🔶 **LEXICAL ANALYSIS**

1. **Q: What is a language processor?**
   A: A language processor is a software that translates or processes programs written in programming languages. Example: Compiler, Interpreter.

2. **Q: What is a compiler?**
   A: A compiler is a program that translates the entire source code into machine code at once.

3. **Q: What is the structure of a compiler?**
   A: A compiler has phases like Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Code Optimization, and Code Generation.

4. **Q: What is the role of the lexical analyzer?**
   A: It breaks the source code into tokens like keywords, identifiers, operators, etc.

5. **Q: What is a token?**
   A: A token is a single meaningful element in a program, like `int`, `x`, `=`, `5`.

6. **Q: What is bootstrapping in compilers?**
   A: Bootstrapping is writing a compiler in the same programming language that it compiles.

7. **Q: What is input buffering?**
   A: It is used to speed up reading the input characters from the source code.

8. **Q: What is a Lexical Analyzer Generator (LEX)?**
   A: LEX is a tool used to automatically create lexical analyzers from regular expressions.

9. **Q: What is a regular expression?**
   A: A regular expression is a pattern used to describe sets of strings, like identifiers or numbers.

10. **Q: What is finite automata?**
   A: Finite automata is a simple machine used to recognize patterns like tokens. It has states and transitions.

11. **Q: How are regular expressions related to finite automata?**
   A: Regular expressions can be converted into finite automata for pattern matching.

12. **Q: What is the difference between DFA and NFA?**
   A: DFA has one unique path for each input symbol, NFA can have multiple paths or none.

13. **Q: What is the design goal of a lexical analyzer?**
   A: To recognize valid tokens efficiently and pass them to the parser.

🔶 **SYNTAX ANALYSIS**

14. **Q: What is the role of the parser?**
   A: The parser checks whether the token sequence follows the grammar of the language.

15. **Q: What is a context-free grammar (CFG)?**
   A: CFG is a set of rules used to define the structure of valid strings in a language.

16. **Q: What is a derivation?**
   A: Derivation is the process of applying grammar rules to generate strings.

17. **Q: What is a parse tree?**
   A: A parse tree shows how a string is derived using grammar rules. Root is the start symbol.

18. **Q: What is ambiguity in grammar?**
   A: A grammar is ambiguous if one string can have more than one parse tree.

19. **Q: What is left recursion?**
   A: Left recursion is when a non-terminal calls itself on the left side, like A → Aα. It can cause infinite loops in parsers.

20. **Q: What is left factoring?**
   A: Left factoring is rewriting a grammar to remove common prefixes, helping predictive parsers.

Here are **20 important viva questions** from **UNIT II: Parsing** (Top Down & Bottom Up) with **very simple and easy-to-understand answers**:

🔷 **TOP DOWN PARSING**

1. **Q: What is top-down parsing?**
   A: It's a parsing method that starts from the start symbol and tries to reach the input string using grammar rules.

2. **Q: What are the preprocessing steps in top-down parsing?**
   A: Remove left recursion and perform left factoring.

3. **Q: What is backtracking in parsing?**
   A: Trying different grammar rules when the first choice doesn't lead to a match. It goes back and tries another path.

4. **Q: What is recursive descent parsing?**
   A: A top-down parser where each non-terminal has a recursive function to parse it.

5. **Q: Why is left recursion a problem in top-down parsing?**
   A: It causes infinite recursion and makes the parser stuck.

6. **Q: What is LL(1) grammar?**
   A: A grammar that can be parsed from Left to right, producing a Leftmost derivation, using 1 lookahead symbol.

7. **Q: What is FIRST set?**
   A: It tells which terminals can appear at the beginning of a string derived from a non-terminal.

8. **Q: What is FOLLOW set?**
   A: It tells which terminals can appear right after a non-terminal in some derivation.

9. **Q: What is non-recursive predictive parsing?**
   A: A top-down parser that uses a stack and a parsing table (no recursion or backtracking).

10. **Q: How is error handled in predictive parsing?**

```
    **A:** By using special entries like "synch" in the parsing table or
skipping unexpected symbols.

---

### 🔷 **BOTTOM UP PARSING**

11. **Q: What is bottom-up parsing?**
    **A:** It starts from the input string and tries to reduce it to the start
symbol using grammar rules in reverse.

12. **Q: What is the main difference between LL and LR parsers?**
    **A:** LL is top-down, works left to right; LR is bottom-up, works left to
right and produces rightmost derivation in reverse.

13. **Q: What are types of LR parsers?**
    **A:** SLR, CLR (canonical LR), and LALR.

14. **Q: What is shift-reduce parsing?**
    **A:** It uses a stack to shift input and reduce it by grammar rules until
the start symbol is reached.

15. **Q: What is an SLR parser?**
    **A:** Simple LR parser that uses FOLLOW sets to decide reductions.

16. **Q: How to construct SLR parsing table?**
    **A:** Create states (items), compute ACTION and GOTO tables using DFA of
items, use FOLLOW sets.

17. **Q: What is a CLR(1) parser?**
    **A:** Canonical LR parser using lookahead symbols to decide shift/reduce
actions more accurately.

18. **Q: What is an LALR parser?**
    **A:** Look-Ahead LR parser, a simplified version of CLR with fewer states,
widely used in practice.

19. **Q: What is the dangling else problem?**
    **A:** It's an ambiguity in nested if-else statements where it's unclear
which `if` an `else` belongs to.

20. **Q: How is error recovery handled in LR parsing?**
    **A:** By using panic mode (skipping input) or inserting error entries in
the parsing table.


---
### 🧠 **Syntax Directed Translation**
---
**1. What is Syntax Directed Translation (SDT)?**
👉 It is a method of translating programming languages using grammar rules along
with semantic actions.
---
**2. What is a Syntax-Directed Definition (SDD)?**
👉 An SDD is a context-free grammar with rules that have attributes and semantic
rules to define the meaning.
---
**3. What are the types of attributes in SDD?**
👉
- **Synthesized attributes**: Computed from children nodes.
- **Inherited attributes**: Passed from parent or siblings.
---
**4. What is the difference between SDD and SDT?**
👉
```

- **SDD**: Uses rules and attributes.
- **SDT**: Adds actions (code) to grammar rules.
---
5. What is L-Attributed SDD?
👉 An SDD is L-attributed if its attributes can be evaluated in left-to-right order (used in top-down parsing).
---
6. What is S-attributed SDD?
👉 An SDD that uses only synthesized attributes.
---
7. What is an Evaluation Order for SDDs?
👉 It's the order in which attribute values are computed so all needed values are available.
---
8. What are applications of Syntax Directed Translation?
👉
- Type checking
- Intermediate code generation
- Symbol table construction
- Expression evaluation
---
9. What is a Syntax Directed Translation Scheme (SDTS)?
👉 It's a grammar with semantic actions inserted within productions for translation.
---
10. How are L-attributed SDDs implemented?
👉 They are implemented using a recursive-descent parser by evaluating attributes in a left-to-right manner.

💻 **Intermediate Code Generation**
---
11. What is Intermediate Code in a compiler?
👉 It is a code between source code and machine code, used for optimization and portability.
---
12. What is a Three Address Code (TAC)?
👉 It's an intermediate code with at most 3 addresses (operands) per instruction like: `t1 = a + b`
---
13. What is a syntax tree?
👉 A tree that shows the structure of a program with operations as nodes and operands as children.
---
14. What is a DAG (Directed Acyclic Graph)?
👉 A compact version of a syntax tree that avoids repeated sub-expressions.
---
15. What are the types of instructions in Three Address Code?
👉
- Assignment (e.g., `x = y + z`)
- Conditional jump
- Unconditional jump
- Procedure calls
- Parameter passing
---
16. What is type checking?
👉 Ensuring operands in an expression have compatible data types.

---

17. What are declarations in a compiler?
👉 Statements that tell the compiler about variable names and their types.
---
18. What is backpatching?
👉 It's a technique to handle jump addresses that are not known until later

(used in control flow like `if`, `while`).
---
19. What is the use of a symbol table?
👉 It stores variable names, types, scope, and other info during compilation.

20. What is control flow in intermediate code?
👉 It refers to the way execution moves through statements (like `if`, `while`, `goto`).

✅ **UNIT IV – CODE OPTIMIZATION**

1. **Q: What is code optimization?**
   A: Code optimization is the process of improving the intermediate code to make it run faster and take less memory.

2. **Q: What are the main goals of code optimization?**
   A: To reduce execution time and memory usage without changing the program's output.

3. **Q: What is a basic block?**
   A: A basic block is a sequence of instructions with no branches in except the first and no branches out except the last.

4. **Q: What are the principle sources of optimization?**
   A: Redundant computations, loop optimization, algebraic simplification, and eliminating dead code.

5. **Q: What is common subexpression elimination?**
   A: It removes repeated calculations and stores the result for reuse.

6. **Q: What is dead code?**
   A: Code that never affects the output and can be removed.

7. **Q: What is constant folding?**
   A: Replacing constant expressions with their computed value at compile time.

8. **Q: What is strength reduction?**
   A: Replacing costly operations like multiplication with cheaper ones like addition.

9. **Q: What is loop unrolling?**
   A: Repeating loop body multiple times to reduce the number of iterations and jumps.

10. **Q: What is loop invariant code motion?**
   A: Moving code that doesn't change inside a loop to outside the loop.

11. **Q: What is peephole optimization?**
   A: Optimizing small sets of instructions (like 2-3 lines) to improve performance.

12. **Q: Give an example of peephole optimization.**
   A: Replacing `x = x * 2` with `x = x + x`.

13. **Q: What is data flow analysis?**
   A: It checks how data moves through the program to find optimization opportunities.

14. **Q: What is copy propagation?**
   A: Replacing variables with their assigned value when possible.

15. **Q: What is a flow graph?**
   A: A diagram showing how control flows between basic blocks.

16. **Q: What is a leader in code optimization?**
   A: The first instruction in a basic block.

17. **Q: What is algebraic simplification?**
   A: Simplifying expressions like replacing `x*1` with `x`.

18. **Q: What is code motion?**
   A: Moving code to a better position to avoid repeated execution.

19. **Q: What is loop fission?**
   A: Splitting a loop into two to improve cache performance.

20. **Q: What is loop fusion?**
   A: Combining two loops into one to reduce loop overhead.

✅ **UNIT V – RUN TIME ENVIRONMENTS & CODE GENERATION**

21. **Q: What is a run-time environment?**
   A: It's the system where the program executes and manages memory during execution.

22. **Q: What is storage organization?**
   A: It is the arrangement of data in memory: stack, heap, static, and code segments.

23. **Q: What is the stack used for in runtime?**
   A: It stores function calls, parameters, local variables, and return addresses.

24. **Q: What is an activation record?**
   A: It’s a block of memory in the stack containing function info like parameters and return value.

25. **Q: What is the heap used for?**
   A: It's used for dynamic memory allocation during program execution.

26. **Q: What is static memory allocation?**
   A: Memory is allocated before program execution starts and stays fixed.

27. **Q: What is dynamic memory allocation?**
   A: Memory is allocated at runtime as needed.

28. **Q: What is a procedure call?**
   A: It's a function or method call in a program.

29. **Q: What is a display in runtime environments?**
   A: It helps manage access to non-local variables in nested functions.

30. **Q: What are the main issues in designing a code generator?**
   A: Instruction selection, register allocation, and efficient use of CPU instructions.

31. **Q: What is object code?**
   A: Code generated by a compiler that the machine can understand.

32. **Q: What are the forms of object code?**
   A: Binary machine code, assembly code, or intermediate low-level code.

33. **Q: What is register allocation?**
   A: Assigning variables to CPU registers to make execution faster.

34. **Q: What is register spilling?**
   A: When there are not enough registers, some variables are stored in memory.

35. **Q: What is instruction selection?**
   A: Choosing the best machine instruction for each operation.

36. **Q: What is instruction scheduling?**
   A: Arranging instructions to avoid pipeline delays and make execution faster.

37. **Q: What is code generation?**
   A: Translating intermediate code into machine-level or assembly code.

38. **Q: What is the difference between intermediate code and object code?**
   A: Intermediate code is platform-independent; object code is machine-specific.

39. **Q: What is the goal of a code generator?**
   A: To generate fast and efficient machine code with correct behavior.

40. **Q: What are calling conventions?**
   A: Rules that define how functions receive parameters and return values.