CS304: Automata and Formal Languages

Lec 14

Parse Trees and Ambiguity

Rachit Nimavat

September 4, 2025

Outline

The World of Regular Languages

Three Faces of Same Power

The Limit: What Can't Finite Automata Do?

Consider the language $L = \{a^n b^n \mid n \geq 0\} = \{\epsilon, ab, aabb, aaabbb, \dots\}$.

To accept L, a DFA/NFA must **count** the number of a's

DFA/NFA has only a finite number of states and cannot remember an arbitrarily large count n.

Next Step?

We need machine with more power:

- Context-Free Grammars: A way to describe languages with recursive structure
- Pushdown Automata: NFA + Stack

Example

Consider $L = \{a^n b^n \mid n \geq 0\}$ over $\Sigma = \{a, b\}$. Let's build a grammar!

$$S \to \varepsilon$$
$$S \to \mathrm{a}S\mathrm{b}$$

Shorthand: $S \to \varepsilon \mid \mathrm{a}S\mathrm{b}$

Derivation

is a sequence of steps applying the production rules. Generating string aabb:

$$\begin{aligned} S &\to \mathrm{a}S\mathrm{b} \\ &\to \mathrm{aa}S\mathrm{bb} \\ &\to \mathrm{aabb} \end{aligned}$$

Defn: DFA

DFA is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$

'symbol'	description
Q	finite set of states
Σ	underlying finite alphabet
δ	transition function between states
$q_0 \in Q$	start state
$F \subseteq Q$	accepting states

Defn: CFG

A CFG is a set of recursive rules used to generate patterns of strings. Think of it as a blueprint for a language.

Definition 1.1

A Context-Free Grammar is a 4-tuple $G = (V, T, P, S)$, where:

V: Finite set of **variables**. think: syntactic categories or placeholders. (e.g., S, E, T in previous examples)

T: Finite set of **terminals** or **tokens**. actual symbols or words of the language. (e.g., $\mathsf{a}, \mathsf{b}, \mathsf{c}, +, \star$ in previous examples)

P: Finite set of **production rules**. Rules have the form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$, i.e., α is a string consisting of variables and terminals. specifies how to replace variables with strings of variables and terminals. (e.g., $S \to \mathsf{a}S\mathsf{b}$ and $E \to T$) in previous examples

S: The **start symbol** ($S \in V$).

Example - II

CFGs are excellent for defining the syntax of programming languages. Let's build a grammar for expressions over $\{a, b, c\}$.

$$E \to E + E \mid E * E \mid T$$
$$T \to \mathsf{a} \mid \mathsf{b} \mid \mathsf{c}$$

Derivations.

Generating $a * b$

$$\begin{aligned} E &\to E * E \\ &\to T * E \\ &\to T * T \\ &\to a * T \\ &\to a * b \end{aligned}$$

Generating $a + a$

$$\begin{aligned} E &\to E + E \\ &\to T + E \\ &\to T + T \\ &\to a + T \\ &\to a + a \end{aligned}$$

Generating $a + b * c$

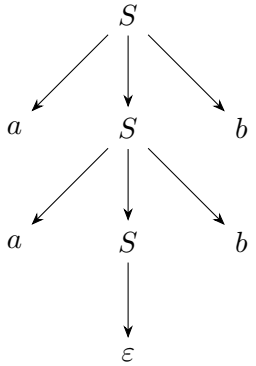
$$\begin{aligned} E &\to E + E \\ &\to E + E * E \\ &\to T + E * E \\ &\to T + T * E \\ &\to T + T * T \\ &\to a + T * T \\ &\to a + b * T \\ &\to a + b * c \end{aligned}$$

Visualizing Derivations: Parse Trees

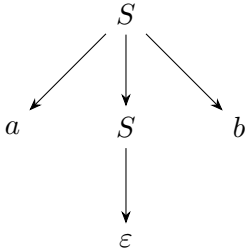
Similar to DFA, wer have **Parse Trees** to visualize derivation in CFGs.
Consider $L = \{a^n b^n \mid n \geq 0\}$ over $\Sigma = \{a, b\}$ and CFG $S \to \varepsilon \mid \mathtt{a}S\mathtt{b}$.

Parse tree for ε:

The parse tree for $aabb$:



Parse tree for ab:



Defn

Consider a CFG $G = (V, T, P, S)$. Given a string $w \in T^*$, a **parse tree** for w is a tree that represents the derivation of w from the start symbol S using the production rules P.

Properties of a Parse Tree

The root node is the start symbol (S).

Each interior node is a variable (V).

Each leaf node is a terminal (T) or ϵ.

If a node A has children $X_1, X_2, \ldots, X_k$, where $X_1, X_2, \ldots \in V \cup T \cup \{\varepsilon\}$, then $A \to X_1 X_2 \ldots X_k$ must be a production rule.

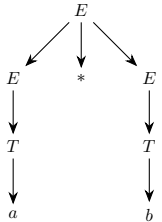
Reading the leaves from left to right yields string w.

Parse trees capture the underlying syntactic structure of the string.

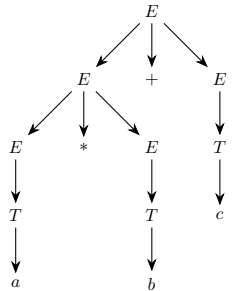
Example

Consider the language of expressions over $\Sigma = \{a, b, c\}$ with CFG:
$E \to E + E \mid E * E \mid T$ and $T \to$ a | b | c.

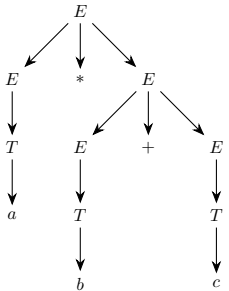
Parsing $a * b$:



Parsing $a * b + c$:



Parsing $a * b + c$:



HW: Think about which parse tree is correct? How do you propose handling ambiguity?

Definition 3.1

A CFG G is **ambiguous** if there exists a string $w \in L(G)$ such that w has more than one distinct parse trees.

Ambiguous CFG has more than one distinct leftmost derivation for some string in the language

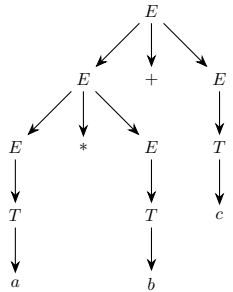
Ambiguous CFG has more than one distinct rightmost derivation for some string in the language

In programming languages, ambiguity means a line of code can be interpreted in multiple ways, which is unacceptable. The compiler must know **exactly** what to do!

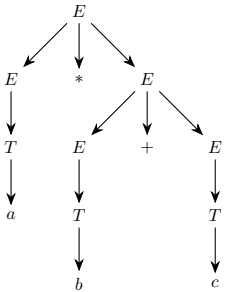
Example - Revisted

Consider the language of expressions over $\Sigma = \{a, b, c, +, *\}$ with CFG:
$E \to E + E \mid E * E \mid T$ and $T \to$ a | b | c.

Parsing $a * b + c$:



Parsing $a * b + c$:



Resolving Ambiguity

Enforcing Precedence and Associativity

Learning from our maths classes, we can enforce precedence and associativity rules in our CFGs to resolve ambiguity. Won't work for all ambiguities though!

The Solution:

Rewrite G with a hierarchy of rules

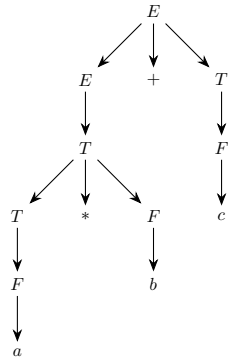
Forces a single, correct interpretation.

$$E \to E + T \mid T$$

$$T \to T * F \mid F$$

$$F \to \mathsf{a} \mid \mathsf{b} \mid \mathsf{c}$$

Unique parse tree of $a * b + c$:



Resolving Ambiguity

Enforcing Precedence and Associativity

The Solution:

Rewrite G with a hierarchy of rules

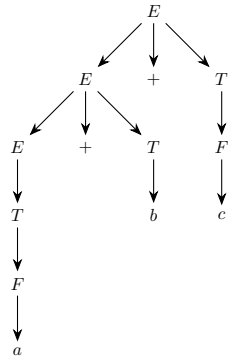
Forces a single, correct interpretation.

$$E \to E + T \mid T$$

$$T \to T * F \mid F$$

$$F \to \text{a} \mid \text{b} \mid \text{c}$$

Unique parse tree of $a + b + c$:



Bonus:

Multiplication is given more precedence than addition What enforces this in G?

Operators are evaluated from left to right What enforces this in G?

Algorithm for Ambiguity Resolution?

No such algorithm!

No algorithm that given a CFG G, decides whether G is ambiguous or not.

There are CFLs for which all possible CFGs are ambiguous.

See, Chapter 5.4 in ALC for more details.