Context-Free Grammars

Derivations, Parse Trees, and Ambiguity

Computer Science Theory

Slide Set 5

Learning Outcomes

By the end of this slide set, you should be able to:

- define a context-free grammar (CFG) and the language it generates;
- distinguish sentential forms, sentences, and leftmost/rightmost derivations;
- prove that a proposed grammar generates exactly a specified language;
- construct parse trees and compute their yields;
- demonstrate ambiguity using parse trees or leftmost derivations;
- rewrite expression and statement grammars to encode intended structure; and
- translate simple BNF/EBNF specifications into CFG productions.

Road Map

Why Context-Free Grammars?

Finite automata recognize patterns using only finite memory. They cannot enforce an unbounded exact match such as

$$\{0^n 1^n \mid n \geq 0\}.$$

A CFG describes a language *generatively*: it recursively builds valid strings from a start variable.

What recursion gives us

A finite set of productions can describe arbitrarily deep nesting, matched delimiters, recursive program syntax, and self-similar structure.

Power and limitation

Every regular language is context-free, but some languages—for example $\{a^n b^n c^n \mid n \geq 0\}$—are not context-free.

An Inductive Definition: Balanced Parentheses

A balanced-parenthesis string can be defined recursively:

1. the empty string is balanced;
2. if x and y are balanced, then $(x)y$ is balanced; and
3. nothing else is balanced.

This becomes the grammar

$$S \to (S)S \mid \varepsilon.$$

Examples

$$\varepsilon, \quad (), \quad (()), \quad ()(), \quad (()())$$

The first S creates nesting inside a pair; the second permits another balanced string to follow.

Formal Definition of a CFG

We use the Linz-style notation $G = (V, T, S, P)$. Sipser commonly writes (V, Σ, R, S); the components are the same up to names and order.

Definition

A context-free grammar is a 4-tuple $G = (V, T, S, P)$, where:

1. V is a finite set of variables (nonterminals);
2. T is a finite set of terminals, with $V \cap T = \varnothing$;
3. $S \in V$ is the start variable; and
4. P is a finite set of productions $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$.

Alternatives such as $A \to x$, $A \to y$, and $A \to z$ are abbreviated $A \to x \mid y \mid z$.

What Does "Context-Free" Mean?

A production has exactly one variable on its left-hand side:

$$A \to \alpha.$$

Therefore A may be replaced by α regardless of the symbols surrounding it.

Valid CFG productions	Not CFG productions
$S \to 0S1$	$AB \to BA$ (two variables on the left)
$A \to aB \mid \varepsilon$	$aA \to Aa$ (a terminal occurs on the left)
$B \to C$	$\varepsilon \to S$ (no variable on the left)

Terminology

"Context-free" describes the form of the productions. It does not mean that the generated strings lack structure or context.

Derivations and Sentential Forms

If $A \to \alpha$ is a production, then for any $u, v \in (V \cup T)^*$,

$$uAv \Rightarrow u\alpha v.$$

- $\Rightarrow$: one derivation step;
- $\Rightarrow^*$: zero or more steps;
- $\Rightarrow^+$: one or more steps;
- **sentential form**: any γ such that $S \Rightarrow^* \gamma$;
- **sentence**: a terminal-only sentential form $w \in T^*$.

The generated language is

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w\}.$$

Because $\Rightarrow^*$ allows zero steps, S itself is a sentential form but normally not a sentence.

Leftmost and Rightmost Derivations

Consider

$$S \to AB, \qquad A \to aA \mid a, \qquad B \to bB \mid b.$$

A leftmost derivation of $aabb$ is

$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aaB \Rightarrow aabB \Rightarrow aabb.$$

A rightmost derivation of the same string is

$$S \Rightarrow AB \Rightarrow AbB \Rightarrow Abb \Rightarrow aAbb \Rightarrow aabb.$$

Key point

The replacement order differs, but both derivations describe the same hierarchical parse tree. Derivation order alone is not semantic structure.

Proving What a Grammar Generates

For

$$G: \quad S \to 0S1 \mid \varepsilon,$$

prove $L(G) = \{0^n 1^n \mid n \geq 0\}$ using two inclusions.

Soundness: $L(G) \subseteq \{0^n 1^n\}$

Every use of $S \to 0S1$ adds one 0 to the left and one 1 to the right. After k such steps the form is $0^k S 1^k$; replacing S by ε yields $0^k 1^k$.

Completeness: $\{0^n 1^n\} \subseteq L(G)$

For any $n \geq 0$, apply $S \to 0S1$ exactly n times and then $S \to \varepsilon$:
$S \Rightarrow^* 0^n S 1^n \Rightarrow 0^n 1^n$.

Testing examples suggests a grammar; the two-inclusion argument proves it.

Reusable Grammar-Design Patterns I

Matched Blocks

$$S \to 0S1 \mid \varepsilon$$

Generates 0^n1^n by adding matched symbols at opposite ends.

Palindromes

$$S \to 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$$

Add the same symbol to both ends; base cases determine parity.

Balanced Strings

$$S \to (S)S \mid \varepsilon$$

One variable position handles nesting and the other concatenation.

Reusable Grammar-Design Patterns II

Suppose grammars with renamed, disjoint variables generate L_1 and L_2 from start variables S_1 and S_2.

Goal	New start production	Meaning
$L_1 \cup L_2$	$S \to S_1 \mid S_2$	choose either grammar
$L_1 L_2$	$S \to S_1 S_2$	generate one string from each
L_1^*	$S \to S_1 S \mid \varepsilon$	repeat zero or more times

Language generation versus ambiguity

These constructions generate the intended languages, but they may produce ambiguous grammars when a string can be decomposed in more than one way.

A Grammar-Design Workflow

1. Identify the recursive or nested structure of valid strings.
2. Choose one variable for each meaningful syntactic category or phase.
3. Write recursive productions that preserve the required invariant.
4. Add base cases so the recursion can terminate.
5. Test the smallest valid strings and several invalid strings.
6. Prove both soundness and completeness.
7. Check whether the grammar accidentally introduces ambiguity.

Debugging question

If an invalid string is generated, which production allowed the invariant to break? If a valid string is missing, which necessary base case or recursive choice is absent?

Formal Structure of a Parse Tree

A parse tree for a CFG $G = (V, T, S, P)$ satisfies:

1. the root is labeled S;
2. every internal node is labeled by a variable;
3. if a node labeled A has children labeled $X_1, \ldots, X_k$ from left to right, then $A \to X_1 \cdots X_k$ is a production; and
4. every leaf is a terminal or ε.

The **yield** is the terminal string obtained by reading the leaves left to right and omitting ε.

Parse-tree theorem

$S \Rightarrow^* w$ iff there exists a parse tree rooted at S with yield w.

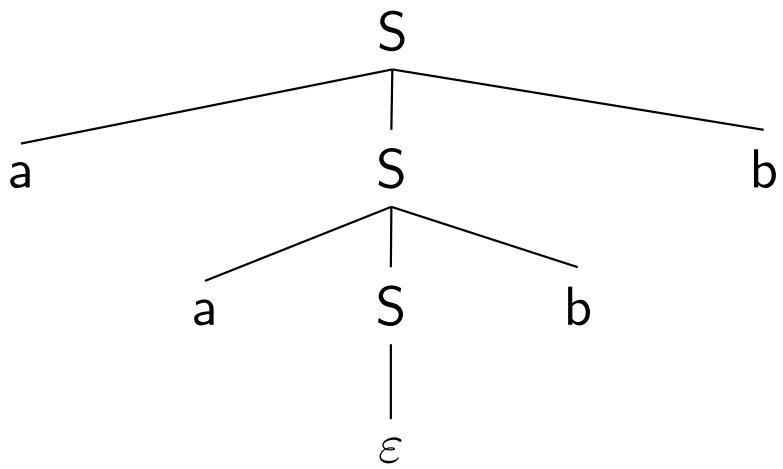
Worked Parse Tree for *aabb*

Grammar: $S \to aSb \mid \varepsilon$.

Derivation

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Each internal S node records one selected production.



Reading the leaves gives $aa\varepsilon bb$, whose yield is $aabb$.

Derivation Order Versus Tree Structure

Suppose a sentential form contains several variables. Expanding the left one first or the right one first may produce different step sequences but the same tree.

Correspondence theorem

Every parse tree determines exactly one leftmost derivation and exactly one rightmost derivation.

Consequently:

- two distinct arbitrary derivations do *not* necessarily prove ambiguity;
- two distinct leftmost derivations of the same string do prove ambiguity; and
- two distinct parse trees of the same string are the clearest structural proof.

This distinction prevents one of the most common mistakes in CFG exercises.

Parse Trees and Abstract Syntax Trees

A parse tree contains every grammar symbol used in the derivation. An abstract syntax tree (AST) keeps the essential structure needed by later compiler phases.

Expression $x + 2 * y$

The parse tree may include variables such as E, T, F, punctuation, and token categories. The AST can be summarized as

$$+(x, *(2, y)).$$

Why the distinction matters

Grammars specify concrete syntax. AST design captures semantic structure such as operators, operands, declarations, and control flow.

What Is an Ambiguous Grammar?

Definition

A CFG is ambiguous if some string in its language has two distinct parse trees.

The following statements are equivalent:

- some string has two distinct parse trees;
- some string has two distinct leftmost derivations;
- some string has two distinct rightmost derivations.

Not enough

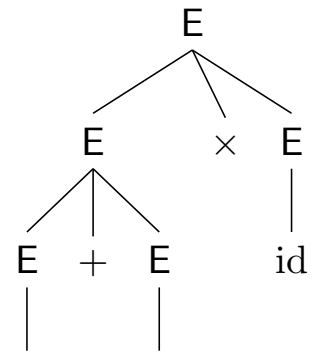
Two unrestricted derivations may simply expand independent variables in different orders and still describe one parse tree. Use parse trees or fixed-order derivations when proving ambiguity.

Ambiguous Arithmetic Grammar

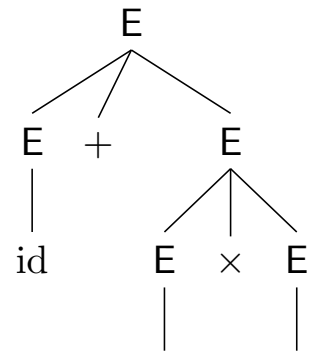
Consider

$$E \to E + E \mid E \times E \mid \mathrm{id}.$$

The string $\mathrm{id} + \mathrm{id} \times \mathrm{id}$ has two structures:



$(\mathrm{id} + \mathrm{id}) \times \mathrm{id}$



$\mathrm{id} + (\mathrm{id} \times \mathrm{id})$

The grammar does not encode precedence or associativity.

The Same Ambiguity as Leftmost Derivations

For $\mathrm{id} + \mathrm{id} \times \mathrm{id}$:

Top-level multiplication

$$E \Rightarrow_{\mathrm{lm}} E \times E \Rightarrow_{\mathrm{lm}} E + E \times E \Rightarrow_{\mathrm{lm}} \mathrm{id} + E \times E$$
$$\Rightarrow_{\mathrm{lm}} \mathrm{id} + \mathrm{id} \times E \Rightarrow_{\mathrm{lm}} \mathrm{id} + \mathrm{id} \times \mathrm{id}.$$

Top-level addition

$$E \Rightarrow_{\mathrm{lm}} E + E \Rightarrow_{\mathrm{lm}} \mathrm{id} + E \Rightarrow_{\mathrm{lm}} \mathrm{id} + E \times E$$
$$\Rightarrow_{\mathrm{lm}} \mathrm{id} + \mathrm{id} \times E \Rightarrow_{\mathrm{lm}} \mathrm{id} + \mathrm{id} \times \mathrm{id}.$$

Encoding Precedence and Associativity

Replace the ambiguous grammar by

$$E \to E + T \mid T,$$
$$T \to T \times F \mid F,$$
$$F \to (E) \mid \mathrm{id}.$$

- Multiplication binds more tightly because it is formed inside T before T participates in addition.
- Left recursion in $E \to E + T$ makes addition left-associative.
- Left recursion in $T \to T \times F$ makes multiplication left-associative.
- Parentheses allow a factor to contain an entire expression.

For right-associative addition, one could instead use $E \to T + E \mid T$.

The Dangling-Else Ambiguity

The natural grammar

$$S \to \textbf{if } c \textbf{ then } S \mid \textbf{if } c \textbf{ then } S \textbf{ else } S \mid a$$

where c stands for a condition and a for any other statement. The grammar is ambiguous: in

$$\textbf{if } c \textbf{ then if } c \textbf{ then } a \textbf{ else } a,$$

the **else** may attach to either **if**.

Standard matched/unmatched solution

$$S \to M \mid U,$$
$$M \to a \mid \textbf{if } c \textbf{ then } M \textbf{ else } M,$$
$$U \to \textbf{if } c \textbf{ then } S \mid \textbf{if } c \textbf{ then } M \textbf{ else } U.$$

This grammar attaches each **else** to the nearest unmatched **if**.

Grammar Ambiguity Versus Inherent Ambiguity

- A **grammar** is ambiguous when it gives some string two parse trees.
- A **language** is inherently ambiguous when every CFG generating it is ambiguous.

An ambiguous grammar may still describe a language that has another unambiguous grammar; arithmetic expressions are an example.

Standard inherently ambiguous language

$$\{a^i b^j c^k \mid i = j \text{ or } j = k,\ i, j, k \geq 1\}.$$

Algorithmic limitation

There is no general algorithm that decides whether an arbitrary CFG is ambiguous.

BNF and EBNF

Backus–Naur Form is a notation for CFG productions:

$$\texttt{<assignment>} ::= \texttt{<id> = <expression>}$$
$$\texttt{<id>} ::= \texttt{x | y | z}.$$

Extended BNF adds notation for repetition and optional material. For example,

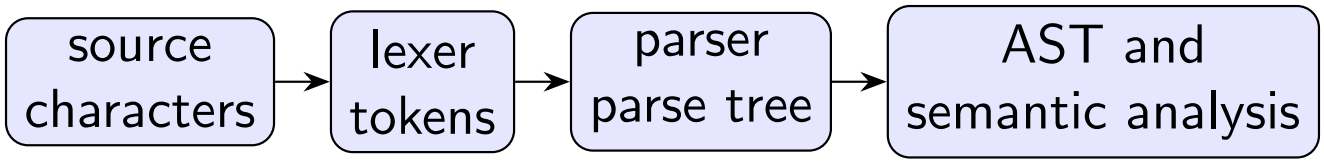
$$\langle \text{list} \rangle ::= \langle \text{id} \rangle \{ \texttt{","} \ \langle \text{id} \rangle \}$$

can be expanded into ordinary CFG productions:

$$L \to \text{id}\, R, \qquad R \to , \text{id}\, R \mid \varepsilon.$$

EBNF conventions vary between standards, so braces, brackets, and parentheses must be defined.

CFGs in a Compiler Pipeline



- Lexical rules are usually regular and identify tokens such as identifiers and numbers.
- A CFG organizes tokens into nested syntactic structure.
- Semantic analysis checks properties not conveniently captured by a CFG, such as types, declarations, and scope.

Syntax is not semantics

A program can be grammatically valid but still contain an undeclared variable or type error.

Top-Down and Bottom-Up Views

A derivation begins with S and expands variables until the input string appears. This is a **top-down** view of syntax.

- A top-down parser predicts which production should expand the current variable.
- A bottom-up parser starts from the input and repeatedly recognizes substrings corresponding to production right-hand sides, eventually reducing to S.

Connection to the next unit

Pushdown automata provide the machine model for context-free languages. Their stack records unfinished grammatical structure during recognition or parsing.

Concept Checks

1. Is S always a sentential form? Is it always a sentence?
2. Why do two arbitrary derivations of the same string not necessarily prove ambiguity?
3. What is the yield of a parse tree containing ε leaves?
4. Does an ambiguous grammar necessarily generate an inherently ambiguous language?
5. Which part of the expression grammar enforces multiplication precedence?
6. Why can a CFG describe syntax but not all static semantic rules of a programming language?

Construction Exercises

1. Give a leftmost derivation of 000111 using $S \to 0S1 \mid \varepsilon$.
2. Draw a parse tree for 0110 using $S \to 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$.
3. Design a CFG for $\{0^i 1^j \mid i, j \geq 0\}$ and explain why it is correct.
4. For $S \to SS \mid a$, give two leftmost derivations of aaa.
5. Modify the unambiguous expression grammar so exponentiation $\uparrow$ has higher precedence than multiplication and is right-associative.
6. Prove, using two inclusions, that $S \to aSb \mid \varepsilon$ generates $\{a^n b^n \mid n \geq 0\}$.

Hints and Short Answers

1. $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000111$.
2. Use outer production $0S0$, then inner production $1S1$, then $S \to \varepsilon$.
3. $S \to AB$, $A \to 0A \mid \varepsilon$, $B \to 1B \mid \varepsilon$.
4. Two leftmost derivations are $S \Rightarrow SS \Rightarrow SSS \Rightarrow aSS \Rightarrow aaS \Rightarrow aaa$ and $S \Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow aaS \Rightarrow aaa$.
5. Keep $T \to T \times F \mid F$ and use $F \to A \uparrow F \mid A$, $A \to (E) \mid \mathrm{id}$. The right recursion makes exponentiation right-associative.
6. Use the invariant $a^k S b^k$ for soundness and apply the recursive rule exactly n times for completeness.

Common Mistakes to Avoid

- Giving examples instead of proving both language inclusions.
- Leaving a variable as a parse-tree leaf in a completed parse.
- Confusing ε with the empty language $\varnothing$.
- Claiming ambiguity from two derivations that differ only in expansion order.
- Forgetting base productions, causing every derivation to continue forever.
- Adding precedence rules without checking associativity.
- Assuming that removing ambiguity from one grammar proves the language is never inherently ambiguous.

Summary

- CFGs use recursive productions $A \to \alpha$ to generate context-free languages.
- Derivations describe substitution sequences; parse trees capture hierarchical structure.
- Grammar correctness is normally proved with soundness and completeness inclusions.
- Ambiguity is structural: one string has two parse trees or two leftmost derivations.
- Carefully designed variables encode precedence, associativity, nesting, and statement structure.
- CFGs are central to parsing, while later compiler phases enforce semantic constraints.

References and Suggested Reading

Peter Linz and Susan H. Rodger,

An Introduction to Formal Languages and Automata, 7th ed.,

Jones & Bartlett Learning, 2023.

Michael Sipser,

Introduction to the Theory of Computation, 3rd ed.,

Cengage Learning, 2013.

Recommended emphasis: Linz for grammar construction and derivation mechanics; Sipser for concise formal definitions, parse-tree reasoning, and ambiguity.