Compiler Design Important Questions PDF

The most important Compiler Design questions are the phases of a compiler, lexical analysis and LEX, left recursion and left factoring, FIRST and FOLLOW sets, LL(1) parsing tables, operator precedence parsing, SLR, CLR and LALR parsing tables, syntax-directed definitions, three-address code (quadruples and triples), symbol tables, runtime storage, basic blocks and flow graphs, DAGs and code optimisation.

Key takeaways

- FIRST and FOLLOW, LL(1) tables and LR parsing tables are the highest-scoring numericals.
- Draw the phases of a compiler with the output of each phase for a sample statement.
- Three-address code and DAG construction are short, reliable long-answer topics.
- These are practice questions, not past papers; check your own syllabus and official previous papers.

About these important questions

This page lists the most important Compiler Design questions for B.Tech CSE and IT exams, arranged unit by unit. The units follow the standard syllabus that most Indian technical universities share, so the order may differ slightly from yours.

Please note: these are practice questions based on common syllabus topics. They are not verified past papers, and we do not claim that any question will appear in your exam. Check your own university syllabus first, then solve the official previous year papers. AKTU students can download them from the AKTU old question paper archive.

Unit 1: Introduction and Lexical Analysis

Covers compiler phases, passes, bootstrapping and lexical analysis.

Short questions (2 marks)

- Differentiate between a compiler and an interpreter.
- Define a token, a pattern and a lexeme.
- What is bootstrapping?

Long questions

1. Explain the phases of a compiler with a diagram, and show the output of each phase for `position = initial + rate * 60`.

2. Explain the role of the lexical analyser and input buffering with sentinels.
3. Explain the structure of a LEX program with an example.
4. Construct a DFA directly from a regular expression such as (a | b)*abb.
5. Compare single-pass and multi-pass compilers, and explain cross compilers.

Unit 2: Syntax Analysis: Top-Down and Operator Precedence

Covers CFGs for parsing, top-down parsing and operator precedence parsing.

Short questions (2 marks)

- What is left recursion? Why must it be removed for top-down parsing?
- What is left factoring?
- When is a grammar LL(1)?

Long questions

1. Remove left recursion from E → E + T | T, T → T * F | F, F → (E) | id.
2. Find FIRST and FOLLOW for the resulting grammar and build the LL(1) parsing table.
3. Parse the input `id + id * id` using the LL(1) table, showing the stack, input and action.
4. Explain recursive descent parsing with an example.
5. Build an operator precedence table and parse a string.

Unit 3: Bottom-Up Parsing and Syntax-Directed Translation

Covers shift-reduce parsing, LR parsers and syntax-directed definitions.

Short questions (2 marks)

- What is a handle?
- Name the conflicts that can occur in shift-reduce parsing.
- Differentiate between synthesised and inherited attributes.

Long questions

1. Construct the canonical LR(0) items and the SLR parsing table for a given grammar.
2. Construct the CLR (LR(1)) and LALR parsing tables for a grammar such as S → CC, C → cC | d.
3. Compare SLR, CLR and LALR parsers.
4. Write a syntax-directed definition for a desk calculator and draw an annotated parse tree.
5. Explain S-attributed and L-attributed definitions.
6. Explain YACC and how it works with LEX.

Unit 4: Intermediate Code, Symbol Table and Runtime Environment

Covers intermediate representations, symbol tables and storage organisation.

Short questions (2 marks)

- What is three-address code?
- Differentiate between quadruples and triples.

- What is an activation record?

Long questions

1. Write three-address code for an expression such as a = b * -c + b * -c and represent it as quadruples, triples and indirect triples.
2. Generate three-address code for an if-else statement and a while loop using backpatching.
3. Explain symbol table organisation using lists, trees and hash tables.
4. Explain the contents of an activation record with a diagram.
5. Explain static, stack and heap storage allocation.
6. Explain parameter passing methods: call by value, reference and name.

Unit 5: Code Optimisation and Code Generation

Covers basic blocks, flow graphs, DAGs, optimisations and code generation.

Short questions (2 marks)

- What is a basic block?
- Define dead code elimination.
- What is peephole optimisation?

Long questions

1. Partition three-address code into basic blocks and draw the flow graph.
2. Construct a DAG for a basic block and explain its uses.
3. Explain local and loop optimisations: common subexpression elimination, constant folding, code motion, strength reduction and induction variable elimination.
4. Explain data flow analysis: reaching definitions and live variable analysis.
5. Explain the issues in the design of a code generator.
6. Explain peephole optimisation with examples.

Sample answer outline: FIRST and FOLLOW sets

For the grammar E → TE', E' → +TE' | ε, T → FT', T' → *FT' | ε, F → (E) | id, answer like this:

1. **State the rules.** FIRST(X) is the set of terminals that can begin a string derived from X. FOLLOW(A) is the set of terminals that can appear right after A in some sentential form, and $ is in FOLLOW of the start symbol.
2. **Compute FIRST bottom-up.** FIRST(F) = { (, id }. FIRST(T) = FIRST(F). FIRST(E) = FIRST(T). FIRST(E') = { +, ε }. FIRST(T') = { *, ε }.
3. **Compute FOLLOW.** FOLLOW(E) = {), $ }. FOLLOW(E') = FOLLOW(E). FOLLOW(T) = { +,), $ }. FOLLOW(T') = FOLLOW(T). FOLLOW(F) = { *, +,), $ }.
4. **Show it in a table.** One row per non-terminal with FIRST and FOLLOW columns.
5. **Use it.** Each production A → α goes in table cell [A, a] for every a in FIRST(α). If ε is in FIRST(α), also add it for every b in FOLLOW(A). If no cell has two entries, the grammar is LL(1).

Marks weightage and how to write answers

A common pattern, used by AKTU and many other universities, has a compulsory short-answer section that covers every unit, followed by long-answer questions with a choice inside each unit. Marks per question change between universities and years, so confirm the scheme on a recent official paper.

- **Short answers (about 2 marks):** a crisp definition, one example or diagram, and nothing more. Three to five lines is enough.
- **Long answers (7 to 10 marks):** start with a definition, add a labelled diagram or table, explain the steps, and finish with an example, advantages or a comparison.
- For parsing tables, draw them with ruled rows and columns. A table that is hard to read loses marks even when correct.
- Always show a parse of a sample string after building a table. It proves the table works.

A 7-day revision plan for Compiler Design

1. **Days 1 to 5:** one unit a day. Read your notes, then write the long answers for that unit from memory and check them against your notes.
2. **Day 6:** redo left recursion removal, FIRST and FOLLOW, one LL(1) table, one SLR and one LALR table, and a DAG.
3. **Day 7:** solve one full official previous year paper in three hours, then check which units cost you time.

Keep a single sheet of formulas, definitions and diagrams for the last hour before the exam.

Frequently asked questions

Which Compiler Design topics are most important?

Phases of a compiler, FIRST and FOLLOW, LL(1) parsing, SLR, CLR and LALR parsing, syntax-directed translation, three-address code, basic blocks and DAGs, and code optimisation are asked most often.

How do I avoid mistakes in LR parsing tables?

Number every production, build the item sets carefully with closure and goto, and draw the DFA of item sets before filling the table. For SLR, use FOLLOW sets for reduce entries; for CLR, use the lookaheads.

Is Compiler Design useful for a software career?

Yes. Parsing, code generation and optimisation ideas help when you build interpreters, query engines, linters or developer tools.

Are these Compiler Design questions from previous year papers?

No. They are practice questions built from topics that appear in most B.Tech Compiler Design syllabi. Use them with your own university's official previous papers, for example the AKTU archive at https://aktu.ac.in/old-question-paper.html.

Is this PDF free?

Yes. Fill in the short form on this page and the PDF downloads right away.

Sources and further reading

- AKTU: old question papers (official archive)
- Stanford: Compilers course materials (CS143)
- NPTEL: online courses from IITs (search for Compiler Design)

About this PDF

These are practice questions for revision, prepared by the Uncodemy trainer team from common B.Tech syllabus topics. They are not past papers, a certificate or a promise of marks. Spot an error? Write to info@uncodemy.com and the team will review it.

More free resources and courses: uncodemy.com | info@uncodemy.com | +91 98183 66550