Generality of Grammars

- For now, we will concentrate on **string** grammars, grammars for generating languages that are sets of strings.

- Other kinds of grammars can be used to generate graphs, trees, pictures, . . .

- There are other formal systems similar to grammars for generating languages that are not grammars. An example is the family of “L systems”.
Definition of Grammars

- A grammar consists of 4 parts:
  - Terminal alphabet $\Sigma$
  - Auxiliary (aka “non-terminal” or “variables”) alphabet $N$
  - Productions (to be defined)
  - Start symbol $S \in N$
Productions

- A grammar consists of 4 parts:
  - Terminal alphabet $\Sigma$
  - Auxiliary alphabet $N$ (such that $N \cap \Sigma = \emptyset$)
  - A finite set of productions $\rightarrow$
  - Start symbol $S \in N$

- Each production is a pair of strings $x \rightarrow y$, where $x \in (\Sigma \cup N)^+$, $y \in (\Sigma \cup N)^*$
  [recall $L^+ = L \cup L^2 \cup L^3 \cup \ldots$, whereas $L^* = \{\varepsilon\} \cup L^+$]

- A production enables rewriting a substring $x$ of a derived string as the string $y$.

- $x$ cannot be $\varepsilon$ because there is nothing to rewrite in that case.
Grammars are Generally Non-Deterministic

- A grammar specifies whether one string can be rewritten as another.
- It does not specify a certain required order.
Nondeterminism in a Grammar with $\Sigma = \{a, b\}$

- Productions
  - $S \rightarrow ab$
  - $S \rightarrow aSb$
  - $S \rightarrow SS$

- Choices (Not a Derivation Tree)

```
S          choices
  / \            /
ab   aSb
```

```
ab          choices
  / \            /
 aabb   aaSbb  aSSb
```

```
aabSb   aSabb
```
Sub-Derivation in a Grammar

- A **sub-derivation** in a grammar is a sequence of strings
  \( x_0 \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \ldots \) each in \( (\Sigma \cup N)^* \) where:
  - For each \( i, x_i \Rightarrow x_{i+1} \) means that there are strings \( u, v, w \) such that
    - \( x_i = uvw \),
    - \( x_{i+1} = uv'w \),
    - \( v \rightarrow v' \) is a production
  - If \( x_0 \) is the start symbol, then the sub-derivation is called a **derivation**.
Yield vs. Language Generated

- The **yield** of a non-terminal is the set of strings that can be derived from the non-terminal as a starting point.

- The **yield of a grammar** is the yield of the start symbol.

- The **language generated** by a grammar is the subset of the yield consisting of only terminal strings.
Yield of a Grammar with \( \Sigma = \{a, b\} \)

- Productions:
  - \( S \rightarrow ab \)
  - \( S \rightarrow aSb \)
  - \( S \rightarrow SS \)

- Examples of strings in the **yield**: the strings between arrows below:
  1. \( S \Rightarrow ab \)
  2. \( S \Rightarrow aSb \Rightarrow aabb \)
  3. \( S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb \)
  4. \( S \Rightarrow SS \Rightarrow abS \Rightarrow abab \)
  5. \( S \Rightarrow SS \Rightarrow SSS \Rightarrow ababab \)
  6. \( S \Rightarrow SS \Rightarrow aSbS \Rightarrow aabbS \Rightarrow aabbaSb \Rightarrow aabbaabb \)
Language of a Grammar with $\Sigma = \{a, b\}$

- Productions:
  1. $S \rightarrow ab$
  2. $S \rightarrow aSb$
  3. $S \rightarrow SS$

- Examples of strings in the language: the strings on the right end of each line:
  1. $S \Rightarrow ab$
  2. $S \Rightarrow aSb \Rightarrow aabb$
  3. $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$
  4. $S \Rightarrow SS \Rightarrow abS \Rightarrow abab$
  5. $S \Rightarrow SS \Rightarrow SSS \Rightarrow ababab$
  6. $S \Rightarrow SS \Rightarrow aSbS \Rightarrow aabbS \Rightarrow aabbaSb \Rightarrow aabbaaabb$
Grammar for Additive Arithmetic Expressions

- The start symbol is $A$.
- The terminals are $\{a, b, c, +\}$.
- The productions are:
  - $A \rightarrow V$
  - $A \rightarrow V + A$
  - $V \rightarrow a$
  - $V \rightarrow b$
  - $V \rightarrow c$
- Sample derivations:
  1. $A \Rightarrow V \Rightarrow a$
  2. $A \Rightarrow V \Rightarrow c$
  3. $A \Rightarrow V + A \Rightarrow c + A \Rightarrow c + V \Rightarrow c + a$
  4. $A \Rightarrow V + A \Rightarrow c + A \Rightarrow c + V + A \Rightarrow c + b + A \Rightarrow c + b + V \Rightarrow c + b + a$
Grammars vs. Machines

• Why sometimes prefer a grammar to a machine that accepts the same language?

• Grammars are **declarative**, pure. They do not necessarily introduce assumptions about how the language will be recognized.

• Grammars are often (but not always) more **succinct**.
Types of Grammars

- Grammars are classified by the kinds of productions they allow, from least restrictive to most restrictive:
  - Type 0: no restriction on productions
  - Type 1: length of LHS ≤ length of RHS, or $S \rightarrow \varepsilon$ (where $S$ is the start symbol)
  - Type 2: LHS is a single auxiliary only
  - Type 3: LHS is a single auxiliary, and RHS is either $\varepsilon$, or $\sigma A$ where $A \in N$ and $\sigma \in \Sigma$
Names for Types of Grammars

- Type 0: **phrase-structure** grammar
- Type 1: **context-sensitive** grammar
- Type 2: **context-free** grammar
- Type 3: **right-linear** grammar

These types are called the “Chomsky Hierarchy”, after linguist Noam Chomsky, who first identified them.

Language classes corresponding to grammars

**Type 0:** phrase-structure or “recursively-enumerable” or “recognizable” languages

**Type 1:** context-sensitive languages

**Type 2:** context-free languages

**Type 3:** right-linear languages

http://en.wikipedia.org/wiki/Chomsky_hierarchy
Example of a Type 1 Grammar Language \( \{a^n b^n c^n \mid n > 0\} \)

- The grammar:
  - Terminal alphabet \( \Sigma = \{a, b, c\} \)
  - Auxiliary alphabet \( \{S, B, W\} \)
  - Productions \( \rightarrow \):
    - \( S \rightarrow abc \quad WX \rightarrow BX \)
    - \( S \rightarrow aSBc \quad BX \rightarrow Bc \)
    - \( cB \rightarrow WB \quad bB \rightarrow bb \)
    - \( WB \rightarrow WX \)
  - Start symbol \( S \)

- A sample derivation is shown on the next page.
- \( WX \) is a “mobile” version of \( cb \). The productions \( WB \rightarrow WX \), \( WX \rightarrow BX \), \( BX \rightarrow Bc \) have the effect of interchanging \( cb \) to get \( bc \).

<table>
<thead>
<tr>
<th>Step</th>
<th>String so Far</th>
<th>Production Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S → aSBc</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>aSBC</td>
<td>S → aSBc</td>
</tr>
<tr>
<td>2</td>
<td>aaSBCBc</td>
<td>S → abc</td>
</tr>
<tr>
<td>3</td>
<td>aaabcBcBc</td>
<td>cB → WB</td>
</tr>
<tr>
<td>4</td>
<td>aaabWBcBc</td>
<td>WB → WX</td>
</tr>
<tr>
<td>5</td>
<td>aaabWXcBc</td>
<td>WX → BX</td>
</tr>
<tr>
<td>6</td>
<td>aaabBXcBc</td>
<td>BX → Bc</td>
</tr>
<tr>
<td>7</td>
<td>aaabBccBc</td>
<td>cB → WB</td>
</tr>
<tr>
<td>8</td>
<td>aaabBcWBc</td>
<td>WB → WX</td>
</tr>
<tr>
<td>9</td>
<td>aaabBcWXc</td>
<td>WX → BX</td>
</tr>
<tr>
<td>10</td>
<td>aaabBcBXc</td>
<td>BX → Bc</td>
</tr>
<tr>
<td>11</td>
<td>aaabBcBcc</td>
<td>cB → WB</td>
</tr>
<tr>
<td>12</td>
<td>aaabBWBcc</td>
<td>WB → WX</td>
</tr>
<tr>
<td>13</td>
<td>aaabBWXcc</td>
<td>WX → BX</td>
</tr>
<tr>
<td>14</td>
<td>aaabBBXcc</td>
<td>BX → Bc</td>
</tr>
<tr>
<td>15</td>
<td>aaabBBccc</td>
<td>bB → bb</td>
</tr>
<tr>
<td>16</td>
<td>aaabbbBccc</td>
<td>bB → bb</td>
</tr>
<tr>
<td>17</td>
<td>aaabbbcccc</td>
<td></td>
</tr>
</tbody>
</table>
A language is **regular** iff it is generated by some type 3 grammar.

- Type 3 productions are of one of two types:
  - $B \rightarrow \sigma C$, where $B, C \in A, \sigma \in \Sigma$
  - $B \rightarrow \varepsilon$

- To prove this result, identify the states of a NFA with auxiliaries in the grammar. Assume a single start state and no $\varepsilon$-transitions (WLOG!).
  - $B \rightarrow \sigma C$ is a production if state $B$ goes to state $C$ via symbol $\sigma$.
  - $B \rightarrow \varepsilon$ is a production iff $B$ is an accepting state in the NFA.

- The language generated by the grammar is the language generated by the NFA. The only way to get rid of the auxiliary in the derived string is to use the production $B \rightarrow \varepsilon$, which corresponds to the NFA being in an accepting state.
Example: NFA vs. Grammar

NFA:

Grammar:
- Start symbol is S
- Productions:
  
  \[
  \begin{align*}
  S & \rightarrow 0S \\
  S & \rightarrow 0C \\
  S & \rightarrow 1B \\
  B & \rightarrow 1S \\
  B & \rightarrow 0C \\
  C & \rightarrow 0B \\
  B & \rightarrow 1C \\
  B & \rightarrow \varepsilon
  \end{align*}
  \]
- \# of productions = \# of arcs + \# of accepting states
There are context-free languages that are not regular.

- \( \{0^n1^n \mid n \geq 0\} \) is known to be non-regular.

- The following context-free grammar generates it:
  - \( S \rightarrow 0S1 \)
  - \( S \rightarrow \varepsilon \)
Summary

- Context-free grammars are strictly more powerful than Type 3 grammars (and thus DFA’s and regular expressions).

- However, as concerns regular expressions, context-free grammars sometime offer a more direct way to represent a regular language.
Grammars vs. Regular Expressions

• Every regular expression corresponds to a context-free grammar in a natural way.

• Each sub-regular-expression and its language is identifiable with an auxiliary or a terminal symbol. The productions are:
  • \( R \rightarrow ST \) if \( R \) is a concatenation of languages
  • \( R \rightarrow S \) and \( R \rightarrow T \) if \( R \) is a union of languages for \( S \) and \( T \)
  • \( R \rightarrow SR \) and \( R \rightarrow \varepsilon \) if \( R \) is \( S^* \)
  • \( R \rightarrow \sigma \) if \( \sigma \in \Sigma \)
  • \( R \rightarrow \varepsilon \) if \( R \) is \( \varepsilon \)
  • none if \( R \) is \( \emptyset \)
Example

- Regular expression: $0((10)^* \cup 01)^*$
- Context-Free Grammar
  
  - $R \rightarrow ST$  
  
  - $S \rightarrow 0$  
  
  - $T \rightarrow VT$  
  
  - $T \rightarrow \varepsilon$  
  
  - $V \rightarrow W$  
  
  - $V \rightarrow X$  
  
  - $W \rightarrow YW$  
  
  - $W \rightarrow \varepsilon$  
  
  - $Y \rightarrow 10$  
  
  - $X \rightarrow 01$
Closure Properties

- From the previous discussion, it can easily be seen that context-free languages are closed under:
  - union
  - product (concatenation)
  - star operator

- Similarly, we can easily see that context free languages are closed under **reversal**, **prefix**, **suffix**, etc.

- We will eventually see that, unlike regular languages, context-free languages are **not closed under intersection**.
Closure Under Substitution (Homomorphism)

• Suppose that L is a language over Σ.
• By a **substitution map**, we mean a function that assigns to each element of Σ a string from an alphabet Δ.

  • Example: Σ = {0, 1}, Δ = {a, b, c}, s(0) = ab, s(1) = cbaba.

• We can “extend” s to map any language over Σ by simply applying s to the letters in each string in the language and concatenating the results for that string.

  • Example: L = {1}*{0}
    s(L) = {cbaba}*{ab}
Both regular and context-free languages are closed under substitution mapping.
Grammar Shorthands: | 

- Suppose that productions are:
  - $A \rightarrow V$
  - $A \rightarrow A + V$
  - $V \rightarrow a$
  - $V \rightarrow b$
  - $V \rightarrow c$

- Group by common left-hand sides
- Use | (read “or”) to represent **alternatives**:
  - $A \rightarrow V \mid A + V$
  - $V \rightarrow a \mid b \mid c$

- Note: | “binds more loosely” than other symbols.
- Same grammar, just a briefer notation.
- | is like union in regular expressions.
Grammar Shorthands: Meta-symbol *

• Suppose that productions are:
  • $A \rightarrow V$
  • $A \rightarrow A + V$

• So
  • $A \Rightarrow V$
  • $A \Rightarrow A + V \Rightarrow V + V$
  • $A \Rightarrow A + V \Rightarrow A + V + V \Rightarrow V + V + V$

• So the combination generates $(V+)^* V$

• Therefore just us instead of the combination:
  • $A \rightarrow (V+)^* V$
Derivation Tree Visualization

\[ A \rightarrow V | V + A \]
\[ V \rightarrow a | b | c \]

Terminal string = “fringe” of tree = “c + a + b”
Derivation Tree Advantage

• The derivation tree has an advantage over linear derivations using ⇒.

• Many different derivations can be shown using a single tree.

• These derivations are, in some sense, equivalent.

• Exercise: List all derivations corresponding to the tree on the previous page.
Ambiguity

- Derivation trees are often used, e.g. in compilers, to assign meaning to generated strings.

- If a string has more than one derivation tree, it is called ambiguous.

- An ambiguous grammar is one that generates at least one ambiguous string.

- Ambiguity is usually undesirable when we must assign a meaning to strings in the language.
Ambiguity

- Consider the grammar
  - $A \rightarrow V \mid A \ast A \mid A + A$
  - $V \rightarrow a \mid b \mid c$

- Show that this grammar is ambiguous.
Aside: Inherent Ambiguity

• For a given language, there may be both ambiguous and unambiguous context-free grammars.

• A language that has no unambiguous context-free grammar is called **inherently ambiguous**.

• An example, which we don’t prove here, of such a language is

\[ \{a^n b^n c^m d^m | n, m > 0\} \cup \{a^n b^m c^m d^n | n, m > 0\} \]
Chomsky Normal Form (CNF)

• This special form for a context-free grammar has a number of important uses. The definition of CNF is:

• Every production, with one possible
• exception, has one of these two forms:

\[ A \to BC, \text{ where } B \text{ and } C \text{ are non-terminals} \]

\[ A \to \sigma, \text{ where } \sigma \in \Sigma \]

• Exception: \( S \to \varepsilon \) is allowed, provided \( S \) is the start symbol and \( S \) is not on the right-hand side of any production.
Conversion to CNF in 5 steps

All of the following steps preserve the language generated.

1. Add a new start symbol $S_0$ and add a production $S_0 \rightarrow S$, where $S$ is the original start symbol.

2. For every production of the form $A \rightarrow \epsilon$, where $A$ is not the start symbol, remove that production and add in its place new productions that represent the effect of the removed production: If $B \rightarrow uAv$, then add $B \rightarrow uv$ where $u, v \in (\Sigma \cup N)^*$ If $A$ occurs multiple times on the RHS, then one production is added for each occurrence of $A$. 
Conversion to CNF continued

3. For each unit production $A \rightarrow B$, where $A, B \in N$, remove that production and for each production of the form $B \rightarrow u$, add $A \rightarrow u$, unless the latter is the same as the unit rule being removed.

4. For productions with RHS longer than 2, remove the production, and add new auxiliaries and productions so that all productions have a RHS at most 2. (See illustration next page.)

5. If a production has RHS length 2 and contains terminals, add new auxiliaries and productions to replace these productions. (See illustration next page+1.)
Illustration of Rule 4

- If $A \rightarrow x_1x_2x_3...x_n$, where $n > 2$, we **remove** it and **add** new auxiliaries $B_2$, ..., $B_{n-1}$ and **add** new productions
  
  $A \rightarrow x_1B_2$
  $B_2 \rightarrow x_2B_3$
  $B_3 \rightarrow x_3B_4$  Each new RHS is length 2.
  ...
  $B_{n-1} \rightarrow x_{n-1}x_n$

  So that $A \Rightarrow x_1B_2 \Rightarrow x_1x_2B_3 \Rightarrow ... \Rightarrow x_1x_2...x_{n-1}x_n$ thus preserving the language.
Illustration of Rule 5

- If $A \rightarrow x_1 x_2$, where either $x_i \in \Sigma$, we replace it with $X_i \in N$, where $X_i$ is a new auxiliary, and add a new production
  
  $$X_i \rightarrow x_i$$

So that $A \Rightarrow X_1 X_2 \Rightarrow x_1 X_2 \Rightarrow x_1 x_2$ thus preserving the language.
Example Conversion to CNF

Consider the grammar with start symbol $S$ and $\Sigma = \{\(', \')\}$:

- $S \rightarrow (L)$
- $L \rightarrow \varepsilon$
- $L \rightarrow SL$

Step 1: New start symbol $S_0$:

- $S_0 \rightarrow S$
- $S \rightarrow (L)$
- $L \rightarrow \varepsilon$
- $L \rightarrow SL$
Example Conversion to CNF

\[
\begin{align*}
S_0 & \to S \\
S & \to (L) \\
L & \to \varepsilon \\
L & \to SL \\
\end{align*}
\]

Step 2: Remove \( \varepsilon \) productions \((L \to \varepsilon)\)

\[
\begin{align*}
S_0 & \to S \\
S & \to (L) \mid () \\
L & \to SL \mid S \\
\end{align*}
\]
Example Conversion to CNF

\[
\begin{align*}
S_0 & \rightarrow S \\
S & \rightarrow (L) \mid () \\
L & \rightarrow SL \mid S
\end{align*}
\]

Step 3: Remove unit productions

\[
(S_0 \rightarrow S, \ L \rightarrow S)
\]

\[
\begin{align*}
S_0 & \rightarrow (L) \mid () \\
S & \rightarrow (L) \mid () \\
L & \rightarrow SL \mid (L) \mid ()
\end{align*}
\]
Example Conversion to CNF

\[
\begin{align*}
S_0 & \rightarrow (L) \mid () \\
S & \rightarrow (L) \mid () \\
L & \rightarrow SL \mid (L) \mid ()
\end{align*}
\]

Step 4: Fix RHS length > 2

\[
\begin{align*}
S_0 & \rightarrow (B \mid () \\
B & \rightarrow L) \\
S & \rightarrow (B \mid () \\
L & \rightarrow SL \mid (B \mid ()
\end{align*}
\]

(To reduce complexity, we reused B.)
Example Conversion to CNF

\[
\begin{align*}
S_0 & \rightarrow (B \mid () \\
B & \rightarrow L) \\
S & \rightarrow (B \mid () \\
L & \rightarrow SL \mid (B \mid ())
\end{align*}
\]

Step 5: Make all RHSs > 1 into non-terminals.
\[
\begin{align*}
S_0 & \rightarrow CB \mid CD \\
B & \rightarrow LD \\
S & \rightarrow CB \mid CD \\
L & \rightarrow SL \mid CB \mid CD \\
C & \rightarrow ( \\
D & \rightarrow )
\end{align*}
\]

The result is Chomsky Normal Form.
There are languages that are type 1 but not type 2.

- \{a^k b^k c^k \mid k > 0\} can be shown to be type 1. However, there is no type 2 grammar that generates it.

- This can be shown the **pumping lemma for context-free languages**.

- Before presenting this, we need to review **derivation trees**.
Pumping Lemma for Context-Free Languages

- Let \( L \) be a context-free language. Then there is a number \( p \) such that if \( s \in L \) and \(|s| > p\) then there are strings \( u, v, x, y, z \), such that
  - \( s = uvxyz \)
  - \(|vy| > 0\) (at least one of \( w \) or \( y \) is non-empty)
  - \(|vxy| \leq p\)
  - (\( \forall m \geq 0 \)) \( u v^m x y^m z \in L \)

- We can use a Skolem function to give \( p \). We name it \( \text{pump}(L) \), the **pumping length** of \( L \).
Pumping Lemma Expressed in Logic

\[ \forall L \ (\text{context-free}(L) \rightarrow \)

\[ (\exists p \ \forall s \ (\ s \in L \land |s| > p \) \rightarrow \)

\[ (\exists u \ \exists v \ \exists x \ \exists y \ \exists z \]

\[ s = uvxyz \]

\[ \land |vy| > 0 \]

\[ \land |vxy| \leq p \]

\[ \land \forall m \ (m \geq 0 \rightarrow u \ v^m x \ y^m z \in L) \)]

\]
Proof of the Pumping Lemma

• The proof is analogous to the proof of the PL for regular languages.

• However, we don’t have states to help us. So we use the auxiliary symbols instead.
How to Remember the Pumping Lemma

Think of this “nested wedges” picture (used in the proof):

Start symbol

Two occurrences of auxiliary A.

∈ L |vxy| ≤ p
Impact of the Pumping Lemma

• The PL can be used to show a language is **not context free**, since it provides a necessary condition \((L \text{ is CF } \rightarrow L \text{ pumpable})\), thus \((L \text{ not pumpable } L \rightarrow \text{ not CF})\).

• It **cannot** be used to show a language is context-free.
What about finite-languages?

- Finite languages (which are regular and thus context-free) obviously cannot be pumped.

- The PL holds **vacuously** for them:
  
  The pumping length is 1 longer than the longest string in the language. So there are no strings that can be pumped.
Proof that \( \{a^k b^k c^k \mid k > 0\} \) is not context-free using the pumping lemma

- Suppose \( \{a^k b^k c^k \mid k > 0\} \) were context-free.

- Let \( p \) be the integer that exists according to the pumping lemma. Consider \( u = a^p b^p c^p \) and decompose into \( uvxyz \).

- At least one of \( v \) and \( y \) is not \( \varepsilon \). Suppose \( v \neq \varepsilon \). The other case is symmetric. By the lemma, \( uv^2xy^2z \) must be in \( L \).

- Analyzing the cases for \( v \) as to whether it consists of all of one letter or of two different letters, in all cases we get a contradiction.
Detailed case analysis

- \( a^p b^p c^p = uvxyz \in L \), where \(|vxy| \leq p\) and \(|vy| > 0\).
- Due to \(|vxy| \leq p\), either
  \( vxy \in \{a\}^*\{b\}^* \) or \( vxy \in \{b\}^*\{c\}^* \).
- If \( vxy \in \{a\}^*\{b\}^* \), then \( uv^2xy^2z \) has the wrong number of \( c \)'s to be in \( L \).
- If \( vxy \in \{b\}^*\{c\}^* \), then \( uv^2xy^2z \) has the wrong number of \( a \)'s to be in \( L \).
- So in either case we have a contradiction.
On Toward Proof of the Pumping Lemma

Binary Tree Height Observation

- The **height** of a binary tree is defined as the number of nodes in a maximal path from the root to any leaf.
- A binary tree with height $n+1$ has at most $2^n$ leaves.
- Thus a binary tree with **at least** $2^n$ leaves has **height at least** $n+1$.
- Examples:

  - Height 1, 1 leaf $(n = 0)$
  - Height 2, 2 leaves $(n = 1)$
  - Height 3, 4 leaves $(n = 2)$
  - Height 4, 8 leaves $(n = 3)$
Proof of the Pumping Lemma (1)

• Suppose L is an infinite context-free language, and (WLOG) G is a Chomsky-Normal Form grammar for L.
• Let n be the number of auxiliary symbols in G.
• We will show that the p that exists in the PL can be satisfied by $p = 2^{n+1}$.

• Let $s \in L$ be such that $|s| > p$. Then the derivation tree for s has at least $2^{n+1}$ leaves, so the height is at least $n+2$.

• Consider a maximal path from leaf to root in this tree. This path has $> n+1$ auxiliary nodes, therefore some auxiliary must be repeated.

• Let $A_1$ be the first instance of the first repeated auxiliary on the path from leaf up toward root and $A_2$ be the second. Such a repetition must take place in $\leq n+1$ nodes.
Proof of the Pumping Lemma (2)

- Here is a picture of our derivation tree:

```plaintext
\[
S 
\quad \downarrow \leq n+1 \quad \downarrow \\
A_2 \quad \quad \quad \quad A_1 \\
\]
```

first place a node is repeated from leaf toward root in a maximal path

derived string s
Proof of the Pumping Lemma (3)

- Identify $u, v, x, y, z$ as follows:

(A binary tree with height $n+1$ has at most $2^n$ leaves.)

We see that $|vxy| \leq 2^n$. Also, $|vy| > 0$ (from CNF).
Example

- $S \rightarrow AC$
- $S \rightarrow AB$
- $C \rightarrow SB$
- $A \rightarrow a$
- $B \rightarrow b$

- Derivation tree for terminal string $aaaabbbbb$

- Note: We can illustrate the principle even though this string is not length or longer 16.
A Long Path
First repeated letters on path from leaf to root
u = aa
v = a
x = ab
y = b
z = bb
Conclusion drawn from pumping

\[\begin{align*}
  u &= aa \\
  v &= a \\
  x &= ab \\
  y &= b \\
  z &= bb \\
\end{align*}\]

Conclusion: \(aaa^kabb^kbb\) is L for all \(k\).
Closure Properties of CFLs: Non-Closure Under Intersection

• The context-free languages are not closed under intersection. Here’s why:

• These can be shown to be context-free:
  • \( \{ a^k b^k c^m \mid k, m > 0 \} \)
  • \( \{ a^m b^k c^k \mid k, m > 0 \} \)

• However, their intersection is:
  • \( \{ a^k b^k c^k \mid k > 0 \} \)
which we know is \textit{not} context free.
Non-Closure Under Complementation

- We know that CFL's are closed under union, but not intersection.

- If they were closed under complementation, they would be closed under intersection, since

\[ L \cap M = \Sigma^* - [(\Sigma^* - L) \cup (\Sigma^* - M)] \]
\( L \cap M = \Sigma^* - [ (\Sigma^* - L) \cup (\Sigma^* - M) ] \)

- Using \( x \in L \leftrightarrow \neg (x \in (\Sigma^* - L)) \) [Complement]
the following statements are equivalent:
- \( x \in L \cap M \) [Meaning of \( \cap \)]
- \((x \in L) \land (x \in M)\) [Meaning of \( \cap \)]
- \(\neg (x \in (\Sigma^* - L)) \land \neg (x \in (\Sigma^* - M))\) [Complement]
- \( \neg [x \in (\Sigma^* - L) \lor x \in (\Sigma^* - M)]\) [DeMorgan]
- \( \neg [x \in [(\Sigma^* - L) \cup (\Sigma^* - M)]]\) [Meaning of \( \cup \)]
- \( x \in \Sigma^* - [ (\Sigma^* - L) \cup (\Sigma^* - M) ]\) [Complement]
Closure Under Intersection with a Regular Language

• If L is context-free and R is regular, then $L \cap R$ is context-free.

• An easy way to see this is to use a machine characterization of context-free languages, which we discuss subsequently.
Example: \( \{ww \mid w \in \{0, 1\}^*\} \) is not context free.

- If this language were context free, so would its intersection with a regular language be.

- Intersecting the original with the regular language \( \{0\}^*\{1\}^*\{0\}^*\{1\}^* \), we get a language
  \[ L = \{0^m 1^n 0^m 1^n \mid m, n \geq 0\} \]

- Let \( p = \text{pump}(L) \). Select \( s = 0^p 1^p 0^p 1^p \) and decompose \( s \) into \( uvxyz \) where \( |vy| > 0 \) and \( |vxy| \leq p \).

- Then show by cases that \( uv^2xy^2z \) cannot be of the form \( 0^m 1^n 0^m 1^n \).
Another Version of Non-Closure Under Complementation

- We just showed the following language is not context-free (using the pumping lemma):
  \[
  \{ww \mid w \in \{0, 1\}^*\}
  \]

- However, the complement:
  \[
  \{0, 1\}^* - \{ww \mid w \in \{0, 1\}^*\}
  \]

  is CF. A grammar for it is given on the next page.
Grammar for \{0, 1\}^* - \{ww \mid w \in \{0, 1\}^*\}

\[
S \rightarrow AB \mid BA \mid A \mid B
A \rightarrow CAC \mid 0 \\
B \rightarrow CBC \mid 1 \\
C \rightarrow 0 \mid 1
\]

Why is this grammar correct?