Subject: Theory Of Computation
Topic: CFG

LISNA THOMAS

ACADEMIC YEAR:2020-21

Context Free Grammars

CIS 361

Introduction

•Finite Automata **accept** all regular languages and only regular languages

•Many simple languages are non regular:

- $\{a^n b^n : n = 0, 1, 2, \ldots\}$
- $\{w : w$ a is palindrome$\}$

and there is no finite automata that accepts them.

• context-free languages are a larger class of languages that encompasses all regular languages and many others, including the two above.

Context-Free Grammars

- Languages that are **generated** by context-free grammars are context-free languages

- Context-free grammars are more expressive than finite automata: if a language L is **accepted** by a finite automata then L can be **generated** by a context-free grammar
  - Beware: The converse is NOT true

Context-Free Grammar

Definition. A context-free grammar is a 4-tuple (Σ, NT, R, S), where:

- Σ is an alphabet (each character in Σ is called **terminal**)
- NT is a set (each element in NT is called **nonterminal**)
- R, the set of rules, is a subset of NT $\times$ ($\Sigma \cup$ NT)*

If $(\alpha,\beta) \in$ R, we write production $\alpha \rightarrow \beta$

β is called a **sentential form**

- S, the **start symbol**, is one of the symbols in NT

CFGs: Alternate Definition

many textbooks use different symbols and terms to describe CFG's

$$G = (V, \Sigma, P, S)$$

V	= variables	a finite set
Σ	= alphabet or terminals	a finite set
P	= productions	a finite set
S	= start variable	$S \in V$

Productions' form, where $A \in V$, $\alpha \in (V \cup \Sigma)^*$:

- $A \rightarrow \alpha$

Derivations

Definition. v is **one-step derivable** from u, written $u \Rightarrow v$, if:

- $u = x\alpha z$
- $v = x\beta z$
- $\alpha \rightarrow \beta$ in R

Definition. v is **derivable** from u, written $u \Rightarrow^* v$, if:
There is a chain of one-derivations of the form:

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow v$$

Context-Free Languages

Definition. Given a context-free grammar $G = (\Sigma, NT, R, S)$, the **language generated** or derived from G is the set:

$$L(G) = \{w : S \Rightarrow^* w\}$$

Definition. A language L is context-free if there is a context-free grammar $G = (\Sigma, NT, R, S)$, such that L is generated from G

CFGs & CFLs: Example 1

$\{\mathbf{a}^n\ \mathbf{b}^n \mid n \geq 0\}$

One of our canonical non-RLs.

$S \rightarrow \varepsilon \mid \mathbf{a}\ S\ \mathbf{b}$

Formally: $G = (\{S\}, \{\mathbf{a},\mathbf{b}\},$
$\{S \rightarrow \varepsilon, S \rightarrow \mathbf{a}\ S\ \mathbf{b}\}, S)$

CFGs & CFLs: Example 2

all strings of balanced parentheses

A core idea of most programming languages.
Another non-RL.

$$P \to \varepsilon \mid (P) \mid P\,P$$

CFGs & CFLs: Lessons

- Both examples used a common CFG technique, “wrapping” around a recursive variable.

$S \rightarrow \mathbf{a}\ S\ \mathbf{b}$ $\qquad\qquad$ $P \rightarrow \mathbf{(}\ P\ \mathbf{)}$

CFGs & CFLs: Example 3

$\{\mathbf{a}^m\ \mathbf{b}^n\ \mathbf{c}^{m+n} \mid m,n \geq 0\}$

?

?

Rewrite as $\{\mathbf{a}^m\ \mathbf{b}^n\ \mathbf{c}^n\ \mathbf{c}^m \mid m,n \geq 0\}$:

$S \rightarrow S' \mid \mathbf{a}\ S\ \mathbf{c}$

$S' \rightarrow \varepsilon \mid \mathbf{b}\ S'\ \mathbf{c}$

CFGs & CFLs: Non-Example

$$\{\mathbf{a}^n\ \mathbf{b}^n\ \mathbf{c}^n \mid n \geq 0\}$$

Can't be done; CFL pumping lemma later.

Intuition: Can count to n, then can count down from n, but forgetting n.

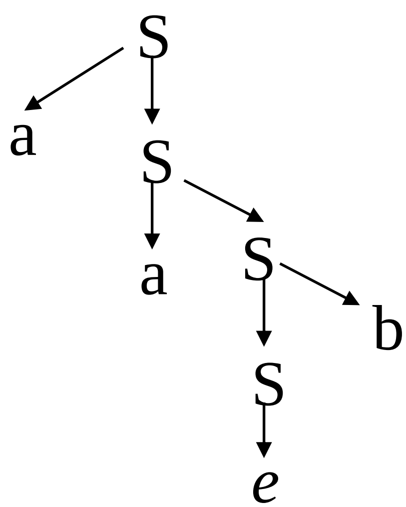
- I.e., a stack as a counter.
- Will see this when using a machine corresponding to CFGs.

Parse Tree

A parse tree of a derivation is a tree in which:

- Each internal node is labeled with a nonterminal

•If a rule $A \rightarrow A_1A_2\ldots A_n$ occurs in the derivation then A is a parent node of nodes labeled A_1, A_2, …, A_n



Parse Trees

$S \rightarrow A \mid A\,B$

$A \rightarrow \varepsilon \mid \mathbf{a} \mid A\,\mathbf{b} \mid A\,A$

$B \rightarrow \mathbf{b} \mid \mathbf{b}\,\mathbf{c} \mid B\,\mathbf{c} \mid \mathbf{b}\,B$

Sample derivations:

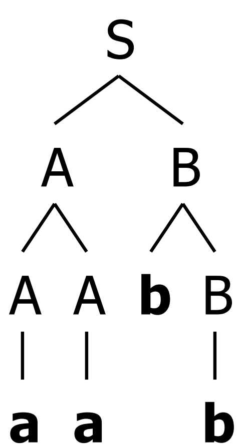
$S \Rightarrow AB \Rightarrow AAB \Rightarrow \mathbf{a}AB \Rightarrow \mathbf{aa}B \Rightarrow \mathbf{aab}B \Rightarrow \mathbf{aabb}$

$S \Rightarrow AB \Rightarrow A\mathbf{b}B \Rightarrow A\mathbf{bb} \Rightarrow AA\mathbf{bb} \Rightarrow A\mathbf{abb} \Rightarrow \mathbf{aabb}$

These two derivations use same productions, but in different orders.

This ordering difference is often uninteresting.

Derivation trees give way to abstract away ordering differences.



Root label = start node.

Each interior label = variable.

Each parent/child relation = derivation step.

Each leaf label = terminal or ε.

All leaf labels together = derived string = *yield*.

Leftmost, Rightmost Derivations

Definition. A **left-most derivation** of a sentential form is one in which rules transforming the left-most nonterminal are always applied

Definition. A **right-most derivation** of a sentential form is one in which rules transforming the right-most nonterminal are always applied

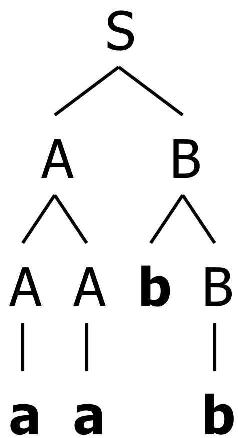
Leftmost & Rightmost Derivations

$S \rightarrow A \mid A\,B$
$A \rightarrow \varepsilon \mid \mathbf{a} \mid A\,\mathbf{b} \mid A\,A$
$B \rightarrow \mathbf{b} \mid \mathbf{b}\,\mathbf{c} \mid B\,\mathbf{c} \mid \mathbf{b}\,B$

Sample derivations:

$S \Rightarrow AB \Rightarrow AAB \Rightarrow \mathbf{a}AB \Rightarrow \mathbf{aa}B \Rightarrow \mathbf{aab}B \Rightarrow \mathbf{aabb}$

$S \Rightarrow AB \Rightarrow A\mathbf{b}B \Rightarrow A\mathbf{bb} \Rightarrow AA\mathbf{bb} \Rightarrow A\mathbf{abb} \Rightarrow \mathbf{aabb}$



These two derivations are special.

1st derivation is *leftmost*.

Always picks leftmost variable.

2nd derivation is *rightmost*.

Always picks rightmost variable.

Left / Rightmost Derivations

- In proofs…
  - Restrict attention to left- or rightmost derivations.
- In parsing algorithms…
  - Restrict attention to left- or rightmost derivations.
  - E.g., recursive descent uses leftmost; `yacc` uses rightmost.

Derivation Trees

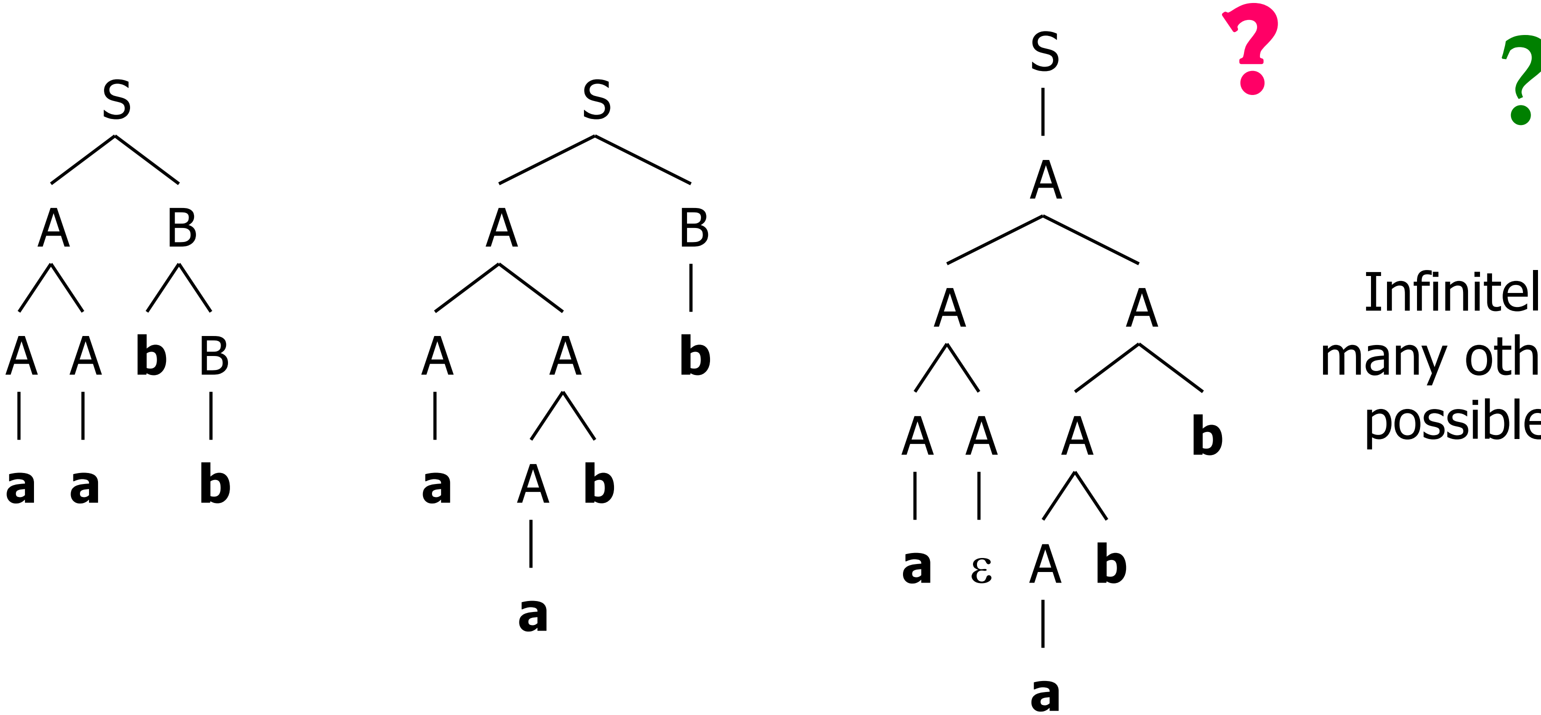
$S \rightarrow A \mid A\ B$

$A \rightarrow \varepsilon \mid \mathbf{a} \mid A\ \mathbf{b} \mid A\ A$

$B \rightarrow \mathbf{b} \mid \mathbf{b\ c} \mid B\ \mathbf{c} \mid \mathbf{b}\ B$

w = **aabb**

Other derivation trees for this string?



Infinitely many others possible.

Ambiguous Grammar

Definition. A grammar G is ambiguous if there is a word $w \in L(G)$ having are least two different parse trees

$$S \rightarrow A$$
$$S \rightarrow B$$
$$S \rightarrow AB$$
$$A \rightarrow aA$$
$$B \rightarrow bB$$
$$A \rightarrow e$$
$$B \rightarrow e$$

Notice that a has at least two left-most derivations

Ambiguity

CFG *ambiguous* $\Leftrightarrow$ any of following equivalent statements:

- $\exists$ string w with multiple derivation trees.
- $\exists$ string w with multiple leftmost derivations.
- $\exists$ string w with multiple rightmost derivations.

Defining ambiguity of <u>grammar</u>, not language.

Ambiguity & Disambiguation

Given an ambiguous grammar, would like an equivalent unambiguous grammar.

- Allows you to know more about structure of a given derivation.
- Simplifies inductive proofs on derivations.
- Can lead to more efficient parsing algorithms.
- In programming languages, want to impose a canonical structure on derivations. E.g., for $\mathbf{1+2}\times\mathbf{3}$.

Strategy: Force an ordering on all derivations.

Disambiguation: Example 1

$$\begin{array}{lll} \text{Exp} & \rightarrow & \mathbf{n} \\ & | & \text{Exp} \mathbin{\mathbf{+}} \text{Exp} \\ & | & \text{Exp} \times \text{Exp} \end{array}$$

What is an equivalent unambiguous grammar?

$$\begin{array}{lll} \text{Exp} & \rightarrow & \text{Term} \\ & | & \text{Term} \mathbin{\mathbf{+}} \text{Exp} \\ \text{Term} & \rightarrow & \mathbf{n} \\ & | & \mathbf{n} \times \text{Term} \end{array}$$

Uses

- operator precedence
- left-associativity

Disambiguation

What is a general algorithm?

None exists!

There are CFLs that are *inherently ambiguous*

Every CFG for this language is ambiguous.

E.g., $\{\mathbf{a}^n\mathbf{b}^n\mathbf{c}^m\mathbf{d}^m \mid n \geq 1, m \geq 1\} \cup \{\mathbf{a}^n\mathbf{b}^m\mathbf{c}^m\mathbf{d}^n \mid n \geq 1, m \geq 1\}$.

So, can't necessarily eliminate ambiguity!

CFG Simplification

Can’t always eliminate ambiguity.

But, CFG simplification & restriction still useful theoretically & pragmatically.

- Simpler grammars are easier to understand.
- Simpler grammars can lead to faster parsing.
- Restricted forms useful for some parsing algorithms.
- Restricted forms can give you more knowledge about derivations.

CFG Simplification: Example

How can the following be simplified?

? ?

$S \rightarrow A\ B$

$S \rightarrow A\ C\ D$

$A \rightarrow A\ \mathbf{a}$

$A \rightarrow \mathbf{a}$

$A \rightarrow \mathbf{a}\ A$

$A \rightarrow \mathbf{a}$

$C \rightarrow \varepsilon$

$D \rightarrow \mathbf{d}\ D$

$D \rightarrow E$

$E \rightarrow \mathbf{e}\ A\ \mathbf{e}$

$F \rightarrow \mathbf{f}\ \mathbf{f}$

1) Delete: B useless because nothing derivable from B.

2) Delete either $A \rightarrow A\mathbf{a}$ or $A \rightarrow \mathbf{a}A$.

3) Delete one of the idential productions.

4) Delete & also replace $S \rightarrow ACD$ with $S \rightarrow AD$.

5) Replace with $D \rightarrow \mathbf{e}A\mathbf{e}$.

6) Delete: E useless after change #5.

7) Delete: F useless because not derivable from S.

CFG Simplification

Eliminate ambiguity.

Eliminate "useless" variables.

Eliminate ε-productions: $A\rightarrow\varepsilon$.

Eliminate unit productions: $A\rightarrow B$.

Eliminate redundant productions.

Trade left- & right-recursion.

Trading Left- & Right-Recursion

Left recursion: $A \rightarrow A\ \alpha$

Right recursion: $A \rightarrow \alpha\ A$

Most algorithms have trouble with one,

In recursive descent, avoid left recursion.