# EECS4302 Compilers and Interpreters
# Fall 2026

## Digest of Lecture 13, October 29

Context-Free Grammars, Derivations, and Grammar Design Trade-offs

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 7

Suggested Practice 8

Where to Review 9

Closing Reminder 9

# Main Ideas

- The session moved from lexical analysis into **syntactic analysis**, using context-free grammars (CFGs) to describe recursive language structure that regular expressions and finite automata cannot capture.
- A CFG separates **non-terminals/variables**, **terminal tokens**, a **start variable**, and **production rules**. Jackie distinguished **base cases**, which can terminate a derivation, from **recursive cases**, which keep expanding language structure.
- CFGs describe syntax, not meaning. A grammar can accept a string syntactically even when the expression contains a type error; semantic checking happens after a parse tree has been built.
- Jackie connected two ways of showing a derivation: a textual **leftmost derivation (LMD)** and a graphical **parse tree**. In an LMD, always expand the leftmost non-terminal; in a parse tree, the leaf sequence read left-to-right gives the derived string.
- The grammar-design exercises showed how small changes to base/recursive cases change the language: all binary palindromes versus strings of the form $ww^R$, and a block language built by separating an individual block from a sequence of blocks.
- The final comparison contrasted a permissive expression grammar (Version 1), which builds a parse tree and relies on later type checking, with a more stratified grammar (Version 2), which rejects some type-inconsistent expressions during parsing.
- For the November 5 Programming Test, Jackie emphasized practising the supplied ANTLR 4 workflow in Eclipse, using the Visitor pattern, and working through the starter guide and practice test before the assessment.

# Roadmap

## 1. Programming Test preparation and logistics

Jackie began with the upcoming Programming Test. The test is on **Thursday, November 5, 4:00–5:20 PM in WSC 105**, uses the full 80-minute lecture period, and is worth **10%**. The optional mockup test is on **Monday, November 2, 3:30–4:30 PM in LAS 1002B**; Jackie said the room was reserved until 5:00 PM. These are different rooms, so check the location carefully.

The test asks you to build a small compiler project using **Eclipse and ANTLR 4**. Jackie said the lexer tokens will be supplied, so the main grammar work is on the context-free grammar rather than writing regular expressions from scratch. You should be comfortable creating the project, generating the ANTLR skeleton, refreshing/building in Eclipse, implementing the model/Visitor work, exporting an executable JAR, and running the supplied testing workflow. The Visitor pattern is the required approach for the test; do not embed Java semantic actions directly in the grammar for this assessment.

Jackie recommended using the starter guide and practice test as the priority preparation, redoing Assignment 2 if useful, checking your EECS login in advance, and being prepared for the basic file-processor/main-driver boilerplate used in the ANTLR workflow. The test environment setup commands shown in the guide will be supplied for copy/paste during the actual test.

## 2. Why context-free grammars are needed

The technical lecture began with the non-regular language

$$L = \{0^n \# 1^n \mid n \geq 0\}.$$

The matching number of $0$s and $1$s requires recursive structure. Jackie used the grammar

$$A \to 0A1, \qquad A \to B, \qquad B \to \#$$

to introduce CFG terminology. $A$ is the start variable. The left-hand side of a production is a non-terminal/variable; the right-hand side contains terminal symbols/tokens and possibly further variables.

A production whose right-hand side contains only terminals is a **base case**; a production containing at least one variable is a **recursive case**. Recursive rules let the language grow without a fixed bound, while base cases provide a way to terminate the derivation. Jackie also stressed that **no semantic analysis is done at the grammar level**; semantic analysis comes after a parse tree is available.

## 3. Textual derivation versus parse tree

Jackie then derived $000\#111$. In a **leftmost derivation**, each step must expand the leftmost variable:

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111.$$

The process is not finished while a non-terminal remains. The final result must contain terminals only.

The same derivation can be represented graphically as a parse tree. The start variable is the root; internal nodes show variable expansions; the external nodes/leaves are terminals. Reading the leaves from left to right gives the string generated by the tree. Unlike an LMD/RMD, the order in which you visually expand nodes does not change the final tree structure.

## 4. Designing CFGs from language descriptions

Jackie worked through three grammar-design exercises.

For binary palindromes, the grammar uses three terminating cases and two symmetric recursive cases:

$$P \to \varepsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1.$$

The base cases cover lengths 0 and 1; the recursive cases add matching symbols to both ends.

For strings of the form $ww^R$, the single-character base cases are removed:

$$P \to \varepsilon \mid 0P0 \mid 1P1.$$

This captures the even-length reverse-concatenation structure. Jackie used the difference between these two grammars to emphasize checking boundary cases such as a one-character palindrome.

For the block-language exercise on page 5 of Jackie's annotated iPad notes, Jackie separated the problem into two levels: one variable describes an individual block of $0$s followed by at least as many $1$s, while the start variable $S$ describes an arbitrary sequence of such blocks using recursion (including the empty sequence). The important design idea is the decomposition into *component* and *sequence* non-terminals rather than compressing the repetition into regex notation.

## 5. Validating a proposed grammar

Jackie drew the target language and the language generated by your grammar as two sets. A proposed grammar is wrong if it produces a string outside the target language. Equivalently, if you can find

$$s \in L(G) \quad \text{but} \quad s \notin L_{\text{target}},$$

then that single counterexample is enough to show the grammar does not specify the requested language exactly. This also motivates comparison questions where two grammar/language descriptions are disproved as equivalent by one distinguishing string.

## 6. Formal CFG notation versus ANTLR conveniences

In theoretical written work, Jackie expects standard CFG productions. Do not use regex-style repetition operators such as $*$ or + as substitutes for recursive CFG productions on a written test or exam. ANTLR may provide such conveniences, but the paper theory questions are about the formal CFG notation taught in class.

## 7. Version 1 versus Version 2 expression grammars

The final part compared two ways to structure an expression grammar. **Version 1** places arithmetic, logical, and relational binary expressions under one broad `BinaryOp` category. That grammar can build a parse tree for an expression such as

$$(1 + 2) \Rightarrow (5/4)$$

even though the implication antecedent is not Boolean. The string is syntactically accepted, so a later Visitor/type-checking pass must report the semantic error.

**Version 2** separates arithmetic, relational, and logical categories. Because the grammar itself encodes more restrictions, some type-inconsistent combinations are rejected by ANTLR as syntax errors before a complete parse tree is built. Jackie framed this as a design trade-off: Version 1 requires more semantic checking but can support more deliberate, user-friendly type-error messages; Version 2 rejects more invalid combinations at the grammar level but may leave error reporting to ANTLR's syntax diagnostics.

### 8. Rooting the implication parse tree correctly

For Version 1, Jackie started constructing the parse tree for $(1 + 2) \Rightarrow (5/4)$. A proposed start at the parenthesized left subexpression only covered part of the input. The correct tree must start at the whole `Expression` and use the top-level implication production, with the left subtree deriving $(1 + 2)$ and the right subtree deriving $(5/4)$. The parse tree must cover **all** input tokens, not just a convenient subexpression.

## Key Terms, Notation, and Structures

**Context-free grammar (CFG)**
: A grammar for recursive syntactic structure. In this session, CFGs were contrasted with regular-language tools and used as the basis for parsing.

**Non-terminal / variable**
: A grammar symbol that can be expanded by a production rule. The first designated variable is the **start variable**.

**Terminal / token**
: A symbol that is not expanded further. A completed derived string contains terminals only.

**Production rule**
: A rule such as $A \to 0A1$ describing how a variable may be replaced.

**Base case**
: A production whose right-hand side contains only terminal symbols/tokens, allowing a derivation to terminate.

**Recursive case**
: A production whose right-hand side contains at least one variable, allowing further expansion.

**Leftmost derivation (LMD)**
: A textual derivation in which the leftmost variable is expanded at every step. Jackie said the requested derivation direction must be followed exactly on an assessment.

**Parse tree**
: A graphical derivation rooted at the start variable. Internal nodes represent variable expansions; the leaves read left-to-right form the derived string (the tree's yield).

$L(G)$
: The language generated by grammar $G$. A grammar specification is incorrect if $L(G)$ contains strings outside the requested target language.

**Syntax versus semantics**
: Parsing establishes whether the token sequence fits the grammar. Type correctness and other semantic properties may still require a later Visitor/semantic-analysis pass.

**Version 1 / Version 2**

The two expression-grammar styles compared in class: one broad binary-expression grammar versus a stratified grammar with arithmetic, relational, and logical categories.

# Key Examples and Demonstrations

## $000\#111$: one string, two derivation views

**Setup.** Use $A \to 0A1$, $A \to B$, and $B \to \#$.
**Main point.** The LMD expands the leftmost variable until terminals only remain; the parse tree records the same recursive structure graphically.
**What to practise.** Reproduce the LMD and then reconstruct the corresponding parse tree from page 4 of Jackie's annotated iPad notes without copying the completed diagram.

## Palindrome versus $ww^R$

**Setup.** Compare the all-palindrome grammar with the reverse-concatenation grammar.
**Main point.** The recursive rules are the same; the difference lies in the base cases. Keeping $0$ and $1$ permits odd-length palindromes, while removing them restricts the language to the even-length $ww^R$ structure.
**What to practise.** Use one-character strings as boundary cases and explain exactly why they distinguish the two languages.

## Block-language decomposition

**Setup.** Each block contains $0$s followed by at least as many $1$s, and a full string may contain multiple blocks.
**Main point.** Jackie separated the grammar into a variable for an individual block and a start variable for a sequence of blocks.
**What to practise.** Return to page 5 of Jackie's annotated iPad notes and convince yourself how each production contributes either to one block or to repetition of blocks.

## Version 1 versus Version 2 on an invalidly typed expression

**Setup.** Consider expressions such as $(3 = 4) + (6 > 7)$ or $(1 + 2) \Rightarrow (5/4)$.
**Main point.** Version 1 can parse such strings and leaves type rejection to semantic analysis; Version 2 encodes more category restrictions directly in the grammar and can reject them during parsing.
**What to practise.** Explain where the error is detected in each design and what kind of parse-tree/Visitor work is still possible afterwards.

### Finding the correct parse-tree root

**Setup.** Construct the Version 1 parse tree for $(1 + 2) \Rightarrow (5/4)$.
**Main point.** Starting with the parenthesized addition only derives part of the input. The whole tree must be rooted at the top-level implication structure.
**What to practise.** Before expanding a parse tree, identify which production can cover the entire input string at the root.

## Common Pitfalls and Misconceptions

- **Stopping a derivation too early.** A sentential form that still contains a variable is not a completed terminal string in $L(G)$.
- **Ignoring the requested derivation direction.** If Jackie asks for an LMD, expand the leftmost variable at each textual step. The parse-tree drawing itself does not impose that same expansion order.
- **Using regex shorthand in a theoretical CFG answer.** On paper, express repetition with production recursion rather than using $*$ or + as if the CFG were an ANTLR shortcut notation.
- **Assuming a syntactically accepted expression is type-correct.** Version 1 intentionally demonstrates that a parse tree may exist even when a later Visitor must report a type error.
- **Rooting a parse tree at a subexpression.** A candidate root is wrong if its tree cannot cover the complete input token sequence.
- **Confusing the two Programming Test rooms.** The optional mockup is in LAS 1002B; the actual November 5 test is in WSC 105.
- **Using embedded Java grammar actions for the Programming Test.** Jackie explicitly required the Visitor approach for this assessment.

## What You Should Be Able to Do

After this session, you should be able to:

- explain why a recursive language such as $\{0^n\#1^n \mid n \geq 0\}$ motivates a CFG rather than a regular-language tool;
- identify the start variable, terminals, non-terminals, production rules, base cases, and recursive cases in a CFG;
- carry out a leftmost derivation and stop only when the sentential form contains terminals only;
- construct the corresponding parse tree and read its leaf yield from left to right;

- design and compare the binary-palindrome and $ww^R$ grammars, including the role of their different base cases;
- explain the component-versus-sequence decomposition used in the block-language grammar;
- test a proposed grammar against its target language and use a counterexample string to disprove an incorrect/equivalent-language claim;
- distinguish formal CFG notation from ANTLR-specific shorthand;
- compare the Version 1 and Version 2 expression grammars in terms of where invalid combinations are rejected and where type checking occurs;
- identify a parse-tree root that covers the complete input string rather than only a subexpression;
- prepare the Programming Test workflow in Eclipse/ANTLR 4 using the required Visitor-based architecture and the supplied practice materials.

# Suggested Practice

1. **Programming Test first.** Work through the eClass starter guide and practice test. Practise the complete Eclipse/ANTLR 4 workflow that Jackie described: project creation, grammar file, skeleton generation, refresh/build, Visitor/model implementation, executable JAR export, and test execution.
2. If useful, **redo Assignment 2** after Jackie posts the solution. Jackie described this as a way to be over-prepared for the simpler Programming Test problem.
3. Attend the optional **November 2 mockup test in LAS 1002B** if you can; otherwise use the remote-lab simulation guidance and verify your EECS account/tool setup before November 5.
4. Reproduce the **leftmost derivation of** $000\#111$ from the start variable, then draw the parse tree independently and verify that its leaves yield the same terminal string.
5. Rebuild the two grammars on page 5 for **binary palindromes** and $ww^R$. Explain why $0$ or $1$ is a useful counterexample when comparing the two languages.
6. Revisit the **block-language grammar** on page 5. Identify which rules describe one block and which rules describe repetition of blocks. Do not replace the recursive CFG structure with regex $*/+$ shorthand in a theoretical answer.
7. For the two expression grammars, explain what happens to $(3 = 4) + (6 > 7)$ or $(1 + 2) \Rightarrow (5/4)$ under Version 1 and Version 2, including whether a complete parse tree is built and where the error is reported.
8. Reconstruct the beginning of the Version 1 parse tree for $(1 + 2) \Rightarrow (5/4)$ and check that your root production covers the entire input before expanding either subtree.

## Where to Review

- **Jackie's annotated iPad notes, page 2:** Programming Test reminder, guide/practice questions, and the November 2 mockup-test location/time.
- **Page 3:** CFG terminology using $\{0^n\#1^n \mid n \geq 0\}$, including start variable, productions, terminals/non-terminals, and base versus recursive cases.
- **Page 4:** the leftmost derivation and parse-tree visualization for $000\#111$.
- **Page 5:** the three grammar-design exercises: binary palindromes, $ww^R$, and the block-language decomposition.
- **Page 6:** Jackie's set-based picture for validating whether the language generated by your grammar matches the requested language and for identifying counterexamples.
- **Page 7:** Version 1 versus Version 2 expression grammars and the syntax-versus-type-checking trade-off.
- **Page 8:** Version 1 parse-tree discussion for $(1 + 2) \Rightarrow (5/4)$ and the correction that the root must cover the complete input.
- **Programming Test materials on eClass:** the guide, practice test, and environment setup/testing instructions Jackie reviewed at the start of the session.
- **Session recording:** return to Jackie's spoken Programming Test clarifications and the class discussion around the two grammar versions when the notes alone do not capture the reasoning or Q&A.

## 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.