EECS4302 Compilers and Interpreters
Fall 2026

Digest of Lecture 4, September 22

Compilation Phases, Semantic Preservation, and AST Optimization

Jackie Wang

How to use this Digest

Use this digest as a **roadmap and checklist**. It highlights the main ideas, important examples, common pitfalls, practice targets, and where to return in the course materials. It does **not** replace attending or reviewing the session, using the recording when needed, studying Jackie's annotated iPad notes and other official course materials, completing assigned work, or practising independently. Use it to identify what you understand, what you need to revisit, and what you should be able to do on your own.

If time is limited, start with the sections most relevant to your current review and return to the rest later; selective use is better than skipping the digest because it looks long.

This is an AI-assisted study roadmap reviewed by Jackie and grounded in the session transcript and Jackie's annotated iPad notes. Like any course document, it may still contain an error or omission. If something seems technically doubtful or inconsistent with the original course materials, return to those materials and ask Jackie for clarification rather than treating the digest as final authority.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 8

Where to Review 9

Closing Reminder 10

Main Ideas

- A compiler can contain several semantics-preserving stages. Jackie extended the front-end/IR/back-end picture by inserting an **optimizer**: the front end produces **IR1**, the optimizer transforms it into **IR2**, and the back end produces the target program. Correctness depends on preserving meaning across every stage.
- The session organized compilation around three kinds of analysis: **lexical analysis** transforms a character sequence into a token sequence; **syntactic analysis** transforms that token sequence into a tree; and **semantic analysis** works on the tree to perform checks or transformations.
- Scanning and parsing are important but comparatively routine. Token patterns can be specified with regular-language machinery, and syntax can be specified with context-free grammars; Jackie emphasized that tools such as Antlr4 can automate much of these first two phases once those specifications are supplied.
- The while-loop example separated **syntactic correctness** from **semantic correctness**. Two programs can both match the grammar and produce parse trees while one still fails type checking because the same variable is declared twice in one scope.
- The optimizer example showed semantic analysis as an internal tree transformation. The expression $2 \times b \times c$ does not change across loop iterations, so the compiler can compute it once into a temporary value and replace the repeated subtree inside the loop.
- The final commuting diagram connected four artifacts: source program, initial AST, transformed AST, and optimized program. **Parsing** moves from program text to a tree; **pretty printing** moves from a tree back to concrete program text; the internal AST-to-AST transformation is hidden from the external user.
- Assignment 1 was announced for release on **September 23**. Jackie said it would use Java/JUnit and regular-language material, with a two-week completion window, and encouraged students to start reviewing the posted lexical-analysis and math-review materials rather than waiting for later lectures.

Roadmap

1. Assignment 1 and the transition into lexical analysis

Jackie began by locating the class in the semester plan. Lecture 4 completes the high-level compilation overview; the next class begins the more formal treatment of lexical analysis. Assignment 1 was to be posted on September 23, shortly after lunch. Jackie described it as Java/JUnit work connected to DFA/NFA-style regular-language material and regular expressions, with two weeks to complete the best attempt. The lexical-analysis slides and math-review slides were already posted, and Jackie explicitly encouraged students to begin self-study before Thursday rather than waiting.

2. Adding an optimizer to the compiler pipeline

Page 3 of Jackie's annotated iPad notes extends the earlier architecture to

$$\text{Source} \rightarrow \text{Front End} \rightarrow \mathrm{IR}_1 \rightarrow \text{Optimizer} \rightarrow \mathrm{IR}_2 \rightarrow \text{Back End} \rightarrow \text{Target}.$$

The optimizer is not limited to one particular notion of efficiency. Jackie discussed runtime improvement and also static structural improvement, using database normalization as an analogy. A real compiler can contain several optimizer passes rather than one box.

The central correctness questions were written directly on the diagram:

- **Q1:** Does the front end accurately represent the source program in IR_1?
- **Q2:** Does the optimizer preserve the semantics/meaning from IR_1 to IR_2?
- **Q3:** Does the back end accurately represent the optimized IR_2 in the target program?

Jackie's point was transitivity: if meaning is preserved at every link, the final target program can preserve the meaning of the original source even though the internal representation has changed. He noted that justification can involve testing or a reasoned argument rather than necessarily a formal proof.

3. Three analysis phases: lexical, syntactic, semantic

Pages 4–6 organize the compiler around three analyses. In **lexical analysis**, the input is a linear sequence of characters and the output is a linear sequence of tokens. In **syntactic analysis**, the input token sequence is linear but the output is a non-linear parse-tree/AST structure. In **semantic analysis**, the input is a tree and the output can be another AST or another form of analysis result. For the optimization case later in the lecture, the semantic phase is specifically an AST-to-AST transformation.

Jackie tied the first two stages to familiar tooling. A **scanner** performs lexical analysis; a **parser** performs syntactic analysis. Antlr4 can automate scanner/parser generation after token patterns and grammar rules have been specified. Jackie characterized these stages as routine compared with the semantic phase, where the compiler-specific engineering and transformation logic is concentrated.

4. Lexical analysis: characters, tokens, and delimiters

Page 4 develops a small Java-like `MyClass` example. The scanner reads the source from left to right as characters and recognizes token patterns. The first tokens shown are the keyword `class`, the identifier `MyClass`, and the left brace. Whitespace between them is treated as a meaningless/useless delimiter and is discarded rather than emitted as a token.

Jackie distinguished such delimiters from useful punctuation such as braces, brackets, commas, parentheses, and semicolons, which can themselves matter to the language. Token patterns are described using regular-language machinery. He previewed four notations that will be reviewed: DFA, NFA, ε-NFA, and regular expressions, together with DFA state minimization.

5. Syntactic analysis: tokens to a non-linear tree

The token sequence produced by the scanner becomes the parser's input. Jackie contrasted linear structures, where an interior element has a unique predecessor and successor, with the non-linear tree produced by syntactic analysis. The parser organizes tokens according to a context-free grammar.

At this overview stage, Jackie introduced two parsing directions: **top-down** and **bottom-up**. These algorithms will be developed later, and Assignment 3 is expected to involve implementing one of them.

6. Essay analogy and a recursive context-free grammar

Page 6 uses essay writing as an analogy: spelling corresponds to lexical analysis, grammatical sentence structure to syntactic analysis, and meaning/coherence to semantic analysis. A sentence can therefore be lexically correct but syntactically wrong, or syntactically correct while still having a problem of meaning.

Page 7 then sketches a recursive context-free grammar for a while loop. Tokens such as `while`, parentheses, braces, and semicolons appear as terminals, while structures such as the Boolean expression and implementation appear as non-terminals. The recursive implementation rule allows a sequence of instructions to grow. Jackie explicitly noted that the displayed grammar was only illustrative and would be treated more carefully in the formal parsing lectures.

7. Syntax can pass while semantic checking fails

Using the grammar, Jackie compared two while-loop programs. Both can be parsed and both therefore pass syntactic analysis. The second program, however, declares the same variable `i` twice in the same scope. That error is not a spelling or grammar error; it is detected by semantic analysis/type checking.

This example also answered a student question about phase boundaries. Jackie said type checking cannot simply be placed in lexical analysis, whose regular-language recognizers are doing a much more limited job. Some checks can be intertwined with parsing, but a separate semantic pass over the constructed tree is a natural place for richer checks.

8. Loop-invariant computation and AST-to-AST optimization

Pages 8–9 develop the main optimization example. The input loop repeatedly computes a product containing $2 \times b \times c$, while b and c remain unchanged between iterations. Jackie identified that subexpression as invariant. Recomputing it every time is unnecessary when the loop repeats many times.

The compiler first has an AST for the unoptimized program. The optimization then changes the tree: it inserts a new assignment before the loop, conceptually

$$\texttt{temp} := 2 \times b \times c,$$

and replaces the corresponding multiplication subtree inside the loop with a reference to `temp`. The important lesson is structural: optimization is carried out on the internal tree,

not by manually editing source text. Multiple AST-to-AST transformations can be chained before the final output stage.

9. The commuting diagram: parsing, transformation, and pretty printing

Page 10 connects the complete flow. The inefficient input program is **parsed** into its initial AST. Semantic analysis/optimization transforms that AST into an optimized AST. The optimized AST is then converted back into concrete program text by **pretty printing**.

From the external user's point of view, this is simply program-to-program compilation. The intermediate trees and transformations are internal implementation artifacts, even though they are the mechanism that makes the optimization possible.

Key Terms, Notation, and Structures

IR1 / IR2 Intermediate representations before and after the optimizer in the Lecture 4 pipeline.

Semantics preservation The requirement that each compiler stage preserve the meaning represented by its input. Lecture 4 organizes this as Q1, Q2, and Q3 across front end, optimizer, and back end.

Lexical analysis Transformation from a sequence of characters to a sequence of tokens. The scanner performs this stage.

Token A recognized lexical unit such as a keyword, identifier, number, or meaningful punctuation/delimiter.

Delimiter A character used around or between tokens. Jackie distinguished meaningless or useless delimiters such as ordinary whitespace in Java-like syntax. Other delimiters, such as braces and semicolons, are useful punctuation.

Regular language The formal setting used to describe token patterns in the lexical-analysis discussion. DFA, NFA, ε-NFA, and regular expressions were named as representations to review.

Syntactic analysis Transformation from the token sequence to a non-linear parse-tree/AST structure according to a context-free grammar. The parser performs this stage.

Context-free grammar (CFG) A recursive grammar with terminals, non-terminals, and production rules used to describe larger syntactic structure.

Top-down / bottom-up parsing The two parsing directions introduced for constructing a tree from the token sequence; their algorithms will be studied later.

Semantic analysis Analysis over the tree after parsing. Its result can be another AST or another form of result; examples in this lecture included type checking and AST-to-AST optimization.

Loop-invariant expression A computation whose value does not change from one loop iteration to the next. In the example, $2 \times b \times c$ is invariant.

Pretty printing Transformation from an internal tree representation back to concrete program text. Jackie contrasted this direction with parsing.

Commuting diagram The page-10 diagram connecting source program, initial AST, transformed AST, and optimized program through parsing, AST transformation, pretty printing, and the external program to program transformation.

Key Examples and Demonstrations

Q1–Q3 on the optimizer pipeline

Setup. Page 3 adds an optimizer between IR_1 and IR_2.

Main point. Correctness is not a property of only the final back end. The front end, optimizer, and back end each have their own semantics-preservation obligation.

What to practise. Redraw the pipeline and state Q1, Q2, and Q3 without looking. Then explain why preserving meaning at every step supports preservation from source to target.

Tokenizing the beginning of `MyClass.java`

Setup. Page 4 reads a Java-like program as a character sequence.

Main point. The scanner recognizes `class` as a keyword, `MyClass` as an identifier, and the left brace as a token while ignoring ordinary whitespace between them. The output is still linear, but its elements are tokens rather than raw characters.

What to practise. Follow the first few source characters from left to right and separate recognized tokens from discarded whitespace. Be able to say why this is lexical rather than syntactic analysis.

Two while-loop programs: same syntax status, different semantic status

Setup. Page 7 shows two inputs that both fit the illustrated while-loop CFG; the second repeats the declaration of `i`.

Main point. Both inputs can have parse trees, so both pass the syntactic phase. The second fails the later type-checking/semantic check because the same variable name is declared twice in the same scope.

What to practise. Explain exactly which phase accepts both programs and which phase distinguishes them. Do not use "compilation error" as one undifferentiated category.

Hoisting $2 \times b \times c$ out of the loop

Setup. Pages 8–9 show the unoptimized and optimized ASTs.

Main point. Because b and c do not change inside the loop, the compiler can insert a pre-loop assignment to `temp` and replace the repeated expression subtree inside the loop with a `temp` reference. The optimized program should preserve the original meaning while reducing repeated computation.

What to practise. On paper, identify the invariant subtree in the input AST, mark where the new assignment node is inserted, and mark the subtree that becomes the `temp` reference.

Reconstructing the commuting diagram

Setup. Page 10 places source and optimized program text above their corresponding ASTs.

Main point. Program → tree is parsing; tree → tree is the optimization; tree → program is pretty printing. The external user sees only the overall program to program transformation.

What to practise. Reproduce the four boxes and label all three internal arrows and the external compilation arrow.

Common Pitfalls and Misconceptions

- **Checking only the final output and ignoring intermediate semantic obligations.** Lecture 4 explicitly separated correctness into Q1, Q2, and Q3. An optimizer is valid only if its transformed IR preserves the meaning of the incoming IR.
- **Treating whitespace as always meaningful program content.** In the Java-like lexical example, ordinary spaces/newlines were meaningless delimiters and were discarded by the scanner. Other punctuation can be useful tokens, so "delimiter" does not automatically mean "discard."
- **Conflating scanning and parsing.** Scanning is character-sequence to token-sequence recognition using regular-language machinery. Parsing consumes tokens and builds a non-linear structure according to a context-free grammar.
- **Assuming syntactically valid code is necessarily semantically valid.** The duplicate declaration example parses successfully but fails the later semantic/type check.
- **Trying to push rich type/scope checking into lexical analysis.** Jackie explicitly rejected this in response to a student question. The scanner's job is token recognition; richer context-dependent checks belong during or after parsing.
- **Confusing parsing with pretty printing.** Parsing moves from concrete program text to an internal tree; pretty printing moves from the tree back to concrete text.
- **Thinking optimization means editing source text directly.** The loop example was developed as a structural AST-to-AST transformation, followed only later by pretty printing of the optimized tree.

- **Optimizing a loop expression without first checking what changes.** In the example, a and d vary across iterations while 2, b, and c do not. The invariance argument is what justifies moving $2 \times b \times c$ outside.

What You Should Be Able to Do

After this session, you should be able to:

- draw the source → front end → IR_1 → optimizer → IR_2 → back end → target pipeline and explain Q1, Q2, and Q3;
- explain why semantic preservation across several stages is a transitive correctness obligation;
- distinguish lexical, syntactic, and semantic analysis by both purpose and input/output structure;
- identify scanner and parser responsibilities and relate token recognition to regular languages and parsing to context-free grammars;
- distinguish a raw character sequence, a token sequence, and a parse-tree/AST structure;
- classify ordinary whitespace versus useful punctuation in the lexical example and explain what the scanner emits;
- explain the roles of DFA, NFA, ε-NFA, regular expressions, and DFA minimization in the upcoming lexical-analysis review at the level introduced here;
- explain why the two while-loop inputs are both syntactically valid while the duplicate declaration fails semantic/type checking;
- distinguish top-down from bottom-up as the two parsing directions introduced for later study;
- trace the loop-invariant optimization from input program to initial AST, transformed AST, and optimized program; and
- explain the difference between parsing and pretty printing using the commuting diagram.

Suggested Practice

1. Before the next lexical-analysis lecture, review the posted lexical-analysis material and the math-review material Jackie identified (sets, relations, and predicate logic). Jackie said those review slides are sufficient for this course.

2. When Assignment 1 is released on September 23, read the instructions promptly. Use it as the intended Java warm-up: inspect the supplied Java/JUnit structure and begin working with the regular-language material rather than waiting until the end of the two-week window.

3. Reproduce page 4 of Jackie's annotated iPad notes from memory: write a short Java-like prefix, mark the discarded whitespace, identify the first few tokens, and label the lexical and syntactic input/output structures.

4. Revisit page 7. Cover the annotations and decide independently whether each of the two while-loop programs passes lexical, syntactic, and semantic analysis. Then compare your explanation with the notes.

5. Redraw the optimizer example on pages 8–9. First identify what changes inside the loop and what remains invariant; only then perform the tree transformation that introduces `temp`.

6. Reconstruct the page-10 commuting diagram without copying it. Label **parsing**, **AST-to-AST transformation**, **pretty printing**, and the external **program-to-program compilation** view.

Where to Review

- **Jackie's annotated iPad notes:** Page 2 contains the announcements for the posted lexical-analysis and math-review materials and the Assignment 1 release on September 23.
- **Page 3:** optimizer pipeline with IR_1, IR_2, and the Q1/Q2/Q3 semantics-preservation questions.
- **Pages 4–6:** lexical → syntactic → semantic analysis, tokenization of `MyClass`, scanner/parser roles, and the essay-writing analogy.
- **Page 7:** while-loop CFG, the two parse-tree examples, and the contrast between syntactic success and semantic/type-checking failure.
- **Pages 8–9:** input and output ASTs for the loop-invariant optimization.
- **Page 10:** the commuting diagram connecting source code, AST transformation, optimized AST, and pretty-printed output.
- **Lecture recording:** return to the spoken development when the visual transition between character sequence, token sequence, parse tree, and transformed AST is not yet clear from the static pages alone.

Closing Reminder

Use this digest as a roadmap for active review, not as a substitute for the learning process itself. Attend Jackie's in-person sessions and pay attention to how he develops ideas and works through the important reasoning step by step. If a transition, explanation, example, or reasoning step remains unclear afterwards, return to the recording to fill that gap.

For structured review, start with Jackie's annotated iPad notes because they most closely reflect the actual flow and development of the class. Work through them in manageable logical units. After each logical unit, study the corresponding part of the presentation slides alongside it. Jackie does not normally teach by reading the slides line by line, so the slides may contain relevant supporting material that was not explicitly read aloud.

Finally, choose a suitable selection of the important reasoning from this session and work through it again on your own. The goal is not to memorize Jackie's recorded steps, but to determine whether you can reproduce the reasoning from your own understanding and explain why the important steps work.