# CONTEXT-FREE LANGUAGES (CFL)

Chap. 5

## Learning Objectives

- Identify whether a particular grammar is Context-Free.
- Discuss the *relationship* between Regular Languages and Context-Free Languages (CFL).
- Construct Context-Free Grammars for simple languages.
- Produce *Leftmost* and *Rightmost derivations* of a string generated by a Context-Free Grammar (CFG).
- Construct *derivation trees* for strings generated by a Context-Free Grammar.
- Show that a Context-Free Grammar is *ambiguous*.
- Rewrite a grammar to remove ambiguity.

## Context-Free Grammars (CFG)

- Many useful languages are not regular.
- Context-Free Grammars are very useful for the definition and processing of programming languages.
- A CFG has *no restrictions* on the *right* side of its productions, while the *left* side must be a single variable.
- A language is Context-Free if it is generated by a CFG.
- Since Regular Grammars are Context-Free, the family of Regular Languages is a *proper subset* of the family of Context-Free Languages: RL $\subset$ CFL.

## Context-Free Languages

- Definition 5.1: A grammar G = (V, T, S, P) is said to be *context-free* if all productions in P have the form

  $A \rightarrow x$,

  where $A \in V$ and $x \in (V \cup T)^*$.

  A language $L$ is said to be *context-free* if and only if there is a context-free grammar G such that $L = L(G)$.
- Example 5.1: Consider the grammar G

  V = { $S$ }, T = { $a, b$ }, and productions

  $S \rightarrow aSa \mid bSb \mid \lambda$ .

  Its sample derivations are:

  $S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabbaa$

  $S \Rightarrow bSb \Rightarrow baSab \Rightarrow baab$
- The language generated by the grammar is

  $L(G) = \{ ww^R \mid w \in \{ a, b \}^* \}$

  : *even-length palindromes* in { $a, b$ }*, CFL. -- cf.) Ex. 4.8

## Example: CFL

- Example 5.2: Consider the grammar G

  V = { S, A, B }, T = { $a$, $b$ }, and productions

  $S \rightarrow abB,\ A \rightarrow aaBb,\ B \rightarrow bbAa,\ A \rightarrow \lambda.$

- Its sample derivations are:

  $S \Rightarrow abB \Rightarrow abbbAa \Rightarrow abbba$

  $S \Rightarrow abB \Rightarrow abbbAa \Rightarrow abbbaaBba \Rightarrow abbbaabbAaba$
  $\Rightarrow abbbaabbaba$

  $S \Rightarrow abB \Rightarrow abbbAa \Rightarrow abbbaaBba \Rightarrow abbbaabbAaba$
  $\Rightarrow abbbaabbaaBbaba \Rightarrow abbbaabbaabbAababa$
  $\Rightarrow abbbaabbaabbababa$

- The language generated by the grammar is

  $L(G) = \{ab(bbaa)^n bba(ba)^n \mid n \geq 0\}$, CFL.

## Example: CFL

- Example 5.3: The language $L = \{ a^n b^m \mid n \neq m \}$ is *CFL*.
- The Context-Free Grammar G, s.t. $L(G) = L$ is:

Case 1: $n > m$

Generate a string with an equal number of $a$'s and $b$'s, i.e. $a^m b^m$,

then add extra $a$'s on the left.

$$S \rightarrow AS_1,\quad S_1 \rightarrow aS_1b \mid \lambda,\quad A \rightarrow aA \mid a.$$

Case 2: $n < m$

Similarly, generate a string with an equal number of $a$'s and $b$'s, $a^n b^n$
then add extra $b$'s on the right.

$$S \rightarrow S_1B,\quad S_1 \rightarrow aS_1b \mid \lambda,\quad B \rightarrow bB \mid b.$$

Thus, CFG G s.t. $L(G) = \{ a^n b^m \mid n \neq m \}$ is:

V = {$S$, $S_1$, $A$, $B$}, T = { $a$, $b$ }, and productions

$S \rightarrow AS_1 \mid S_1B,\quad S_1 \rightarrow aS_1b \mid \lambda,\quad A \rightarrow aA \mid a,\quad B \rightarrow bB \mid b.$

# Example: CFL

- Example 5.4: Consider the grammar

  V = { S }, T = { $a$, $b$ }, and productions

  $S \to aSb \mid SS \mid \lambda$ .
- Its sample derivations are:

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

  $S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \Rightarrow abab$
- The language generated by the grammar is

  $\{ w \in \{ a, b \}^* \mid n_a(w) = n_b(w)$ and

  $n_a(v) \geq n_b(v)$ where $v$ is any prefix of $w$ },

  -- CFL.

# Leftmost and Rightmost Derivations

- Definition 5.2:
- In a *leftmost derivation*, the *leftmost variable* in a sentential form is replaced at each step.
- In a *rightmost derivation*, the *rightmost variable* in a sentential form is replaced at each step.
- Example 5.5:

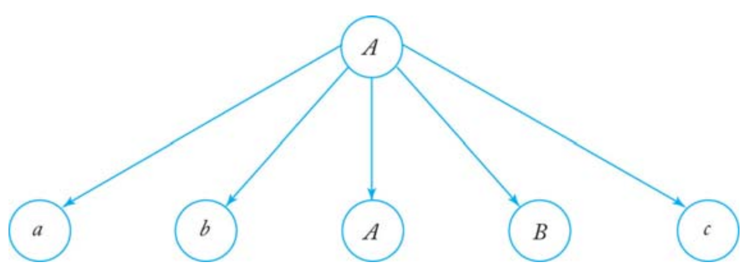
  V = { $S$, $A$, $B$ }, T = { $a$, $b$ }, and productions

  $S \to aAB$, $A \to bBb$, $B \to A \mid \lambda$
- The string *abb* has two distinct derivations:
  - Leftmost deriv.: $S \Rightarrow aAB \Rightarrow abBbB \Rightarrow abbB \Rightarrow abb$
  - Rightmost der. : $S \Rightarrow aAB \Rightarrow aA \Rightarrow abBb \Rightarrow abb$

## Derivation Trees

- A way to show derivations, independent of the order in which productions are used.
- Definition 5.3: An ordered tree is a *derivation tree* or *parse tree* of a CFG $G=(V, T, S, P)$, iff it has the following properties:
  - the *root* is labeled $S$.
  - Every *leaf* has a label from $T \cup \{\lambda\}$.
  - Every *internal nodes* are labeled from a variable $V$, on the *left* side of a production.
  - the children of an internal node labeled (from left to right) $a_1, a_2, .., a_n$, are contained on the corresponding right side of a production on the form: $A \rightarrow a_1 a_2 \ldots a_n$.
  - A leaf labeled $\lambda$ has no siblings.

## Derivation Trees




For example,

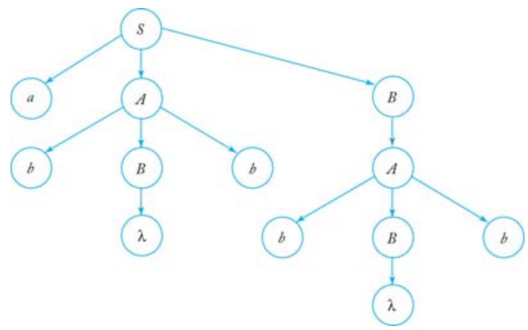
- the production $A \rightarrow abABc$ shows the corresponding partial derivation tree.

## Derivation Trees (Cont.)

- The yield of a derivation tree is the string of terminals produced by *a leftmost depth-first traversal* of the tree.
- Example 5.5: Using the production of grammar

  $S \rightarrow aAB, \quad A \rightarrow bBb, \quad B \rightarrow A \mid \lambda$

  the derivation tree below yields the string *abbbb*.




## Sentential Forms and Derivation Trees

- Theorem 5.1: Given a CFG G, for every string $w \in L(G)$, there exists a *derivation tree* that yields *w:*

  $\forall w \in L(G)$, $\exists$ a derivation tree, $root(S) \Rightarrow^* w$
- The converse is also true: the yield of any derivation tree formed with productions from G is in L(G):

  $\forall w$ *s.t.* $root(S) \Rightarrow^* w$ in any derivation tree, $w \in L(G)$.
- Derivation trees show which productions are used in obtaining a sentence, but do not give the order of their application.

## Sentential Forms and Derivation Trees

- Theorem 5.1: Proof →)

Show that for every sentential form of L(G), there is a corresponding partial derivation tree. Prove it by induction on the *number of steps* in the derivation.

Basis: True for every sentential form derivable in one step.

Since $S \Rightarrow u$ implies that there is a production $S \to u$, this follows immediately from Def. 5.3.

I.H: Assume that for every sentential form derivable in $n$ steps, there is a corresponding partial derivation tree.

I.S: Any $w$ derivable in $n$+1 steps must be such that

$S \Rightarrow^* xAy$, $x, y \in (V \cup T)^*$, $A \in V$, in $n$ steps,

and $xAy \Rightarrow xa_1a_2 \cdots a_m y$, $a_i \in V \cup T$.

By the I. H., there is a partial derivation tree with yield $xAy$, and since the grammar must have production $A \to a_1a_2 \cdots a_m$, we see that by expanding the leaf labeled A, we get a partial derivation tree with yield $xa_1a_2 \cdots a_m y = w$.

Therefore, the result is true for all sentential forms.

Proof ←) Similar.

## Parsing and Membership

- The *Parsing Problem*:
  - Given a grammar G and a string $w$, find a sequence of derivations using the productions in G to produce $w$.
  - Can be solved in top-down parsing such as a so-called *exhaustive search parsing* (or *brute force parsing*), but not very efficient fashion.
  - Excessively large number of sentential forms are generated.
  - Exhaustive parsing is guaranteed to yield all strings in L(G) (i.e. $w \in L(G)$) eventually, but it may never terminate for a string not in L(G).

## Parsing and Membership

- Theorem 5.2: Suppose G = (V, T, S, P) is a CFG that doesn't have any rules of the forms

  $A \rightarrow \lambda$ ($\lambda$ production) or

  $A \rightarrow B$ (unit production), where $A, B \in V$.

  Then, the exhaustive parsing decides/stops for $w \notin L(G)$.

i.e. the exhaustive parsing can be made into

an (membership) *algorithm* that, for any $w \in \Sigma^*$,

either produces a parsing of w or tells us that no parsing is possible.

- Problems with $A \rightarrow \lambda$ :

  It can be used to decrease the length of successive sentential forms, so that we can't tell easily when to stop.

- Problems with $A \rightarrow B$:

  It can be used to increase the length of successive sentential forms, so that it might yield a cycle or derivation, never stop the derivation.

## Parsing and Membership

- Theorem 5.2: proof)

Since the exhaustive parsing can generate excessively large number of sentential forms though its exact numbers depend on a given production rule, let's put a *rough upper bounds* on it.

If we restrict to the leftmost derivation,

- After one round, we can have no more than $|P|$ sentential forms,
- After the $2^{nd}$ round, no more than $|P|^2$ sentential forms,
- … etc. may be generated.

Since the parsing can't involve more than $2 \cdot |w|$ rounds,

The total number of sentential forms can not exceed

$$M = |P| + |P|^2 \ldots + |P|^{2|w|} = O( P^{2|w|+1} )$$

It indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive.

# Parsing and Membership (cont.)

- Example 5.7: Consider the production rules of G and $w = aabb$:

  $S \to SS \mid aSb \mid bSa \mid \lambda.$

Round 1: 1) $S \Rightarrow SS$, 2) $S \Rightarrow aSb$, 3) $S \Rightarrow bSa$, 4) $S \Rightarrow \lambda$.

Round 2: By 1): 1) $S \Rightarrow^{1)} SS \Rightarrow^{1)} SSS$, 2) $S \Rightarrow^{1)} SS \Rightarrow^{2)} aSbS$,

3) $S \Rightarrow^{1)} SS \Rightarrow^{3)} bSaS$, 4) $S \Rightarrow^{1)} SS \Rightarrow^{4)} S$.

By 2): 1) $S \Rightarrow^{2)} aSb \Rightarrow^{1)} aSSb$, 2) $S \Rightarrow^{2)} aSb \Rightarrow^{2)} aaSbb$,

3) $S \Rightarrow^{2)} aSb \Rightarrow^{3)} abSab$, 4) $S \Rightarrow^{2)} aSb \Rightarrow^{4)} ab$.

Round 3: 2) $S \Rightarrow^{2)} aSb \Rightarrow^{2)} aaSbb \Rightarrow^{4)} aabb = w$ . Thus, $aabb \in L(G)$!

Claim: Parse w'=$abbb$ to decide $w' \in L(G)$?

- Example 5.8: Consider G' from G in Ex.5.7 with the production rules

  $S \to SS \mid aSb \mid bSa \mid ab \mid ba$ without $\lambda$.

Then, L(G') = L(G) $- \{\lambda\}$.

Given any $w \in \{a, b\}^+$, the exhaustive parsing always terminates within |w| (i.e. $\leq$ |w|) rounds because the length of the sentential form grows by at least one symbol in each round. After |w| rounds, we have either produced a parsing or we know that $w \notin L(G)$.

# Parsing and Membership (cont.)

- Definition 5.4: A CFG G=(V, T, S, P) is said to be a *simple grammar* (or *s-grammar*) if all its productions are of the form

  $A \to ax$, where

  $A \in V$, $a \in T$, $x \in V^*$, and any pair $(A, a)$ occurs *at most once* in $P$.
- Example 5.9:
  - A CFG with production rules

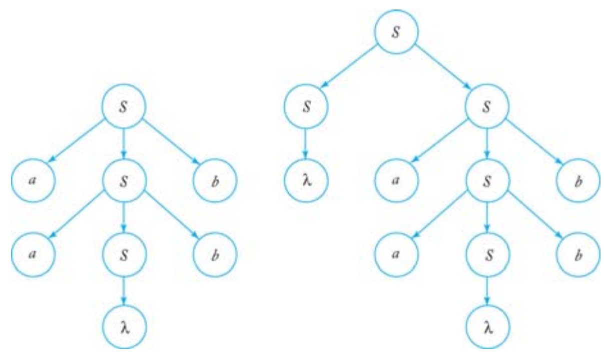
    $S \to aS \mid bSS \mid c$ is an s-grammar.
  - A CFG with $S \to aS \mid bSS \mid aSS \mid c$ is *not an s-grammar* because the pair $(S, a)$ occurs in the two productions $S \to aS$ and $S \to aSS$.

## Parsing and Ambiguity

- Definition 5.5:

  A CFG G is *ambiguous* if there exists some string $w \in L(G)$ that has *more than one derivation tree*.

- Example 5.10: The grammar with productions S → *aSb* | *SS* | λ is ambiguous since the string *aabb* has two derivation trees.




## Ambiguity in Programming Languages

Example 5.11:

- Consider the CFG G, designed to generate simple arithmetic expressions such as $(a+b)*c$ and $a*b+c$.

  G = (V, T, E, P) with

  V = { E, I }, T = { *a, b, c,* +, ∗, (, ) }, and productions

  $E \rightarrow I \mid E+E \mid E*E \mid (E)$

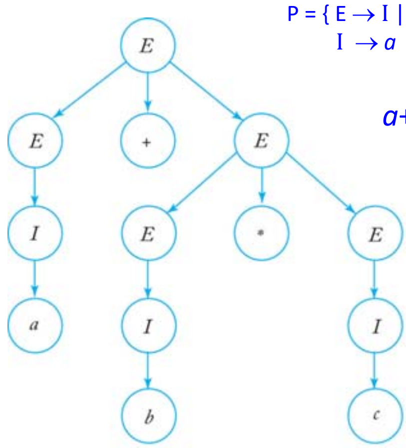
  $I \rightarrow a \mid b \mid c$

- The grammar G is *ambiguous* because strings such as $a+b*c$ have more than one derivation tree, as shown in Figure 5.5

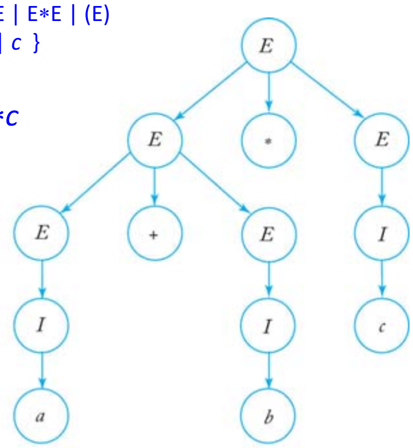
# Ex. 5.11: (cont.) Derivation Trees from Ambiguous Grammar

V = { E, I }, T = { *a, b, c*, +, $*$, (, ) },
P = { E $\rightarrow$ I | E+E | E$*$E | (E)
I $\rightarrow$ *a* | *b* | *c* }

*a*+*b*$*$*c*




(a)




(b)

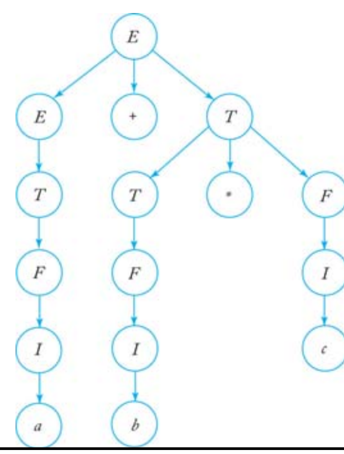
# Resolving Ambiguity

- Ambiguity can often be removed by rewriting the grammar so that only one parsing is possible.
- Consider the grammar G'

Cf) E $\rightarrow$ I | E+E | E$*$E | (E)
I $\rightarrow$ *a* | *b* | *c* }

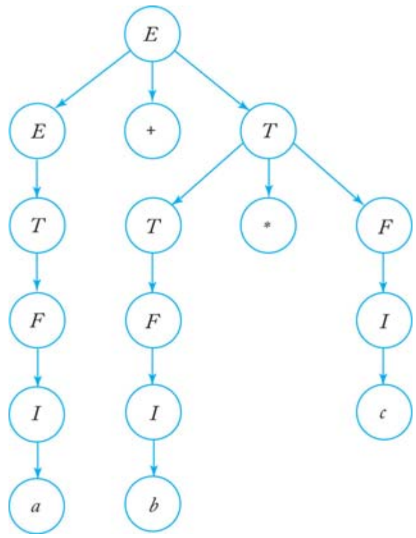
V = { E, $T_0$, F, I }, T = { *a, b, c*, +, $*$, (, ) }, and productions

$E \rightarrow T_0$
$T_0 \rightarrow F$
$F \rightarrow I$
$E \rightarrow E + T_0$
$T_0 \rightarrow T_0 * F$
$F \rightarrow (E)$
$I \rightarrow a \mid b \mid c$




- Only one derivation tree yields the string *a*+*b*$*$*c*

# Derivation Tree for *a+b∗c* using Unambiguous Grammar G’




# Ambiguous Languages

- For *some* languages, it is always possible to find an unambiguous grammar, as shown in the previous examples.
- However, there are *inherently ambiguous* languages, for which every possible grammar is ambiguous.
- Definition 5.6:

  If *L* is a CFL for which there exists an unambiguous grammar, then *L* is said to be unambiguous.

  If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

# Ambiguous Languages

- Example 5.13: The language $L=\{ a^n b^n c^m \} \cup \{ a^n b^m c^m \}$, $n, m \geq 0$, is *Inherently ambiguous CFL.*

Claim: *Generate a CFG that generates L. Then, L is CFL.*

Let $L=\{ a^n b^n c^m \} = L_1$ and $\{ a^n b^m c^m \} = L_2$. Then, $L = L_1 \cup L_2$.

The CFG $G_1$ that generates $L_1$ is: $S_1 \to S_1 c | A$, $A \to aAb | \lambda$ and

the CFG $G_2$ that generates $L_2$ is: $S_2 \to aS_2 | B$, $B \to bBc | \lambda$ .

Then, L is generated by $G_1 \cup G_2$: $S \to S_1 \mid S_2$

This grammar (and every other equivalent grammar) is ambiguous, because any string of the form $a^n b^n c^n$ has two distinct derivations. Thus, $L$ is inherently ambiguous.

# CFG and Programming Languages

- Application of Theory of Formal language:

  the definition of programming languages (PL) and the construction of interpreters and compilers for them -- Regular language and CFL.
- Define a PL by a grammar in Backus-Naur form (BNF).
- In BNF, Variables are enclosed in triangular bracket while Terminal symbols are with no marking.

  <*expression*> ::= <*term*> | <*expression*> + <*term*>

  <*term*> ::= <*factor*> | <*term*> * <*factor*>, where +, * ∈ Terminal, etc.
- In C-like PL, <*while statement*> ::= ***while*** <*expression*> <*statement*> .
- The aspects of a PL that can be modeled by a CFG are its syntax. However, not all syntactically correct programs are acceptable programs.

  e.g.) In C,
  ```
  char a, b, c;
  c = 3.2;
  ```
  This is syntactically correct but semantically incorrect (i.e. unacceptable)