Module 6

Pushdown automata

The automata for context-free languages

CS 360: Introduction to the Theory of Computing
Winter 2022

Collin Roberts
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; 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.

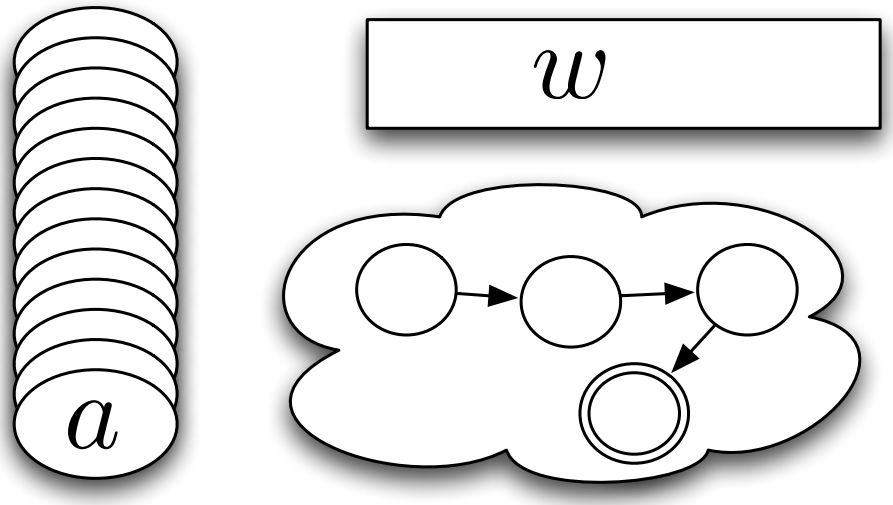
- The basic limitation of DFA/NFAs: no ability to keep an unbounded amount of memory.
  - Important: FAs can keep only a *finite* amount of memory: k bits can be stored by a finite automaton having 2^k states.
- Pushdown automata have an unbounded amount of memory, but only accessed in specific order.
  - Things added to the memory recently must be read before things added long in the past.
  - This is why $L = \{ss \mid s \in \Sigma^*\}$ will not be accepted by such an automaton: we would 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 the next letter in the input word 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 a 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.
(We will see later that 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 is 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 is the structure of an ε-NFA underlying the PDA:

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

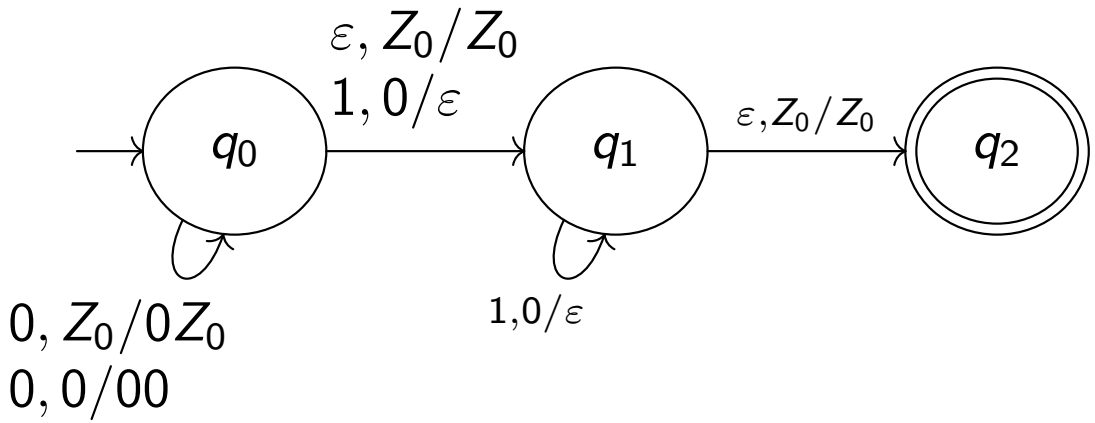
- This means that δ is of the form:
  $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to$ finite subsets of $\{Q \times \Gamma^*\}$
- (Why finite subsets? There are an infinite number of words in Γ^* that we could put on the stack! The machine description must be finite.)

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. E.g. if the stack is empty, and we push $w = 10$ onto the stack, then 1 is on top of the stack and 0 is below 1.
- 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 configuration of its computation?

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

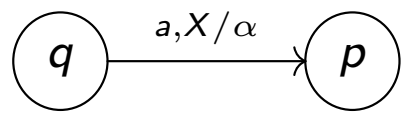
The PDA's instantaneous description is: (q, w, γ).
We need a way to characterize one step in the PDA's computation:

- This is more complicated than the $\hat{\delta}$ notation for finite automata: we need to show what happens to the stack.
- Notation describes what could follow the current configuration of the computation.
- Our new notation describes "one path" of the PDA's computation: to simulate the whole PDA, you would need to present all nondeterministic possibilities.

Transitions in the PDA

What happens if we go forward one step?

- Let (q, w, γ) and (p, z, ζ) be two instantaneous descriptions of a PDA's configuration.
- From the first to the second in one transition: $(q, w, \gamma) \vdash (p, z, \zeta)$.
- What does that really mean?
  - If $\gamma = X\beta$ and $w = az$, 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, there is a transition:



    Remember also that α is of finite length.
  - Then we write $(q, w, \gamma) \vdash (p, z, \zeta)$
  - Read $\vdash$ as "produces in one step".

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

How does that work?

We start in the instantaneous description (q, w, γ)
Two choices: either we consume a letter from the input, or we do not.

- Suppose first that 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 $\delta(q, \varepsilon, X)$.
  - By definition $\delta(q, \varepsilon, X)$ is a finite set of members from $Q \times \Gamma^*$. Suppose the chosen member of the set 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 $(p, w, \alpha\beta)$.

Second possibility: consume an input symbol

- Or suppose that we take a transition that consumes a symbol from w.
  - Again, let $\gamma = X\beta$ and $w = az$.
  - 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 $(p, z, \alpha\beta)$.

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 $I \overset{*}{\underset{P}{\vdash}} J$.

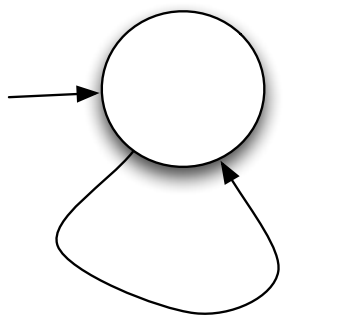
(There exists a finite sequence of instantaneous descriptions $K_1, K_2, \ldots, K_n$ such that $I = K_1$, $J = K_n$ and for all $i = 1, \ldots, n-1$, $K_i \underset{P}{\vdash} K_{i+1}$.)

We will not attempt to define an analogue to $\hat{\delta}$ for PDAs.

Bad things in PDAs: running forever

There are two bad kinds of events too worry about here. Both are also possible with ε-NFAs.
Running forever, using ε-transitions:

- There are only a finite number of possible states we can reach following ε-transitions, and they will cycle.
- But now we also can grow the stack, so the configurations are different at each step.



$\varepsilon, Z_0/Z_0Z_0$

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

Bad things in PDAs: crashing

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

- No available transitions: if the current instantaneous description is (q, x, z), where $x = aw$ and $z = X\beta$ for some alphabet symbol a and stack symbol X, and if the sets $\delta(q, a, X)$ and $\delta(q, \varepsilon, X)$ are both empty, then the PDA can make no transitions and therefore crashes.
- Empty stack: If the current instantaneous description is (q, x, ε) for some $x \neq \varepsilon$, then the PDA can make no transitions and therefore crashes.
  - It is supposed to remove the top stack symbol as it makes a transition, but there is no top stack symbol!
  - Note that, if the machine arrives at an instantaneous description $(q, \varepsilon, \varepsilon)$, then the machine will
    - accept if q is an accept state, and
    - reject if q is not an accept state.

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 $K_1 \underset{P}{\vdash} K_2 \underset{P}{\vdash} \cdots \underset{P}{\vdash} K_n$.

A valid computation remains valid if we:

1. Add string $w \in \Sigma^*$ to the end of the input for all K_i.
2. Add string $\gamma \in \Gamma^*$ to the end of the stack for all K_i.
3. Remove an unused suffix from the input 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 strings $w \in \Sigma^*$ 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 uses 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 a common suffix from the input, the computation remains 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, never add to the input.
- As such, these transitions are also valid to show that $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$.

How does a PDA accept?

- Like a DFA: accept if, at end of the input, the machine is in an accept state.
- This is called acceptance by final state.
- Later: we will accept if the stack empties at the same moment that the last input symbol has been processed (called acceptance by empty stack).

To formalize:

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

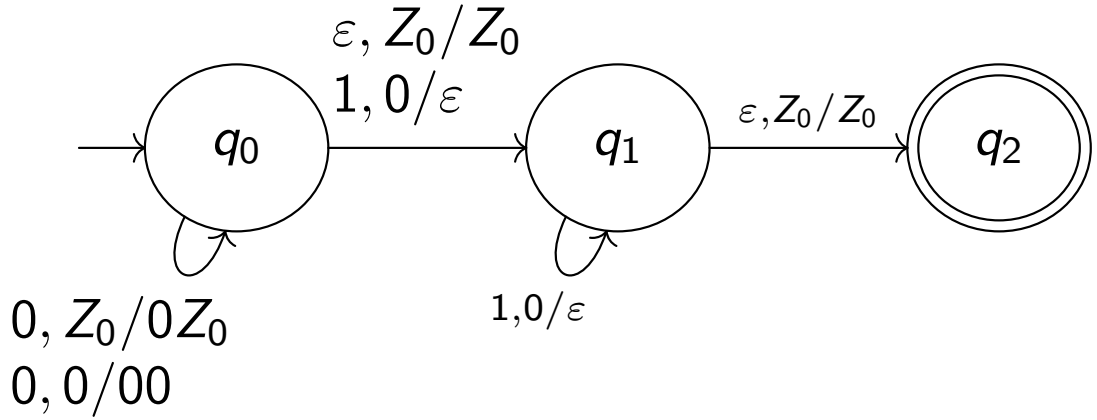
The language of a PDA P is

$$L(P) = \{w \in \Sigma^* \mid P \text{ accepts } w\}.$$

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

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

- (Recall that this language is not regular.)
- PDAs can be hard to draw!
- This one has three states:
  - q_0 for pushing 0s onto the stack, while reading them from the input word,
  - q_1 for popping 0s off of the stack, while reading corresponding 1s from the input word, and
  - q_2 for accepting at end of word.



Why does this work?

For words in L:

- Start with stack Z_0
- Add 0 symbols from w to the stack.
- ... until we reach the first 1 symbol. Then, we absorb 0s from the stack and corresponding 1s the input.
- ... until we are done, and then we follow the ε transition to q_2 and accept.

For example, on the input 0011, a computation witnessing acceptance 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 $(q_0, 0^i w, Z_0) \overset{*}{\vdash} (q_0, w, 0^i Z_0)$.
- Now consider what occurs upon processing $0^i 1^k$.
- If $k < i$, then all threads crash after reaching q_1. So from now on, we will consider $k \geq i$.
- The only valid computation after reading in $0^i 1^k$ is $(q_0, 0^i 1^k w, Z_0) \overset{*}{\vdash} (q_1, w, 0^{i-k} Z_0)$, 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 $0^i 1^i$, there are no more symbols to read, so we need not fear accepting $0^i 1^k$ 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 can we build a PDA for this?
Something like:

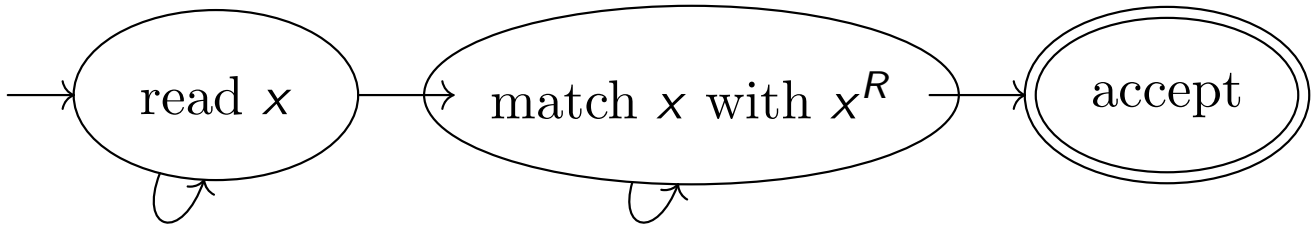
- Push x on the stack, until we get to !
- Then read in the !
- Then pop letters off the stack. If all stack and input letters match, then the word is in the language.

Reject when:

- There is not exactly one ! symbol
- The string after the ! is not the same length as what is before.
- The two substrings do not match each other.

An idea of how to build it

Like this:



- In the reading state, add the letters read from the input to the stack.
- In the matching state, compare input letters against letters from stack.
- If we get to the stack end and end of input string at the same time, then accept.

Transitions in the PDA

In reading state:

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

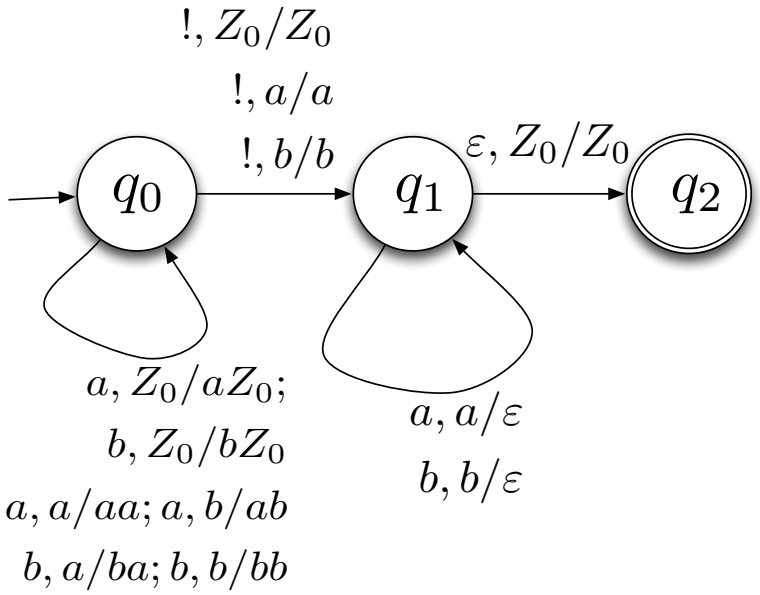
In matching state:

- If we read in an a from input, and the top of stack is a, keep going. Do not 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

$$L(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (p, \varepsilon, \alpha) \text{ for accept state } p, \text{ stack string } \alpha\}$$

New model of acceptance:

- The language of the machine is

$$N(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (p, \varepsilon, \varepsilon) \text{ for any state } p\}.$$

This is called acceptance by empty stack, for obvious reasons.
Notation: $N(P)$.
A PDA accepting by empty stack does not have accept states, so is a 6-tuple: $(Q, \Sigma, \Gamma, \delta, q_0, Z_0)$.

Key fact about acceptance by empty stack

Recall the two notions of acceptance for a PDA.

- by final state: $L(P_F)$ = every word x such that at least one thread of the machine P_F is in an accept state at the end of reading x.
- by empty stack: $N(P_N)$ = every word x such that at least one thread of the machine P_N has empty stack at the end of reading x.

Theorem:

1. Suppose that P_F is a PDA that accepts by final state. Then there exists a PDA, P_N, which accepts $L(P_F)$ by empty stack.
2. Suppose that P_N is a PDA that accepts by empty stack. Then there exists a PDA, P_F, which accepts $N(P_N)$ by final state.

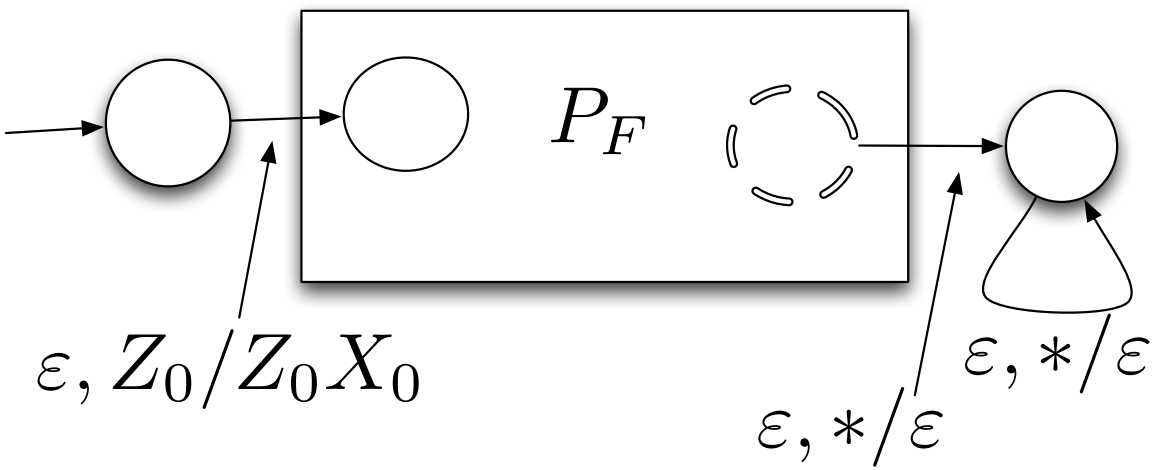
Proving the theorem

We will use acceptance by empty stack to show that PDAs accept exactly the class of context-free languages, which is why we care about this Theorem.
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 does not empty its stack at the end of the input word at any other time.

The construction

- To make P_N accept whenever P_F accepts:
  - From every accept state in P_F, take an ε-transition to a "drain" state that empties the stack.
- To prevent P_N from accepting whenever P_F does not accept:
  - If there are input letters remaining when the above ε-transition is taken, then the thread crashes.
  - Before doing any computation, we put a special character X_0 at the bottom of the stack, below even the Z_0 character.
  - It is crucial that X_0 **not** be in the stack alphabet for P_F.
  - We only remove the Z_0 and X_0 characters when we are transitioning to or already in our new "drain" state.
  - All computations in P_F happen as before, so we can only remove Z_0 and X_0 from the stack when P_F would have already accepted.



From empty stack to final state

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

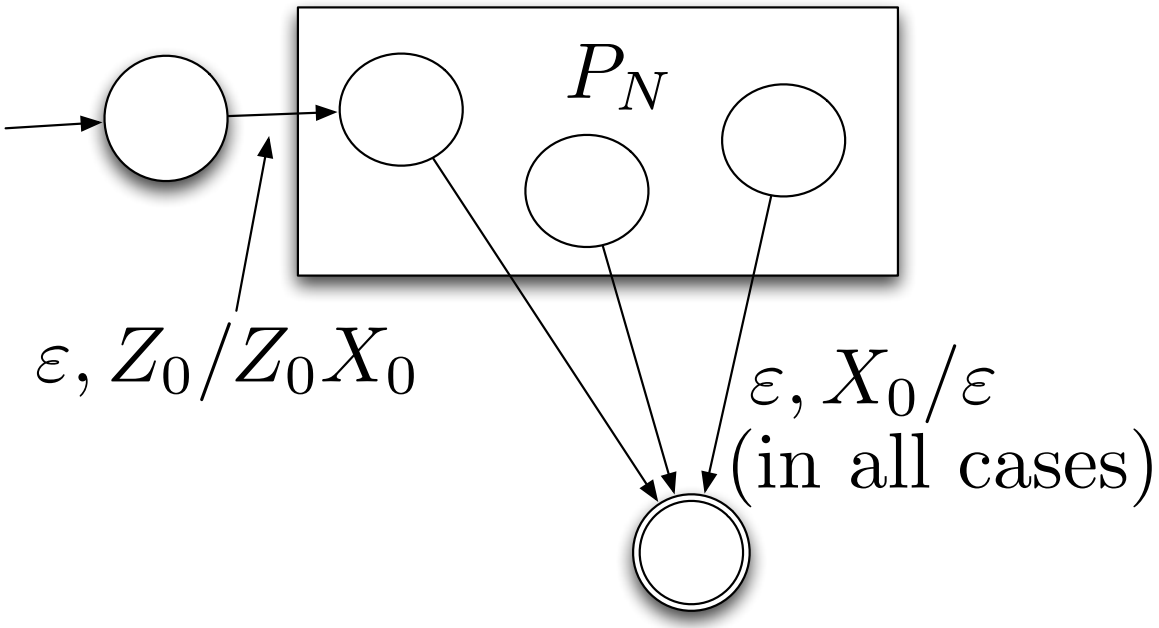
- Again, it is crucial that X_0 **not** be in the stack alphabet for P_N.
- Here, the X_0 character alerts us that we have removed all of the stack characters inside of P_N.
- Without this special symbol, we would crash trying to detect empty stack.
- When we see this special character, we take an ε-transition 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 outgoing transitions from the new accept state: if we have not processed the entire input word, the machine crashes instead of accepting.

The actual construction

New pushdown automaton P_F:



Note: We have not rigourously proved either construction. Both proofs are in the text, not especially enlightening.

Proving that PDAs and CFGs both accept CFLs

Our reason for introducing PDAs is that they accept exactly the class of context-free languages (analogous to how FAs accept exactly the class of regular 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: We will prove using acceptance by empty stack in both cases. We started with acceptance by final state since it makes PDAs "feel" like ε-NFAs.
We will 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 a given word w, then the PDA accepts w.

- If the derivation produces terminals, they should match letters from w.
- If the derivation produces non-terminals, keep the derivation going.
- Stack memory keeps the portion of the string of unresolved variables and terminals from the derivation, which has yet to be matched with input symbols.

More specific description

Example: A grammar for palindromes with no !: $S \to aSa|bSb|a|b|\varepsilon$
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.
- The stack alphabet is Σ unioned with the list of variables in G.
- 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:

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

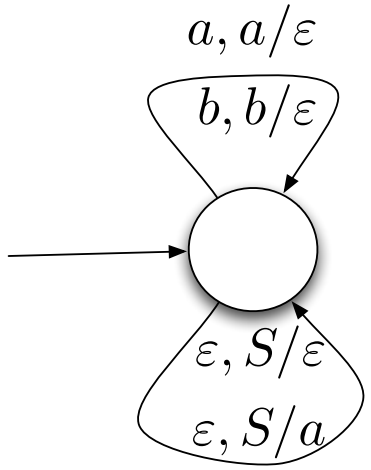
Consider the word $abbba$. We have the leftmost derivation:
$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abbba$. Keep the string on the stack as we derive it, and match the first characters in the input string. Something like:

$$\begin{aligned}
(q, abbba, S) &\vdash (q, abbba, aSa) \\
&\vdash (q, bbba, Sa) \\
&\vdash (q, bbba, bSba) \\
&\vdash (q, bba, Sba) \\
&\vdash (q, bba, bba) \\
&\vdash (q, ba, ba) \\
&\vdash (q, a, a) \\
&\vdash (q, \varepsilon, \varepsilon)
\end{aligned}$$

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).
- Notice, we only have 1 state!



Does this work?

Yes, as we had hoped.

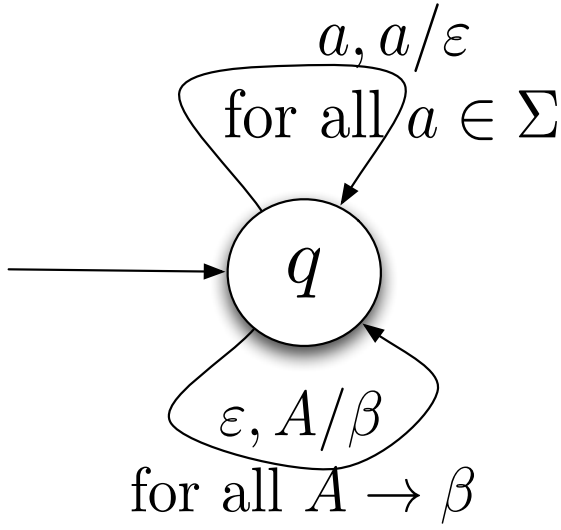
- Start with S on the stack
- Try all possible derivations for S
- Match characters in x until we get to another variable in the derivation
- Try derivations for that variable
- If we reach end of w and no more variables to derive, we accept.

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

General structure

Start with the grammar $G = (V, T, P, S)$. 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 $w \in L(G)$ be arbitrary.
- Then there exists some leftmost derivation for w in G : $S = \gamma_1 \underset{lm}{\Rightarrow} \gamma_2 \underset{lm}{\Rightarrow} \gamma_3 \underset{lm}{\Rightarrow} \cdots \underset{lm}{\Rightarrow} \gamma_n = w$.
- For all of the γ_i except γ_n, there exists a leftmost variable.
  - Decompose $\gamma_i = x_i \alpha_i$, where α_i begins with the first variable in γ_i, A_i. (The exception is $\alpha_n = \varepsilon$.)
  - Since $\gamma_i \overset{*}{\underset{lm}{\Rightarrow}} w$, and since x_i contains only terminals, we know that x_i is a prefix of w.
  - Write $w = x_i y_i$, for some string y_i.
- In the PDA, because the transitions are defined from the productions of G, we have $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$ for all i. (We will prove this rigourously, on the next slide.)
- In particular, taking $i = n$ gives $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$, and so $w \in N(P)$, because P accepts w by empty stack.

Finishing the proof in one direction

Now we want to prove:

- For all $1 \leq i \leq n$, we have $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$.

Proof by induction on i:

- Base case ($i = 1$): $x_1 = \varepsilon, y_1 = w, \alpha_1 = S$, so we have that $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$ trivially.

Finishing the proof in one direction

- Inductive case ($1 < i \leq n$): The inductive hypothesis is that $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_j, \alpha_j)$, for all $j < i$.
  - By the induction hypothesis $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_{i-1}, \alpha_{i-1})$, where $w = x_{i-1}y_{i-1}$, $\gamma_{i-1} = x_{i-1}\alpha_{i-1}$ and $\alpha_{i-1} = A_{i-1}\eta_{i-1}$ for some η_{i-1}.
  - Everything constructed here comes from a leftmost derivation of w in G, so there exists a production $A_{i-1} \to \beta_{i-1}$ in G, for some β_{i-1}.
  - The production $A_{i-1} \to \beta_{i-1}$ gives that $\delta(q, \varepsilon, A_{i-1})$ contains (q, β_{i-1}) so $(q, y_{i-1}, \alpha_{i-1}) \underset{P}{\vdash} (q, y_{i-1}, \beta_{i-1}\eta_{i-1})$.
  - As everything constructed here comes from a derivation of w in G, y_{i-1} and β_{i-1} must have a (possibly empty) prefix of matching terminals.
  - Match all of the terminals at the top of the stack against the terminals in the input (using $\delta(q, a, a) = \{q, \varepsilon\}$ for all $a \in T$).
  - Consume all matching terminals to obtain $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$.

The proof in the other direction

We have proved $L(G) \subseteq N(P)$. Now we prove $N(P) \subseteq L(G)$. Suppose that $w \in N(P)$. To show that $w \in L(G)$, we will 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:
  - By definition $w \in N(P)$ means that $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$.
  - Then applying the above implication, with $A = S$ gives us that $S \overset{*}{\underset{G}{\Rightarrow}} w$, in other words, $w \in L(G)$.

The proof is by induction on n, the length of the chosen computation that takes (q, w, A) to $(q, \varepsilon, \varepsilon)$:

- Base case ($n = 1$; $n = 0$ is impossible since $A \neq \varepsilon$):
  - In the construction of the *PDA* P, the only transitions defined for a non-terminal A at the top of the stack are ε-transitions corresponding to productions for A in G.
  - So our computation must take one of these ε-transitions as its first (and hence its only) step.
  - As this ε-transition takes w to ε, therefore $w = \varepsilon$.
  - Also $(q, \varepsilon) \in \delta(q, \varepsilon, A)$, so by construction there is a rule $A \to \varepsilon$ in G.
  - Therefore $A \overset{*}{\underset{G}{\Rightarrow}} w$ as desired.

The inductive case

Inductive case ($n > 0$): The inductive hypothesis is that for all variables A, if $(q, w, A) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$ in fewer than n steps, then $A \overset{*}{\underset{G}{\Rightarrow}} w$. Now consider an n-step computation $(q, w, A) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$.

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

The inductive case

- Rewriting the last statement, we have $(q, w_1 w', Y_1 Y_2 \cdots Y_K) \overset{*}{\underset{P}{\vdash}} (q, w', Y_2 \cdots Y_k)$.
- Because we can delete the unread suffix w' of the input word without affecting the validity of the computation, we therefore have that $(q, w_1, Y_1 Y_2 \cdots Y_k) \underset{P}{\vdash} (q, \varepsilon, Y_2 \cdots Y_K)$.
- Moreover, since in the computation, we never touch the symbols of $Y_2 \cdots Y_k$ on the stack, we also have that $(q, w_1, Y_1) \underset{P}{\vdash} (q, \varepsilon, \varepsilon)$.
- If Y_1 is a terminal, then by the shape of the PDA, $(q, w_1, Y_1) \underset{P}{\vdash} (q, \varepsilon, \varepsilon)$ implies that $w_1 = Y_1$ is also a terminal, so that $Y_1 \overset{*}{\underset{G}{\Rightarrow}} w_1$ holds trivially.
- Otherwise Y_1 is a variable, and by construction, this computation witnessing $(q, w_1, Y_1) \underset{P}{\vdash} (q, \varepsilon, \varepsilon)$ takes fewer than n steps, therefore $Y_1 \overset{*}{\underset{G}{\Rightarrow}} w_1$, by our inductive hypothesis.

End of the inductive case

- To summarize, whether Y_1 is a terminal or a variable, we have $Y_1 \overset{*}{\underset{G}{\Rightarrow}} w_1$.
- We then apply this argument to each subsequent Y_i, obtaining a derivation $Y_i \overset{*}{\underset{G}{\Rightarrow}} w_i$ for all i, with $w = w_1 w_2 \cdots w_k$.
- Finally, we have a derivation for w, by combining these together: $A \Rightarrow Y_1 \cdots Y_k \overset{*}{\underset{G}{\Rightarrow}} w_1 Y_2 \cdots Y_k \overset{*}{\underset{G}{\Rightarrow}} w_1 w_2 Y_3 \cdots Y_k \overset{*}{\underset{G}{\Rightarrow}} w_1 \cdots w_k = w$
- This shows that $A \overset{*}{\underset{G}{\Rightarrow}} w$, as desired.

Next, from a PDA to a CFG

- We have shown the easier case: for a CFG G with language $L(G) = L$, we can exhibit a PDA P such that $N(P) = L$.
- Now, given a PDA P (which accepts by empty stack), we must exhibit a grammar G such that $N(P) = L(G)$.
- This is harder.
  - Suppose that, in P,

    $$(q, w, X) \stackrel{*}{\vdash} (p, \varepsilon, \varepsilon),$$

    for some $X \in \Gamma$.
  - How does that happen?

How do we consume one symbol from the stack?

- Suppose the first step in the computation is $(q, az, X) \vdash (r, z, Y_1 Y_2 \cdots Y_k)$.
- We may be following a transition that does or does not consume the first symbol of w, so $a \in \Sigma$ or $a = \varepsilon$.
- The process begins by removing X from the stack, replacing it with some string $Y_1 \cdots Y_k$, 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 $Y_2 \cdots Y_k$; this process may consume some input letters,from w, or it might not.
- Eventually, we wind up in state p, in the instantaneous description $(p, \varepsilon, \varepsilon)$, having read all of the letters of w and all of the symbols in $Y_1 \cdots Y_k$.

Making a grammar from this

To construct a grammar which generates the same language that the PDA accepts, we need to construct our productions to mimic the action of the transitions in the PDA.

- We will have a non-terminal $[qXp]$ for every stack symbol X and every pair of states q and p.
- Each of these $[qXp]$ triplets is a variable in the new grammar we are building.
- A string generated by the non-terminal $[qXp]$ in G corresponds with a (possibly partial) computation in the PDA. It will capture the input letters consumed so far and the current stack contents.
- We want: $[qXp] \overset{*}{\underset{G}{\Rightarrow}} w$ if and only if $(q, w, X) \overset{*}{\underset{P}{\vdash}} (p, \varepsilon, \varepsilon)$.
- Now we need to construct the productions of the grammar to mimic the transitions in the given PDA.

Making a grammar from this

- Examine each rule of the transition function, one at a time.
- Suppose that one element of $\delta(q, a, X)$ is $(r, Y_1 Y_2 \cdots Y_k)$.
  - Note: a can be ε, and so can $Y_1 \cdots Y_k$.
  - If the output pair is (r, ε), so that $k = 0$, then add a production $[qXr] \to a$.
    - Read a, pop X, move to state r.
  - Otherwise, if $k \geq 1$, then add productions $[qXr_k] \to a[rY_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k]$ to the grammar, for all choices of $r_1, \ldots, r_k$ from our state set Q.
    - Read a, pop X, push $Y_1 \cdots Y_k$ onto the stack, allowing for any sequence $r_1, \ldots, r_k$ of states to be used to pop $Y_1 \cdots Y_k$ in the end, move to state r.
- The language of the PDA is the union of the language corresponding to emptying the stack in any possible state, so for every state p we add a rule: $S \to [q_0 Z_0 p]$.
- Then by construction $(q_0, w, Z_0) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$ if and only if $S \Rightarrow [q_0 Z_0 p] \overset{*}{\Rightarrow} w$, as we had wanted.

How does this work?

The basic idea, again: Each PDA transition

- either consumes a symbol from the input, or does not,
- changes state and
- takes one symbol off the stack, and then puts some new symbols on the stack, or does not.

To mimic this in one step of a leftmost derivation in the grammar which we have just constructed:

- Look at the right hand side of a particular production.
- The terminal (or ε) at the beginning of the right hand side is exactly what we expect to read from the input, according to the transition function in the given PDA.
- Then the string of nonterminals, each of the form $[rY_1r_1]$ or $[r_{i-1}Y_ir_i]$ indicate, in the Y_i characters, the symbols that get pushed onto the stack.
- The first state in the first non-terminal (namely r) is the state that the PDA goes into.
- The following states correspond to the states the automaton will be in when we have shrunk the stack.

The proof

Actually, we are not going to do it.

- The basic structure is not especially interesting; it is in Section 6.3.2 of the text.
- 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 is 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 transition is determined.

- Can have ε-transitions, but each 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, then 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 $|\delta(q_0, \varepsilon, X)| = 0$.
- And $|\delta(q_0, a, X)| \leq 1$ always.

Why do we care about DPDAs?

Some CFLs are not accepted by DPDAs.

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

One important difference versus DFAs:

- It is possible that no transition is available.
- The machine can still crash.

An example of a language accepted by a DPDA

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

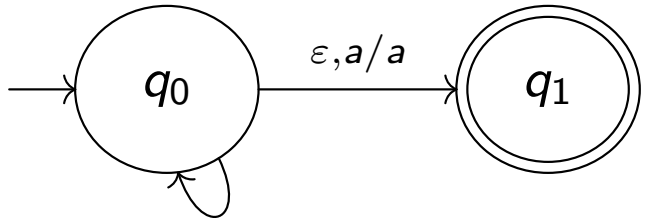
- (Words with more a's than b's.)

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 is still an a on the top of the stack at the end.

A nondeterministic PDA for L

This gives a 2-state PDA like this:



$a, Z_0/aZ_0$
$b, Z_0/bZ_0$
$a, b/\varepsilon$
$b, a/\varepsilon$
$a, a/aa$
$b, b/bb$

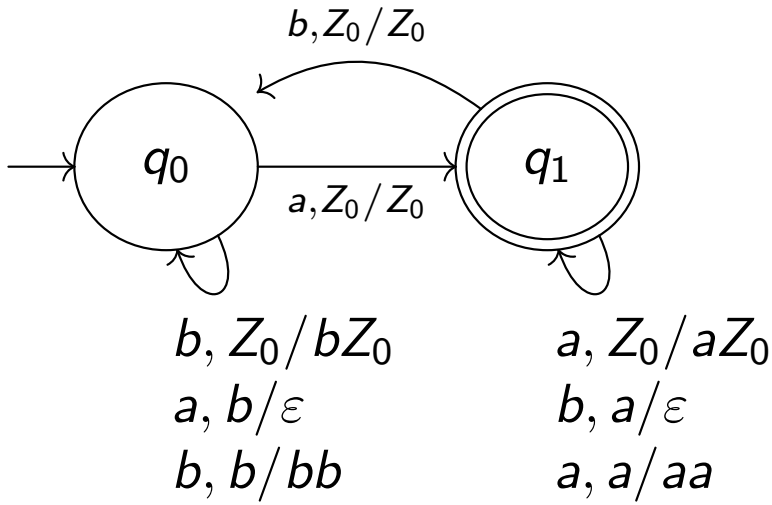
- First two transitions in state q_0: prefix is balanced
- Second two transitions in state q_0: reducing imbalance
- Last two transitions in state q_0: adding to imbalance
- Nondeterministic: ε-transition to state q_1.

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 do not. 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 is always at most one choice for what to do next.

But the stack still remembers the imbalance: we have just divided the transitions from before between the two states.
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 the text):
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 will 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.

Sketch of direct proof that DPDAs do not 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 is not accepted by any 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 is the end of the string, and be ready to read in more letters, in case it is just the first half of the string.

Determinism makes this impossible

In a DPDA, that is 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.
- We cannot 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.

- The 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 symbols, $(p, Y_1 Y_2 \cdots Y_n)$ turned into a pile of distinct new rules in the grammar.
- However, since the PDA is deterministic, we can see that only one derivation in the grammar will actually lead to the word that is accepted by the DPDA.

You can modify the proof to work for acceptance by final state, too. (That is Theorem 6.21 in the text.)

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 (we just saw that $L = \{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$ is a DCFL - it is an exercise to prove that L is not regular).
- DCFLs are all not inherently ambiguous (Theorem on previous slide).
- There are languages that are not accepted by DPDAs that are not inherently ambiguous (e.g. palindromes).
- Some CFLs that are not inherently ambiguous are not DCFLs (e.g. palindromes).
- All CFLs are accepted by PDAs, and the languages accepted by PDAs are exactly the CFLs.

What about languages that are not context free?

End of module 6

We have 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.