Properties of Context-Free Languages

COMP 455 - 002, Spring 2019
Simplification of CFGs

We can simplify CFGs by removing:

- **Useless symbols.**
  - \(X\) is **generating** if \(X \Rightarrow^* w\), where \(w \in T^*\).
  - \(X\) is **reachable** if \(S \Rightarrow^* \alpha X \beta\) (\(S\) is the start symbol).
  - \(X\) is **useful** only if it is both reachable and generating.

- **\(\varepsilon\)-productions**, of the form \(A \rightarrow \varepsilon\).
  - If \(\varepsilon\) is in the language, we will still need one \(\varepsilon\)-production.

- **Unit productions**, of the form \(A \rightarrow B\).
Finding Generating Variables

**Theorem 7.4:** The following algorithm correctly finds all generating variables.

```
old_vars := ∅;
new_vars := {A | A → w exists, and w ∈ T*};
while old_vars ≠ new_vars:
    old_vars = new_vars;
    new_vars = old_vars ∪ {A | A → α, α ∈ (T ∪ old_vars)*};
return new_vars;
```
Proof of Theorem 7.4:
We want to show that $X$ is added to $\text{new} \_\text{vars}$ if and only if $X \Rightarrow^* w$ for some $w \in T^*$.

"Only if": We must show that if $X$ is added to $\text{new} \_\text{vars}$ then $X \Rightarrow^* w$.

This can be proven by induction on the number of iterations of the algorithm (specifics are left as an exercise).
Finding Generating Variables

**Proof** of Theorem 7.4:

We want to show that $X$ is added to `new_vars` if and only if $X \Rightarrow w$ for some $w \in T^*$.

**“If”**: We must show that if $X \Rightarrow w$, then $X$ is eventually added to `new_vars`.

This can be proven by induction on the length of the derivation (specifics are left as an exercise).

```plaintext
old_vars := ∅;
new_vars := {A | A → w exists, and w ∈ T*};
while old_vars ≠ new_vars:
    old_vars = new_vars;
    new_vars = old_vars ∪ {A | A → α, α ∈ (T ∪ old_vars)*};
return new_vars;
```
Finding Reachable Variables

**Theorem 7.5:** There exists an iterative algorithm that will correctly find all *reachable* symbols.

This is similar to the previous algorithm, except this time you’ll start with a set containing the start symbol and look for new reachable symbols in each iteration.

```plaintext
old_vars := Ø;
new_vars := {S};
while old_vars ≠ new_vars:
    old_vars = new_vars;
    new_vars = old_vars ∪ {A | A is produced by something in old_vars};
return new_vars;
```
Eliminating Useless Symbols

**Theorem 7.2**: (Abbreviated) Every nonempty CFL is generated by a CFG with no useless symbols.

**Proof**: Let $L$ be the language of some CFG $G$, where $L \neq \emptyset$. Define:

- Remove non-generating using Theorem 7.4
- Remove non-reachable using Theorem 7.5

This order matters!
Eliminating Useless Symbols

Proof, continued:
Assume $G_2$ contains a useless variable, $X$.

- By Theorem 7.5, $S \xRightarrow{G_2} \alpha X \beta$.
  - In other words, we know that $X$ is reachable in $G_2$.
- Any production in $G_2$ must be a production in $G_1$, so $S \xRightarrow{G_1} \alpha X \beta$ must be a production in $G_1$.
- By Theorem 7.4, $S \xRightarrow{G_1} \alpha X \beta \xRightarrow{G_1} w$.
  - In other words, we know that $X$ is producing in $G_1$.
- Every symbol in this derivation is reachable from $S$, so none will be eliminated by Theorem 7.5. So, $S \xRightarrow{G_2} \alpha X \beta \xRightarrow{G_2} w$. This contradicts the assumption that $X$ was useless in $G_2$.

It should be intuitively clear that removing useless symbols won’t change the language of a grammar.
Eliminating Useless Symbols: Example

Incorrect order:

- $S \rightarrow AB \mid a$
- $A \rightarrow a$
- $S \rightarrow AB \mid a$
- $A \rightarrow a$
- $S \rightarrow a$
- $A \rightarrow a$

Remove Unreachable

Remove Non-generating

Correct order:

- $S \rightarrow AB \mid a$
- $A \rightarrow a$
- $S \rightarrow a$
- $A \rightarrow a$
- $S \rightarrow a$

Remove Unreachable

Remove Non-generating

Note that these are *not* the same!
Removing Nullable Symbols

A symbol $A$ is nullable if $A \vDash \varepsilon$.

**Theorem 7.7:** There exists an algorithm that will correctly identify all nullable symbols.

We will not prove this—it should be intuitively similar to what we’ve done before.

```
old_vars := ∅;
new_vars := \{A \mid A \rightarrow \varepsilon \text{ exists}\};
while old_vars ≠ new_vars:
    old_vars = new_vars;
    new_vars = old_vars ∪ \{A \mid A \rightarrow \alpha, \alpha \in old_vars^*\};
return new_vars;
```
Removing $\varepsilon$-Productions

**Theorem 7.9**, reworded: If $L = L(G)$ for a CFG $G$, then there exists a CFG $G_1$ with no $\varepsilon$-Productions such that $L(G_1) = L(G) - \{\varepsilon\}$.

**Proof:**

**To construct $G_1$:** If $A \to X_1 \ldots X_k$ is in $P$, then add all productions of the form $A \to \alpha_1 \ldots \alpha_k$ to $P_1$, where:

1. If $X_i$ is *not* nullable, then $\alpha_i = X_i$,
2. If $X_i$ is nullable, then $\alpha_i$ is either $X_i$ or $\varepsilon$, and
3. Not all $\alpha_i$’s are $\varepsilon$.

This requires adding *two* production rules for each nullable $X_i$. 
Removing $\epsilon$-Productions: Example

CFG $P$:

- $S \rightarrow ABC$
- $A \rightarrow \epsilon$
- $B \rightarrow b \mid \epsilon$
- $C \rightarrow c$

$A$ and $B$ are nullable. CFG $P_1$:

- $S \rightarrow ABC \mid BC \mid AC \mid C$
- $B \rightarrow b$
- $C \rightarrow c$

$A$ is now useless in $P_1$. (If you want to eliminate both $\epsilon$-productions and useless symbols, you must remove $\epsilon$-productions first.)
Removing $\varepsilon$-Productions: Proof

Proof, continued:

**Claim:** For all $A \in V$ and $w \in T^*$, $A \xrightarrow{\ast} w$ if and only if $w \neq \varepsilon$ and $A \xrightarrow{\ast} w$.

"If": Assume $A \xrightarrow{\ast} w$ and $w \neq \varepsilon$. We prove by induction on $i$ that $A \xrightarrow{\ast} w$.

**Base case:** $i = 1$ (one derivation step)

$A \rightarrow w$ must be a production in $P$. Because $w \neq \varepsilon$, $A \rightarrow w$ is also a production in $G_1$. 

---

$G_1$: The CFG without $\varepsilon$-productions producing $L(G) - \{\varepsilon\}$. 

Jim Anderson (modified by Nathan Otterness)
Removing $\epsilon$-Productions: Proof

**Inductive step:** $i > 1$ (more than one derivation step)

Assume $A \Rightarrow_{G} Y_1 \ldots Y_m \Rightarrow_{G} w$. Then $Y_j \Rightarrow_{G} w_j$ and $w = w_1 \ldots w_m$.

If $w_j \neq \epsilon$, then $Y_j \Rightarrow_{G} w_j$, by the induction hypothesis.

If $w_j = \epsilon$, then $Y_j$ is nullable.

Therefore, $A \Rightarrow \beta_1 \ldots \beta_m$ is in the productions of $G_1$, where

- $\beta_j = Y_j$ if $w_j \neq \epsilon$,
- $\beta_j = \epsilon$ if $w_j = \epsilon$.

And we have the following derivation in $G_1$:

$A \Rightarrow \beta_1 \beta_2 \ldots \beta_m \Rightarrow_{G} w_1 \beta_2 \ldots \beta_m \Rightarrow_{G} w_1 w_2 \ldots w_m = w$.
Removing $\varepsilon$-Productions: Proof

“Only if”: Assume $A \xrightarrow{G_1}^i w$. Then, $w \neq \varepsilon$.

We will prove by induction on $i$ that $A \xrightarrow{G}^* w$.

Base case: $i = 1$.

$A \rightarrow w$ is in the productions of $G_1$. Therefore, $A \xrightarrow{G} \alpha$ is in the productions of $G$ where $w = \alpha$ with nullable symbols replaced by $\varepsilon$. 

Claim: For all $A \in V$ and $w \in T^*$, $A \xrightarrow{G_1}^* w$ if and only if $w \neq \varepsilon$ and $A \xrightarrow{G}^* w$. 

Removing $\varepsilon$-Productions: Proof

“Only if”, continued:

We must show that the derivation $A \xrightarrow{G} \alpha \xrightarrow{G} w$ exists in $G$.

Inductive step: Suppose that $A \xrightarrow{G_1} X_1 \ldots X_k \xrightarrow{i-1} w$. Then, $A \rightarrow \beta$ is in the productions of $G$, where $X_1 \ldots X_k = \beta$ with some nullable symbols removed.

As in the base case, $A \xrightarrow{G} X_1 \ldots X_k$. And, by the inductive hypothesis, we can show that $X_1 \ldots X_k \xrightarrow{G} w$. 

“Assume the derivation with $i - 1$ steps is correct…” etc.
Removing Unit Productions

**Theorem 7.13** (reworded): If $L = L(G)$ for a CFG $G$, then there exists a CFG $G_1$ with no unit productions such that $L = L(G_1)$.

**Proof:** Let $G = (V, T, P, S)$

To construction $G_1$, first add all non-unit productions in $P$ to $P_1$.

Next, if $A \Rightarrow^*_{G} B$ and $B \rightarrow \alpha$ is a non-unit production in $P$, then add $A \rightarrow \alpha$ to $P_1$.

We can find all pairs of $(A, B)$ where $A \Rightarrow^*_{G} B$ using an iterative algorithm like before (see Section 7.1.4 of the book).
Removing Unit Productions: Example

Consider the CFG $G$ with the following productions:

- $S \rightarrow A \mid b$
- $A \rightarrow AAAa$
Removing Unit Productions: Proof

Claim: $L(G_1) \subseteq L(G)$

Proof:

If $A \rightarrow \alpha$ is in $P_1$, then $A \Rightarrow^* \alpha$. Therefore, $A \Rightarrow^* \alpha$ implies $A \Rightarrow^* \alpha$.
Removing Unit Productions: Proof

**Claim:** $L(G) \subseteq L(G_1)$

**Proof:**
Suppose $w \in L(G)$.
Let $S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n = w$ be a leftmost derivation.
If $\alpha_i \Rightarrow_{G} \alpha_{i+1}$ is due to a non-unit production, then $\alpha_i \Rightarrow_{G_1} \alpha_{i+1}$.

$G$: The grammar containing unit productions.
$G_1$: The grammar with unit productions removed.
Removing Unit Productions: Proof

Claim: \( L(G) \subseteq L(G_1) \)

Consider the following leftmost derivation with unit productions in \( G: \alpha_{i-1} \Rightarrow_G^* \alpha_i \Rightarrow_G^* \alpha_{i+1} \Rightarrow_G^* ... \Rightarrow_G^* \alpha_j \Rightarrow_G^* \alpha_{j+1} \).

Unit productions just replace the symbol at the same (leftmost) position, so \( \alpha_{i-1} \Rightarrow \alpha_{j+1} \) will also hold by some production in \( P_1 – P \).
Putting it all Together

**Theorem 7.14:** If $L$ is the language of a CFG $G$, and $L$ contains at least one string other than $\varepsilon$, then there exists a CFG $G_1$ with no $\varepsilon$-productions, unit productions, or useless symbols such that $L(G_1) = L - \{\varepsilon\}$.

(See the book for a formal proof.)

The *order* in which we apply the previous results is important.
Putting it all Together

The order of previous results:

- We saw on slide 12 that eliminating $\varepsilon$-productions may cause some symbols to become useless, so we must eliminate $\varepsilon$-productions before removing useless symbols.

- In the same example on slide 12, removing $\varepsilon$-productions also introduced a unit production ($S \rightarrow C$), so $\varepsilon$-productions must be eliminated before unit productions.
Finally, unit productions must be eliminated before useless symbols. Consider this example:

So, the only viable order is: 1) Eliminate $\varepsilon$-productions, 2) Eliminate unit productions, and 3) Eliminate useless symbols.
Chomsky Normal Form (CNF)

A CFG is in *Chomsky Normal Form* if all of its productions are of the form \( A \rightarrow BC \) or \( A \rightarrow a \), and it contains no useless symbols.

**Theorem 7.16** (reworded): Any CFL that doesn’t include \( \varepsilon \) can be generated by a CFG in CNF.

**Proof**: Let \( L = L(G) \) for some CFG \( G \), where \( \varepsilon \not\in L \). Use Theorem 7.14 to convert \( G \) into \( G_1 = (V, T, P, S) \), where \( G_1 \) contains no \( \varepsilon \)-productions, unit productions, or useless symbols.
Converting to Chomsky Normal Form

**Proof** (continued):

If $A \to X$ is a production, then $X \in T$ (which is already in the correct form).

Otherwise, consider $A \to X_1 X_2 \ldots X_m$, where $m \geq 2$.

- If $X_i$ is a terminal, introduce a new variable $C_a$ and a new production $C_a \to a$, then replace $X_i$ by $C_a$.
- Call the resulting grammar $G_2$ (after making all such replacements).

**Claim**: $L(G_1) = L(G_2)$. (The proof is left as an exercise.)
Converting to Chomsky Normal Form

The remaining problem is that we need to replace productions of the form $A \rightarrow B_1 B_2 \ldots B_m$ (where $m \geq 3$).

Replace such a production by:

$A \rightarrow B_1 D_1$, $D_1 \rightarrow B_2 D_2$, ..., $D_{m-1} \rightarrow B_{m-1} B_m$, using newly added $D_i$ variables.

Call the resulting grammar $G_3$.

Claim: $L(G_3) = L(G_2)$. (The proof is left as an exercise.)
Conversion to CNF: Example

Starting CFG with no $\varepsilon$-productions, unit productions, or useless symbols.

\[ S \rightarrow bA \mid aB \]
\[ A \rightarrow bAA \mid aS \mid a \]
\[ B \rightarrow aBB \mid bS \mid b \]

Replace $A \rightarrow C_b AA$ by:
\[ A \rightarrow C_b D_1, \ D_1 \rightarrow AA \]

Replace $B \rightarrow C_a BB$ by:
\[ B \rightarrow C_a D_2, \ D_2 \rightarrow BB \]

\[ S \rightarrow C_b A \mid C_a B \]
\[ A \rightarrow C_b D_1 \mid C_a S \mid a \]
\[ B \rightarrow C_a D_2 \mid C_b S \mid b \]
\[ D_1 \rightarrow AA \]
\[ D_2 \rightarrow BB \]
\[ C_a \rightarrow a \]
\[ C_b \rightarrow b \]

CFG in CNF.
The Pumping Lemma for CFLs

**Theorem 7.18** (the Pumping Lemma for CFLs):
Let $L$ be any CFL. Then there exists a value $n$ such that for all $z \in L$, where $|z| \geq n$, there exist strings $u, v, w, x, y$ such that:

1. $z = uvwx$,  
2. $|vx| \geq 1$,  
3. $|vwx| \leq n$, and  
4. For all $i \geq 0$, $uv^iwx^iy \in L$.  

Jim Anderson (modified by Nathan Otterness)
Proof of the Pumping Lemma for CFLs

If \( L \) is a CFL, let \( G \) be a CFG generating \( L - \{ \varepsilon \} \).

Claim: Let \( z \in L - \{ \varepsilon \} \). If a parse tree for \( z \) in \( G \) has no path longer than \( n \), then \( |z| \leq 2^{n-1} \). (This is Theorem 7.17 in the book.)

Proof, by induction on \( n \):

Base case: \( n = 1 \). String \( z = a \). Tree: \[ S \quad |z| = 1 = 2^{n-1} \]

The Pumping Lemma only applies to strings longer than \( n \), so removing \( \varepsilon \) doesn’t matter.
Proof of the Pumping Lemma for CFLs

Inductive step: $n > 1$.

Suppose that a tree exists with some path of length $n$, but no path exceeding a length $n$. It looks like this:

\[
\begin{align*}
S & \quad \leq 2^{n-2} \\
A & \quad \leq 2^{n-2} \\
B & \quad \leq 2^{n-2} \\
\end{align*}
\]
Proof of the Pumping Lemma for CFLs

- Let $m$ equal the number of variables in the CFG $G$.
- Let $n = 2^m$.
- Suppose $z \in L(G)$, where $|z| \geq n$.
  - **Note:** $|z| > 2^{m-1}$
- We claim that any parse tree for $z$ has a path of length $\geq m + 1$.
  - To see this, suppose all paths in a tree are shorter than $m + 1$ (no path is $> m$). Then, $|z| \leq 2^{m-1}$, contradicting the claim that $|z| > 2^{m-1}$.

Jim Anderson (modified by Nathan Otterness)
Proof of the Pumping Lemma for CFLs

- As stated on the previous slide, any parse tree for $z$ has a path of length $\geq m + 1$.
- Such a path has at least $m + 2$ nodes, $m + 1$ of which are variables.

$z = uvwxy$
$|vx| \geq 1$
$|vwx| \leq n$
For all $i \geq 0$, $uv^iwx^i y \in L$. 

The CNF grammar requires replacing variables with a terminal at the end of the path.

Some path from the start symbol to a terminal in $z$ must contain $m + 1$ variables.
Proof of the Pumping Lemma for CFLs

- Since the CFG contains \( m \) variables, but the path in \( z \)'s parse tree contains \( m + 1 \) variables, at least one variable must be repeated in the path.
- If \( A \) is the repeated variable, the path looks like this:

\[
\begin{align*}
& z = uvwxy \\
& |vx| \geq 1 \\
& |vwx| \leq n \\
& \text{For all } i \geq 0, \ uv^iwx^iy \in L.
\end{align*}
\]

Jim Anderson (modified by Nathan Otterness)
Proof of the Pumping Lemma for CFLs

Consider the subtrees rooted at each occurrence of $A$:

- $z = uvwxy$
- $|vx| \geq 1$
- $|vwx| \leq n$
- For all $i \geq 0$, $uv^iwx^iy \in L$. 
Proof of the Pumping Lemma for CFLs

A subtree rooted at $A$ has (at least) two possible yields: $wwx$ and $w$.

$z = uvwxy$

$|vx| \geq 1$

$|vwx| \leq n$

For all $i \geq 0$, $uv^iwx^iy \in L$. 

$vwx = \text{yield of first subtree of } A$

$w = \text{yield of second subtree of } A$
Proof of the Pumping Lemma for CFLs

We can replace the possible subtrees rooted at $A$ with each other to generate different strings.

This tree has a yield $uvwxy = uv^0wx^0y$

This string must also be in $L$. 

- $z = uvwxy$
- $|vx| \geq 1$
- $|vwx| \leq n$
- For all $i \geq 0$, $uv^iwx^i y \in L$. 
Proof of the Pumping Lemma for CFLs

We can replace the possible subtrees rooted at A with each other to generate different strings.

\[ z = uvwxy \]
\[ |vx| \geq 1 \]
\[ |vw| \leq n \]
\[ \text{For all } i \geq 0, \quad uv^iwx^i y \in L. \]

This tree has a yield \( uvvwxy = uv^2wx^2y \)
This string must also be in \( L. \)
Proof of the Pumping Lemma for CFLs

We can repeat this process indefinitely to keep generating strings in $L$ of the form $uv^iwx^i y$.

So, $uv^iwx^i y \in L$ for all $i \geq 0$. 

- $z = uvwxy$
- $|vx| \geq 1$
- $|vwx| \leq n$
- For all $i \geq 0$, $uv^iwx^i y \in L$. 

Jim Anderson (modified by Nathan Otterness)
Application of the CFL Pumping Lemma

- $L = \{a^ib^ic^i \mid i \geq 1\}$.
- Select any $n$, and let $z = a^n b^n c^n$. Clearly, $z \in L$.
- Let $z = uvwxy$, $|vx| \geq 1$, and $|vwx| \leq n$. This means:
  - $vx$ is all $a$'s, $b$'s, or $c$'s, or
  - $vx$ is all $a$'s and $b$'s or all $b$'s and $c$'s
  - (The key point is that $vx$ can not possibly contain $a$'s, $b$'s, and $c$'s all at the same time.)
- If $vx$ contains only one type of symbol, $uv^0wx^0y$ will contain too few of that symbol.
- If $vx$ contains two types of symbols, $uv^0wx^0y$ will contain too many of the symbol not in $vx$.
- Therefore, $uv^0wx^0y \notin L$, and the Pumping Lemma for CFLs does not hold for $L$. 
Application of the CFL Pumping Lemma

- $L = \{ww \mid w \in (0 + 1)^*\}$.
- Select any $n$, and let $z = 0^n1^n0^n1^n$. Clearly, $z \in L$.
- Let $z = uvwxy$, $|vx| \geq 1$, and $|vwx| \leq n$. This means:
  - $vx$ is all 0s or all 1s
  - $vx$ is some 0s followed by some 1s
  - $vx$ is some 1s followed by some 0s.
- If $vx$ contains only 0s or only 1s, $uv^0wx^0y$ will contain too few 0s or 1s in one half of the string.
- If $vx$ contained 0s followed by 1s, then either the first or second half of $uv^0wx^0y$ will contain fewer 0s and 1s than the other half.
- If $vx$ contained 1s followed by 0s ($vx$ is in the middle of $z$), then $uv^0wx^0y$ will have more 0s in the first group of 0s than the second group of 0s. (The number of 1s will also be similarly imbalanced.)
- In any of these cases, $uv^0wx^0y \notin L.$
Closure of CFLs under Substitution

- Recall that a homomorphism maps characters in some alphabet $\Sigma$ to strings over another alphabet $\Delta$.

- A homomorphism is actually a special case of substitution, which maps characters in one alphabet to any string in a language over another alphabet.

- Consider this example substitution $f$:
  - $\Sigma = \{0, 1\}$, $\Delta = \{a, b\}$, $f(0) = a + b^*$, $f(1) = a^*b$.
  - $f(0^*1^*) = (a + b^*)^*(a^*b)^*$.
  - (Note that this particular example uses regular languages.)
Closure of CFLs under Substitution

**Theorem 7.23** (reworded): The CFLs are closed under substitution and, by extension, homomorphism.

**Proof:**

The main idea is to replace all terminals in a CFG with start symbols of another CFG.

- Let $L$ be a CFL, and $L \subseteq \Sigma^*$. For all $a \in \Sigma$, let $L_a$ be a CFL.
- Let $L = L(G)$. For all $a \in \Sigma$, let $L_a = L(G_a)$.
- Assume these grammars have distinct variables.
Closure under Substitution, Proof contd.

Let $G = (V, T, P, S)$, and for all $a \in \Sigma$, $G_a = (V_a, T_a, P_a, S_a)$

Define $G' = (V', T', P', S')$, where:

$V' = (\bigcup_{a \in \Sigma} V_a) \cup V$

$T' = \bigcup_{a \in \Sigma} T_a$

$S' = S$

$P' = \bigcup_{a \in \Sigma} P_a \cup \{ A \to \alpha' \mid A \to \alpha \text{ is in } P, \text{ and } \alpha' = \alpha \text{ with each } a \in \Sigma \text{ replaced by } S_a \}$.

The language defined by substitution equals $L(G')$. (The proof is left as an exercise—or see Theorem 7.23 in the book.)
Theorem 7.24: CFLs are closed under Union, Concatenation, \( ^* \)-closure, and \( ^+ \)-closure.

Proof:

Union: Let \( L_1 \) and \( L_2 \) be CFLs. \( L_1 \cup L_2 = s(L) \), where \( L = \{1, 2\} \) (which is clearly a CFL), and \( s \) is the substitution defined by \( s(1) = L_1 \) and \( s(2) = L_2 \). The proofs for the others are similar.
Closure under Reversal

**Theorem 7.25:** CFLs are closed under reversal.

**Proof:**

If $L = L(G)$, where $G = (V, T, P, S)$, then $L^R = L(G^R)$, where $G^R = (V, T, P^R, S)$, and $P^R = \{A \rightarrow \alpha^R \mid A \rightarrow \alpha \text{ is a production in } P\}$.

(The full formal proof is left as an exercise.)

The productions in $G^R$ are just the productions in $G$ written backwards.
(Lack of) Closure under Intersection

**Theorem:** CFLs are not closed under intersection.

**Proof:**

- \( L_1 = \{a^i b^i c^i \mid i \geq 1 \} \). We know \( L_1 \) isn’t a CFL from earlier slides.
- \( L_2 = \{a^i b^i c^j \mid i \geq 1, j \geq 1 \} \). This is a CFL.
- \( L_3 = \{a^i b^j c^j \mid i \geq 1, j \geq 1 \} \). This is also a CFL.
- However, \( L_1 = L_2 \cap L_3 \).

See example 7.26 in the book for CFGs.
(Lack of) Closure under Complementation

Corollary to the previous theorem: CFLs are not closed under complementation.

Proof:

- CFLs are closed under union.
- \( L_1 \cap L_2 \equiv \overline{L_1 \cup L_2} \).
- So, if CFLs are closed under complementation, they would be closed under intersection, too.
Intersection with Regular Languages

**Theorem 7.26:** If $L$ is a CFL and $R$ is a regular language, then $L \cap R$ is a CFL.

**Proof:**
Let $L$ be the language of some PDA $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$. Let $R$ be the language of some DFA $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$.

**Idea:** Create a new PDA combining the states of $P$ and $A$, similar to combining two DFAs.
Proof: Intersection with Reg. Languages

Let $P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$, where:

$\delta(((q, p), a, X) \text{ contains } ((r, s), \gamma) \text{ if and only if } \delta_A(p, a) = s \text{ and } \delta_P(q, a, X) \text{ contains } (r, \gamma)$.}

**Claim:** $((q_P, q_A), w, Z_0) \uparrow_{P'} ((q, p), \varepsilon, \gamma)$ if and only if $q_P, w, Z_0 \uparrow_P (q, \varepsilon, Y)$ and $\delta(q_A, w) = p$.

(This can be proven by induction on $i$.)
Closure under Inverse Homomorphism

**Theorem 7.30:** CFLs are closed under inverse homomorphism.

**Proof:**
Consider $L$, where $L$ is the language of some PDA $P$. 

$P = (Q, \Delta, \Gamma, \delta, q_0, Z_0, F)$. 

Let $h: \Sigma \to \Delta^*$. 

Construct a PDA $P'$ that accepts $h^{-1}(L)$. 
Closure under Inverse Homomorphism

Proof, continued:
The key idea is the same as for regular languages: When processing an input $a$, $P'$ simulates $P$ on the input $h(a)$. In a DFA, simulation required just a single state transition.

However, $P$, being a PDA, does more than just change state on input $h(a)$—it may change the stack contents or make nondeterministic choices.

Solution: Use a buffer to hold the symbols of $h(a)$. This buffer will really be part of $P'$'s (finite!) state.
Closure under Inverse Homomorphism

Conceptually:

The buffer must be large enough to hold the longest string produced by $h$. 

Input $a$ goes through $h$ to $h(a)$, which is buffered. The original PDA state is then accepted or rejected.
Closure under Inverse Homomorphism

Short example: $h(a) = 01$, and $h(b) = 111$
Closure under Inverse Homomorphism

- We have a homomorphism \( h: \Sigma \rightarrow \Delta^* \).
- Recall that \( P = (Q, \Delta, \Gamma, \delta, q_0, Z_0, F) \).
- Let \( P' = (Q', \Sigma, \Gamma, \delta', (q_0, \epsilon), Z_0, F \times \{\epsilon\}) \).
- States in \( Q' \) is a set of pairs \((q, x)\) such that
  - \( q \) is a state in \( Q \), and
  - \( x \) is the finite “buffer” — \( h(a) \) or a suffix of \( h(a) \) for some symbol \( a \in \Sigma \).

Start in the original start state, with an empty buffer.

Accept in any of the original accepting states, if the buffer is empty.
Closure under Inverse Homomorphism

Transitions in $P'$:

- $\delta'( (q, x), \varepsilon, X )$ contains all $((p, x), \gamma)$ such that $\delta(q, \varepsilon, X)$ contains $((p, \gamma))$.
  - “Simulate $\varepsilon$-transitions”
- $\delta'( (q, bx), \varepsilon, X )$ contains all $((p, x), \gamma)$ such that $\delta(q, b, X)$ contains $((p, \gamma))$.
  - “Simulate non-$\varepsilon$ transitions”
- $\delta'( (q, \varepsilon), a, X )$ contains $((q, h(a)), X)$ for all $a \in \Sigma$ and $X \in \Gamma$.
  - “Load the buffer”
Closure under Inverse Homomorphism

Claim: If $(q_0, h(w), Z_0) \overset{*}{\vdash}_P (p, \varepsilon, \gamma)$, then $(q_0, \varepsilon, w, Z_0) \overset{*}{\vdash}_P' ((p, \varepsilon), \varepsilon, \gamma)$.

Proof sketch:

For each $a \in \Sigma$, whatever sequence of moves $P$ makes on $h(a)$, $P'$ can make a corresponding sequence of moves on input $a$. 
Closure under Inverse Homomorphism

Claim: If \( ((q_0, \varepsilon), w, Z_0)_P^* ((p, \varepsilon), \varepsilon, \gamma), \) then
\( (q_0, h(w), Z_0)_P^* (p, \varepsilon, \gamma). \)

Proof sketch:

\( P' \) can only process \( w \) one character at a time. For each character \( a \), \( P' \) does what \( P \) does on \( h(a) \).

The first claim implied that \( L(P') \supseteq h^{-1}(L(P)) \).
This claim implies that \( L(P') \subseteq h^{-1}(L(P)) \).
Decision Properties of CFLs

As with regular languages, we’ll focus less on efficiency and more on simplicity than what’s done in the book.

- **Theorem:** There exist algorithms to determine if a CFL is (a) empty, (b) finite, or (c) infinite.
Detecting if a CFL is Empty

Nonemptiness:

- Use the iterative algorithm for detecting useless symbols to test if $A \Rightarrow^* w$ for all $A \in V$.
- The CFL is nonempty if and only if $S \Rightarrow^* w$ for some terminal string $w$. 
Detecting if a CFL is Finite or Infinite

Assume $L$ does not contain $\varepsilon$. (If $\varepsilon \in L$, then consider $L - \{\varepsilon\}$ instead.)

Suppose $L = L(G)$, where $G = (V, T, P, S)$ is in CNF (and therefore has no useless symbols).

Consider the directed graph $(V, E)$, where $(A, B) \in E$ if $A \rightarrow BC$ or $A \rightarrow CB$ is in $P$ for some variable $C$.

**Claim:** $L$ is finite if and only if $(V, E)$ has no cycles.
Detecting if a CFL is Finite or Infinite

Claim (again): \( L \) is finite if and only if \((V,E)\) has no cycles.

“Only if”: A cycle has the form \( A_0, A_1, \ldots, A_n, A_0 \).

Therefore, we have \( A_0 \Rightarrow \alpha_1 A_1 \beta_1 \Rightarrow \alpha_2 A_2 \beta_2 \Rightarrow \cdots \Rightarrow \alpha_n A_n \beta_n \Rightarrow \alpha_{n+1} A_0 \beta_{n+1} \), where \( |\alpha_i \beta_i| = i \).

Since \( G \) has no useless symbols:

- \( \alpha_{n+1} \Rightarrow w \),
- \( \beta_{n+1} \Rightarrow x \),
- \( S \Rightarrow y A_0 z \), and
- \( A_0 \Rightarrow v \),

where \( w, x, y, z, \) and \( v \) are terminal strings.

This is due to the grammar being in CNF — we can only produce \( i \) symbols in \( i \) derivation steps.
Detecting if a CFL is Finite or Infinite

“Only if” proof, continued:
From:

- \( \alpha_{n+1} \Rightarrow w \)
- \( \beta_{n+1} \Rightarrow x \)
- \( S \Rightarrow yA_0z \)
- \( A_0 \Rightarrow \nu \)
we have:

\[ S \Rightarrow yA_0x \Rightarrow ywA_0xz \Rightarrow yw^2A_0x^2z \Rightarrow ... \Rightarrow yw^iA_0x^iz \Rightarrow yw^i\nu x^i z. \]

Therefore, \( L \) is infinite.
Detecting if a CFL is Finite or Infinite

Claim (again): \( L \) is finite if and only if \((V, E)\) has no cycles.

“If”: Suppose \((V, E)\) has no cycles.

> **Definition**: The *rank* of \( A \in V \) is the longest path beginning from \( A \). If a graph has no cycles, then all ranks are finite.

> **Claim**: If \( A \) has a rank \( r \), then \( A \) derives no terminal string of length exceeding \( 2^r \).

Fred This can be proven by induction on \( r \), similarly to some claims prior to the Pumping Lemma.

> Since \((V, E)\) has no cycles, \( S \) has a finite rank. Therefore, the longest string derivable from \( S \) has a finite length, so \( L \) must be finite.
Membership: CYK Algorithm

We want to know if a string $x$ is in $L$.

- Let $L - \{ \varepsilon \} = L(G)$ for a CFG $G$ in CNF.
- Let $|x| = n$.
- Let $x_{ij}$ be a substring of $x$ of length $j$ beginning at position $i$.
- Inductively determine all variables $A$ such that $A \Rightarrow^* x_{ij}$.
- $x \in L$ if and only if $S \Rightarrow^* x_1n$. 

Jim Anderson (modified by Nathan Otterness)
Membership: CYK Algorithm

Base case: \( j = 1 \). \( A \Rightarrow x_{ij} \) if and only if \( A \rightarrow x_{ij} \).

Inductive step: \( j > 1 \). \( A \Rightarrow x_{ij} \) if and only if there exists \( A \rightarrow BC \) and \( k, 1 \leq k < j \), such that \( B \Rightarrow x_{ik} \) and \( C \Rightarrow x_{i+k \ j-k} \).

\( j = 1 \), so \( x_{ij} = x_{i1} \). \( x_{i1} \) has a length of 1, so it’s a terminal symbol.
Membership: CYK Algorithm

Pseudocode for determining the sets of variables, $V_{ij}$, that can produce the substring $x_{ij}$:

```
for i := 1 to n:
    V_{i1} := \{A \mid A \rightarrow x_{i1} \}

for j := 2 to n:
    for i := 1 to n - j + 1:
        V_{ij} := \emptyset;
        for k := 1 to j - 1:
            V_{ij} := V_{ij} \cup \{A \mid A \rightarrow BC, B \in V_{ik}, C \in V_{i+k, j-k} \}
```

The time complexity is $O(n^3)$. 
CYK Algorithm: Example

\[ S \rightarrow AB \mid BC \]
\[ A \rightarrow BA \mid a \]
\[ B \rightarrow CC \mid b \]
\[ C \rightarrow AB \mid a \]

\[ x = baaba \quad (n = 5) \]

for \( i := 1 \) to \( n \):
\[ V_{i1} := \{ A \mid A \rightarrow x_{i1} \} \]

for \( j := 2 \) to \( n \):
for \( i := 1 \) to \( n - j + 1 \):
\[ V_{ij} := \emptyset; \]
for \( k := 1 \) to \( j - 1 \):
\[ V_{ij} := V_{ij} \cup \{ A \mid A \rightarrow BC, B \in V_{ik}, C \in V_{i+k-j-k} \} \]
**CYK Algorithm: Example**

\[ S \rightarrow AB \mid BC \]
\[ A \rightarrow BA \mid a \]
\[ B \rightarrow CC \mid b \]
\[ C \rightarrow AB \mid a \]

\( x = baaba \) (\( n = 5 \))

---

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>A, C</td>
<td>A, C</td>
<td>B</td>
<td>A, C</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

\[
\text{for } i := 1 \text{ to } n: \quad V_{i1} := \{ A \mid A \rightarrow x_{i1} \} \\
\text{for } j := 2 \text{ to } n: \quad \\
\text{for } i := 1 \text{ to } n - j + 1: \quad V_{ij} := \emptyset; \\
\text{for } k := 1 \text{ to } j - 1: \quad V_{ij} := V_{ij} \cup \{ A \mid A \rightarrow BC, B \in V_{ik}, C \in V_{i+k-j-k} \} 
\]
CYK Algorithm: Example

\[ S \rightarrow AB \mid BC \]
\[ A \rightarrow BA \mid a \]
\[ B \rightarrow CC \mid b \]
\[ C \rightarrow AB \mid a \]

\[ x = baaba \ (n = 5) \]

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
b & a & a & b & a \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & B & A, C & A, C & B \\
2 & S, A & B & S, C & S, A \\
3 & & & & \\
4 & & & & \\
5 & & & & \\
\end{array}
\]

\[
\begin{align*}
\text{for } i & := 1 \text{ to } n: \\
V_{i1} & := \{ A \mid A \rightarrow x_{i1} \} \\
\text{for } j & := 2 \text{ to } n: \\
\text{for } i & := 1 \text{ to } n - j + 1: \\
V_{ij} & := \emptyset; \\
\text{for } k & := 1 \text{ to } j - 1: \\
V_{ij} & := V_{ij} \cup \{ A \mid A \rightarrow BC, B \in V_{ik}, C \in V_{i+k-j-k} \}
\end{align*}
\]
### CYK Algorithm: Example

S → AB | BC
A → BA | a
B → CC | b
C → AB | a

x = baaba (n = 5)

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>V_{ij}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>B, A, C</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>S, A, B</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>∅, B, B</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

for i := 1 to n:
V_{i1} := \{A | A → x_1\}

for j := 2 to n:
for i := 1 to n - j + 1:
V_{ij} := ∅;
for k := 1 to j - 1:
V_{ij} := V_{ij} \cup \{A | A → BC, B ∈ V_{ik}, C ∈ V_{i+k-j-k}\}
CYK Algorithm: Example

\[ S \rightarrow AB \mid BC \]
\[ A \rightarrow BA \mid a \]
\[ B \rightarrow CC \mid b \]
\[ C \rightarrow AB \mid a \]

\[ x = baaba \ (n = 5) \]

---

**Table:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_ j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>A, C</td>
<td>A, C</td>
<td>B</td>
<td>A,C</td>
</tr>
<tr>
<td>2</td>
<td>S, A</td>
<td>B</td>
<td>S, C</td>
<td>S, A</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>∅</td>
<td>B</td>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>∅</td>
<td>S, A</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>∅</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Algorithm:**

\[
\begin{align*}
\text{for } i := 1 \text{ to } n: & \\
& V_{i1} := \{ A \mid A \rightarrow x_{i1} \} \\
\text{for } j := 2 \text{ to } n: & \\
& \text{for } i := 1 \text{ to } n-j+1: & \\
& V_{ij} := \emptyset; & \\
& \text{for } k := 1 \text{ to } j-1: & \\
& V_{ij} := V_{ij} \cup \{ A \mid A \rightarrow BC, B \in V_{ik}, C \in V_{i+k-j-k} \} \\
\end{align*}
\]
**CYK Algorithm: Example**

\[ S \to AB \mid BC \]
\[ A \to BA \mid a \]
\[ B \to CC \mid b \]
\[ C \to AB \mid a \]

\[ x = baaba \ (n = 5) \]

---

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>A, C</td>
<td>A, C</td>
<td>B</td>
<td>A, C</td>
</tr>
<tr>
<td>2</td>
<td>S, A</td>
<td>B</td>
<td>S, C</td>
<td>S, A</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>∅</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>∅</td>
<td>S, A, C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S, A, C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

\[ S \in V_{1n}, \text{ so } x \in L. \]