Computational Models - Lecture 6
Handout Mode

Iftach Haitner and Ronitt Rubinfeld.
Tel Aviv University.

April 4/6, 2016

---

Based on frames by Benny Chor, Tel Aviv University, modifying frames by Maurice Herlihy, Brown University. Also with modifications of Yishay Mansour.
Outline

- Push Down Automata (PDA)
- Closure properties for CFL and testing properties.
- Equivalence of CFGs and PDAs

- Sipser’s book, 2.1, 2.2 & 2.3
Part I

Push-Down Automata: Review
Diagram Notation

When drawing the automata diagram, we use the following notation

- Transition from state $q$ to state $q'$ labelled by $a, b \rightarrow c$ means $(q', c) \in \delta(q, a, b)$, and informally means the automata
  - read $a$ from input
  - pop $b$ from stack
  - push $c$ onto stack
Diagram Notation

When drawing the automata diagram, we use the following notation

- Transition from state $q$ to state $q'$ labelled by $a, b \rightarrow c$ means $(q', c) \in \delta(q, a, b)$,
and informally means the automata
  - read $a$ from input
  - pop $b$ from stack
  - push $c$ onto stack

- Meaning of $\varepsilon$ transitions ((informally)):
  - $a = \varepsilon$: don’t read input
  - $b = \varepsilon$: don’t pop any symbol
  - $c = \varepsilon$: don’t push any symbol
How to define $\hat{\delta}(q, w, s) \text{ contains } (q', s')$?

Given (start) state $q$, substring $w$ of the input, and $s, s'$ descriptions of strings on a stack:
There is a legal way to get from state $q$ with stack contents $s$ to state $q'$ with stack contents $s'$ by reading from $w$ at each step.
The following is with respect to $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$.

**Definition 1 ($\delta^*$)**

For $w \in \Sigma^*$ let $\hat{\delta}(q, w, s)$ be all pairs $(q', s') \in Q \times \Gamma^*$ for which exist $w'_1, \ldots, w'_m \in \Sigma_{\varepsilon}$, states $r_1, \ldots, r_m \in Q$ and strings $s_0, s_1, \ldots s_m \in \Gamma^*$ s.t.:

1. $w = w'_1, \ldots, w'_m$, $r_0 = q$, $r_m = q'$, $s_0 = s$ and $s_m = s'$
2. For every $i \in \{0, \ldots, m-1\}$ exist $a, b \in \Gamma_{\varepsilon}$ and $t \in \Gamma^*$ s.t.:
   2.1 $(r_{i+1}, b) \in \delta(r_i, w'_i+1, a)$
   2.2 $s_i = at$ and $s_{i+1} = bt$

Namely, $(q', s') \in \hat{\delta}(q_0, w, \varepsilon)$ if after reading $w$ (possibly with in-between $\varepsilon$ moves), $M$ can find itself in state $q'$ and stack value $s'$. 
Model of Computation

The following is with respect to \( M = (Q, \Sigma, \Gamma, \delta, q_0, F) \).

**Definition 1 \((\delta^*)\)**

For \( w \in \Sigma^* \) let \( \hat{\delta}(q, w, s) \) be all pairs \((q', s') \in Q \times \Gamma^*\) for which exist \( w'_1, \ldots, w'_m \in \Sigma_\varepsilon \), states \( r_1, \ldots, r_m \in Q \) and strings \( s_0, s_1, \ldots s_m \in \Gamma^*\) s.t.:

1. \( w = w'_1, \ldots, w'_m \), \( r_0 = q \), \( r_m = q' \), \( s_0 = s \) and \( s_m = s' \)
2. For every \( i \in \{0, \ldots, m - 1\} \) exist \( a, b \in \Gamma_\varepsilon \) and \( t \in \Gamma^* \) s.t.:
   2.1 \( (r_{i+1}, b) \in \delta(r_i, w'_{i+1}, a) \)
   2.2 \( s_i = at \) and \( s_{i+1} = bt \)

Namely, \((q', s') \in \hat{\delta}(q_0, w, \varepsilon)\) if after reading \( w \) (possibly with in-between \( \varepsilon \) moves), \( M \) can find itself in state \( q' \) and stack value \( s' \).

\( M \) accepts \( w \in \Sigma^* \) if \( \exists q' \in F \) such that \((q', t) \in \hat{\delta}(q_0, w, \varepsilon)\) for some \( t \).
Knowing when stack is empty

It is convenient to be able to know when the stack is empty, but there is no built-in mechanism to do that.

1. Start by pushing $\text{STOP}$ onto stack.
2. When you see it again, stack is empty.
Knowing when stack is empty

It is convenient to be able to know when the stack is empty, but there is no built-in mechanism to do that.

Solution

1. Start by pushing $\$ \text{onto stack.}$

2. When you see it again, stack is empty.
Example 3 – Palindrome

A palindrome is a string $w$ satisfying $w = w^R$.

- “Madam I’m Adam”
- “Dennis and Edna sinned”
- “Red rum, sir, is murder”
- “Able was I ere I saw Elba”
- “In girum imus nocte et consumimur igni” (Latin: "we go into the circle by night, we are consumed by fire").
- “νιψον ανόμηματα μη μοναν οψιν”

- Palindromes also appear in nature. For example as DNA restriction sites – short genomic strings over $\{A, C, T, G\}$, being cut by (naturally occurring) restriction enzymes.
Example 3 – Palindrome

A palindrome is a string $w$ satisfying $w = w^R$.

- “Madam I’m Adam”
- “Dennis and Edna sinned”
- “Red rum, sir, is murder”
- “Able was I ere I saw Elba”
- “In girum imus nocte et consumimur igni” (Latin: “we go into the circle by night, we are consumed by fire”).
- “νιψον ανομηματα μη μοναν ϋψων”

Palindromes also appear in nature. For example as DNA restriction sites – short genomic strings over $\{A, C, T, G\}$, being cut by (naturally occurring) restriction enzymes.

What the difference from $\{ww^R\}$?
A PDA for Palindromes

Algorithm 2

1. Start pushing $x \in \Sigma^*$ into stack.
2. At some point, guess that the midpoint of $x$ has reached.
3. Pops and compares to input, letter by letter.
4. Accept if end of input occurs together with emptying of stack.

▶ This PDA accepts palindromes of even length over the alphabet (all lengths is an easy modification).
▶ Again, non-determinism (at which point to make the switch) seems necessary.
A PDA for Palindromes

Algorithm 2

Input: $x \in \Sigma^*$

1. Start pushing $x$ into stack.
2. At some point, guess that the mid point of $x$ has reached.
3. Pops and compares to input, letter by letter.
4. Accept If end of input occurs together with emptying of stack.

This PDA accepts palindromes of even length over the alphabet (all lengths is an easy modification). Again, non-determinism (at which point to make the switch) seems necessary.
Algorithm 2

Input: \( x \in \Sigma^* \)

1. Start pushing \( x \) into stack.
2. At some point, guess that the mid point of \( x \) has reached.
3. Pops and compares to input, letter by letter.
4. **Accept** If end of input occurs **together** with emptying of stack.

- This PDA accepts palindromes of **even length** over the alphabet (all lengths is an easy modification).
Algorithm 2

Input: $x \in \Sigma^*$

1. Start pushing $x$ into stack.
2. At some point, guess that the mid point of $x$ has reached.
3. Pops and compares to input, letter by letter.
4. Accept If end of input occurs together with emptying of stack.

- This PDA accepts palindromes of even length over the alphabet (all lengths is an easy modification).
- Again, non-determinism (at which point to make the switch) seems necessary.
PDA Languages

The Push-Down Automata Languages, $\mathcal{L}_{\text{PDA}}$, is the set of all languages that can be described by some PDA:

\[ \mathcal{L}_{\text{PDA}} = \{ \mathcal{L}(M) : M \text{ is a PDA} \} \]
PDA Languages

The Push-Down Automata Languages, $\mathcal{L}_{PDA}$, is the set of all languages that can be described by some PDA:

$\mathcal{L}_{PDA} = \{ \mathcal{L}(M) : M \text{ is a PDA} \}$

It is immediate that $\mathcal{L}_{PDA} \supseteq \mathcal{L}_{DFA}$: every DFA is just a PDA that ignores the stack.
PDA Languages

The Push-Down Automata Languages, $\mathcal{L}_{PDA}$, is the set of all languages that can be described by some PDA:

$\mathcal{L}_{PDA} = \{\mathcal{L}(M) : M \text{ is a PDA}\}$

It is immediate that $\mathcal{L}_{PDA} \supseteq \mathcal{L}_{DFA}$: every DFA is just a PDA that ignores the stack.

$\mathcal{L}_{CFG} \subseteq \mathcal{L}_{PDA}$?
PDA Languages

The Push-Down Automata Languages, \( L_{PDA} \), is the set of all languages that can be described by some PDA:

- \( L_{PDA} = \{L(M) : M \text{ is a PDA}\} \)

It is immediate that \( L_{PDA} \supseteq L_{DFA} \): every DFA is just a PDA that ignores the stack.

- \( L_{CFG} \subseteq L_{PDA} \)?
- \( L_{PDA} \subseteq L_{CFG} \)?
PDA Languages

The Push-Down Automata Languages, \( \mathcal{L}_{PDA} \), is the set of all languages that can be described by some PDA:

\[ \mathcal{L}_{PDA} = \{ \mathcal{L}(M) : M \text{ is a PDA} \} \]

It is immediate that \( \mathcal{L}_{PDA} \supset \mathcal{L}_{DFA} \): every DFA is just a PDA that ignores the stack.

\[ \mathcal{L}_{CFG} \subseteq \mathcal{L}_{PDA} \] ?

\[ \mathcal{L}_{PDA} \subseteq \mathcal{L}_{CFG} \] ?

\[ \mathcal{L}_{PDA} = \mathcal{L}_{CFG} \] !!!
Part II

Closure Properties
Simple Closure Properties of Context-Free Languages

- CFLs are closed under
  - Union: $S \rightarrow S_1 \mid S_2$
  - Concatenation: $S \rightarrow S_1S_2$
  - Star: $S_{\text{new}} \rightarrow \varepsilon \mid S_{\text{old}} \mid S_{\text{old}}S_{\text{new}}$

What about complement and intersection?
Simple Closure Properties of Context-Free Languages

- CFL’s are closed under

- Union:

- Concatenation:

- Star:
Simple Closure Properties of Context-Free Languages

- CFL’s are closed under
  - Union:

\[ S \to S_1 | S_2 \]
Simple Closure Properties of Context-Free Languages

- CFL’s are closed under
  - Union:

  $$S \rightarrow S_1 \mid S_2$$

- Concatenation:
  $$S \rightarrow S_1 S_2$$

- Star:
  $$S_{\text{new}} \rightarrow \varepsilon \mid S_{\text{old}} \mid S_{\text{old}} S_{\text{new}}$$
Simple Closure Properties of Context-Free Languages

- CFL’s are closed under
  - Union: $S \rightarrow S_1 \mid S_2$
Simple Closure Properties of Context-Free Languages

- CFL’s are closed under
  - Union: $S \rightarrow S_1 \mid S_2$
  - Concatenation:
Simple Closure Properties of Context-Free Languages

- CFL's are closed under
  - Union: $S \rightarrow S_1 \mid S_2$
  - Concatenation:
Simple Closure Properties of Context-Free Languages

- CFL’s are closed under
  - Union: $S \rightarrow S_1 \mid S_2$
  - Concatenation: $S \rightarrow S_1 S_2$
Simple Closure Properties of Context-Free Languages

- CFL’s are closed under
  - Union: $S \rightarrow S_1 \mid S_2$
  - Concatenation: $S \rightarrow S_1 S_2$
  - Star:
Simple Closure Properties of Context-Free Languages

- CFL’s are closed under
  - Union: $S \rightarrow S_1 \mid S_2$
  - Concatenation: $S \rightarrow S_1 S_2$
  - Star:
CFL’s are closed under
- Union: \( S \rightarrow S_1 | S_2 \)
- Concatenation: \( S \rightarrow S_1 S_2 \)
- Star: \( S_{\text{new}} \rightarrow \epsilon | S_{\text{old}} | S_{\text{old}} S_{\text{new}} \)
Simple Closure Properties of Context-Free Languages

- CFL’s are closed under
  - Union: $S \rightarrow S_1 \mid S_2$
  - Concatenation: $S \rightarrow S_1 S_2$
  - Star: $S_{new} \rightarrow \epsilon \mid S_{old} \mid S_{old} S_{new}$

- What about complement and intersection?
Intersection idea?

- Idea: Can’t we run two PDA’s in parallel, and accept iff both accept??
Intersection

\[ S_1 \rightarrow A_1 B_1 \]
\[ A_1 \rightarrow 0A_1 1|\varepsilon \]
\[ B_1 \rightarrow 2B_1|\varepsilon \]
\[ S_2 \rightarrow A_2 B_2 \]
\[ A_2 \rightarrow 0A_2 |\varepsilon \]
\[ B_2 \rightarrow 1B_2 2|\varepsilon \]

\[ L_1 = 0^n 1^n 2^* \]
\[ L_2 = 0^* 1^n 2^n \]
Intersection

\[
\begin{align*}
S_1 & \rightarrow A_1 B_1 & S_2 & \rightarrow A_2 B_2 \\
A_1 & \rightarrow 0A_1 1|\varepsilon & A_2 & \rightarrow 0A_2 |\varepsilon \\
B_1 & \rightarrow 2B_1 |\varepsilon & B_2 & \rightarrow 1B_2 2 |\varepsilon \\
\end{align*}
\]

\[
L_1 = 0^n 1^n 2^* \\
L_2 = 0^* 1^n 2^n
\]

\[
L_1 \cap L_2 = 0^n 1^n 2^n
\]
Intersection

\[ S_1 \rightarrow A_1B_1 \quad S_2 \rightarrow A_2B_2 \]
\[ A_1 \rightarrow 0A_11|\varepsilon \quad A_2 \rightarrow 0A_2|\varepsilon \]
\[ B_1 \rightarrow 2B_1|\varepsilon \quad B_2 \rightarrow 1B_22|\varepsilon \]

\[ L_1 = 0^n1^n2^* \quad L_2 = 0^*1^n2^n \]

- \[ L_1 \cap L_2 = 0^n1^n2^n \]
- \[ L_1 \text{ and } L_2 \text{ are CFLs (why?),} \]
Intersection

\[
\begin{align*}
S_1 & \rightarrow A_1B_1 & S_2 & \rightarrow A_2B_2 \\
A_1 & \rightarrow 0A_11|\varepsilon & A_2 & \rightarrow 0A_2|\varepsilon \\
B_1 & \rightarrow 2B_1|\varepsilon & B_2 & \rightarrow 1B_22|\varepsilon \\
\end{align*}
\]

\[L_1 = 0^n1^n2^n \quad L_2 = 0^*1^n2^n\]

- \(L_1 \cap L_2 = 0^n1^n2^n\)
- \(L_1\) and \(L_2\) are CFLs (why?),
- But \(L_1 \cap L_2\) is not a CFL.
Intersection

\[ S_1 \rightarrow A_1 B_1 \quad S_2 \rightarrow A_2 B_2 \]
\[ A_1 \rightarrow 0A_1 1|\varepsilon \quad A_2 \rightarrow 0A_2|\varepsilon \]
\[ B_1 \rightarrow 2B_1|\varepsilon \quad B_2 \rightarrow 1B_2 2|\varepsilon \]

\[ L_1 = 0^n 1^n 2^* \quad L_2 = 0^* 1^n 2^n \]

- \[ L_1 \cap L_2 = 0^n 1^n 2^n \]
- \[ L_1 \text{ and } L_2 \text{ are CFLs (why?)}, \]
- But \[ L_1 \cap L_2 \text{ is not a CFL.} \]
- So.. we can’t we run two PDA’s in parallel, and accept iff both accept.
Intersection

\[ S_1 \rightarrow A_1 B_1 \quad S_2 \rightarrow A_2 B_2 \]
\[ A_1 \rightarrow 0A_1 1 | \varepsilon \quad A_2 \rightarrow 0A_2 | \varepsilon \]
\[ B_1 \rightarrow 2B_1 | \varepsilon \quad B_2 \rightarrow 1B_2 2 | \varepsilon \]

\[ \mathcal{L}_1 = 0^n 1^n 2^n \quad \mathcal{L}_2 = 0^* 1^n 2^n \]

- \( \mathcal{L}_1 \cap \mathcal{L}_2 = 0^n 1^n 2^n \)
- \( \mathcal{L}_1 \) and \( \mathcal{L}_2 \) are CFLs (why?),
- But \( \mathcal{L}_1 \cap \mathcal{L}_2 \) is not a CFL.
- So.. we can’t we run two PDA’s in parallel, and accept iff both accept.
- What about intersection of a CFL with a regular language?
When CFL Intersects Regular Language

- Are the context free languages closed under intersection with a regular language?

YES!

Run PDA $L_1$ and DFA $L_2$ "in parallel" (just like the intersection of two regular languages).

Formal details omitted (but you should be able to figure them out).
When CFL Intersects Regular Language

- Are the context free languages closed under intersection with a regular language?
- That is, if $L_1$ is context free languages, and $L_2$ is regular, must $L_1 \cap L_2$ be context free languages?

YES!

Run PDA $L_1$ and DFA $L_2$ “in parallel” (just like the intersection of two regular languages).

Formal details omitted (but you should be able to figure them out).
Are the context free languages closed under intersection with a regular language?

That is, if \( L_1 \) is context free languages, and \( L_2 \) is regular, must \( L_1 \cap L_2 \) be context free languages?

YES!
Are the context free languages closed under intersection with a regular language?

That is, if $L_1$ is context free languages, and $L_2$ is regular, must $L_1 \cap L_2$ be context free languages?

YES!

Run PDA $L_1$ and DFA $L_2$ “in parallel” (just like the intersection of two regular languages).
When CFL Intersects Regular Language

- Are the context free languages closed under intersection with a regular language?
- That is, if $L_1$ is context free languages, and $L_2$ is regular, must $L_1 \cap L_2$ be context free languages?
- YES!
  - Run PDA $L_1$ and DFA $L_2$ “in parallel” (just like the intersection of two regular languages).
  - Formal details omitted (but you should be able to figure them out).
An Application:

▶ Is $L = \{w \in \{0, 1, 2\}^* : \#_0(w) = \#_1(w) = \#_2(w)\}$ context free?
An Application:

- Is $\mathcal{L} = \{ w \in \{0, 1, 2\}^* : \#_0(w) = \#_1(w) = \#_2(w) \}$ context free?

- $0^*1^*2^*$ is regular.

- Context free languages intersected with regular languages are context free.

- $L \cap 0^*1^*2^* = \{ 0^n1^n2^n : n \geq 0 \}$ is not context free.

- So $L$ is not a context free language.

- This could also be established using pumping lemma, but proof above is more elegant.
An Application:

- Is \( \mathcal{L} = \{ w \in \{0, 1, 2\}^* : \#_0(w) = \#_1(w) = \#_2(w) \} \) context free?

  - \( 0^*1^*2^* \) is regular.

Context free languages intersected with a regular language are context free.

\[ \mathcal{L} \cap 0^*1^*2^* = \{ 0^n1^n2^n : n \geq 0 \} \] is not context free.

So \( \mathcal{L} \) is not a context free language.

This could also be established using the pumping lemma, but the proof above is more elegant.
An Application:

- Is $\mathcal{L} = \{ w \in \{0, 1, 2\}^* : \#_0(w) = \#_1(w) = \#_2(w) \}$ context free?

  - $0^*1^*2^*$ is regular.
  - Context free languages intersected with a regular languages are context free.
An Application:

- Is $\mathcal{L} = \{w \in \{0, 1, 2\}^* : \#_0(w) = \#_1(w) = \#_2(w)\}$ context free?

- $0^*1^*2^*$ is regular.
- Context free languages intersected with a regular languages are context free.
- $\mathcal{L} \cap 0^*1^*2^* = \{0^n1^n2^n : n \geq 0\}$ is not context free.
An Application:

- Is $\mathcal{L} = \{ w \in \{0, 1, 2\}^* : \#_0(w) = \#_1(w) = \#_2(w) \}$ context free?

  - $0^*1^*2^*$ is regular.
  - Context free languages intersected with a regular languages are context free.
  - $\mathcal{L} \cap 0^*1^*2^* = \{0^n1^n2^n : n \geq 0\}$ is not context free.
  - So $\mathcal{L}$ is not a context free language
An Application:

- Is \( \mathcal{L} = \{ w \in \{0, 1, 2\}^* : \#_0(w) = \#_1(w) = \#_2(w) \} \) context free?

- \( 0^*1^*2^* \) is regular.
- Context free languages intersected with a regular languages are context free.
- \( \mathcal{L} \cap 0^*1^*2^* = \{ 0^n1^n2^n : n \geq 0 \} \) is not context free.
- So \( \mathcal{L} \) is not a context free language
An Application:

- Is $\mathcal{L} = \{w \in \{0, 1, 2\}^*: \#_0(w) = \#_1(w) = \#_2(w)\}$ context free?
  
  - $0^*1^*2^*$ is regular.
  - Context free languages intersected with a regular languages are context free.
  - $\mathcal{L} \cap 0^*1^*2^* = \{0^n1^n2^n: n \geq 0\}$ is not context free.
  - So $\mathcal{L}$ is not a context free language

- This could also be established using pumping lemma, but proof above is more elegant.
Closure under complementation?

CFLs are closed under union. If CFLs are also closed under complementation, then they would be closed under intersection because of:

\[ L_1 \cap L_2 = \overline{L_1 \cup L_2} \]

But, CFLs are not closed under intersection, so they cannot be closed under complementation.
Complementation: an example

We give a simple example where $\mathcal{L}$ is not CFL but $\overline{\mathcal{L}}$ is.

- Take $L = \{ww : w \in \{0, 1\}^*\}$.
- $L$ is not a CFL (why?)
- We prove that $\overline{\mathcal{L}}$ is a CFL

Let's restate:

$L$ are the strings of length $2\ell$ for which for all $1 \leq i \leq \ell$, $w_i = w_i + \ell$.

$L$ are strings for which either (1) $|w|$ is odd, or (2) $|w|$ is even and there exists $i$, for which $w_i \neq w_i + \ell$. 

Iftach Haitner and Ronitt Rubinfeld (TAU)
Computational Models Lecture 6
April 4/6, 2016 18 / 49
Complementation: an example

We give a simple example where \( \mathcal{L} \) is not CFL but \( \overline{\mathcal{L}} \) is.

- Take \( \mathcal{L} = \{ww : w \in \{0,1\}^*\} \).
Complementation: an example

We give a simple example where $\mathcal{L}$ is not CFL but $\overline{\mathcal{L}}$ is.

- Take $\mathcal{L} = \{ww : w \in \{0, 1\}^*\}$.
- $\mathcal{L}$ is not a CFL (why?)
Complementation: an example

We give a simple example where $\mathcal{L}$ is not CFL but $\overline{\mathcal{L}}$ is.

- Take $\mathcal{L} = \{ww : w \in \{0, 1\}^*\}$.
- $\mathcal{L}$ is not a CFL (why?)
- $\mathcal{L}$ is not a CFL (why?)
- We prove that $\overline{\mathcal{L}}$ is a CFL
Complementation: an example

We give a simple example where $\mathcal{L}$ is not CFL but $\overline{\mathcal{L}}$ is.

- Take $\mathcal{L} = \{ww : w \in \{0, 1\}^*\}$.
- $\mathcal{L}$ is not a CFL (why?)
- We prove that $\overline{\mathcal{L}}$ is a CFL
- Let’s restate:
Complementation: an example

We give a simple example where $\mathcal{L}$ is not CFL but $\overline{\mathcal{L}}$ is.

- Take $\mathcal{L} = \{ww : w \in \{0, 1\}^*\}$.
- $\mathcal{L}$ is not a CFL (why?)
- We prove that $\overline{\mathcal{L}}$ is a CFL
- Let’s restate:
  - $\mathcal{L}$ are the strings of length $2\ell$ for which for all $1 \leq i \leq \ell$, $w_i = w_{i+\ell}$. 

Iftach Haitner and Ronitt Rubinfeld (TAU) Computational Models Lecture 6 April 4/6, 2016 18 / 49
Complementation: an example

We give a simple example where $L$ is not a CFL but $\overline{L}$ is.

- Take $L = \{ww : w ∈ \{0, 1\}^*\}$.
- $L$ is not a CFL (why?)
- We prove that $\overline{L}$ is a CFL
- Let’s restate:
  - $L$ are the strings of length $2\ell$ for which for all $1 ≤ i ≤ \ell$, $w_i = w_{i+\ell}$.
  - $\overline{L}$ are strings for which either (1) $|w|$ is odd, or (2) $|w|$ is even and there exists $i$, for which $w_i ≠ w_{i+\ell}$.
Complementation cont.

- For any \( y \in \overline{\mathcal{L}} \), either

\[
\begin{align*}
\text{y's length is odd, or} \\
\text{y's length is even, 2} \ell, \text{ and } \exists i \geq 1 \text{ such that } y_i \neq y_{\ell+i}.
\end{align*}
\]
Complementation cont.

- For any $y \in \overline{L}$, either
  - $y$’s length is odd, or
Complementation cont.

For any $y \in \overline{L}$, either

- $y$’s length is odd, or
- $y$’s length is even, $2\ell$, and $\exists i \geq 1$ such that $y_i \neq y_{\ell+i}$.
Complementation cont.

- For any $y \in \overline{L}$, either
  - $y$’s length is odd, or
  - $y$’s length is even, $2\ell$, and $\exists i \geq 1$ such that $y_i \neq y_{\ell+i}$.

- It suffice to construct a PDA/CFG for $\overline{L}_{even}$ – the even length members of $\overline{L}$ (why?)
Complementation cont.

- For any $y \in \overline{L}$, either
  - $y$'s length is odd, or
  - $y$'s length is even, $2\ell$, and $\exists i \geq 1$ such that $y_i \neq y_{\ell+i}$.

- It suffice to construct a PDA/CFG for $\overline{L}_{even}$ – the even length members of $\overline{L}$ (why?)

- Let $\overline{L}_{even}^\sigma = \{(0, 1)^k\sigma(0, 1)^j(0, 1)^k\sigma(0, 1)^j : k, j \geq 0\}$
Complementation cont.

- For any $y \in \overline{L}$, either
  - $y$’s length is odd, or
  - $y$’s length is even, $2\ell$, and $\exists i \geq 1$ such that $y_i \neq y_{\ell+i}$.
- It suffice to construct a PDA/CFG for $\overline{L}_{even}$ — the even length members of $\overline{L}$ (why?)
- Let $\overline{L}^\sigma_{even} = \{\{0, 1\}^k \sigma \{0, 1\}^j \{0, 1\}^k \sigma \{0, 1\}^j : k, j \geq 0\}$
- Note that $\overline{L}_{even} = \overline{L}^0_{even} \cup \overline{L}^1_{even}$
Complementation cont.

- For any $y \in \overline{L}$, either
  - $y$’s length is odd, or
  - $y$’s length is even, $2\ell$, and $\exists i \geq 1$ such that $y_i \neq y_{\ell+i}$.

- It suffice to construct a PDA/CFG for $\overline{L}_{\text{even}}$ — the even length members of $\overline{L}$ (why?)

- Let $\overline{L}^\sigma_{\text{even}} = \{0, 1\}^k \sigma \{0, 1\}^j \{0, 1\}^k \bar{\sigma} \{0, 1\}^j : k, j \geq 0\}$

- Note that $\overline{L}_{\text{even}} = \overline{L}^0_{\text{even}} \cup \overline{L}^1_{\text{even}}$

- and that $\overline{L}^\sigma_{\text{even}} = \{0, 1\}^k \sigma \{0, 1\}^k \{0, 1\}^j \bar{\sigma} \{0, 1\}^j : k, j \geq 0\}$
Complementation cont.

- For any \( y \in \overline{L} \), either
  - \( y \)'s length is odd, or
  - \( y \)'s length is even, \( 2\ell \), and \( \exists i \geq 1 \) such that \( y_i \neq y_{\ell+i} \).

- It suffice to construct a PDA/CFG for \( \overline{L}_{\text{even}} \) — the even length members of \( \overline{L} \) (why?)

- Let \( \overline{L}_{\text{even}}^\sigma = \{ \{0, 1\}^k \sigma \{0, 1\}^j \{0, 1\}^k \overline{\sigma} \{0, 1\}^j : k, j \geq 0 \} \)

- Note that \( \overline{L}_{\text{even}} = \overline{L}_{\text{even}}^0 \cup \overline{L}_{\text{even}}^1 \)

- and that \( \overline{L}_{\text{even}}^\sigma = \{ \{0, 1\}^k \sigma \{0, 1\}^j \{0, 1\}^i \overline{\sigma} \{0, 1\}^j : k, j \geq 0 \} \)

- CFG for \( \overline{L}_{\text{even}}^0 \)

\[
S \to AB \\
A \to CAC \mid 0 \\
B \to CBC \mid 1 \\
C \to 0 \mid 1
\]
A PDA for $\overline{L}^0_{\text{even}}$
A PDA for $\overline{L}_{\text{even}}$

Idea: Guess $k, j \geq 0$, and accept $w$ if it is of the form:

$\{0, 1\}^k0\{0, 1\}^k0\{0, 1\}^j1\{0, 1\}^j$
Homomorphism and Inverse Homomorphism

- Homomorphism: replaces each letter with a word

Example:
\[ h(1) = aba, \quad h(0) = aa \]
\[ L_1 = \{0^n1^n \mid n \geq 0\} \]
\[ h(L_1) = \{a^{2n}(aba)^n \mid n \geq 0\} \]

Claim: Assuming that \( L \) is a CFL, then so is \( h(L) \)

Inverse homomorphism: \( h^{-1}(w) = \{x : h(x) = w\} \)
\[ h^{-1}(L_2) = \{x : h(x) \in L\} \]

Example:
\[ L_2 = \{a^n b^n a^i \mid n, i \geq 0\} \]
\[ h^{-1}(L_2) = \{10^i, 0^i \mid i \geq 0\} \]

Claim: Assuming that \( L \) is a CFL, then so is \( h^{-1}(L) \)
Homomorphism and Inverse Homomorphism

- **Homomorphism**: replaces each letter with a word

- **Example**: $h(1) = aba$, $h(0) = aa$
  
  $h(010) = aa aba aa$

$L_1 = \{0^n1^n|n \geq 0\}$, $h(L_1) = \{a^{2n}(aba)^n|n \geq 0\}$. 

Homomorphism and Inverse Homomorphism

- **Homomorphism**: replaces each letter with a word
- **Example**: $h(1) = aba$, $h(0) = aa$  
  $h(010) = aa \ aba \ aa$  
  $\mathcal{L}_1 = \{0^n1^n|n \geq 0\}$, $h(\mathcal{L}_1) = \{a^{2n}(aba)^n|n \geq 0\}$.
- **Claim**: Assuming that $\mathcal{L}$ is a CFL, then so is $h(\mathcal{L})$
Homomorphism and Inverse Homomorphism

- **Homomorphism**: replaces each letter with a word
  - **Example**: \( h(1) = aba, \ h(0) = aa \)
    
    \( h(010) = aa \ aba \ aa \)
    
    \( \mathcal{L}_1 = \{0^n1^n|n \geq 0\}, \ h(\mathcal{L}_1) = \{a^{2n}(aba)^n|n \geq 0\} \).

- **Claim**: Assuming that \( \mathcal{L} \) is a CFL, then so is \( h(\mathcal{L}) \)

- **Proof?**
Homomorphism and Inverse Homomorphism

- **Homomorphism**: replaces each letter with a word
  - Example: \( h(1) = aba, \ h(0) = aa \)
    \[ h(010) = aa \ aba \ aa \]
  - \( \mathcal{L}_1 = \{0^n1^n \mid n \geq 0\}, \ h(\mathcal{L}_1) = \{a^{2n}(aba)^n \mid n \geq 0\} \).

- **Claim**: Assuming that \( \mathcal{L} \) is a CFL, then so is \( h(\mathcal{L}) \)

- **Proof?**
Homomorphism and Inverse Homomorphism

- **Homomorphism**: replaces each letter with a word

- **Example**: \( h(1) = \text{aba}, \ h(0) = \text{aa} \)
  \[ h(010) = \text{aa aba aa} \]
  \[ \mathcal{L}_1 = \{0^n1^n|n \geq 0\}, \ h(\mathcal{L}_1) = \{a^{2n}(\text{aba})^n|n \geq 0\} \].

- **Claim**: Assuming that \( \mathcal{L} \) is a CFL, then so is \( h(\mathcal{L}) \)

- **Proof?** use the grammar
Homomorphism and Inverse Homomorphism

- **Homomorphism**: replaces each letter with a word

- **Example**: $h(1) = aba$, $h(0) = aa$
  $h(010) = aa aba aa$

- **Claim**: Assuming that $\mathcal{L}$ is a CFL, then so is $h(\mathcal{L})$
- **Proof? use the grammar**

- **Inverse homomorphism**: $h^{-1}(w) = \{x : h(x) = w\}$,
  $h^{-1}(\mathcal{L}) = \{x : h(x) \in \mathcal{L}\}$
Homomorphism and Inverse Homomorphism

- **Homomorphism**: replaces each letter with a word
  
  - **Example**: \( h(1) = aba, \ h(0) = aa \)
  
  \( h(010) = aa aba aa \)

  \( L_1 = \{0^n1^n|n \geq 0\}, \ h(L_1) = \{a^{2n}(aba)^n|n \geq 0\}. \)

- **Claim**: Assuming that \( L \) is a CFL, then so is \( h(L) \)

- **Proof? use the grammar**

- **Inverse homomorphism**: \( h^{-1}(w) = \{x: h(x) = w\}, \)

  \( h^{-1}(L) = \{x: h(x) \in L\} \)

- **Example**: \( L_2 = \{a^n b^n a^i|n, i \geq 0\}, \ h^{-1}(L_2) = \{10^i, 0^i|i \geq 0\}. \)
Homomorphism and Inverse Homomorphism

- **Homomorphism**: replaces each letter with a word

  - **Example**: \( h(1) = aba \), \( h(0) = aa \)
    
    \( h(010) = aa \ aba \ aa \)
    
    \( \mathcal{L}_1 = \{0^n1^n|n \geq 0\} \), \( h(\mathcal{L}_1) = \{a^{2n}(aba)^n|n \geq 0\} \).

- **Claim**: Assuming that \( \mathcal{L} \) is a CFL, then so is \( h(\mathcal{L}) \)

- **Proof? use the grammar**

- **Inverse homomorphism**: \( h^{-1}(w) = \{x: h(x) = w\} \),
    
    \( h^{-1}(\mathcal{L}) = \{x: h(x) \in \mathcal{L}\} \)

- **Example**: \( \mathcal{L}_2 = \{a^n b^n a^i|n, i \geq 0\} \), \( h^{-1}(\mathcal{L}_2) = \{10^i, 0^i|i \geq 0\} \).

- **Claim**: Assuming that \( \mathcal{L} \) is a CFL, then so is \( h^{-1}(\mathcal{L}) \)
Part III

Algorithmic Questions
Emptiness of CFGs

Question 3
Given a CFG, $G$, is $\mathcal{L}(G) = \emptyset$?
Emptiness of CFGs

**Question 3**

Given a CFG, $G$, is $L(G) = \emptyset$?

In other words, is there a string generated by $G$?
Question 3

Given a CFG, $G$, is $L(G) = \emptyset$?

In other words, is there a string generated by $G$?

Theorem 4

*There is an algorithm that solves this problem (and always halts).*
Question 3
Given a CFG, $G$, is $\mathcal{L}(G) = \emptyset$?

In other words, is there a string generated by $G$?

Theorem 4
There is an algorithm that solves this problem (and always halts).

Possible approaches for a proof:
- Not So Great Idea: We know how to test whether $w \in \mathcal{L}(G)$ for any string $w$, so just try it for each $w$... (when can we stop?)
Emptiness of CFGs

**Question 3**

Given a CFG, $G$, is $L(G) = \emptyset$?

In other words, is there a string generated by $G$?

**Theorem 4**

*There is an algorithm that solves this problem (and always halts).*

Possible approaches for a proof:

- **Not So Great Idea:** We know how to test whether $w \in L(G)$ for any string $w$, so just try it for each $w$... (when can we stop?)

- **Better Idea:** Can the start variable generate a string of terminals?
Emptiness of CFGs

Question 3
Given a CFG, $G$, is $\mathcal{L}(G) = \emptyset$?

In other words, is there a string generated by $G$?

Theorem 4
There is an algorithm that solves this problem (and always halts).

Possible approaches for a proof:

- Not So Great Idea: We know how to test whether $w \in \mathcal{L}(G)$ for any string $w$, so just try it for each $w$... (when can we stop?)
- Better Idea: Can the start variable generate a string of terminals?
- A more holistic approach: Can a particular variable generate a string of terminals?
Checking Emptiness

Idea: Mark variables that can produce a string of terminals

1. Mark all terminal symbols in $G$.
2. Repeat until no new variable become marked:
   Mark any $A$ where $A \rightarrow U_1 U_2 \ldots U_k$ and all $U_i$ have already been marked.
3. Remove all unmarked variables, and any rule they appear in.
4. If $S$ is removed, then $\mathcal{L}(G) = \emptyset$. Correctness?

Recall cleanup in the CNF conversion process?
Checking Emptiness

Idea: Mark variables that can produce a string of terminals

1. Mark all terminal symbols in $G$.
2. Repeat until no new variable become marked:
   Mark any $A$ where $A \rightarrow U_1 U_2 \ldots U_k$ and all $U_i$ have already been marked.
3. Remove all unmarked variables, and any rule they appear in.
4. If $S$ is removed, then $\mathcal{L}(G) = \emptyset$.

Correctness?
Checking Emptiness

Idea: Mark variables that can produce a string of terminals

1. Mark all terminal symbols in $G$.
2. Repeat until no new variable become marked:
   Mark any $A$ where $A \rightarrow U_1 U_2 \ldots U_k$ and all $U_i$ have already been marked.
3. Remove all unmarked variables, and any rule they appear in.
4. If $S$ is removed, then $\mathcal{L}(G) = \emptyset$.

Correctness?

Recall cleanup in the CNF conversion process?
Question 5

Given a CFG $G$, is $L(G) = \Sigma^*$?
Question 5

Given a CFG $G$, is $L(G) = \Sigma^*$?

- We just saw an algorithm to determine, given a CFG $G$, whether $L(G) = \emptyset$.
Question 5
Given a CFG $G$, is $\mathcal{L}(G) = \Sigma^*$?

- We just saw an algorithm to determine, given a CFG $G$, whether $\mathcal{L}(G) = \emptyset$.
- $\mathcal{L}(G) = \Sigma^*$ iff $\overline{\mathcal{L}(G)} = \emptyset$. Why not modify the algorithm so it determines emptiness of the complement?
Question 5
Given a CFG $G$, is $\mathcal{L}(G) = \Sigma^*$?

- We just saw an algorithm to determine, given a CFG $G$, whether $\mathcal{L}(G) = \emptyset$.
- $\mathcal{L}(G) = \Sigma^*$ iff $\overline{\mathcal{L}(G)} = \emptyset$. Why not modify the algorithm so it determines emptiness of the complement?
- Unfortunately, CFGs are not closed under complement.
Question 5

Given a CFG $G$, is $L(G) = \Sigma^*$?

- We just saw an algorithm to determine, given a CFG $G$, whether $L(G) = \emptyset$.
- $L(G) = \Sigma^*$ iff $\overline{L(G)} = \emptyset$. Why not modify the algorithm so it determines emptiness of the complement?
- Unfortunately, CFGs are not closed under complement.
Question 5

Given a CFG $G$, is $\mathcal{L}(G) = \Sigma^*$?

- We just saw an algorithm to determine, given a CFG $G$, whether $\mathcal{L}(G) = \emptyset$.
- $\mathcal{L}(G) = \Sigma^*$ iff $\overline{\mathcal{L}(G)} = \emptyset$. Why not modify the algorithm so it determines emptiness of the complement?
- Unfortunately, CFGs are not closed under complement.

Fact 6

There is no algorithm to solve CFG fullness.
Question 5
Given a CFG $G$, is $\mathcal{L}(G) = \Sigma^*$?

- We just saw an algorithm to determine, given a CFG $G$, whether $\mathcal{L}(G) = \emptyset$.
- $\mathcal{L}(G) = \Sigma^*$ iff $\overline{\mathcal{L}(G)} = \emptyset$. Why not modify the algorithm so it determines emptiness of the complement?
- Unfortunately, CFGs are not closed under complement.

Fact 6
There is no algorithm to solve CFG fullness.

- We are not prepared to prove this remarkable fact (yet).
Finiteness of CFGs

Question 7

Given a CFG $G$, is $|\mathcal{L}(G)|$ finite?
Finiteness of CFGs

Question 7

Given a CFG $G$, is $|\mathcal{L}(G)|$ finite?

First, a useful subroutine.
CLEANUP: Removing redundant variables and terminals

1. Mark all terminal symbols in $G$.
2. Repeat until no new variable become marked:
   Mark any $A$ where $A \rightarrow U_1 U_2 \ldots U_k$ and all $U_i$ have already been marked.
3. Remove all unmarked variables, and any rule they appear in.
4. If $S$ is removed, then $L(G) = \emptyset$.
5. Remove any variable $A$ not reachable from $S$.
6. Remove any terminal which does not appear in some rule.
# Back to finiteness of CFGs

## Question 8

Given a CFG $G$, is $|\mathcal{L}(G)|$ finite?

---

1. Remove redundant variables and terminals.
2. Turn into a CNF form
3. Create a graph $C$ whose nodes are variables and its directed edges are derivations.
4. Return TRUE iff $C$ has no cycles.

**Correctness?**

Iftach Haitner and Ronitt Rubinfeld (TAU)
Back to finiteness of CFGs

Question 8

Given a CFG $G$, is $|\mathcal{L}(G)|$ finite?

1. Remove redundant variables and terminals.
2. Turn into a CNF form
3. Create a graph $C$ whose nodes are variables and its directed edges are derivations.
4. Return $\text{TRUE}$ iff $C$ has no cycles.
Question 8

Given a CFG $G$, is $|\mathcal{L}(G)|$ finite?

1. Remove redundant variables and terminals.
2. Turn into a CNF form
3. Create a graph $C$ whose nodes are variables and its directed edges are derivations.
4. Return $\text{TRUE}$ iff $C$ has no cycles.

Correctness?
Question 9

Given a CFG $G$, is $L(G)$ inherently ambiguous?

This means that for any CFG that generates $L(G)$, there is a word in the language with two different parse trees.
Question 9

Given a CFG $G$, is $\mathcal{L}(G)$ inherently ambiguous?

This means that for any CFG that generates $\mathcal{L}(G)$, there is a word in the language with two different parse trees.

Fact 10

There is no algorithm to solve CFG inherent ambiguity.

- We will not prove this fact, yet you want to know it to put things in context.
When Are Two CFGs equivalent?

**Question 11**

Given two CFG $G_1$ and $G_2$, test if $L(G_1) = L(G_2)$. Is there an algorithm to solve this problem?
Part IV

Equivalence Theorem
The CFG–PDA Equivalence Theorem

Theorem 12

$L_{PDA} = L_{CFG}$: A language is context free if and only if some pushdown automata accepts it.
The CFG–PDA Equivalence Theorem

Theorem 12

\[ \mathcal{L}_{PDA} = \mathcal{L}_{CFG} : \text{A language is context free if and only if some pushdown automata accepts it.} \]

This time (unlike the regular expression vs. regular languages theorem), both the proof “if” part and of the “only if” part are non trivial.
The CFG–PDA Equivalence Theorem

**Theorem 12**

\[ \mathcal{L}_{\text{PDA}} = \mathcal{L}_{\text{CFG}} : A \text{ language is context free if and only if some pushdown automata accepts it.} \]

This time (unlike the regular expression vs. regular languages theorem), both the proof “if” part and of the “only if” part are non trivial.

Proof sketch follows.
Lemma 13

\[ \mathcal{L}_{\text{CFG}} \subseteq \mathcal{L}_{\text{PDA}} \]: If a language is context free, then some PDA accepts it.
Lemma 13

\[ \mathcal{L}_{\text{CFG}} \subseteq \mathcal{L}_{\text{PDA}} : \text{If a language is context free, then some PDA accepts it.} \]

- Let \( \mathcal{L} \) be a context-free language, and let \( G = (V, \Sigma, R, S) \) be context-free grammar for \( \mathcal{L} \).
Lemma 13

\[ L_{\text{CFG}} \subseteq L_{\text{PDA}} : \text{If a language is context free, then some PDA accepts it.} \]

- Let \( L \) be a context-free language, and let \( G = (V, \Sigma, R, S) \) be a context-free grammar for \( L \).
- We build a PDA \( P = (Q, \Sigma, \Gamma, \delta, q_0, F) \), such that on input \( w \) it "figures out" if there is a derivation of \( w \) using \( G \).
Lemma 13

\( \mathcal{L}_{\text{CFG}} \subseteq \mathcal{L}_{\text{PDA}} : \text{If a language is context free, then some PDA accepts it.} \)

- Let \( \mathcal{L} \) be a context-free language, and let \( G = (V, \Sigma, R, S) \) be context-free grammar for \( \mathcal{L} \)
- We build a PDA \( P = (Q, \Sigma, \Gamma, \delta, q_0, F) \), such that on input \( w \) it “figures out” if there is a derivation of \( w \) using \( G \).
Lemma 13

\( \mathcal{L}_{\text{CFG}} \subseteq \mathcal{L}_{\text{PDA}} \): If a language is context free, then some PDA accepts it.

- Let \( \mathcal{L} \) be a context-free language, and let \( G = (V, \Sigma, R, S) \) be context-free grammar for \( \mathcal{L} \).
- We build a PDA \( P = (Q, \Sigma, \Gamma, \delta, q_0, F) \), such that on input \( w \) it “figures out” if there is a derivation of \( w \) using \( G \).

Question 14

How does \( P \) figure out which substitution to make?
Lemma 13

\[ \mathcal{L}_{\text{CFG}} \subseteq \mathcal{L}_{\text{PDA}} : \text{If a language is context free, then some PDA accepts it.} \]

- Let \( \mathcal{L} \) be a context-free language, and let \( G = (V, \Sigma, R, S) \) be context-free grammar for \( \mathcal{L} \).
- We build a PDA \( P = (Q, \Sigma, \Gamma, \delta, q_0, F) \), such that on input \( w \) it “figures out” if there is a derivation of \( w \) using \( G \).

Question 14

How does \( P \) figure out which substitution to make?

Answer: It guesses.
Simplifying Assumptions

1. In a single move, a PDA can push a whole word (from some fixed set) into the stack (first letter at the top)
Simplifying Assumptions

1. In a *single* move, a PDA can push a *whole* word (from some fixed set) into the stack (first letter at the top)

Can we justify it?
Simplifying Assumptions

1. In a single move, a PDA can push a whole word (from some fixed set) into the stack (first letter at the top)
   Can we justify it?

2. When deriving a word from a CFL, we always substitute the left most variable
Simplifying Assumptions

1. In a **single** move, a PDA can push a **whole** word (from some fixed set) into the stack (first letter at the top)

   Can we justify it?

2. When deriving a word from a CFL, we always substitute the **left most** variable

   Does it change the derived language?
Informal Description of $P$

Algorithm 15 ($P$)

1. Push $S$ on stack
2. While top of the stack $t$ is not $\$$:
   2.1 If $t$ is variable $A$, 
       (non-deterministically) select rule $A \rightarrow \alpha$ and substitute (i.e. push $\alpha$ to stack).
   2.2 If $t$ is a terminal $a$,
       read next input and compare; Reject if different.
   2.3 Accept if end of input and stack is empty
State Diagram for $P$

- Initial state: $q_{start}$
- Transition: $\epsilon, \epsilon \rightarrow S \, $ for a rule $A \rightarrow \alpha$
- Transition: $\alpha, a \rightarrow \epsilon$ for a terminal $a$
- Transition: $\epsilon, \, \$ \rightarrow \epsilon$
- Accepting state: $q_{accept}$
Example

consider the CFG:
\[ S \rightarrow 0S1|\varepsilon. \]

The related PDA:
Claim: $\mathcal{L}(P) = \mathcal{L}(G)$
Claim: $\mathcal{L}(P) = \mathcal{L}(G)$

Claim 16

$S \xrightarrow{*} \alpha$ iff $\alpha = \alpha_1 \alpha_2$ such that $(q_{\text{loop}}, \alpha_2) \in \hat{\delta}(q_{\text{loop}}, \alpha_1, \$)$.

(note that $\alpha_1$ is made of terminals, $\alpha_2$ can be variables and terminals)
Claim: $\mathcal{L}(P) = \mathcal{L}(G)$

Claim 16

$S \xrightarrow{*} \alpha$ iff $\alpha = \alpha_1 \alpha_2$ such that $(q_{\text{loop}}, \alpha_2 \$) \in \widehat{\delta}(q_{\text{loop}}, \alpha_1, \$).

(note that $\alpha_1$ is made of terminals, $\alpha_2$ can be variables and terminals)

Does the above yield that $\mathcal{L}(P) = \mathcal{L}(G)$?
$S \xrightarrow{*} \alpha \implies \alpha = \alpha_1 \alpha_2$ such that $(q_{loop}, \alpha_2) \in \hat{\delta}(q_{loop}, \alpha_1, S)$

Proof by induction on the number of derivations steps used to yield $\alpha$ from $S$. 
\[ S \xrightarrow{*} \alpha \implies \alpha = \alpha_1 \alpha_2 \text{ such that } (q_{\text{loop}}, \alpha_2 \dollar) \in \hat{\delta}(q_{\text{loop}}, \alpha_1, S\dollar) \]

Proof by induction on the number of derivations steps used to yield \( \alpha \) from \( S \).

- 1 derivation steps: hence there is a rule \( S \rightarrow \alpha \). Thus \( (q_{\text{loop}}, \alpha \dollar) \in \hat{\delta}(q_{\text{loop}}, \varepsilon, S\dollar) \), and the proof follows for \( \alpha_1 = \varepsilon \) and \( \alpha_2 = \alpha \).
$S \rightarrow^* \alpha \implies \alpha = \alpha_1 \alpha_2$ such that $(q_{\text{loop}}, \alpha_2) \in \hat{\delta}(q_{\text{loop}}, \alpha_1, S)$

Proof by induction on the number of derivations steps used to yield $\alpha$ from $S$.

- 1 derivation steps: hence there is a rule $S \rightarrow \alpha$. Thus $(q_{\text{loop}}, \alpha) \in \hat{\delta}(q_{\text{loop}}, \varepsilon, S)$, and the proof follows for $\alpha_1 = \varepsilon$ and $\alpha_2 = \alpha$.

- Assume $S \rightarrow^* \alpha$ in $k > 1$ derivation steps, and let $\alpha'$ be the string derived by the first $(k - 1)$ steps.
$$S \xrightarrow{*} \alpha \implies \alpha = \alpha_1 \alpha_2 \text{ such that } (q_{\text{loop}}, \alpha_2 S) \in \hat{\delta}(q_{\text{loop}}, \alpha_1, S)$$

Proof by induction on the number of derivations steps used to yield $\alpha$ from $S$.

1. **1 derivation steps:** hence there is a rule $S \rightarrow \alpha$. Thus

   $$(q_{\text{loop}}, \alpha S) \in \hat{\delta}(q_{\text{loop}}, \varepsilon, S),$$

   and the proof follows for $\alpha_1 = \varepsilon$ and $\alpha_2 = \alpha$.

2. **Assume $S \xrightarrow{*} \alpha$ in $k > 1$ derivation steps, and let $\alpha'$ be the string derived by the first $(k-1)$ steps.**

3. **By i.h $\alpha' = \alpha_1' \alpha_2'$ such that $(q_{\text{loop}}, \alpha_2' S) \in \hat{\delta}(q_{\text{loop}}, \alpha_1', S)$**
$$S \rightarrow^* \alpha \quad \Rightarrow \quad \alpha = \alpha_1 \alpha_2 \text{ such that } (q_{\text{loop}}, \alpha_2\$) \in \hat{\delta}(q_{\text{loop}}, \alpha_1, S\$)$$

Proof by induction on the number of derivations steps used to yield $\alpha$ from $S$.

- **1 derivation steps**: hence there is a rule $S \rightarrow \alpha$. Thus $(q_{\text{loop}}, \alpha\$) \in \hat{\delta}(q_{\text{loop}}, \varepsilon, S\$), and the proof follows for $\alpha_1 = \varepsilon$ and $\alpha_2 = \alpha$.
- **Assume $S \rightarrow^* \alpha$ in $k > 1$ derivation steps**, and let $\alpha'$ be the string derived by the first $(k - 1)$ steps.
- **By i.h** $\alpha' = \alpha'_1 \alpha'_2$ such that $(q_{\text{loop}}, \alpha'_2\$) \in \hat{\delta}(q_{\text{loop}}, \alpha'_1, S\$)
- **Write** $\alpha'_2 = w_1 Aw_2$ where $A$ is the left most variable in $\alpha'_2$. 
\( S \xrightarrow{*} \alpha \implies \alpha = \alpha_1 \alpha_2 \) such that \((q_{\text{loop}}, \alpha_2 \$$) \in \hat{\delta}(q_{\text{loop}}, \alpha_1, S$$)

Proof by induction on the number of derivations steps used to yield \( \alpha \) from \( S \).

- **1 derivation steps:** hence there is a rule \( S \rightarrow \alpha \). Thus \((q_{\text{loop}}, \alpha \$$) \in \hat{\delta}(q_{\text{loop}}, \varepsilon, S$$), and the proof follows for \( \alpha_1 = \varepsilon \) and \( \alpha_2 = \alpha \).

- Assume \( S \xrightarrow{*} \alpha \) in \( k > 1 \) derivation steps, and let \( \alpha' \) be the string derived by the first \( (k - 1) \) steps.

- By i.h \( \alpha' = \alpha'_1 \alpha'_2 \) such that \((q_{\text{loop}}, \alpha'_2 \$$) \in \hat{\delta}(q_{\text{loop}}, \alpha'_1, S$$)

- Write \( \alpha'_2 = w_1 Aw_2 \) where \( A \) is the left most variable in \( \alpha'_2 \).

- The \( k' \)th derivation step replaces this occurrence of \( A \) with a string \( t \) (why?)
$S \rightarrow^* \alpha \implies \alpha = \alpha_1 \alpha_2$ such that $(q_{\text{loop}}, \alpha_2 \$) \in \hat{\delta}(q_{\text{loop}}, \alpha_1, S\$)$

Proof by induction on the number of derivations steps used to yield $\alpha$ from $S$.

- **1 derivation steps:** hence there is a rule $S \rightarrow \alpha$. Thus $(q_{\text{loop}}, \alpha \$) \in \hat{\delta}(q_{\text{loop}}, \varepsilon, S\$)$, and the proof follows for $\alpha_1 = \varepsilon$ and $\alpha_2 = \alpha$.

- Assume $S \rightarrow^* \alpha$ in $k > 1$ derivation steps, and let $\alpha'$ be the string derived by the first $(k - 1)$ steps.

- By i.h $\alpha' = \alpha'_1 \alpha'_2$ such that $(q_{\text{loop}}, \alpha'_2 \$) \in \hat{\delta}(q_{\text{loop}}, \alpha'_1, S\$)$

- Write $\alpha'_2 = w_1 Aw_2$ where $A$ is the left most variable in $\alpha'_2$.

- The $k$’th derivation step replaces this occurrence of $A$ with a string $t$ (why?)

- It is easy to see that $(q_{\text{loop}}, tw_2 \$) \in \hat{\delta}(q_{\text{loop}}, \alpha'_1 w_1, S\$)$.
\[ S \rightarrow^* \alpha \implies \alpha = \alpha_1 \alpha_2 \text{ such that } (q_{\text{loop}}, \alpha_2 \epsilon) \in \hat{\delta}(q_{\text{loop}}, \alpha_1, S) \]

Proof by induction on the number of derivations steps used to yield \( \alpha \) from \( S \).

- **1 derivation steps:** hence there is a rule \( S \rightarrow \alpha \). Thus \( (q_{\text{loop}}, \alpha \epsilon) \in \hat{\delta}(q_{\text{loop}}, \epsilon, S) \), and the proof follows for \( \alpha_1 = \epsilon \) and \( \alpha_2 = \alpha \).

- **Assume** \( S \rightarrow^* \alpha \) in \( k > 1 \) derivation steps, and let \( \alpha' \) be the string derived by the first \( (k - 1) \) steps.

  - By i.h \( \alpha' = \alpha'_1 \alpha'_2 \) such that \( (q_{\text{loop}}, \alpha'_2 \epsilon) \in \hat{\delta}(q_{\text{loop}}, \alpha'_1, S) \)
  
  - Write \( \alpha'_2 = w_1 Aw_2 \) where \( A \) is the left most variable in \( \alpha'_2 \).

  - The \( k \)'th derivation step replaces this occurrence of \( A \) with a string \( t \) (why?)

  - It is easy to see that \( (q_{\text{loop}}, tw_2 \epsilon) \in \hat{\delta}(q_{\text{loop}}, \alpha'_1 w_1, S) \).

  - To complete the proof take \( \alpha_1 = \alpha'_1 w_1 \) and \( \alpha_2 = tw_2 \).
\[ \alpha = \alpha_1 \alpha_2 \text{ such that } (q_{\text{loop}}, \alpha_2 \$) \in \hat{\delta}(q_{\text{loop}}, \alpha_1, \$) \implies S \xrightarrow{*} \alpha \]
\[ \alpha = \alpha_1 \alpha_2 \text{ such that } (q_{\text{loop}}, \alpha_2 \$) \in \hat{\delta}(q_{\text{loop}}, \alpha_1, \$) \implies S \xrightarrow{*} \alpha \]

Proof by induction on the number of steps used by \( P \) to process \( \alpha_1 \).
\[ \alpha = \alpha_1 \alpha_2 \text{ such that } (q_{loop}, \alpha_2 \$) \in \hat{\delta}(q_{loop}, \alpha_1, \$) \implies S \xrightarrow{*} \alpha \]

Proof by induction on the number of steps used by \( P \) to process \( \alpha_1 \).

- A single step: \( \alpha_1 = \varepsilon \) and \( \alpha_2 = S\$ \), and the proof follows since \( S \xrightarrow{*} S \).
\[ \alpha = \alpha_1 \alpha_2 \text{ such that } (q_{loop}, \alpha_2 \$) \in \hat{\delta}(q_{loop}, \alpha_1, \$) \implies S \xrightarrow{*} \alpha \]

Proof by induction on the number of steps used by \( P \) to process \( \alpha_1 \).

- A single step: \( \alpha_1 = \varepsilon \) and \( \alpha_2 = S\$ \), and the proof follows since \( S \xrightarrow{*} S \).
- Assume \( \alpha_1 \) was processed in \( k > 1 \) steps, and let \( \alpha'_1 \) and \( \alpha'_2 \) be the input string read and the stack value before the last step.
$\alpha = \alpha_1 \alpha_2$ such that $(q_{loop}, \alpha_2$) $\in \hat{\delta}(q_{loop}, \alpha_1, \$) \implies S \rightarrow^* \alpha$

Proof by induction on the number of steps used by $P$ to process $\alpha_1$.

- A single step: $\alpha_1 = \varepsilon$ and $\alpha_2 = S\$, and the proof follows since $S \rightarrow^* S$.

- Assume $\alpha_1$ was processed in $k > 1$ steps, and let $\alpha'_1$ and $\alpha'_2$ be the input string read and the stack value before the last step.

- Note that $(q_{loop}, \alpha'_2\$) $\in \hat{\delta}(q_{loop}, \alpha'_1, \$)$. 
Proof by induction on the number of steps used by $P$ to process $\alpha_1$.

- A single step: $\alpha_1 = \varepsilon$ and $\alpha_2 = S\$, and the proof follows since $S \Rightarrow S$.

- Assume $\alpha_1$ was processed in $k > 1$ steps, and let $\alpha'_1$ and $\alpha'_2$ be the input string read and the stack value before the last step.

- Note that $(q_{loop}, \alpha'_2\$) $\in \widehat{\delta}(q_{loop}, \alpha'_1, \$).

- By i.h $S \Rightarrow \alpha' = \alpha'_1 \alpha'_2$. 
\( \alpha = \alpha_1 \alpha_2 \) such that \((q_{\text{loop}}, \alpha_2$) \(\in \hat{\delta}(q_{\text{loop}}, \alpha_1$, $) \implies S \xrightarrow{*} \alpha \)

Proof by induction on the number of steps used by \(P\) to process \(\alpha_1\).

- **A single step:** \(\alpha_1 = \varepsilon\) and \(\alpha_2 = S$\), and the proof follows since \(S \xrightarrow{*} S\).

- **Assume \(\alpha_1\) was processed in \(k > 1\) steps, and let \(\alpha'_1\) and \(\alpha'_2\) be the input string read and the stack value before the last step.

  - **Note that** \((q_{\text{loop}}, \alpha'_2$) \(\in \hat{\delta}(q_{\text{loop}}, \alpha'_1$, $)\).

  - **By i.h** \(S \xrightarrow{*} \alpha' = \alpha'_1 \alpha'_2\).

  - **In case the \(k\)'th move of \(P\) is reading an input character,** then \(\alpha_1 \alpha_2 = \alpha'_1 \alpha'_2\), and therefore \(S \xrightarrow{*} \alpha_1 \alpha_2\).
\[ \alpha = \alpha_1 \alpha_2 \text{ such that } (q_{\text{loop}}, \alpha_2) \in \hat{\delta}(q_{\text{loop}}, \alpha_1, \$) \implies S \xrightarrow{*} \alpha \]

Proof by induction on the number of steps used by \( P \) to process \( \alpha_1 \).

- A single step: \( \alpha_1 = \varepsilon \) and \( \alpha_2 = S\$, and the proof follows since \( S \xrightarrow{*} S \).

- Assume \( \alpha_1 \) was processed in \( k > 1 \) steps, and let \( \alpha'_1 \) and \( \alpha'_2 \) be the input string read and the stack value before the last step.

- Note that \( (q_{\text{loop}}, \alpha'_2) \in \hat{\delta}(q_{\text{loop}}, \alpha'_1, \$) \).

- By i.h. \( S \xrightarrow{*} \alpha' = \alpha'_1 \alpha'_2 \).

- In case the \( k \)'th move of \( P \) is reading an input character, then \( \alpha_1 \alpha_2 = \alpha'_1 \alpha'_2 \), and therefore \( S \xrightarrow{*} \alpha_1 \alpha_2 \).

- Otherwise, \( \alpha'_1 = \alpha_1, \alpha'_2 = Aw \) and \( \alpha_2 = tw \) for some rule \( A \rightarrow t \in R \).
\[ \alpha = \alpha_1 \alpha_2 \text{ such that } (q_{\text{loop}}, \alpha_2) \in \hat{\delta}(q_{\text{loop}}, \alpha_1, $) \implies S \xrightarrow{*} \alpha \]

Proof by induction on the number of steps used by \( P \) to process \( \alpha_1 \).

- A single step: \( \alpha_1 = \varepsilon \) and \( \alpha_2 = S$, and the proof follows since \( S \xrightarrow{*} S \).
- Assume \( \alpha_1 \) was processed in \( k > 1 \) steps, and let \( \alpha'_1 \) and \( \alpha'_2 \) be the input string read and the stack value before the last step.
- Note that \((q_{\text{loop}}, \alpha'_2$) \in \( \hat{\delta}(q_{\text{loop}}, \alpha'_1, $).
- By i.h \( S \xrightarrow{*} \alpha' = \alpha'_1 \alpha'_2 \).
- In case the \( k \)'th move of \( P \) is reading an input character, then \( \alpha_1 \alpha_2 = \alpha'_1 \alpha'_2 \), and therefore \( S \xrightarrow{*} \alpha_1 \alpha_2 \).
- Otherwise, \( \alpha'_1 = \alpha_1 \), \( \alpha'_2 = Aw \) and \( \alpha_2 = tw \) for some rule \( A \rightarrow t \in R \).
- Hence \( S \xrightarrow{*} \alpha_1 \alpha_2 \).
Lemma 17

\( L_{PDA} \subseteq L_{CFG} \): If a PDA accepts a language then it is context free.
Lemma 17

$L_{PDA} \subseteq L_{CFG}$: If a PDA accepts a language then it is context free.

We prove the lemma by constructing a CFG $G$ for a language $L$ accepted by a PDA $P$. 

[Proof details]
Lemma 17

\( \mathcal{L}_{PDA} \subseteq \mathcal{L}_{CFG} \): If a PDA accepts a language then it is context free.

We prove the lemma by constructing a CFG \( G \) for a language \( \mathcal{L} \) accepted by a PDA \( P \).

Let \( P = (Q, \Sigma, \Gamma, \delta, q_0, F) \). We assume w.l.o.g. that:

1. A single accepting state \( q_a \in F \).
2. \( P \) empties the stack before accepting.
3. Each transition either pops or pushes.
Lemma 17

$L_{\text{PDA}} \subseteq L_{\text{CFG}}$: If a PDA accepts a language then it is context free.

We prove the lemma by constructing a CFG $G$ for a language $L$ accepted by a PDA $P$.

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$. We assume w.l.o.g. that:

- A single accepting state $q_a \in F$. 
Lemma 17

\( \mathcal{L}_{\text{PDA}} \subseteq \mathcal{L}_{\text{CFG}} \): If a PDA accepts a language then it is context free.

We prove the lemma by constructing a CFG \( G \) for a language \( \mathcal{L} \) accepted by a PDA \( P \).

Let \( P = (Q, \Sigma, \Gamma, \delta, q_0, F) \). We assume w.l.o.g. that:

- A single accepting state \( q_a \in F \).
- \( P \) empties the stack before accepting.
Lemma 17

$L_{PDA} \subseteq L_{CFG}$: If a PDA accepts a language then it is context free.

We prove the lemma by constructing a CFG $G$ for a language $L$ accepted by a PDA $P$.

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$. We assume wlg. that:

- A single accepting state $q_a \in F$.
- $P$ empties the stack before accepting.
- Each transition either pops or pushes.
Lemma 17

\[ \mathcal{L}_{\text{PDA}} \subseteq \mathcal{L}_{\text{CFG}} : \text{If a PDA accepts a language then it is context free.} \]

We prove the lemma by constructing a CFG \( G \) for a language \( \mathcal{L} \) accepted by a PDA \( P \).

Let \( P = (Q, \Sigma, \Gamma, \delta, q_0, F) \). We assume wlg. that:

- A single accepting state \( q_a \in F \).
- \( P \) empties the stack before accepting
- Each transition either pops or pushes
Lemma 17

$L_{\text{PDA}} \subseteq L_{\text{CFG}}$: If a PDA accepts a language then it is context free.

We prove the lemma by constructing a CFG $G$ for a language $L$ accepted by a PDA $P$.

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$. We assume w.l.o.g. that:

- A single accepting state $q_a \in F$.
- $P$ empties the stack before accepting.
- Each transition either pops or pushes.

Can we justify the above?
Proof Idea

- Suppose string $x$ takes $P$ from state $p$ with empty stack to state $q$ with empty stack:
Proof Idea

- Suppose string $x$ takes $P$ from state $p$ with empty stack to state $q$ with empty stack:
- First move that touches the stack must be a push, last must be a pop.
Proof Idea

- Suppose string $x$ takes $P$ from state $p$ with empty stack to state $q$ with empty stack:
- First move that touches the stack must be a push, last must be a pop.
- In between:
Proof Idea

- Suppose string $x$ takes $P$ from state $p$ with empty stack to state $q$ with empty stack:
- First move that touches the stack must be a push, last must be a pop.
- In between:
  - *Either stack is empty only at start and finish:*
Proof Idea

- Suppose string $x$ takes $P$ from state $p$ with empty stack to state $q$ with empty stack:
- First move that touches the stack must be a push, last must be a pop.
- In between:
- *Either stack is empty only at start and finish:*
- Simulate by $A_{pq} \rightarrow aA_{rs}b$, where $a, b$ are first and last symbols in $x$, $r$ is state that $p$ can reach in a step and $s$ is state that can reach $q$ in a step.
Proof Idea

- Suppose string $x$ takes $P$ from state $p$ with empty stack to state $q$ with empty stack:
- First move that touches the stack must be a push, last must be a pop.
- In between:
  - *Either stack is empty only at start and finish:*
    - Simulate by $A_{pq} \rightarrow aA_{rs}b$, where $a, b$ are first and last symbols in $x$, $r$ is state that $p$ can reach in a step and $s$ is state that can reach $q$ in a step.
  - *or stack was empty at some point in between:*
Proof Idea

- Suppose string $x$ takes $P$ from state $p$ with empty stack to state $q$ with empty stack:
- First move that touches the stack must be a push, last must be a pop.
- In between:
  - Either stack is empty only at start and finish:
    - Simulate by $A_{pq} \rightarrow aA_{rs}b$, where $a, b$ are first and last symbols in $x$, $r$ is state that $p$ can reach in a step and $s$ is state that can reach $q$ in a step.
  - Or stack was empty at some point in between:
    - Simulate by $A_{pq} \rightarrow A_{pr}A_{rq}$ where $r$ is intermediate state and $P$ has empty stack.
Defining $G = (V, \Sigma, R, S)$

$V = \{A_{pq} : p, q \in Q\}$
Defining $G = (V, \Sigma, R, S)$

$\triangleright \ V = \{A_{pq} : p, q \in Q\}$

Idea: $A_{pq}$ will generate all strings that take $P$ from $p$ with an empty stack, to $q$ with an empty stack
Defining $G = (V, \Sigma, R, S)$

- $V = \{ A_{pq} : p, q \in Q \}$
  Idea: $A_{pq}$ will generate all strings that take $P$ from $p$ with an empty stack, to $q$ with an empty stack

- $S = A_{q_0, q_a}$
Defining \( G = (V, \Sigma, R, S) \)

- \( V = \{ A_{pq} : p, q \in Q \} \)
  
  Idea: \( A_{pq} \) will generate all strings that take \( P \) from \( p \) with an empty stack, to \( q \) with an empty stack

- \( S = A_{q_0, q_a} \)

- Initially \( R = \emptyset \) and
Defining $G = (V, \Sigma, R, S)$

- $V = \{A_{pq} : p, q \in Q\}$
  
  Idea: $A_{pq}$ will generate all strings that take $P$ from $p$ with an empty stack, to $q$ with an empty stack

- $S = A_{q_0, q_a}$

- Initially $R = \emptyset$ and
  
  1. Add $\{A_{pq} \rightarrow A_{p,r}A_{r,q} : p, q, r \in Q\}$ to $R$
Defining $G = (V, \Sigma, R, S)$

- $V = \{A_{pq} : p, q \in Q\}$
  - Idea: $A_{pq}$ will generate all strings that take $P$ from $p$ with an empty stack, to $q$ with an empty stack
- $S = A_{q_0,q_a}$
- Initially $R = \emptyset$ and
  1. Add $\{A_{pq} \rightarrow A_p, rA_r, q : p, q, r \in Q\}$ to $R$
  2. Add $\{A_{qq} \rightarrow \varepsilon : q \in Q\}$ to $R$
Defining $G = (V, \Sigma, R, S)$

- $V = \{A_{pq} : p, q \in Q\}$
  
  Idea: $A_{pq}$ will generate all strings that take $P$ from $p$ with an empty stack, to $q$ with an empty stack

- $S = A_{q_0, q_a}$

- Initially $R = \emptyset$ and
  
  1. Add $\{A_{pq} \rightarrow A_p, rA_r, q : p, q, r \in Q\}$ to $R$
  2. Add $\{A_{qq} \rightarrow \varepsilon : q \in Q\}$ to $R$
  3. For all $p, r, s, q \in Q$, $a, b \in \Sigma_\varepsilon$ and $t \in \Gamma$ such that
Defining $G = (V, \Sigma, R, S)$

- $V = \{A_{pq}: p, q \in Q\}$
  
  Idea: $A_{pq}$ will generate all strings that take $P$ from $p$ with an empty stack, to $q$ with an empty stack

- $S = A_{q_0,q_a}$

- Initially $R = \emptyset$ and
  
  1. Add $\{A_{pq} \rightarrow A_p, rA_r, q: p, q, r \in Q\}$ to $R$
  2. Add $\{A_{qq} \rightarrow \varepsilon: q \in Q\}$ to $R$
  3. For all $p, r, s, q \in Q, a, b \in \Sigma_\varepsilon$ and $t \in \Gamma$ such that
     3.1 $(r, t) \in \delta(p, a, \varepsilon)$ and
Defining $G = (V, \Sigma, R, S)$

- $V = \{ A_{pq} : p, q \in Q \}$
  
  Idea: $A_{pq}$ will generate all strings that take $P$ from $p$ with an empty stack, to $q$ with an empty stack

- $S = A_{q_0, q_a}$

- Initially $R = \emptyset$ and
  
  1. Add $\{ A_{pq} \rightarrow A_{p,r}A_{r,q} : p, q, r \in Q \}$ to $R$
  2. Add $\{ A_{qq} \rightarrow \epsilon : q \in Q \}$ to $R$
  3. For all $p, r, s, q \in Q, a, b \in \Sigma_\epsilon$ and $t \in \Gamma$ such that
     3.1 $(r, t) \in \delta(p, a, \epsilon)$ and
     3.2 $(q, \epsilon) \in \delta(s, b, t)$
Defining $G = (V, \Sigma, R, S)$

- $V = \{ A_{pq} : p, q \in Q \}$
  Idea: $A_{pq}$ will generate all strings that take $P$ from $p$ with an empty stack, to $q$ with an empty stack

- $S = A_{q_0,q_a}$

- Initially $R = \emptyset$ and
  1. Add $\{ A_{pq} \rightarrow A_p, rA_r, q : p, q, r \in Q \}$ to $R$
  2. Add $\{ A_{qq} \rightarrow \varepsilon : q \in Q \}$ to $R$
  3. For all $p, r, s, q \in Q, a, b \in \Sigma_\varepsilon$ and $t \in \Gamma$ such that
     3.1 $(r, t) \in \delta(p, a, \varepsilon)$ and
     3.2 $(q, \varepsilon) \in \delta(s, b, t)$
Defining $G = (V, \Sigma, R, S)$

- $V = \{A_{pq}: p, q \in Q\}$
  
  Idea: $A_{pq}$ will generate all strings that take $P$ from $p$ with an empty stack, to $q$ with an empty stack

- $S = A_{q_0, q_a}$

- Initially $R = \emptyset$ and
  
  1. Add $\{A_{pq} \rightarrow A_{p,r,A_{r,q}}: p, q, r \in Q\}$ to $R$
  2. Add $\{A_{qq} \rightarrow \varepsilon: q \in Q\}$ to $R$
  3. For all $p, r, s, q \in Q, a, b \in \Sigma_\varepsilon$ and $t \in \Gamma$ such that
     - 3.1 $(r, t) \in \delta(p, a, \varepsilon)$ and
     - 3.2 $(q, \varepsilon) \in \delta(s, b, t)$

    add $A_{pq} \rightarrow aA_{r,s}b$ to $R$
Example PDA to CFG

\[
\begin{align*}
q_1 & \quad \varepsilon, \varepsilon \rightarrow \$ \\
q_2 & \quad 0, \varepsilon \rightarrow 0 \\
q_3 & \quad 1, 0 \rightarrow \varepsilon \\
q_4 & \quad \varepsilon, \$ \rightarrow \varepsilon \\
q_2, q_3 & \quad 0 \rightarrow \varepsilon \\
q_2, q_3 & \quad 1 \rightarrow \varepsilon \\
q_2, q_2 & \quad \varepsilon \rightarrow \varepsilon \\
q_1 & \quad \varepsilon \rightarrow \varepsilon
\end{align*}
\]
Example PDA to CFG

A_{q_1,q_4} \rightarrow A_{q_2,q_3}
A_{q_2,q_3} \rightarrow 0A_{q_2,q_3} 1
A_{q_2,q_3} \rightarrow 0A_{q_2,q_2} 1.
A_{q_2,q_2} \rightarrow \varepsilon.
PDA Computation corresponding to $A_{pq} \rightarrow A_{p,r} A_{r,q}$
PDA Computation corresponding to $A_{pq} \rightarrow aA_{r,s}b$
Claim: $\mathcal{L}(G) = \mathcal{L}(P)$

Claim 18

$A_{pq} \xrightarrow{*} w \in \Sigma^*$ iff $(q, \varepsilon) \in \hat{\delta}(p, w, \varepsilon)$
Claim: $\mathcal{L}(G) = \mathcal{L}(P)$

Claim 18

$A_{pq} \xrightarrow{*} w \in \Sigma^*$ iff $(q, \varepsilon) \in \hat{\delta}(p, w, \varepsilon)$

Proof by induction on the number of derivation rules/ transitions
A Short Summary

- Regular Languages \equiv Finite Automata.
- Context Free Languages \equiv Push Down Automata.
- Closure properties of regular languages and of CFLs.
- Most algorithmic problems for finite automata are solvable.
- Some algorithmic problems for finite automata are not solvable.
- Pumping lemmata for both classes of languages.
- There are additional languages out there.
The View Over The Horizon

- enumerable
- decidable
- context free
- regular