Recall

Theorem 1: \( L \) is a context-free language \( \Rightarrow \) for some PDA \( M \), \( L = N(M) \)
(and therefore \( L = L(M') \) for a PDA \( M' \)).

Idea: Use a single-state PDA.

- The stack keeps track of variables that have not yet been converted to terminal symbols.
- When a variable is turned into a terminal symbol, it is popped from the stack, and the input consumed.

Thus: \( S \Rightarrow w.A \) in the context-free grammar for \( L \),
using a leftmost derivation with \( w \in T^* \), \( A \in V^* \)
if and only if

\( (\text{state } q, \text{ input } w, \text{ stack } S) \xrightarrow[*]{\delta} (q, \varepsilon, A) \)
in the PDA \( M \).

The machine execution simulates a leftmost derivation.

Now we will show the converse statement:

Theorem 2: \( L = N(M) \), the empty-stack language, for a PDA \( M \)
implies

\( L \) is context-free, i.e., \( L = L(G) \) for a context-free grammar \( G \).
**Theorem 2:** \( L = N(M) \) for PDA \( M \Rightarrow L \) is context-free \[6.3\]

**Note:** Converting a recognizing machine to a generating grammar should be the more complicated direction, and it is.

**Recall:** How we converted from an \( \varepsilon \)-NFA to a regular expression:

State elimination argument

\[ \begin{array}{c}
  \xrightarrow{a} \rightarrow 1 \xrightarrow{a} 2 \xrightarrow{a} 3 \quad \text{becomes} \quad \xrightarrow{ab*a} 1 \xrightarrow{a} 3 \xrightarrow{a} 3 \quad \text{becomes} \quad ab*a \\
\end{array} \]

- we eliminated one state at a time, putting regexps on the transitions.
- interpretation: a substring matching the regexp label can be consumed to make the transition.

For PDAs, this is harder; instead of regexps, can use CFGs — but do we put them on the stack, or...?

**Example 1:** For one-state PDAs, converting to a CFG is easy:

\[ M : \]
\[ a, z, \varepsilon / A, Z_0 \]
\[ a, A / AA \]
\[ a, Z_0 / \varepsilon \]
\[ b, Z_0 / B, Z_0 \]
\[ b, A / \varepsilon \]
\[ b, \varepsilon / B, B \]
\[ \varepsilon, Z_0 / \varepsilon \]

\[ N(M) = \{ w \in \{a, b\}^* | n_a(w) : n_b(w) \} \]
- each \( a \) puts an \( A \) counter on the stack, or eats a \( B \) counter
- each \( b \) places a \( B \) counter, or eats an \( A \)

\[ G : \]
\[ Z_0 \rightarrow aAZ_0 | bBZ_0 | \varepsilon \]
\[ A \rightarrow aAA | b \]
\[ B \rightarrow a | bBB \]

\[ L(A) = \{ akb^{k+1} | k \geq 0 \} \]
\[ L(B) = \{ lbk^{k+1} | k \geq 0 \} \]
\[ L(Z_0) = N(M) \]

**Example:**

\[ (p, \text{abba}, Z_0) \Rightarrow (p, \text{bba}, A, Z_0) \]
\[ \Rightarrow (p, \text{a}, B, Z_0) \]
\[ \Rightarrow (p, \varepsilon, Z_0) \Rightarrow (p, \varepsilon, \varepsilon) \]

**Idea:** Have productions of old stack symbols \( \rightarrow \) consumed input \( \rightarrow \) new stack

\[ \xrightarrow{a, X / \varepsilon} \]

i.e. \( (p, a, e) \in \delta(p, a, X) \)

\[ \text{becomes} \quad X \rightarrow a \]
Example 2: \[ L = \{ w \in \{a, b\}^* \mid n_b(w) = 2n_a(w) \} \]

M:

- accepts \( L \) by empty stack since each \( b \) eats one \( A \) counter, or places a \( B \); and each \( a \) puts two \( A \) counters on the stack, or eats up to two \( B \) counters.

Exercise: Eliminate the state \( q\).

- problem: How can an \( a \) eat two \( B \) counters without extra states?

- one possible approach: Introduce a [BS] counter, a single stack symbol that represents two \( B \)'s.

- problem: What if there have been an odd number of \( b \)'s, and an \( a \) arrives. How can we make sure a \([BS]\) is on top?

- we can't!

Another approach: Guess a context-free grammar for \( L \), then convert it to a PDA.

\[
S \rightarrow ASASBS | ASBSAS | BSASAS | \epsilon
\]

\[ A \rightarrow b, \quad B \rightarrow a \]

- works:

To prove Theorem 2, we need a general technique!

Possible approach? Just as we generalized \( \epsilon \)-NFAs to allow regexpressions on arrows, generalize PDAs to allow consuming multiple stack symbols? - complicated..
Main Idea: Keep all state information on the stack.

- If the top stack symbol includes the current state's name, then the machine doesn't have to change states ever—doing so would be redundant.

First idea: Let stack alphabet $\Gamma' = Q \times \Gamma$, i.e., have each stack symbol be a pair $[p A]$ for $p \in Q$, $A \in \Gamma$.

Example:

$$M: \quad 1 \xrightarrow{a, z_0/[2A]} 2 \xrightarrow{a, A/E} \omega \xrightarrow{b, B/A} M'$$

becomes

$$M': \quad 1 \xrightarrow{a, z_0/[2A]} 2 \xrightarrow{a, [zA]^i/E} \omega \xrightarrow{b, [zB]^i/[2A][3B]}$$

Note: You can only use a stack symbol if its state information is correct.

Note: When we add symbols to the stack, we need to guess where that symbol will be read.

$M'$ will accept the same language but we cannot collapse the states all down to one.

Problem: We are not enforcing consistency between states. Eg, if we use the $a, z_0/[3B][2B]$ transition, then $[3B]$ is at the stack top, as if we had jumped to state 3! The top of the stack should tell you the current state, and it doesn't necessarily.
Second idea: We want an evolution in \( M' \) to expose \([q, A] \) on the stack if and only if a corresponding evolution in \( M \) goes to state \( q \) with \( A \) on top. To enforce this consistency, put on the stack both where a symbol is used, and where it is going:

\[
\begin{align*}
1 & \rightarrow A/E \rightarrow 2 \rightarrow E/E \rightarrow 3 \\
1 & \rightarrow A/E \rightarrow 2 \rightarrow A[IA]/E \rightarrow 3 \\
1 & \rightarrow A/A[E] \rightarrow 2 \rightarrow A[2]/E \rightarrow 3
\end{align*}
\]

and if before we placed \( ABBA \) on the stack,

but this doesn't work!!!

E.g., if stack is \( [IA] \), and we follow

\[
\begin{align*}
1 & \rightarrow [IA]/E \rightarrow 3 \\
1 & \rightarrow [IA]/E \rightarrow 3 \rightarrow [2] \rightarrow 5
\end{align*}
\]

then the stack becomes

\[
\]

even though we are in state 5, top of stack says "3"!

Third idea: To fix this, keep track, instead, of each symbol's ultimate destination, i.e., of the state the machine will be in when that level of the stack is ultimately cleared. Thus the above example becomes

\[
\begin{align*}
1 & \rightarrow [IA]/[3B5] \rightarrow 3 \\
1 & \rightarrow [IA]/[3B5] \rightarrow 3 \rightarrow [2B2] \rightarrow 5 \\
1 & \rightarrow [2A] \rightarrow 3 \rightarrow [2B2] \rightarrow 5
\end{align*}
\]

and the stack is always consistent \([q, A, p, [A, p[, [A, p[, [A, p[, ... \]

even as it grows and shrinks.

Then the state underneath \([IA]/E \) has to start with a \( q, [E, [qC2] \), and this rule keeps that...
At this point, remembering what state the machine is in is redundant, since it is stored safely on the stack. Therefore can put all the rules together onto a single state.

**Example 2 (continued)**

\[ M' : \]
\[
\begin{align*}
\text{start} & \rightarrow [pZop] \\
\rightarrow & \left\{ \\
E, [pZop] & \rightarrow [pAP][pAP] \\
a, [pZop] & \rightarrow [pAP][pAP] \\
b, [pZop] & \rightarrow [pBP][pZop] \\
b, [pZop] & \rightarrow [pBP][pBP] \\
b, [pZop] & \rightarrow [pBP][pBP] \\
b, [pZop] & \rightarrow [pBP][pBP] \\
b, [pZop] & \rightarrow [pBP][pBP] \\
b, [pZop] & \rightarrow [pBP][pBP] \\
b, [pZop] & \rightarrow [pBP][pBP] \\
\end{align*}
\]

Using our conversion rules for single-state PDAs, this gives a context-free grammar: Renaming symbols \([pZop]\) as \(S\), \([pAP]\) as \(A\), \([pZop]\) as \(T\), \([pBP]\, [pBP]\) as \(B', B, B''\), we get:

\[
\begin{align*}
S & \rightarrow E \\
S & \rightarrow aAAS \\
A & \rightarrow aAAA \\
S & \rightarrow bBS \\
S & \rightarrow bBT \\
B & \rightarrow bBB \\
B' & \rightarrow a \\
B'' & \rightarrow E \\
T & \rightarrow AS
\end{align*}
\]

Simplify it:

\[
\begin{align*}
S & \rightarrow E | aAAS | bBS | bBS \\
A & \rightarrow b | aAAA \\
B & \rightarrow bBB | bB' \\
B' & \rightarrow a | bBS' \\
\end{align*}
\]

thus \(S \Rightarrow bbbba \rightarrow aabb \)

(Compare to \(S \rightarrow E | bBS | a \rightarrow bSBS \rightarrow aABSBS \); recall that it is undecidable whether two CFGs generate the same language!)
Corollary 1: CFGs and PDAs have the same power, i.e., accept the same set of languages (context-free languages).

Corollary 2: Although the intersection of two CFLs is not necessarily context-free, the intersection of a context-free language with a regular language is always context-free. 
\((L \text{ is CFL, } R \text{ is REG } \Rightarrow L \cap R \text{ is CFL})\)

Proof sketch:

Given: DFA \(D\) for \(R\), PDA \(P\) for \(L\)

Make PDA \(M\) for \(R \cap L\)

states of \(M\): \((q, p)\)

state of \(D\): state of \(P\)

transitions: if \(\delta_D(q, a) = q'\), let \(\delta_M((q, p), a, X) = (q', p, X)\)

acceptance: assuming \(P\) accepts by final state, \(L = L(P)\), let \(F\) for \(M\) be \(\{ (q, p) \mid \text{both } q \text{ and } p \text{ are final } \}\)

\[= F_D \times F_P.\]

Closure properties

<table>
<thead>
<tr>
<th>Closed under...</th>
<th>Regular</th>
<th>Context-Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Concatenation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Kleene Star</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reversal</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Complement</td>
<td>✓</td>
<td>No. not always</td>
</tr>
<tr>
<td>Intersection</td>
<td>✓</td>
<td>No. not always</td>
</tr>
<tr>
<td>Homomorphism</td>
<td>✓</td>
<td>✓ Proof: replace a with h(a)</td>
</tr>
<tr>
<td>Inverse Homomorphism</td>
<td>✓</td>
<td>✓ Exercise</td>
</tr>
</tbody>
</table>
Theorem 2: \( L = N(M) \) for PDA \( M \Rightarrow L \) is CFL.

Proof sketch: Given \( M = (Q, \Sigma, \Gamma, S, q_0, Z_0, \delta) \),

Construct \( G = (V, T, P, S) \)

Variables: \( V = \Sigma \cup S \cup \{ q_Ap \} \mid p, q \in Q \) and \( A \in T \)

Intuition: Productions will ensure that

\[ q_Ap \Rightarrow \omega \iff (q, \omega, A)^* = (p, \epsilon, \epsilon) \]

Productions are of 2 types:

**Type I**: \( p \in Q, S \Rightarrow [q_0, Z_0] \)

Thus \( w \in N(M) \iff (q_0, w, Z_0)^*(p, \epsilon, \epsilon) \)

**Type II** will ensure validity of intuition

Intuition: Suppose \( (p, \omega, V_1 \cdots V_n) \in S(q, a, x) \)

Then add productions

\[ L(G) \]

for all possible choices of \( a, \cdots, r \in Q \)

**Example**

PDA \( M \):

\[ \begin{align*}
\epsilon, z_0/x \\
\epsilon, z_0/x \\
\epsilon, x/x \\
\epsilon, x/x \\
\end{align*} \]

\( q_0 \)

Note: \( N(M) = \{ \epsilon, z_0 \} \) only!

**Grammar G:**

**Type I**: \( S \Rightarrow [q_0, Z_0] [q_0, Z_0] \)

**Type II**: \( a \cdot (x, x) \Rightarrow [q_0, Z_0] \Rightarrow \epsilon \)

for \( (x, x) \)

Thus can simplify to

\( S \Rightarrow [q_0, Z_0] \)

\( [q_0, Z_0] \Rightarrow \epsilon \)

Clearly \( L(G) = \{ \epsilon \} \).

all variables are useless
(non-generating) except
\( S \) and \( [q_0, Z_0] \)