Module 6

Pushdown automata

The automaton for context-free languages

CS 360: Introduction to the Theory of Computing
Fall 2023

Daniel G. Brown
University of Waterloo

Topics for this module

- Pushdown automata definitions
- Languages of pushdown automata
- The equivalence of pushdown automata and context-free grammars
- Deterministic PDAs

The theorem that relates CFLs and PDAs was discovered by a few people, but one was the famous linguist Noam Chomsky.

Pushdown automaton

We need a new machine to accept context-free languages.
Our new machine is called a pushdown automaton.

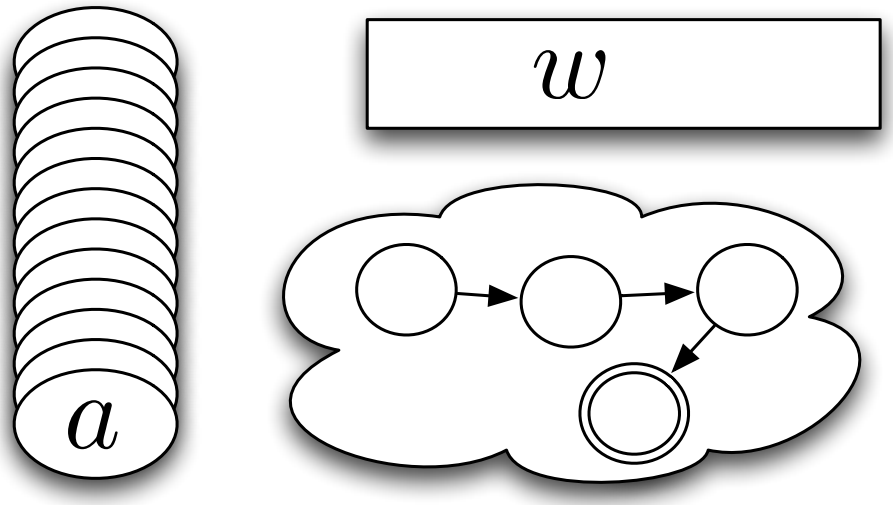
- The basic limitation of DFA/NFAs: no ability to keep infinite memory.
  - Important: FAs can keep a *finite* amount of memory: k bits can be stored by multiplying the state space by 2^k.
- Pushdown automata have infinite memory, but only accessed in specific order.
  - Things added to the memory recently must be read before things added long in the past.
  - This is why $L = \{ss \mid s \in \Sigma^*\}$ won't be accepted by them: we'd have to look back very far in the memory.

From an ε-NFA to a PDA

We treat the memory as a stack, and have a finite automaton controlling it.
Each transition is of the form:

- If top letter on the memory stack is b, and
  next letter in the input is a (or ε, if we take an ε transition),
  and the controlling FA is in state q, then:
- Go to state r,
  take the b off the top of the memory stack,
  eat the letter a from the input (again, a might be ε),
  and write the word w onto the top of the memory stack.

Envisioning a PDA



Each move:

- Take top plate off stack memory, and next letter in input sequence (or ε-transition).
- Based on current state, go to different state, (maybe) put new plates on stack.

Important: PDAs are also nondeterministic.

More tweaks in how they work

The stack must never be empty:

- There is a special "stack empty" symbol Z_0.
  - If Z_0 is on the top of the stack, we make sure it gets returned to bottom of stack after the next move.
- The string put on the stack may be empty (stack gets 1 symbol shorter), or long.
  - It does not have to just be 0 or 1 letters long!
  - It *must* be of finite length.

And, again, very important: PDAs are non-deterministic unless otherwise stated.
(To see later: deterministic PDAs are not as powerful as nondeterministic ones!)

Formal definitions

A PDA is a 7-tuple, $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where:

- Q = finite set of states for control
- Σ = finite input alphabet
- Γ = finite stack alphabet (often, it's just $\Gamma = \{Z_0\} \cup \Sigma$)
- δ = transition function
- q_0 = start state for machine
- Z_0 = stack start letter (bottom of stack character)
- F = accepting states for finite control

What do all of these elements mean?

There's the structure of an ε-NFA underlying the PDA:

- Q = states of the NFA
- Σ = alphabet for the NFA (that is, for the input string)
- q_0 = starting state of the NFA
- F = accept states of the NFA

There is the basis of the stack itself:

- Z_0: the character with which the stack is initialized
- Γ: the alphabet of symbols that are used in the stack itself.

And, lastly, there's the transition function, δ.

How does the transition function work?

How does the δ function look?

- From a state, a letter on the input (or ε), and a letter on the stack, pull the top letter off the stack,
- Then go to another state, and put a finite-length word on the stack.

We let the transition put many letters onto the stack, not necessarily only zero or one.
Also, PDAs are nondeterministic by default: we may have a variety of choices for a given states/input letter/stack symbol combination.

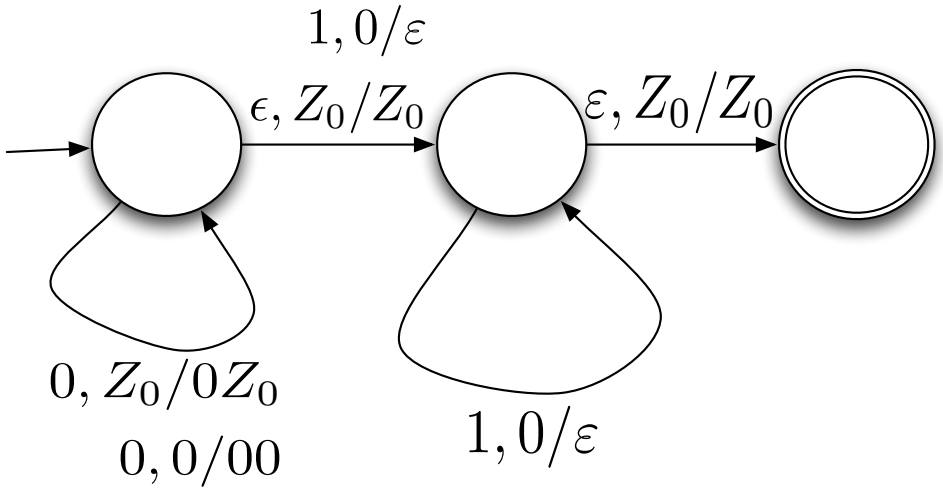
- This means that δ is of the form:
  $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to$ finite subsets of $Q \times \Gamma^*$
- (Why finite subsets? There are an infinite number of words in Γ^* we could put on the stack! Machine description must be finite.)

Drawing PDAs: notation

We draw them like DFAs, except for the labels of edges:

- Edges are labelled by both the input letter (or ε) and the top symbol of the stack; these are separated by a comma.
- We also must show what is pushed onto the stack: this follows a / character. The beginning of the string is the *top* of the stack after the transition occurs.
- It is possible that ε is pushed onto the stack: this makes the stack become shorter.

Example:



Instantaneous description of a PDA

Given a PDA, what characterizes the current state of its computation?

- What state the machine is in, q.
- What's left to read in the input, w.
- What's on the stack, γ.

PDA's instantaneous description: (q, w, γ).
We need a way to characterize one step in the PDA's computation:

- More complicated than the $\hat{\delta}$ notation: we need to show what happens to the stack.
- Notation describes what could follow the current state of the computation.
- Our new notation describes "one path" of the PDA's computation: to simulate the whole PDA, you'd need to present all nondeterministic possibilities.

Transitions in the PDA

What happens if we go forward one step?

- Let (q, z, γ) and (p, w, ζ) be two instantaneous descriptions of a PDA's state.
- From the first to the second in one transition: $(q, z, \gamma) \vdash (p, w, \zeta)$.
- What does that really mean?
  - If $\gamma = X\beta$ and $z = aw$ (where X is one letter long, a is either one letter long or ε) and $\delta(q, a, X)$ contains (p, α), where $\zeta = \alpha\beta$.
  - (That is, state q has a transition to state p labelled $a, X/\alpha$; remember also that α is of finite length)
  - Then $(q, z, \gamma) \vdash (p, w, \zeta)$
  - Read "produces in one step".

If the machine P needs to be indicated, write $I \underset{P}{\vdash} J$.

How does that work?

We start in the instantaneous description (q, z, γ)
Two choices: either we consume a letter from the input, or we don't.

- Suppose we take an ε-transition and do not read any input symbol.
  - Let $\gamma = X\beta$, so the top symbol of the stack is X.
  - We must follow a transition from $\delta(q, \varepsilon, X)$.
  - $\delta(q, \varepsilon, X)$ is a finite set of members from $Q \times \Gamma^*$. Suppose chosen member is (p, α).
  - We remove no letters from the input string, take the X off the stack, and add α to the beginning of the stack.
  - This brings us to the instantaneous description $(p, z, \alpha\beta)$.

Second possibility: consume an input symbol

- Or we take a transition that consumes a symbol from z.
  - Again, let $\gamma = X\beta$ and $z = aw$.
  - Follow a transition from $\delta(q, a, X)$; suppose we choose (p, α).
  - Remove a from the input, X from the stack, move to state p, and push α to the top of the stack.
  - Arrive at $(p, w, \alpha\beta)$.

Longer derivations

- 1 step in P: $(q, x, \beta) \underset{P}{\vdash} (q', x', \alpha)$.
- An arbitrary (finite) number of steps: $(q, x, \beta) \overset{*}{\underset{P}{\vdash}} (q', x', \alpha)$.

A proper definition

Proper inductive definition of $I \overset{*}{\underset{P}{\vdash}} J$:

- Base case: For any instantaneous description I, $I \overset{*}{\underset{P}{\vdash}} I$.
- Inductive case: If $I \underset{P}{\vdash} K$ and $K \overset{*}{\underset{P}{\vdash}} J$, then $I \overset{*}{\underset{P}{\vdash}} J$.

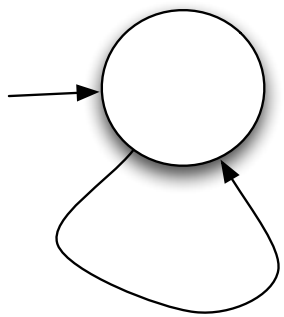
(A finite sequence of instantaneous descriptions $K_1, K_2, \ldots K_n$ such that $I = K_1$, $J = K_n$ and for all $i = 1 \ldots n-1$, $K_i \underset{P}{\vdash} K_{i+1}$.)

We won't give a parallel to $\hat{\delta}$ for PDAs.

Bad things in PDAs: running forever

Two bad kinds of events; both also possible with ε-NFAs.
Running forever, using ε transitions:

- Only a finite number of possible states we can reach following ε transitions, and they'll cycle.
- But we also can grow the stack, so the configurations are different.



$\varepsilon, Z_0/Z_0Z_0$

- $(q_0, x, Z_0) \vdash (q_0, x, Z_0Z_0) \vdash (q_0, x, Z_0Z_0Z_0) \vdash (q_0, x, Z_0Z_0Z_0Z_0) \overset{*}{\vdash}$ ...

Bad things in PDAs: crashing

The PDA crashes if there are no available transitions from the current instantaneous description, or stack is empty.

- No available transitions: if the current instantaneous description is (q, x, z), where $x = aw$ and $z = X\beta$, and the sets $\delta(q, a, X)$ and $\delta(q, \varepsilon, X)$ are both empty, then the PDA can make no transitions.
- Empty stack: If the current instantaneous description is (q, x, ε), the PDA can make no transitions: it is supposed to remove the top stack symbol, and there is none!
  - Note: if the machine arrives in an instantaneous description $(q, \varepsilon, \varepsilon)$, and q is an accept state, the machine accepts.

Basic rules about how PDAs work

A valid computation in a PDA P: sequence of instantaneous descriptions $K_1, K_2, \ldots, K_n$ where $K_1 \underset{P}{\vdash} K_2 \underset{P}{\vdash} \ldots \underset{P}{\vdash} K_n$.

A valid computation remains valid if we:

1. Add string $w \in \Sigma^*$ to the end of the input for all K_i.
2. Remove the same suffix from the input for all K_i.
3. Add string $\gamma \in \Gamma^*$ to the end of the stack for all K_i.

Why does this matter?

- Lets us isolate part of a computation.
- Adding to the end of the stack will help us with some theorems about the equivalence of PDAs and CFGs.

We can prove the first two principles in the same theorem:

Theorem: Suppose that for a given PDA P, $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$. Then for any string $w \in \Sigma^*$ and $\gamma \in \Gamma^*$, it is also the case that $(q, xw, \alpha) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta)$ and $(q, x, \alpha\gamma) \overset{*}{\underset{P}{\vdash}} (p, y, \beta\gamma)$.

Proof: Consider all transitions that show: $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$.

- None of these transitions is use characters of w or γ.
- As such, each move is valid if those symbols are added to the end of the input or stack.

Removing common suffixes

Similarly, if we remove common suffixes from the input, still valid:

Theorem: If $(q, xw, \alpha) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta)$, then $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$.

Proof: This is easily shown by following all of the transitions in the first derivation.

- None of them consumes any of the letters of w, since we can only remove symbols from the input of a PDA, not add to it.
- As such, they are also valid to show that $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$.

How does a PDA accept?

- Like a DFA: at end of the input, the machine can be in an accept state. [Later: if the stack is empty.]
- This is acceptance by final state.

To formalize:

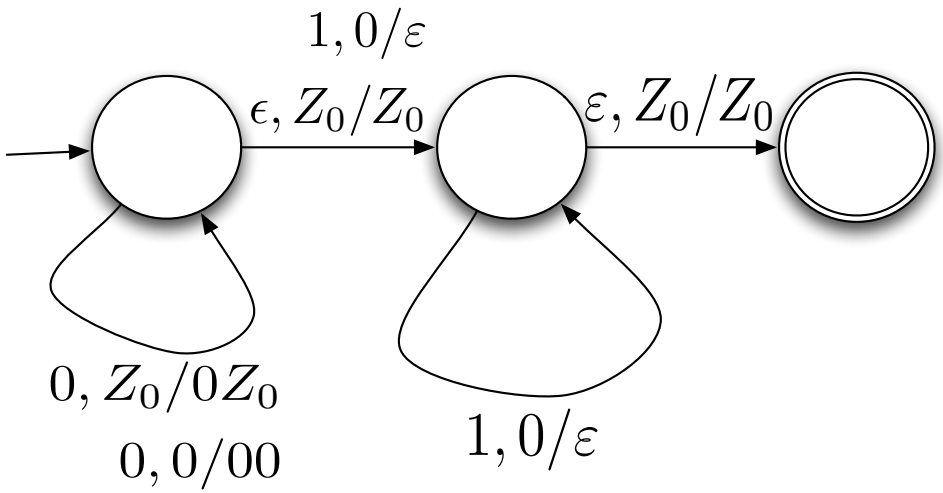
- PDA P accepts word x if $(q_0, x, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, y)$ for any string y and state $q \in F$.

Language of the PDA: $L(P) = \{\text{all words in } \Sigma^* \text{ accepted by the PDA}\}$.

An example: $L = \{0^i 1^i\}$

A PDA for $L = \{0^i 1^i \mid i \geq 0\}$:

- These can be hard to draw!
- 3 states: one for pushing 0's onto stack, while reading them, one for popping 1's off, one for accepting at end of word.
- We'll call these states q_0, q_1, and q_2.



Why does this work?

For words in L:

- Start with stack Z_0
- Add symbols from w to it.
- ... until we reach the first 1 symbol. Then, we absorb 1's from the stack and the input.
- ... until we're done, and then we follow the ε transition to q_2 and accept.

More specifically, on the input 0011, the computation is:

$$\begin{aligned}(q_0, 0011, Z_0) &\vdash (q_0, 011, 0Z_0)\\ &\vdash (q_0, 11, 00Z_0)\\ &\vdash (q_1, 1, 0Z_0)\\ &\vdash (q_1, \varepsilon, Z_0)\\ &\vdash (q_2, \varepsilon, Z_0)\end{aligned}$$

Some formality

- Only words $w \in 0^*1^*$ can have the property that $(q_0, w, y) \overset{*}{\vdash} (q_2, \varepsilon, z)$ for any strings y and z, since we transition from q_0 to q_1 by consuming a word from $0^*(1 + \varepsilon)$, and we transition from q_1 to q_2 by consuming a word from $1^*\varepsilon$; the concatenation of these is 0^*1^*.
- The only valid computation after reading in 0^i for $i > 0$ is $(q_0, 0^i w, Z_0) \overset{*}{\vdash} (q_0, w, 0^i Z_0)$.
- The only valid computation after reading in 0^i1^k is $(q_0, 0^i1^k w, Z_0) \overset{*}{\vdash} (q_1, w, 0^{i-k} Z_0)$, unless $k = i$, at which point we can transition to (q_2, ε, Z_0). If $k > i$, then the machine has no valid computations that read the first $i + k$ symbols.
- We can only accept if after reading in 0^i1^i, there are no more symbols to read, so we need not fear accepting 0^i1^k when $k > i$.

Another example: a PDA for palindromes

The language $L = \{x!x^R \mid x \in \{a, b\}^*\}$ is context-free.

- (A grammar is: $S \to aSa|bSb|!$)

How to build a PDA for this?
Something like:

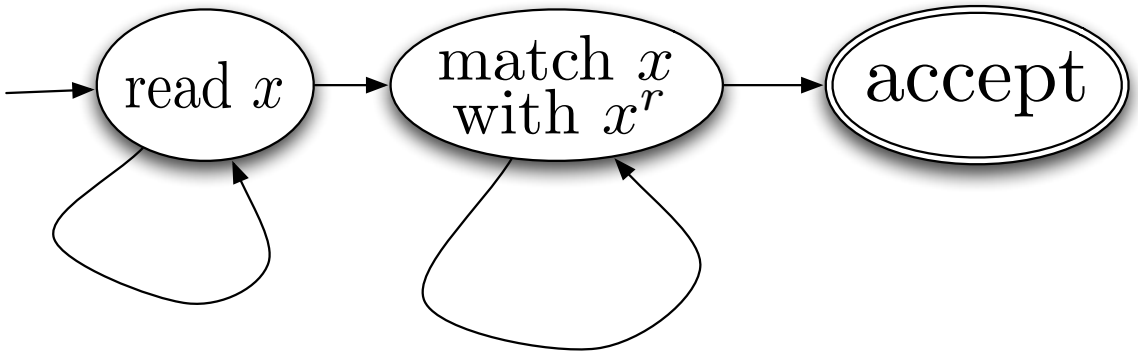
- Push x on the stack, until we get to !
- Then read in the !
- Then pop letters off the stack. If all stack and input letters match, then the word is in the language.

Reject when:

- There's not exactly one ! symbol
- The string after the ! isn't the same length as what's before.
- The two substrings don't match each other.

An idea of how to build it

Like this:



- In reading state, add the letters read to stack
- In matching state, compare against letters from stack.
- If we get to the stack end and end of input string at the same time, accept.
- Again, we'll call these states q_0, q_1, and q_2.

Transitions in the PDA

In reading state:

- If we read in a from input, and top of stack is b, add ab to front of the stack.
- (In general, if we read in x and the top of stack is y, add xy to the stack.)

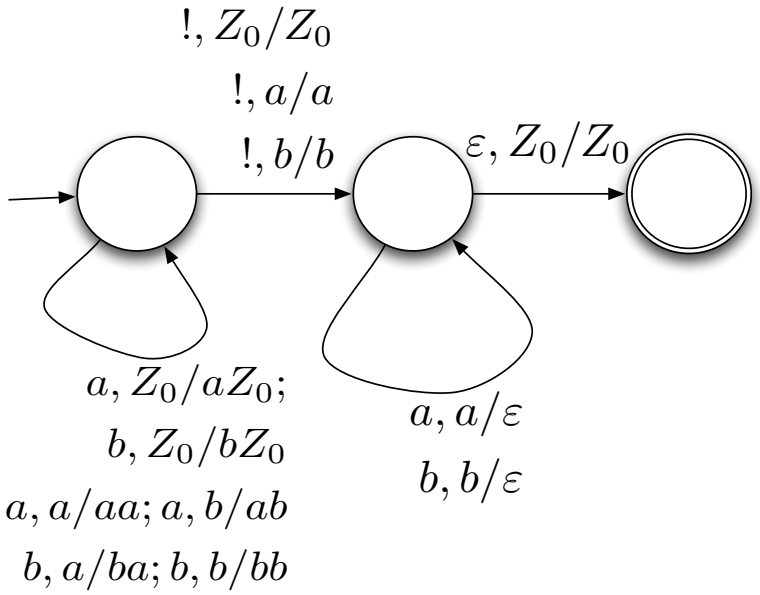
In matching state:

- If we read in an a from input, and the top of stack is a, keep going. Don't replace the a.
- If input/stack mismatch, crash.
- Keep an ε-transition from matching state to accept state, for when the top letter of the stack is the empty-stack character, Z_0.

The final PDA

Last transitions:

- When reading !, go to the matching state, and keep stack from changing.
- If we reach the end of the input, take the ε-transition to q_2, and accept the palindrome.
- Machine crashes if we go into the accept state and there are still letters to read from input.



An example computation with this PDA

Consider the input sequence $ab!ba$. Here's the accepting path:

$$\begin{aligned}
(q_0, ab!ba, Z_0) &\vdash (q_0, b!ba, aZ_0) \\
&\vdash (q_0, !ba, baZ_0) \\
&\vdash (q_1, ba, baZ_0) \\
&\vdash (q_1, a, aZ_0) \\
&\vdash (q_1, \varepsilon, Z_0) \\
&\vdash (q_2, \varepsilon, Z_0).
\end{aligned}$$

Since we end in an accepting state, q_2, and have processed the entire input, the PDA accepts.

Acceptance by empty stack

Current model of acceptance:

- The language of the machine is $L(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (p, \varepsilon, \alpha)$ for some accept state $p \in F$ and any stack string $\alpha\}$.

New model of acceptance:

- The language of the machine is $N(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (p, \varepsilon, \varepsilon)$ for any state $p\}$.

Called acceptance by empty stack, for obvious reasons.
Notation: $N(P)$.
A PDA accepting in this manner doesn't have accept states, so is a 6-tuple: $(Q, \Sigma, \Gamma, \delta, q_0, Z_0)$.

Fact about acceptance by empty stack

Theorem: Suppose that P_F is a PDA that accepts in the normal way (by terminating the word in an accept state.)
Then there exists a PDA, P_N, which accepts $L(P_F)$ by empty stack.
That is,

- $L(P_F)$ = every word x such that the machine P_F can end in an accept state after reading the entirety of input x.
- $N(P_N)$ = every word x such that P_N, run on x, has empty stack at the end of reading x, on at least one valid computation.

For any P_F, we can find P_N, and vice versa.

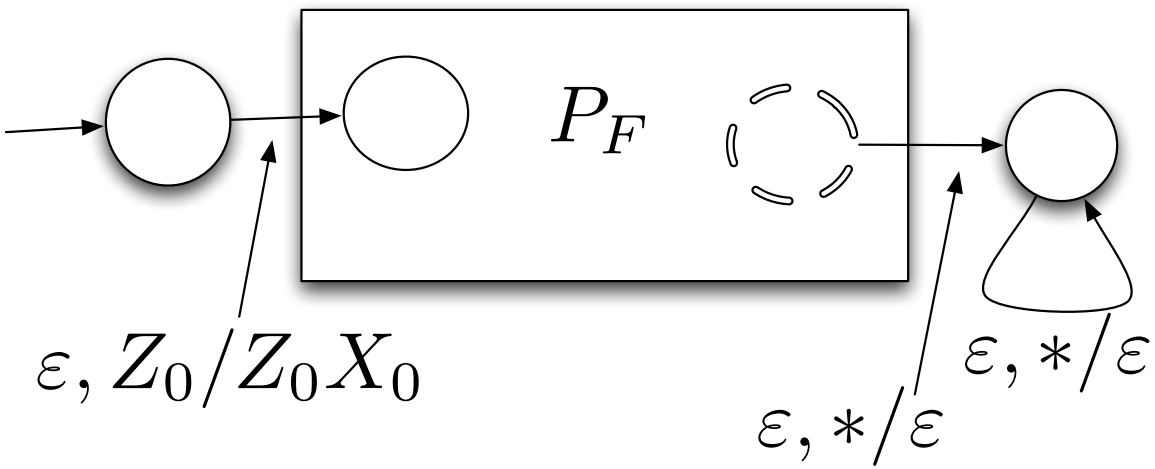
Proving the theorem

We'll use acceptance by empty stack to show that PDAs accept exactly the class of context-free languages, which is why we care.
Suppose we are given a PDA P_F whose language, using acceptance by final state, is L.

- How can we construct a PDA P_N that accepts L by empty stack?
- We must ensure two properties hold about our new PDA, P_N:
  - It reads its input and empties its stack whenever the original PDA, P_F accepts the word.
  - It doesn't empty its stack at the end of the input word at any other time.

The new construction

- From every accept state in P_F, we take an ε-transition to a "drain" state that empties the stack. (If there are input letters remaining, then the machine crashes.)
- Before doing any computation, we put a special character X_0 at the end of the stack, behind even the Z_0 character.
  - We only remove the Z_0 character when we are transitioning to or already in our "drain" state.
  - All computations happen as before, so we can only remove X_0 when P_F would have already accepted.



From empty stack to final state

A similar construction can be used to transform a PDA P_N with $N(P_N) = L$ to a PDA P_F with $L(P_F) = L$.
Again, need special character X_0 added to the end of the stack that only gets removed after P_N would have accepted.

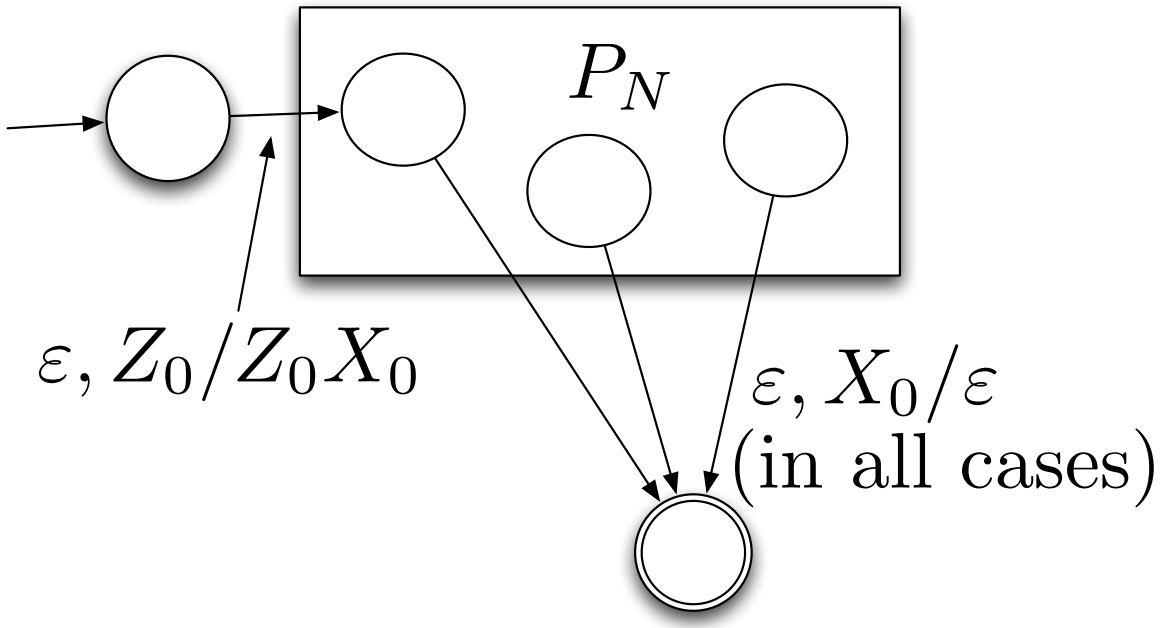
- Here, the character alerts that we've removed all of the stack characters.
- Without a special symbol, we'd crash trying to detect empty stack.
- We take an ε-transition when we see this special character to the machine's only accept state.

To ensure that the new PDA P_F only accepts (by final state) the words P_N accepts:

- No transitions from the new accept state: if we've not processed the entire input word, the machine crashes instead of accepting.

The actual construction

New pushdown automaton P_F:



Note: not proved either side. Both proofs in your book, not especially enlightening.

Proving that PDAs and CFGs both accept CFLs

Main reason for introducing PDAs: accept exactly the class of context-free languages
We must prove two things:

- Given a CFG G whose language is L, we can create a PDA P such that $N(P) = L$.
- Given a PDA P where $N(P) = L$, we can create a CFG G such that $L(G) = L$.

Note: prove using acceptance by empty stack in both cases. We showed acceptance by final state since it makes PDAs feel like ε-NFAs.
We'll start with the first of these.

How to make a PDA from a grammar?

What do we need to do?
Simulate leftmost derivations of words in the language. If one of those can generate w, accept!

- If the derivation produces terminals, they should match letters from w
- If the derivation produces non-terminals, keep the derivation going.
- Stack memory keeps unresolved variables from the derivation

More specific description

Example: A grammar for palindromes with no !: $S \to aSa|bSb|b|\varepsilon|a$
We want to try possible derivations from S:
Idea:

- PDA produces the word by leftmost derivation.
- Top character on the stack is either a terminal to match against w or a variable. Start the PDA with the stack character S.
- Use all possible expansions for a variable: when we expand a variable, remove it from the stack and replace it with the right hand side of its rule.
- Nondeterminism: simulate all possible rule expansions, implicitly generating the whole language

A rough idea

We have our grammar:

- $S \to aSa|bSb|b|\varepsilon|a$

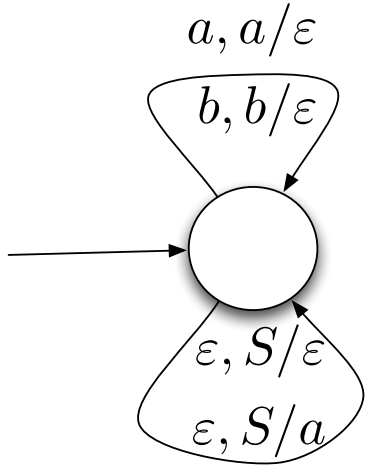
Consider the word $abbba$. We have the leftmost derivation:
$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abbba$.
Keep the string on the stack as we derive it, and match the first characters in the input string. Something like:

$$\begin{aligned}
(*, abbba, Z_0) &\vdash (*, abbba, SZ_0) \\
&\vdash (*, abbba, aSaZ_0) \\
&\vdash (*, bbba, SaZ_0) \\
&\vdash (*, bbba, bSbaZ_0) \\
&\vdash (*, bba, SbaZ_0) \\
&\vdash (*, bba, bbaZ_0) \\
&\vdash (*, ba, baZ_0) \\
&\vdash (*, a, aZ_0) \\
&\vdash (*, \varepsilon, Z_0)
\end{aligned}$$

How to make a PDA from that?

Basic idea:

- Allow every production that consumes the top variable, nondeterministically.
- Also have match transitions, corresponding to when the stack matches the input.
- Start with an S on the stack (start character for grammar).
- 1 state!



Does this work?

Yes, as we'd hoped.

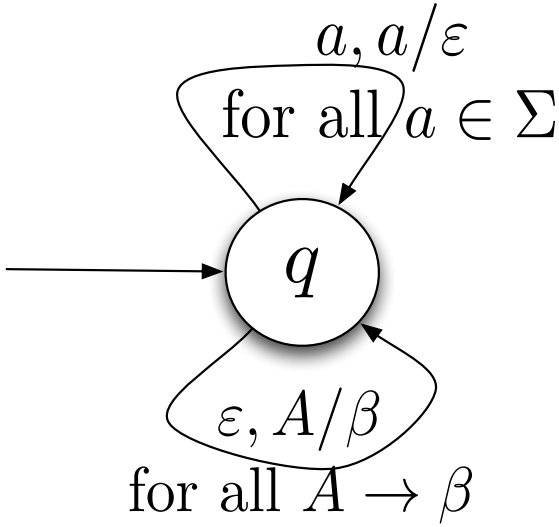
- Start with S on the stack
- Try all possible derivations for S
- Match characters in x until we get to another variable in the derivation
- Try derivations for that variable
- If we reach end of w and no more variables to derive, we accept.

This can, of course, be generalized for any grammar $G = (V, \Sigma, S, P)$

General structure

Consider the PDA
$P = \{\{q\}, T, V \cup T, \delta, q, S\}$,
where δ has the following members:

- For all rules $A \to \beta$ in the grammar, $\delta(q, \varepsilon, A)$ includes (q, β). There are no other members of $\delta(q, \varepsilon, A)$.
- For all terminals $a \in T$, $\delta(q, a, a) = \{(q, \varepsilon)\}$.
- All other values of $\delta(q, x, y)$ are the empty set.



Statement of the theorem

Theorem: This PDA, P, accepts $L(G)$ by empty stack. Two parts: $L(G) \subseteq N(P)$, and $N(P) \subseteq L(G)$.
We will begin with showing that $L(G) \subseteq N(P)$.

- Let w be a word in $L(G)$.
- Then there exists some leftmost derivation for w in G:

$$S = \gamma_1 \underset{lm}{\Rightarrow} \gamma_2 \underset{lm}{\Rightarrow} \gamma_3 \underset{lm}{\Rightarrow} \ldots \underset{lm}{\Rightarrow} \gamma_n = w.$$

- For all of the γ_i except γ_n, there exists a leftmost variable.
  - Divide $\gamma_i = x_i \alpha_i$, where α_i begins with the first variable in γ_i, A_i. (The exception is that $\alpha_n = \varepsilon$.)
  - Since $\gamma_i \overset{*}{\underset{lm}{\Rightarrow}} w$, we know that x_i is a prefix of w: let $w = x_i y_i$.
- In the PDA, $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$ for all i
- In particular, $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$, and so $w \in N(P)$, because P accepts w by empty stack.

Finishing the proof in one direction

What we want to prove:

- For all i, $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$.

Proof by induction:

- Base case: $x_1 = \varepsilon, y_1 = w, \alpha_i = S$, so we have that $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$ trivially.
- Inductive case: Suppose that $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$, where $i < n$.
  - Then $S \underset{lm}{\Rightarrow} \gamma_i$, where $w = x_i y_i$, $\gamma_i = x_i \alpha_i$ and $\alpha_i = A_i \eta_i$.
  - Follow next rule: $A_i \to \beta_i$. We know that there is also an entry in $\delta(q, \varepsilon, A)$ with value (q, β_i), so $(q, y_i, \alpha_i) \underset{P}{\vdash} (q, y_i, \beta_i \eta_i)$.
  - Next, we match all of the terminals at the top of the stack against the terminals in the input, since y_i and β_i must have a (possibly empty) prefix of matching terminals. This is possible since $\delta(q, a, a) = \{q, \varepsilon\}$ for all $a \in T$.
  - Consume all terminals until $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_{i+1}, \alpha_{i+1})$. This is sufficient to prove that $L(G) \subseteq N(P)$.

The proof in the other direction

Suppose that $w \in N(P)$. We need to show that $w \in L(G)$.
We'll show something more general:

- For any variable A, if $(q, w, A) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$, then $A \overset{*}{\underset{G}{\Rightarrow}} w$.
- This is sufficient: if $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$, then $S \overset{*}{\underset{G}{\Rightarrow}} w$, and $w \in L(G)$.

Proof by induction on the length of the computation that takes (q, w, A) to $(q, \varepsilon, \varepsilon)$:

- Base case: one step.
  - If so, then (q, ε) is a member of the set $\delta(q, \varepsilon, A)$, meaning that there is a rule of the form $A \rightarrow \varepsilon$ in G.
  - Then $w = \varepsilon$ and $A \overset{*}{\underset{G}{\Rightarrow}} w$ as desired.

The inductive case

Inductive case: Suppose that for all variables A, if $(q, w, A) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$ in fewer than k steps, then $A \overset{*}{\underset{G}{\Rightarrow}} w$. Now consider a k-step computation $(q, w, A) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$.

- The first step in the computation must be a rule derived from the form $A \to Y_1 Y_2 \ldots Y_k$ (where the Y_i are in $T \cup V$), since those are the only rules valid when A is on top of the stack.
  So we move to the configuration $(q, w, Y_1 \ldots Y_k)$.
- Divide the computation from this time forward into phases: there will be some first point when the stack contains just $Y_2 \ldots Y_k$, then some first point when the stack contains just $Y_3 \ldots Y_k$, and so on, until the first time it has just Y_k.
  (Each step removes only at most one symbol from the stack, and we eventually empty the stack.)
- Consider when the stack contains $Y_2 \ldots Y_k$: we know that $(q, w, S) \underset{P}{\vdash} (q, w, Y_1 Y_2 \ldots Y_k) \overset{*}{\underset{P}{\vdash}} (q, w', Y_2 \ldots Y_k)$, for some suffix w' of w, where $w = w_1 w'$.

End of the inductive case

- By one of our rules about reasoning in PDAs, we can see that $(q, w_1, Y_1 Y_2 \ldots Y_k) \underset{P}{\vdash} (q, \varepsilon, Y_2 \ldots Y_k)$.
  Moreover, since in the computation, we never touch the symbols of $Y_2 \ldots Y_k$ on the stack, we also see that $(q, w_1, Y_1) \underset{P}{\vdash} (q, \varepsilon, \varepsilon)$.
- This transformation must take fewer than n steps, so we see that $Y_1 \overset{*}{\underset{G}{\Rightarrow}} w_1$, by our inductive hypothesis.
- We then apply this to each subsequent Y_i, giving a derivation $Y_i \overset{*}{\underset{G}{\Rightarrow}} w_i$ for all i, where $w = w_1 w_2 \ldots w_k$.
- Finally, we have a derivation for w, by combining these together:

$$S \Rightarrow Y_1 Y_2 \ldots Y_k \overset{*}{\underset{G}{\Rightarrow}} w_1 Y_2 \ldots Y_k \overset{*}{\underset{G}{\Rightarrow}} w_1 w_2 Y_3 \ldots Y_k \overset{*}{\underset{G}{\Rightarrow}} w_1 w_2 \ldots w_k = w$$

Next, from a PDA to a CFG

We've shown the easier case: for a CFG G with language $L(G) = L$, we can show a PDA P such that $N(P) = L$.
Now, from a PDA P, we must find a grammar G such that $N(P) = L(G)$.
This is harder.

- Suppose that P can move from the instantaneous description (p, w, X) to $(q, \varepsilon, \varepsilon)$ for some stack alphabet letter X.
- How does that happen?

How do we make the stack lose a letter?

Suppose the first step is $(p, az, X) \vdash (r, z, Y_1 Y_2 \ldots Y_k)$. (We may be following a transition that does or does not consume the first symbol of w, so $a = \varepsilon$ or a is a symbol in Σ.)

- The process begins by removing X from the stack, replacing it with some string $(Y_1 \ldots Y_k)$, and moving to state r.
- Then, we go from state r to some state r_1, eventually removing all of the symbols of Y_1 from the stack, so that it consists of just $(Y_2 \ldots Y_k)$; this process may consume some input letters,from w, or it might not.
- Eventually, we wind up in state q, in the instantaneous description $(q, \varepsilon, \varepsilon)$, having read all of the letters of w and all of the symbols in $Y_1 \ldots Y_k$.

Making a grammar from this

To build a grammar, we need to turn this into a formal set of rules.

- We'll have a non-terminal $[pXq]$ for every stack symbol X and every pair of states p and q. (Each of these $[pXq]$ triplets is a letter in the new grammar we are building.)
  - Want: $[pXq] \stackrel{*}{\underset{G}{\Rightarrow}} w$ exactly when $(p, w, X) \stackrel{}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$.
  - Depends on the productions of the grammar.
- Suppose that one of the productions in $\delta(p, a, X)$ is $(r, Y_1 Y_2 \ldots Y_k)$
  - Note: a can be ε, and so can $Y_1 \ldots Y_k$.
  - Then add a production $[pXq] \rightarrow a[rY_1r_1][r_1Y_2r_2]\ldots[r_{k-1}Y_kr_k]$ to the grammar, for all choices of $r_1, \ldots r_{k-1}$ from our state set Q; let $r_k = q$, to ensure the simulated PDA ends in state q.
- The language of the PDA is the union of the language corresponding to emptying the stack in all possible states, so for every state q we need a rule:
  - $S \rightarrow [q_0 Z_0 q]$.

How's this work?

Basic idea, again:

- Each PDA transition either reads a symbol from the input, or doesn't.
- Each PDA transition changes state.
- Each PDA transition takes one symbol off the stack, and puts some on.

In a leftmost derivation in our grammar:

- The terminals at the beginning of the word we're producing are exactly the ones that could have been read off the input.
- The string of nonterminals, each of the form $[pYr]$, indicate in the Y characters the symbols on the current stack.
- The first state in the first non-terminal is the current state of the PDA.
- The other states correspond to the state the automaton will be in when we have shrunk the stack.

The proof

Actually, we're not going to do it.

- The basic structure is not especially interesting; it's in Section 6.3.2 of your book.
- The really interesting thing is the idea; the proof is just about filling in the details

One thing to note: this construction is finite-size.

- This may not be obvious, but it's because the strings put on the stack are always of finite length. This means we add a finite number of rules, though potentially enormous. (How big?)

Deterministic PDAs

In a DPDA, every move is fixed.

- Can have ε-transitions, but they must be the only valid transition.
- If there is a transition from q_0 with X on the top of the stack that does consume input letters, there are no ε-transitions.
- Must be at most 1 transition from every state for every (input letter, stack letter) pair.

More formally, for a given state q_0:

- If $|\delta(q_0, a, X)| = 1$ for some letter a, then $|\delta(q_0, \varepsilon, X)| = 0$.
- $|\delta(q_0, a, X)| \leq 1$ always.
- $|\delta(q_0, \varepsilon, X)| \leq 1$ also, always.

Why care about DPDAs?

Some CFLs aren't accepted by DPDAs.

- Maybe the languages accepted by DPDAs are somehow simpler than regular CFLs.

One important difference versus DFAs:

- Possible no transition is available.
- Can still crash.

An example of a language accepted by a DPDA

Ordinary PDA for the language: $L = \{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$.

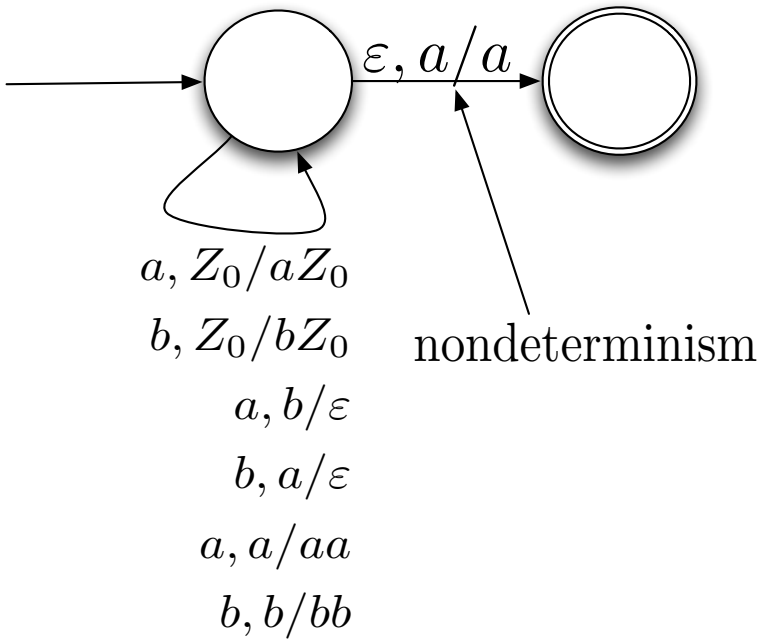
- (Words with more a's than b's.)

As we scan the word left-to-right, there will be more of one letter than the other.

- Keep track of the number of letters that we have more of.
- (Example: read in $aaaabab$ so far, then we should have three a's on the stack, since we have 5 a's and 2 b's.)
- Accept when there's still an a on the stack at the end.

A nondeterministic PDA for L

This gives a 2-state PDA like this:



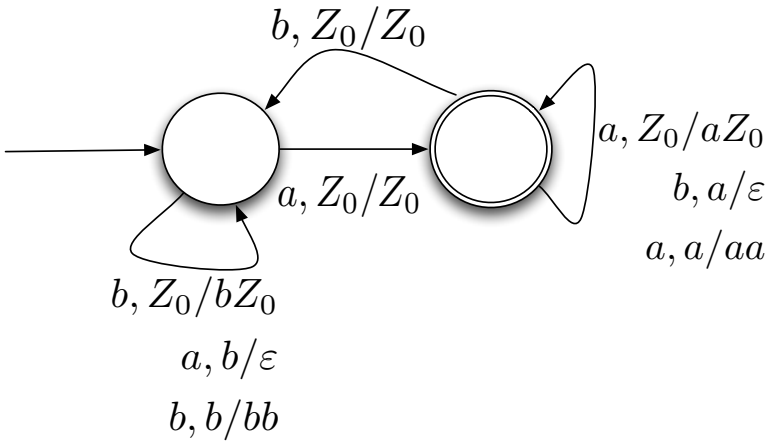
- First two transitions in state q_0: prefix is balanced
- Second two transitions in state q_0: reducing imbalance
- Last two transitions in state q_0: adding to imbalance
- Nondeterministic: ε-transition to state q_1.

Can we make this PDA deterministic?

The current PDA is nondeterministic, because we have transitions when we pull an a off the stack that do eat input letters, and that don't.
Can we make small changes to make it deterministic?

- As before, but with states for "balanced or accumulating b's" and for "accumulating a's".
- The second of these is the accept state.

This is deterministic



This is a deterministic PDA:

- There's always at most one choice for what to do next.

But the computation remains fundamentally the same: we've just divided the transitions from before into the two steps. (In the accept state, the stack is one symbol short in the new DPDA than in the previous nondeterministic PDA.)
We jump from one state to the other when we switch from more *a*'s than *b*'s to not, or the other way around.

Not all CFLs are accepted by DPDAs

Fact (not proved in your book):
There exist context-free languages that are not accepted by DPDAs.

- Nondeterminism specifically makes PDAs more powerful.
- (This is not true for either Turing machines or for finite automata.)

Two different ways to prove this:

- Directly.
- Indirectly (next module)

We'll talk a little about the first, but rely on the second.
Easy theorem: Any regular language L is accepted by a DPDA.

- The DPDA just ignores the stack, and simulates the DFA that accepts L.

Direct proof sketch that DPDAs don't accept every CFL

We know that the language $L = \{x \in \{a, b\}^* | x \text{ is a palindrome}\}$ is a CF language, and hence accepted by a PDA.
We can argue that it's not accepted by a DPDA.

- Why? There has to be a way to identify the palindrome's middle.
- After reading the first 8 letters of the string *aabaabaa*, must both be ready to accept, if that's the end of the string, and be ready to read in more letters, in case it's just the first half of the string.

Determinism makes this impossible

In a DPDA, that's just not possible.
Especially bad: need to keep track of lots of other possibilities:

- the 8-letter string is the beginning of the 11-letter palindrome *aabaabaabaa*,
- Etc.
- Can't keep track of all of these choices for the middle.
- (Only one transition available at a time.)

This can be made into a proof if we know how many states the DPDA is claimed to have; try it.

DPDAs and ambiguity

One straightforward theorem:
Theorem: If P is a DPDA, then $N(P)$ has an unambiguous grammar.

- Proof is not hard: follow the grammar produced in the theorem that transformed a PDA into a CFG.
- The only place of some concern is that transitions where $\delta(q, a, X)$ included multiple new stack symbols, $(p, Y_1 Y_2 \ldots Y_n)$ turned into a pile of distinct new rules in the grammar.
- However, since the PDA is deterministic, we can see that only one parse tree in the grammar will actually lead to the word that is accepted by the DPDA, so the grammar will still be unambiguous.

You can modify the proof to work for acceptance by final state, too. (That's Theorem 6.21 in your book.)

Overall hierarchy of languages

We now know that:

- All finite languages are regular.
- All regular languages can be accepted by a DPDA, so are DCFLs. There are non-regular DCFLs.
- DCFLs are all not inherently ambiguous. There are languages that are not accepted by DPDAs that are not inherently ambiguous, and some CFLs that are not inherently ambiguous are not DCFLs.
- All CFLs are accepted by PDAs, and the languages accepted by PDAs are exactly the CFLs.

What about languages that *aren't* context free?

End of module 6

We've seen a lot in this module!

- Pushdown automata: an automaton that accepts context-free languages.
- Different ways for PDAs to accept: empty stack or final state.
- Proofs that CFLs and PDAs represent the same classes of languages
- Deterministic languages: in the case of PDAs, nondeterminism gives a different class of language entirely.