Modern Compiler Design: Practice Questions.

1. What is the analysis-synthesis model of compilation, and how does it contribute to the understanding of compiler structure?
2. Enumerate and explain the various phases of a compiler in the order of their execution.
3. Describe the role and significance of lexical analysis in the compilation process.
4. What is the primary purpose of syntax analysis in the compilation process, and how does it differ from lexical analysis?
5. Explain the concept of intermediate code generation and its importance in the compilation process.
6. How does optimization contribute to the efficiency of a compiler? Provide examples of common optimization techniques.
7. Detail the process of code generation and its role in producing executable code from intermediate code.
8. Discuss the challenges and strategies involved in error handling within a compiler.
9. Compare and contrast lexical analysis and syntax analysis, highlighting their respective functions and outputs.
10. Explore the concept of intermediate code and its advantages in the context of compiler design.
11. Provide an overview of optimization techniques used in compilers and explain how they enhance program performance.
12. Discuss the significance of code generation in the overall compilation process, emphasizing its role in creating executable code.
13. Explain the role of the lexical analyzer in the compilation process and its significance in program understanding.
14. Define and differentiate between a token, lexeme, and pattern in the context of lexical analysis.
15. What challenges and difficulties can arise in the process of lexical analysis, and how can they be addressed?
16. Discuss the importance of error reporting in lexical analysis and its impact on the overall compilation process.
17. Provide an overview of regular expressions and their role in specifying patterns in lexical analysis.
18. Describe the concept of finite automata and its relevance to lexical analysis.
19. Illustrate the process of transitioning from regular expressions to finite automata, emphasizing the key steps involved.
20. Explain the construction and representation of transition diagrams in the context of finite automata.
21. Demonstrate the conversion of a regular expression into a deterministic finite automaton (DFA) with a step-by-step example.
    a. $(0|1)^*0(0|1)^+$
    b. $(0|1)^*01^+$
    c. $1^*(0|1)^+$
22. Discuss the significance of minimizing the number of states in a DFA and the techniques involved in achieving this.
23. How does a lexical analyzer use finite automata to recognize tokens in a programming language? Provide insights into the process.

24. Explore the concept of lexemes and their relationship with patterns in the lexical analysis phase.
25. Compare and contrast non-deterministic finite automata (NFA) and deterministic finite automata (DFA) in lexical analysis.
26. Analyze the transition from regular expressions to transition diagrams, emphasizing the connections between the two.
27. Define context-free grammars and explain their significance in the syntactic specification of programming languages. Provide an example to illustrate your explanation.
28. Describe the concept of derivations in the context of context-free grammars. How do derivations contribute to generating valid syntactic structures for programming languages?
29. Explain the role of parse trees in representing the syntactic structure of programming language constructs. Provide a step-by-step example of constructing a parse tree for a simple grammar and the given string id + id * id.

    S -> E
    E -> E + T | T
    T -> T * F | F
    F -> (E) | id

30. What is the role of context-free grammars in syntax analysis, and how do they contribute to defining the syntax of programming languages?
31. Explain the concepts of derivation and parse trees in the context of syntax analysis. Provide an example to illustrate these concepts.
32. Discuss the challenges associated with ambiguity in context-free grammars. How does ambiguity impact syntax analysis, and what strategies can be employed to address it?
33. Explore the principles of top-down parsing. What is the significance of recursive descent parsing, and how does it work?
34. Explain the concept of predictive parsing. How does it differ from other parsing techniques, and what advantages does it offer?
35. Define and discuss the principles of bottom-up parsing. How do bottom-up parsers work, and what are their advantages in syntax analysis?
36. What are operator precedence grammars, and how do they handle the parsing of expressions with different levels of precedence?
37. Define LR parsers and discuss their role in syntax analysis. How do LR parsers handle context-free grammars, and what makes them advantageous?
38. Compare and contrast top-down and bottom-up parsing techniques, highlighting their strengths and weaknesses.
39. How does recursive descent parsing handle left-recursive productions in context-free grammars? Provide an example to illustrate.
40. Explain the concept of LR(1) parsing. How does LR(1) parsing address some of the limitations of LR parsing?
41. Define syntax-directed definitions. How do they enhance the specification of attributes in the context of syntax analysis?
42. Differentiate between inherited and synthesized attributes in the context of syntax-directed definitions. Provide examples to illustrate each type of attribute.
43. Explain the concept of a dependency graph in the context of syntax-directed definitions. How does it represent relationships between attributes?

44. Discuss the importance of evaluation order in syntax-directed definitions. How does the order of attribute evaluation impact the overall process?
45. Compare and contrast bottom-up and top-down evaluation of attributes. In what scenarios is each approach more suitable?
46. Provide a step-by-step example of bottom-up evaluation of attributes in a syntax-directed definition. Illustrate the process with a small grammar and associated attributes.
47. Provide a step-by-step example of top-down evaluation of attributes in a syntax-directed definition. Use a simple grammar to demonstrate the process.
48. Explain how synthesized attributes can be evaluated in a bottom-up fashion. Provide an example to demonstrate the evaluation process.
49. Discuss the concept of type expressions in the context of type checking. How are type expressions used to represent and verify data types in programs?
50. Explore the importance of type checking in the compilation process. How does type checking contribute to the detection of potential errors in programs?
51. Explain the importance of a symbol table in the context of programming languages and compilers. How does it contribute to the compilation process?
52. Discuss the essential contents of a symbol table. What information is typically stored for each symbol, and how is this information used during compilation?
53. Explore the various data structures that can be used for implementing symbol tables. Compare and contrast the advantages and disadvantages of different data structures in this context.
54. Define the Run-Time System and explain its significance in the execution of programs. How does it contribute to managing program execution at runtime?
55. Describe the organization of storage in the Run-Time System. How is memory structured to accommodate variables, data, and program instructions during program execution?
56. Explain the concept of an activation tree in the context of the Run-Time System. How does it represent the dynamic execution of a program and the invocation of procedures?
57. Define an activation record and discuss its role in managing the execution of procedures. What information does an activation record typically contain?
58. Explore different methods of parameter passing in the Run-Time System. Discuss the advantages and disadvantages of techniques such as pass-by-value and pass-by-reference.
59. Explain the role of the symbol table in the Run-Time System. How does it aid in the dynamic management of identifiers and their associated information during program execution?
60. Discuss the challenges associated with dynamic storage allocation in the Run-Time System. What are common techniques for allocating and deallocating memory dynamically?
61. Provide examples to illustrate the process of activation record creation and management during the execution of nested procedures.
62. Explore the impact of dynamic storage allocation on program efficiency and memory utilization. How do different allocation strategies affect program performance?
63. Explain the concept of Intermediate Representations (IR) in the context of compiler design.
64. Discuss the importance of intermediate code in the compilation process. How does it facilitate the translation from source code to machine code?

65. Describe the translation process of declarations into intermediate code. Include examples to illustrate the steps involved.
66. Explain the intermediate code generation process for assignments. Provide a step-by-step example for clarity.
67. Discuss the translation of control flow constructs (e.g., if statements, loops) into intermediate code. How are these high-level constructs represented in the intermediate code?
68. Explore the generation of intermediate code for Boolean expressions. Provide examples demonstrating how logical expressions are translated.
69. Explain the procedure for translating procedure calls into intermediate code. How are parameters passed, and how is the control flow managed during procedure invocation?
70. Discuss the challenges and considerations in the implementation of intermediate code generation. How can efficiency and accuracy be optimized?
71. Describe the role of temporaries in intermediate code. How are they generated, managed, and utilized during the translation process?
72. Discuss the advantages of using Three-Address Code as an intermediate representation. Provide examples to illustrate its structure and benefits.
73. Explain the concept of quadruples in intermediate code. How are they used to represent operations in a program? Provide a detailed example.
74. Discuss the generation of intermediate code for array operations. Include examples to demonstrate how array indexing and manipulation are translated.
75. Explore the concept of abstract syntax trees (ASTs) and their relationship with intermediate code. How is information passed between these two representations during compilation?
76. Explain the concept of basic blocks in the context of intermediate code generation. How are they identified and utilized in control flow translation?
77. Discuss the considerations and techniques for optimizing intermediate code. Provide examples to illustrate common optimization strategies during this phase of compilation.
78. Given the C statements. Provide the corresponding three-address code. Explain each step of the code generation process.
    a. e = a + b * c
    b. **result = x * y - z / w**
79. Draw the DAG for the expressions. Identify and eliminate common subexpressions in the DAG.
    a. q = a * b + a * c
    b. **result = x * y - z + x * y**
    c. **output = a * b + c * d**
80. Explain the concept of loop unrolling and how it improves program performance.
81. Provide an example code snippet with a loop and demonstrate the process of loop unrolling.
82. Define loop fusion and its purpose in optimizing code.
83. Provide an example with two separate loops and demonstrate the process of loop fusion.
84. Discuss scenarios where loop fusion is advantageous and situations where it might not be suitable.
85. Explain the concept of loop tiling (blocking) and its impact on cache locality.
86. Provide a code snippet with a loop and demonstrate how loop tiling is applied.
87. Define loop-invariant code motion(LICM) and its role in optimizing loops.
88. Provide an example loop with invariant and non-invariant code and demonstrate LICM.

89. Discuss the benefits and potential drawbacks of applying LICM. Explain how LICM contributes to reducing redundant computations.
90. Discuss the major issues involved in code generation during the compilation process.
91. Explain how code generation is influenced by the target machine architecture.
92. Outline the essential steps performed by a simple code generator.
93. Provide a basic code generation example for a simple expression.
94. Explain the importance of register allocation in code generation. Discuss different strategies for register allocation.
95. Define peephole optimization and its purpose in improving generated code. Provide an example where peephole optimization can be applied.
96. Discuss the impact of peephole optimization on code size and execution speed.
97. Explain the process of instruction selection in code generation. Discuss different techniques for choosing appropriate machine instructions.
98. Find the LR(1) item closure for a simple grammar. Augment the grammar if needed.

    S' -> S
    S -> E
    E -> E + T
    E -> T
    T -> T * F
    T -> F
    F -> (E)
    F -> id

    Find the
    a. Closure for [S' -> .S, $].
    b. Closure for [E -> .E + T, $]
    c. Closure for [T -> .T * F, $]
    d. Closure for [F -> .(E), $]
    e. Closure for [E -> .T, $]
99. Numerical/Solving problems based on
    a. Generating Regular Expressions.
    b. Drawing NFA from a give regular expressions.
    c. Minimization of finite Automata.
    d. Parsing using LL(1) Parsing Table
    e. Parsing using SPM
    f. Parsing Using OPM
    g. Parsing using LR parsing Table
100. Consider the following context-free grammars (G1, G2, G3, G4, G5)
    a. $S \rightarrow aSb \mid \varepsilon$
    b. $S \rightarrow aS \mid bS \mid \varepsilon$
    c. $S \rightarrow 0S1 \mid 1S0 \mid \varepsilon$
    d. $S \rightarrow AB$
       $A \rightarrow aA \mid \varepsilon$
       $B \rightarrow bB \mid \varepsilon$

    For each grammar (G1 to G5), compute and provide the language L(G) generated by the grammar.