the structure of the input program.

---

68. Explain the Difference Between Context-Free Grammar and Regular Grammar?

Answer:

- **Context-Free Grammar (CFG):** Can generate complex languages, including nested structures.
- **Regular Grammar:** Represents simpler languages, typically used for lexical analysis.

---

69. What is Type Checking in Compilers?

Answer:
Type checking is the process of verifying and enforcing the constraints of types in a program, ensuring operations are performed between compatible data types.
Example: Ensuring that an integer isn't added to a string.

---

70. What is a Shift-Reduce Parser?

Answer:
Shift-Reduce Parsing is a type of bottom-up parsing where the parser shifts symbols onto a stack and reduces them to grammar rules when possible.

---

71. What is a LALR(1) Parser?

Answer:
LALR(1) (Look-Ahead LR) is a type of parser that combines the simplicity of an SLR parser with the power of an LR parser. It resolves some of the conflicts found in SLR parsing by looking ahead one token.

---

72. What is the Role of a Compiler Backend?

Answer:
The **backend** of the compiler is responsible for generating machine code or intermediate code, optimizing the code for performance, and ensuring compatibility with the target architecture. Key tasks include register allocation, instruction scheduling, and code generation.

---

73. What is a Finite Automaton?

Answer:
A **Finite Automaton (FA)** is a mathematical model used in lexical analysis to recognize patterns in strings. There are two types:

1. **Deterministic Finite Automaton (DFA).**
2. **Non-deterministic Finite Automaton (NFA).**

---

74. What is Grammar in Compiler Design?

Answer:
Grammar defines the syntax of a programming language. It consists of a set of rules that specify how sentences in the language can be formed.
Types of Grammar:

1. **Context-Free Grammar (CFG).**
2. **Regular Grammar.**

---

75. What is a Hash Table and Its Role in a Compiler?

Answer:
A **hash table** is a data structure used to store and retrieve keys in constant time. In compilers, hash tables are used for fast lookups of variables, functions, and keywords in the symbol table.

---

76. What is Dead Code Detection in Compiler Optimization?

Answer:
Dead code detection identifies code that does not affect the program's output, such as unreachable code, unused variables, and redundant expressions, and removes it to improve performance.

---

77. What is the Role of a Parser in a Compiler?

Answer:
The parser's role is to analyze the structure of the source code and build a syntax tree or parse tree. It checks if the code follows the grammar of the programming language and helps the compiler understand the program's structure.

78. What is Loop Inversion?

Answer:
Loop inversion is a technique used in compilers to optimize loops. It involves changing a loop's order of execution to reduce the overhead of loop conditions and increase performance.

---

79. What is Copy Propagation in Compiler Optimization?

Answer:
Copy propagation replaces variables with their assigned values to simplify expressions and reduce the number of instructions generated.
Example:

css

```
a = b;
x = a + c;
```

Can be optimized to:

css

```
x = b + c;
```

---

80. What is Constant Propagation?

Answer:
Constant propagation is an optimization where the compiler replaces variables with known constant values. This reduces computation during runtime.
Example:

arduino

```
int x = 5;
y = x + 10;
```

Becomes:

makefile

```
y = 5 + 10;
```

---

81. What is a Context-Sensitive Grammar?

Answer:
Context-sensitive grammar is more powerful than context-free grammar. It allows the production rules to depend on the context in which non-terminals appear.

Example:
The rule A -> BC may depend on whether A appears in a specific context.

---

82. What is Code Optimization in Compilers?

Answer:
Code optimization improves the efficiency of generated code by reducing execution time, memory usage, or both. Techniques include:

1. **Dead code elimination.**
2. **Constant folding.**
3. **Loop unrolling.**

---

83. What is Register Allocation in Compilers?

Answer:
Register allocation assigns variables to CPU registers to improve performance. It minimizes memory access and optimizes execution speed.
Techniques:

1. **Graph coloring.**
2. **Linear scan.**

---

84. What is Inline Expansion?

Answer:
Inline expansion replaces function calls with the actual function body, reducing the function call overhead and possibly improving performance.

---

85. What is Partial Evaluation in Compilers?

Answer:
Partial evaluation is an optimization technique that evaluates parts of the program that can be determined at compile time, leaving the rest for runtime. This reduces runtime overhead.

---

86. What is Grammar Ambiguity?

Answer:
Grammar ambiguity occurs when a string can be parsed in multiple ways using the same grammar, leading to different parse trees. It can cause confusion in understanding the meaning of the code.

---

87. What is a Register-Transfer Language (RTL)?

Answer:
RTL is an intermediate representation used in compilers that specifies the transfer of data between registers in machine-level code.

---

88. What is the Difference Between an Optimizing and a Non-Optimizing Compiler?

Answer:

- **Optimizing Compiler:** Performs optimizations during the compilation process to improve the generated code's performance.
- **Non-Optimizing Compiler:** Directly translates the source code to machine code without optimizations.

---

89. What is a Semantic Error?

Answer:
A semantic error occurs when the code is syntactically correct but doesn't produce the intended result. For example, using the wrong operator or incorrect data types.

---

90. What is the Role of Type Systems in a Compiler?

Answer:
A type system ensures that operations are performed on compatible data types, preventing errors such as trying to add a string to an integer. It also helps in optimization and memory allocation.

91. What is a Shift-Reduce Conflict in Parsing?

Answer:
A **shift-reduce conflict** occurs in bottom-up parsing (e.g., LR parsing) when the parser must choose between shifting the next input symbol onto the stack or reducing a rule. This happens when both actions are valid for a given situation, leading to ambiguity.

---

92. What is the Role of a Preprocessor in a Compiler?

Answer:
A **preprocessor** is a tool that processes the source code before it is passed to the compiler. It handles tasks like:

1. **Macro expansion**
2. **File inclusion**
3. **Conditional compilation**
   It helps in improving code reusability and modularity.

---

93. What is the Role of a Semantic Analyzer in a Compiler?

Answer:
The **semantic analyzer** ensures that the program adheres to the semantic rules of the language. It checks for:

1. **Type mismatches**
2. **Undeclared variables**
3. **Function argument consistency**

---

94. What is the Difference Between a Top-Down and Bottom-Up Parser?

Answer:

- **Top-Down Parser:** Builds the parse tree from the root to the leaves. Examples include LL and recursive descent parsers.

- **Bottom-Up Parser:** Builds the parse tree from the leaves to the root. Examples include LR and SLR parsers.

---

95. What is Syntax-Directed Translation?

Answer:
Syntax-Directed Translation is the process where the translation of the source language constructs is determined by the syntactic structure of the source code. It uses the grammar rules and annotations to generate intermediate code or target code.

---

96. What is Chaining in Compiler Design?

Answer:
Chaining refers to combining multiple smaller actions (such as expression evaluation) into a single, optimized process. In compilers, it can be used to optimize intermediate code or reduce unnecessary operations during code generation.

---

97. What is a Deadlock in Compiler Design?

Answer:
A **deadlock** in compiler design occurs when two or more processes in the system are unable to proceed due to waiting for each other to release resources. This situation can halt the compiler or result in inefficient resource usage.

---

98. What is a Memoization Technique in Compiler Optimization?

Answer:
Memoization is a technique where intermediate results of computations are stored to avoid redundant calculations, improving the speed and efficiency of recursive functions in compilers and interpreters.

---

99. What is a Yacc (Yet Another Compiler Compiler)?

Answer:
Yacc is a tool used for generating parsers. It reads a grammar specification and generates code for a parser based on that grammar. Yacc is commonly used in compiler design to handle syntax analysis.

---

100. What is a Virtual Machine in Compiler Design?

Answer:
A **Virtual Machine (VM)** is an abstract machine that executes the intermediate code generated by the compiler. It provides a platform-independent environment, enabling the same intermediate code to run on different physical machines.

---

101. What is Tail Recursion Optimization in Compiler?

Answer:
Tail recursion optimization (TRO) is an optimization technique where the compiler optimizes recursive calls that are in the tail position (i.e., the last action in a function), turning them into iterative loops to reduce the call stack overhead.

102. What is a Context-Free Grammar's Ambiguity?

Answer:
Ambiguity in a context-free grammar occurs when there are multiple parse trees for a single string in the language. This ambiguity can make parsing and understanding difficult, as the grammar does not have a unique structure for every string.

---

103. What is a Finite State Machine (FSM)?

Answer:
A **Finite State Machine (FSM)** is a mathematical model of computation that can be in one of a finite number of states at any given time. It transitions between states based on input symbols, commonly used in lexical analysis for pattern matching.

---

104. What is an Intermediate Code in Compiler Design?

Answer:
Intermediate code is an abstract representation of the source code, which is independent of machine architecture. The compiler generates this code after parsing and semantic analysis. It is later translated into machine code during the code generation phase.

---

105. What is Code Generation in Compiler Design?

Answer:
Code generation is the phase in the compiler where intermediate code is translated into machine code or assembly code. This phase is responsible for mapping the intermediate representations to instructions that the target machine can execute.

---

106. What is the Purpose of Semantic Rules in a Compiler?

Answer:
Semantic rules define the meaning of the program constructs and check whether the program's logic is correct. They ensure that operations are performed on valid types and values, and they handle scope checking, type checking, and other semantic operations.

---

107. What is the Role of the Code Optimizer in a Compiler?

Answer:
The **code optimizer** improves the performance and efficiency of the generated code by reducing resource consumption. It eliminates redundant instructions, optimizes loops, and applies various optimization strategies to make the code more efficient.

---

108. What is a Semantic Error in Compiler Design?

Answer:
A **semantic error** occurs when the syntax of the code is correct, but the program does not perform as expected due to logical issues, such as using an incorrect operator or calling functions with the wrong arguments.

---

109. What is the Role of a Context-Sensitive Grammar?

Answer:
A **context-sensitive grammar** allows rules to be applied based on the context of the symbols around a non-terminal. This is more powerful than context-free grammar and can generate languages that context-free grammars cannot.

---

110. What is the Function of a Scanner in Compiler Design?

Answer:
A **scanner** (or lexical analyzer) reads the source code and breaks it down into tokens (the smallest units of syntax). It helps in identifying keywords, operators, variables, and constants in the program.

---

111. What is Recursive Descent Parsing?

Answer:
Recursive descent parsing is a top-down parsing technique where each non-terminal in the grammar is represented by a recursive function. It is simple and effective for grammars that do not cause ambiguities but can be inefficient for complex grammars.

---

112. What is Lexical Analysis?

Answer:
Lexical analysis is the first phase of the compiler, where the source code is converted into tokens (keywords, variables, operators). It also removes unnecessary whitespace and comments from the program to make it ready for parsing.

---

113. What is the Difference Between Bottom-Up and Top-Down Parsing?

Answer:

- **Top-Down Parsing:** Starts with the start symbol and tries to match input tokens to generate the parse tree.
- **Bottom-Up Parsing:** Starts with the input symbols and tries to build the parse tree up to the start symbol.

---

114. What is a Shift-Reduce Conflict?

Answer:
A **shift-reduce conflict** occurs in bottom-up parsing when the parser cannot decide whether to shift the next token onto the stack or to reduce the symbols on the stack using a grammar rule. This type of ambiguity can be resolved with additional lookahead tokens.

---

115. What is the Purpose of Error Handling in a Compiler?

Answer:
Error handling in a compiler involves detecting and reporting errors during lexical analysis, parsing, semantic analysis, and code generation. It ensures that the compiler can gracefully handle incorrect or invalid input without crashing.

---

116. What is the Difference Between a Syntax Tree and an Abstract Syntax Tree (AST)?

Answer:

- **Syntax Tree:** A detailed representation of the syntactic structure of a program, including all syntax rules.
- **Abstract Syntax Tree (AST):** A simplified version of the syntax tree, omitting unnecessary nodes like parentheses and intermediate grammar rules. The AST focuses on the essential elements of the program.

---

117. What is a Grammar Conflict in Compiler Design?

Answer:
A **grammar conflict** arises when a grammar rule cannot be applied unambiguously in a given context. It typically leads to issues like shift-reduce conflicts or reduce-reduce conflicts during parsing, making it difficult to correctly parse the code.

---

118. What is an LR Parser?**

Answer:
An **LR parser** is a bottom-up parser used for efficient syntax analysis. It reads the input from left to right and constructs the rightmost derivation. It is powerful enough to handle a large subset of context-free grammars.

---

119. What are Error Recovery Techniques in Parsing?

Answer:
Error recovery in parsers helps the parser continue processing the input after detecting errors. Common techniques include:

1. **Panic Mode:** Skipping over input until a synchronization point is found.
2. **Phrase Level Recovery:** Using known grammar rules to recover from common syntax errors.
3. **Error Productions:** Adding special error rules to the grammar to handle expected mistakes.

---

120. What is Code Emission in a Compiler?

Answer:
Code emission refers to the generation of the final machine or intermediate code after semantic analysis. The code emitter maps the semantic operations to specific machine instructions or low-level intermediate code.