

AuxFlow — Review 1 Viva Preparation

Static Analysis for Multi-Pass Document Compilation · Upayan Mazumder (24BDS0367) · Compiler Design Lab

1. Why is this a compiler project?

- Full compiler front-end: lexer, parser, AST, symbol tables, an IR, and dataflow/graph analysis.
- Only the **target** differs — it emits an analysis report, not machine code.

2. Why not simply run compilation multiple times?

- Running shows *what happened*; it does not *explain* the dependency structure.
- AuxFlow reasons statically from source — which references depend on which, and where feedback is risky.
- Complements running the engine; it is not a replacement.

3. What compiler phases are involved?

- Lexical → syntax → AST → semantic (symbol tables) → IR generation.
- Then static / dataflow analysis → graph analysis (Tarjan SCC) → classification & report.

4. Why create an intermediate representation?

- Separates dependency reasoning from surface syntax.
- Keeps only convergence-relevant facts and edges; analysed in its own terms.
- Analyses never read back into the AST → self-contained and reusable.

5. Why use graphs?

- Cross-references form a dependency graph; the multi-pass fixpoint is naturally defined over it.
- Standard tools apply: traversal, Tarjan SCC for cycles, fixpoint over the condensation.

6. What are the limitations?

- Defined only over a **restricted subset** — arbitrary reference analysis is undecidable; outside → opaque.
- Abstracts fine-grained layout effects, so some real behaviours are unseen — **declared**, not hidden.
- Conservative: may over-flag or answer *unknown*; never claims certainty.

7. What makes this feasible for one student?

- Standard compiler construction, built phase by phase, each independently testable.
- No third-party core dependencies; the analysis runs with no typesetting engine installed.
- The parser is the heaviest phase — the restricted subset bounds its scope.

8. What is the expected output?

- An inspectable **dependency graph** of the document's cross-references.
- A conservative, convergence-related **classification**.
- Explainable, **source-located diagnostics** — names the dependency and its location, not just “cycle detected”.

Quick backups (likely follow-ups)

- **Is this AI/ML?** No — deterministic static analysis; no model, no training data.
- **Does a cycle mean failure?** No — ordinary documents (e.g. a contents list) are cyclic and settle; only feedback where a value affects *how much content* is generated is a risk.
- **Biggest risk?** The parser — mitigated by the restricted subset and a fail-safe rule for unsupported constructs.
- **One-line contribution?** A compiler dependency analysis applied to multi-pass compilation, with explainable, conservative output.