Computational Models - Lecture 6[1]

Handout Mode

Iftach Haitner and Ronitt Rubinfeld.

Tel Aviv University.

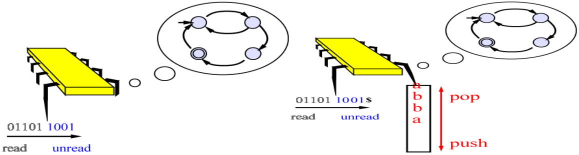
April 4/6, 2016

---

[1] 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 $a, b \to c$ 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 $a, b \to c$ 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 ε 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 $\widehat{\delta}(q, w, s)$ 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.

Model of Computation

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

Definition 1 (δ^*)

For $w \in \Sigma^*$ let $\widehat{\delta}(q, w, s)$ 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, $(q', s') \in \widehat{\delta}(q_0, w, \varepsilon)$ if after reading w (possibly with in-between ε 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 (δ^*)

For $w \in \Sigma^*$ let $\widehat{\delta}(q, w, s)$ 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, $(q', s') \in \widehat{\delta}(q_0, w, \varepsilon)$ if after reading w (possibly with in-between ε moves), M can find itself in state q' and stack value s'.

- M accepts $w \in \Sigma^*$ if $\exists q' \in \mathcal{F}$ such that $(q', t) \in \widehat{\delta}(q_0, w, \varepsilon)$ 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.

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 $\$$ onto stack.
2. When you see it again, stack is empty.

Example 3 – Palindrome

A palindrome is a string w satisfying $w = w^{\mathcal{R}}$.

- “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 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 $w = w^{\mathcal{R}}$.

- “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 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.

What the difference from $\{ww^R\}$?

A PDA for Palindromes

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.

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).

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.

PDA Languages

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

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

PDA Languages

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

- $\mathcal{L}_{\mathsf{PDA}} = \{\mathcal{L}(M) \colon M \text{ is a PDA}\}$

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

PDA Languages

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

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

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

- $\mathcal{L}_{\text{CFG}} \subseteq \mathcal{L}_{\text{PDA}}$?

PDA Languages

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

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

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

- $\mathcal{L}_{\text{CFG}} \subseteq \mathcal{L}_{\text{PDA}}$?
- $\mathcal{L}_{\text{PDA}} \subseteq \mathcal{L}_{\text{CFG}}$?

PDA Languages

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

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

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

- $\mathcal{L}_{\text{CFG}} \subseteq \mathcal{L}_{\text{PDA}}$?
- $\mathcal{L}_{\text{PDA}} \subseteq \mathcal{L}_{\text{CFG}}$?
- $\mathcal{L}_{\text{PDA}} = \mathcal{L}_{\text{CFG}}$!!!

Part II

Closure Properties

Simple Closure Properties of Context-Free Languages

Simple Closure Properties of Context-Free Languages

- CFL's are closed under

Simple Closure Properties of Context-Free Languages

- CFL’s are closed under
  - Union:

Simple Closure Properties of Context-Free Languages

- CFL's are closed under
    - Union:

Simple Closure Properties of Context-Free Languages

- CFL's are closed under
  - Union: $S \to S_1 \mid S_2$

Simple Closure Properties of Context-Free Languages

- CFL's are closed under
  - Union:$S \to S_1 \mid S_2$
  - Concatenation:

Simple Closure Properties of Context-Free Languages

- CFL's are closed under
  - Union: $S \to S_1 \mid S_2$
  - Concatenation:

Simple Closure Properties of Context-Free Languages

- CFL's are closed under
  - Union:$S \to S_1 \mid S_2$
  - Concatenation:$S \to S_1 S_2$

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:

Simple Closure Properties of Context-Free Languages

- CFL's are closed under
    - Union:$S \to S_1 \mid S_2$
    - Concatenation:$S \to S_1 S_2$
    - Star:

Simple Closure Properties of Context-Free Languages

- CFL's are closed under
  - Union: $S \to S_1 \mid S_2$
  - Concatenation: $S \to S_1 S_2$
  - Star: $S_{new} \to \varepsilon \mid S_{old} \mid S_{old} S_{new}$

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:$S_{new} \rightarrow \varepsilon \mid S_{old} \mid S_{old} S_{new}$
- What about complement and intersection?

Intersection idea?

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

Intersection

$$\begin{array}{ll} S_1 \to A_1 B_1 & \quad S_2 \to A_2 B_2 \\ A_1 \to 0A_1 1|\varepsilon & \quad A_2 \to 0A_2|\varepsilon \\ B_1 \to 2B_1|\varepsilon & \quad B_2 \to 1B_2 2|\varepsilon \end{array}$$

$$\mathcal{L}_1 = 0^n 1^n 2^* \qquad \mathcal{L}_2 = 0^* 1^n 2^n$$

Intersection

$$\begin{aligned} S_1 &\to A_1 B_1 \\ A_1 &\to 0A_1 1 \mid \varepsilon \\ B_1 &\to 2B_1 \mid \varepsilon \end{aligned} \qquad \begin{aligned} S_2 &\to A_2 B_2 \\ A_2 &\to 0A_2 \mid \varepsilon \\ B_2 &\to 1B_2 2 \mid \varepsilon \end{aligned}$$

$$\mathcal{L}_1 = 0^n 1^n 2^* \qquad \mathcal{L}_2 = 0^* 1^n 2^n$$

- $\mathcal{L}_1 \cap \mathcal{L}_2 = 0^n 1^n 2^n$

Intersection

$$\begin{aligned} S_1 &\to A_1 B_1 \\ A_1 &\to 0A_1 1 | \varepsilon \\ B_1 &\to 2B_1 | \varepsilon \end{aligned} \qquad \begin{aligned} S_2 &\to A_2 B_2 \\ A_2 &\to 0A_2 | \varepsilon \\ B_2 &\to 1B_2 2 | \varepsilon \end{aligned}$$

$$\mathcal{L}_1 = 0^n 1^n 2^* \qquad \mathcal{L}_2 = 0^* 1^n 2^n$$

- $\mathcal{L}_1 \cap \mathcal{L}_2 = 0^n 1^n 2^n$
- $\mathcal{L}_1$ and $\mathcal{L}_2$ are CFLs (why?),

Intersection

$$\begin{aligned} S_1 &\to A_1B_1 \\ A_1 &\to 0A_11|\varepsilon \\ B_1 &\to 2B_1|\varepsilon \end{aligned} \qquad \begin{aligned} S_2 &\to A_2B_2 \\ A_2 &\to 0A_2|\varepsilon \\ B_2 &\to 1B_22|\varepsilon \end{aligned}$$

$$\mathcal{L}_1 = 0^n1^n2^* \qquad \mathcal{L}_2 = 0^*1^n2^n$$

- $\mathcal{L}_1 \cap \mathcal{L}_2 = 0^n1^n2^n$
- $\mathcal{L}_1$ and $\mathcal{L}_2$ are CFLs (why?),
- But $\mathcal{L}_1 \cap \mathcal{L}_2$ is not a CFL.

Intersection

$$\begin{aligned} S_1 &\to A_1 B_1 \\ A_1 &\to 0A_1 1 | \varepsilon \\ B_1 &\to 2B_1 | \varepsilon \end{aligned} \qquad \begin{aligned} S_2 &\to A_2 B_2 \\ A_2 &\to 0A_2 | \varepsilon \\ B_2 &\to 1B_2 2 | \varepsilon \end{aligned}$$

$$\mathcal{L}_1 = 0^n 1^n 2^* \qquad \mathcal{L}_2 = 0^* 1^n 2^n$$

- $\mathcal{L}_1 \cap \mathcal{L}_2 = 0^n 1^n 2^n$
- $\mathcal{L}_1$ 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.

Intersection

$$\begin{aligned} S_1 &\to A_1B_1 \\ A_1 &\to 0A_1 1|\varepsilon \\ B_1 &\to 2B_1|\varepsilon \end{aligned} \qquad \begin{aligned} S_2 &\to A_2B_2 \\ A_2 &\to 0A_2|\varepsilon \\ B_2 &\to 1B_2 2|\varepsilon \end{aligned}$$

$$\mathcal{L}_1 = 0^n1^n2^* \qquad \mathcal{L}_2 = 0^*1^n2^n$$

- $\mathcal{L}_1 \cap \mathcal{L}_2 = 0^n1^n2^n$
- $\mathcal{L}_1$ 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?

When CFL Intersects Regular Language

- Are the context free languages closed under intersection with a regular language?
- That is, if $\mathcal{L}_1$ is context free languages, and $\mathcal{L}_2$ is regular, must $\mathcal{L}_1 \cap \mathcal{L}_2$ be context free languages?

When CFL Intersects Regular Language

- Are the context free languages closed under intersection with a regular language?
- That is, if $\mathcal{L}_1$ is context free languages, and $\mathcal{L}_2$ is regular, must $\mathcal{L}_1 \cap \mathcal{L}_2$ be context free languages?
- YES!

When CFL Intersects Regular Language

- Are the context free languages closed under intersection with a regular language?
- That is, if $\mathcal{L}_1$ is context free languages, and $\mathcal{L}_2$ is regular, must $\mathcal{L}_1 \cap \mathcal{L}_2$ be context free languages?
- YES!
  - Run PDA $\mathcal{L}_1$ and DFA $\mathcal{L}_2$ “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 $\mathcal{L}_1$ is context free languages, and $\mathcal{L}_2$ is regular, must $\mathcal{L}_1 \cap \mathcal{L}_2$ be context free languages?
- YES!
  - Run PDA $\mathcal{L}_1$ and DFA $\mathcal{L}_2$ “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 $\mathcal{L} = \{w \in \{0,1,2\}^* \colon \#_0(w) = \#_1(w) = \#_2(w)\}$ context free?

An Application:

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

An Application:

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

An Application:

- Is $\mathcal{L} = \{w \in \{0,1,2\}^* \colon \#_0(w) = \#_1(w) = \#_2(w)\}$ context free?
    - $0^*1^*2^*$ is regular.
    - Context free languages intersected with a regular languages are context free.

An Application:

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

$$\mathcal{L}_1 \cap \mathcal{L}_2 = \overline{\overline{\mathcal{L}_1} \cup \overline{\mathcal{L}_2}}$$

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.

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?)
- 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 2ℓ for which for all $1 \leq i \leq \ell$, $w_i = w_{i+\ell}$.

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 2ℓ for which for all $1 \leq i \leq \ell$, $w_i = w_{i+\ell}$.
  - $\overline{\mathcal{L}}$ are strings for which either (1) $|w|$ is odd, or (2) $|w|$ is even and there exists i, for which $w_i \neq w_{i+\ell}$.

Complementation cont.

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

Complementation cont.

- For any $y \in \overline{\mathcal{L}}$, either
    - y's length is odd, or

Complementation cont.

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

Complementation cont.

- For any $y \in \overline{\mathcal{L}}$, either
  - y's length is odd, or
  - y's length is even, 2ℓ, and $\exists i \geq 1$ such that $y_i \neq y_{\ell+i}$.
- It suffice to construct a PDA/CFG for $\overline{\mathcal{L}}_{even}$ – the even length members of $\overline{\mathcal{L}}$ (why?)

Complementation cont.

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

Complementation cont.

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

Complementation cont.

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

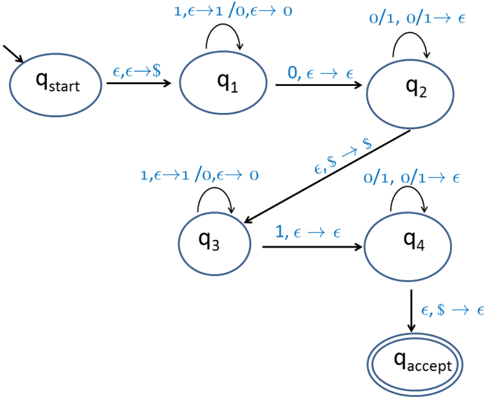
Complementation cont.

- For any $y \in \overline{\mathcal{L}}$, either
    - y's length is odd, or
    - y's length is even, 2ℓ, and $\exists i \geq 1$ such that $y_i \neq y_{\ell+i}$.
- It suffice to construct a PDA/CFG for $\overline{\mathcal{L}}_{even}$ – the even length members of $\overline{\mathcal{L}}$ (why?)
- Let $\overline{\mathcal{L}}^{\sigma}_{even} = \{\{0,1\}^k \sigma \{0,1\}^j \{0,1\}^k \overline{\sigma} \{0,1\}^j \colon k, j \geq 0\}$
- Note that $\overline{\mathcal{L}}_{even} = \overline{\mathcal{L}}^{0}_{even} \cup \overline{\mathcal{L}}^{1}_{even}$
- and that $\overline{\mathcal{L}}^{\sigma}_{even} = \{\{0,1\}^k \sigma \{0,1\}^k \{0,1\}^j \overline{\sigma} \{0,1\}^j \colon k, j \geq 0\}$
- CFG for $\overline{\mathcal{L}}^{0}_{even}$
  $S \rightarrow AB$
  $A \rightarrow CAC \mid 0$
  $B \rightarrow CBC \mid 1$
  $C \rightarrow 0 \mid 1$

A PDA for $\overline{\mathcal{L}}^{0}_{even}$

A PDA for $\overline{\mathcal{L}}^{0}_{even}$

Idea: Guess $k, j \geq 0$, and accept w if it is of the form:
$\{0,1\}^k 0 \{0,1\}^k \{0,1\}^j 1 \{0,1\}^j$



Homomorphism and Inverse Homomorphism

- *Homomorphism:* replaces each letter with a word

Homomorphism and Inverse Homomorphism

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

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: $h(1) = aba$, $h(0) = aa$
  $h(010) = aa\,aba\,aa$
  $\mathcal{L}_1 = \{0^n1^n | n \geq 0\}$, $h(\mathcal{L}_1) = \{a^{2n}(aba)^n | n \geq 0\}$.
- 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: $h(1) = aba$, $h(0) = aa$
  $h(010) = aa\,aba\,aa$
  $\mathcal{L}_1 = \{0^n 1^n | n \geq 0\}$, $h(\mathcal{L}_1) = \{a^{2n}(aba)^n | n \geq 0\}$.
- Claim: Assuming that $\mathcal{L}$ is a CFL, then so is $h(\mathcal{L})$
- Proof? use the grammar

Homomorphism and Inverse Homomorphism

- *Homomorphism:* replaces each letter with a word
- Example: $h(1) = aba$, $h(0) = aa$
  $h(010) = aa\,aba\,aa$
  $\mathcal{L}_1 = \{0^n 1^n | n \geq 0\}$, $h(\mathcal{L}_1) = \{a^{2n}(aba)^n | n \geq 0\}$.
- Claim: Assuming that $\mathcal{L}$ is a CFL, then so is $h(\mathcal{L})$
- Proof? use the grammar
- *Inverse homomorphism:* $h^{-1}(w) = \{x \colon h(x) = w\}$,
  $h^{-1}(\mathcal{L}) = \{x \colon h(x) \in \mathcal{L}\}$

Homomorphism and Inverse Homomorphism

- *Homomorphism:* replaces each letter with a word
- Example: $h(1) = aba$, $h(0) = aa$
  $h(010) = aa\,aba\,aa$
  $\mathcal{L}_1 = \{0^n 1^n | n \geq 0\}$, $h(\mathcal{L}_1) = \{a^{2n}(aba)^n | n \geq 0\}$.
- Claim: Assuming that $\mathcal{L}$ is a CFL, then so is $h(\mathcal{L})$
- Proof? use the grammar
- *Inverse homomorphism:* $h^{-1}(w) = \{x \colon h(x) = w\}$,
  $h^{-1}(\mathcal{L}) = \{x \colon h(x) \in \mathcal{L}\}$
- Example: $\mathcal{L}_2 = \{a^n b^n a^i | n, i \geq 0\}$, $h^{-1}(\mathcal{L}_2) = \{10^i, 0^i | i \geq 0\}$.

Homomorphism and Inverse Homomorphism

- *Homomorphism:* replaces each letter with a word
- Example: $h(1) = aba$, $h(0) = aa$
  $h(010) = aa\,aba\,aa$
  $\mathcal{L}_1 = \{0^n 1^n | n \geq 0\}$, $h(\mathcal{L}_1) = \{a^{2n}(aba)^n | n \geq 0\}$.
- Claim: Assuming that $\mathcal{L}$ is a CFL, then so is $h(\mathcal{L})$
- Proof? use the grammar
- *Inverse homomorphism:* $h^{-1}(w) = \{x \colon h(x) = w\}$,
  $h^{-1}(\mathcal{L}) = \{x \colon h(x) \in \mathcal{L}\}$
- 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 $\mathcal{L}(G) = \emptyset$?

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

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).

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?)

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?

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 $A \to U_1 U_2 \dots U_k$ 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 $\mathcal{L}(G) = \emptyset$.

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 $A \to U_1 U_2 \dots U_k$ 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 $A \to U_1 U_2 \ldots U_k$ 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?

CFGs Fullness

Question 5

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

CFGs Fullness

Question 5

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

- We just saw an algorithm to determine, given a CFG G, whether $\mathcal{L}(G) = \emptyset$

CFGs 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?

CFGs 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.

CFGs 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.

CFGs 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.

CFGs 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 $A \to U_1 U_2 \ldots U_k$ 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 $\mathcal{L}(G) = \emptyset$.
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?

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 a no cycles.

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 a no cycles.

Correctness?

CFGs Inherent Ambiguity

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.

CFGs Inherent Ambiguity

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

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

The CFG–PDA Equivalence Theorem

Theorem 12

$\mathcal{L}_{\text{PDA}} = \mathcal{L}_{\text{CFG}}$ *: 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 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.

CFL $\Longrightarrow$ PDA

Lemma 13

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

CFL $\implies$ PDA

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 $\mathcal{L}$

CFL $\implies$ PDA

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.

CFL $\Longrightarrow$ PDA

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.

CFL $\Longrightarrow$ PDA

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?

CFL $\implies$ PDA

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?

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

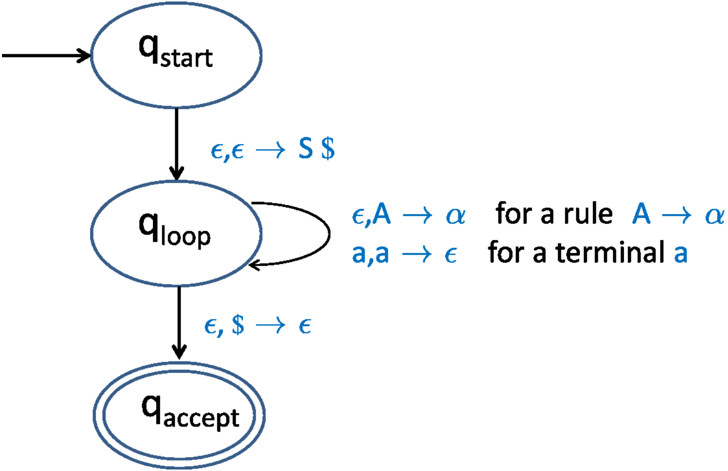
   Does it change the derived language?

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 $A \to \alpha$ 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*

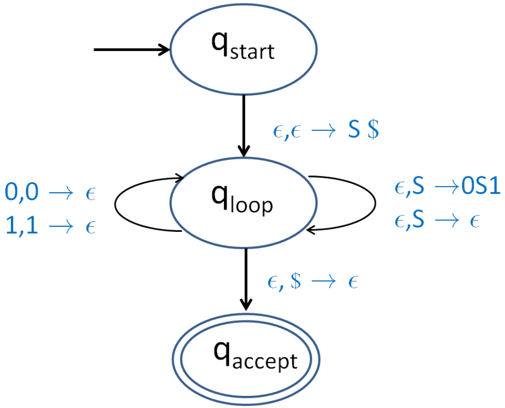


Example

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 $(q_{loop}, \alpha_2\$) \in \widehat{\delta}(q_{loop}, \alpha_1, \$)$.

(note that α_1 is made of terminals, α_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 $(q_{loop}, \alpha_2\$) \in \widehat{\delta}(q_{loop}, \alpha_1, \$)$.

(note that α_1 is made of terminals, α_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 $(q_{loop}, \alpha_2\$) \in \widehat{\delta}(q_{loop}, \alpha_1, S\$)$

Proof by induction on the number of derivations steps used to yield α from S.

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

Proof by induction on the number of derivations steps used to yield α from S.

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

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

Proof by induction on the number of derivations steps used to yield α from S.

- 1 derivation steps: hence there is a rule $S \to \alpha$. Thus $(q_{loop}, \alpha\$) \in \widehat{\delta}(q_{loop}, \varepsilon, S\$)$, and the proof follows for $\alpha_1 = \varepsilon$ and $\alpha_2 = \alpha$.
- Assume $S \xrightarrow{*} \alpha$ in $k > 1$ derivation steps, and let α' be the string derived by the first $(k-1)$ steps.

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

Proof by induction on the number of derivations steps used to yield α from S.

- 1 derivation steps: hence there is a rule $S \rightarrow \alpha$. Thus $(q_{loop}, \alpha\$) \in \widehat{\delta}(q_{loop}, \varepsilon, S\$)$, and the proof follows for $\alpha_1 = \varepsilon$ and $\alpha_2 = \alpha$.
- Assume $S \xrightarrow{*} \alpha$ in $k > 1$ derivation steps, and let α' be the string derived by the first $(k-1)$ steps.
- By i.h $\alpha' = \alpha'_1\alpha'_2$ such that $(q_{loop}, \alpha'_2\$) \in \widehat{\delta}(q_{loop}, \alpha'_1, S\$)$

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

Proof by induction on the number of derivations steps used to yield α from S.

- 1 derivation steps: hence there is a rule $S \to \alpha$. Thus $(q_{loop}, \alpha\$) \in \widehat{\delta}(q_{loop}, \varepsilon, S\$)$, and the proof follows for $\alpha_1 = \varepsilon$ and $\alpha_2 = \alpha$.
- Assume $S \xrightarrow{*} \alpha$ in $k > 1$ derivation steps, and let α' be the string derived by the first $(k-1)$ steps.
- By i.h $\alpha' = \alpha'_1\alpha'_2$ such that $(q_{loop}, \alpha'_2\$) \in \widehat{\delta}(q_{loop}, \alpha'_1, S\$)$
- Write $\alpha'_2 = w_1 A w_2$ where A is the left most variable in α'_2.

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

Proof by induction on the number of derivations steps used to yield α from S.

- 1 derivation steps: hence there is a rule $S \to \alpha$. Thus $(q_{loop}, \alpha\$) \in \widehat{\delta}(q_{loop}, \varepsilon, S\$)$, and the proof follows for $\alpha_1 = \varepsilon$ and $\alpha_2 = \alpha$.
- Assume $S \xrightarrow{*} \alpha$ in $k > 1$ derivation steps, and let α' be the string derived by the first $(k-1)$ steps.
- By i.h $\alpha' = \alpha'_1\alpha'_2$ such that $(q_{loop}, \alpha'_2\$) \in \widehat{\delta}(q_{loop}, \alpha'_1, S\$)$
- Write $\alpha'_2 = w_1 A w_2$ where A is the left most variable in α'_2.
- The k'th derivation step replaces this occurrence of A with a string t (why?)

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

Proof by induction on the number of derivations steps used to yield α from S.

- 1 derivation steps: hence there is a rule $S \to \alpha$. Thus $(q_{loop}, \alpha\$) \in \widehat{\delta}(q_{loop}, \varepsilon, S\$)$, and the proof follows for $\alpha_1 = \varepsilon$ and $\alpha_2 = \alpha$.
- Assume $S \xrightarrow{*} \alpha$ in $k > 1$ derivation steps, and let α' be the string derived by the first $(k - 1)$ steps.
- By i.h $\alpha' = \alpha'_1\alpha'_2$ such that $(q_{loop}, \alpha'_2\$) \in \widehat{\delta}(q_{loop}, \alpha'_1, S\$)$
- Write $\alpha'_2 = w_1 A w_2$ where A is the left most variable in α'_2.
- The k'th derivation step replaces this occurrence of A with a string t (why?)
- It is easy to see that $(q_{loop}, tw_2\$) \in \widehat{\delta}(q_{loop}, \alpha'_1 w_1, S\$)$.

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

Proof by induction on the number of derivations steps used to yield α from S.

- 1 derivation steps: hence there is a rule $S \to \alpha$. Thus $(q_{loop}, \alpha\$) \in \widehat{\delta}(q_{loop}, \varepsilon, S\$)$, and the proof follows for $\alpha_1 = \varepsilon$ and $\alpha_2 = \alpha$.
- Assume $S \xrightarrow{*} \alpha$ in $k > 1$ derivation steps, and let α' be the string derived by the first $(k-1)$ steps.
- By i.h $\alpha' = \alpha'_1\alpha'_2$ such that $(q_{loop}, \alpha'_2\$) \in \widehat{\delta}(q_{loop}, \alpha'_1, S\$)$
- Write $\alpha'_2 = w_1Aw_2$ where A is the left most variable in α'_2.
- The k'th derivation step replaces this occurrence of A with a string t (why?)
- It is easy to see that $(q_{loop}, tw_2\$) \in \widehat{\delta}(q_{loop}, \alpha'_1 w_1, S\$)$.
- To complete the proof take $\alpha_1 = \alpha'_1 w_1$ and $\alpha_2 = tw_2$.

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

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

Proof by induction on the number of steps used by P to process α_1.

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

Proof by induction on the number of steps used by P to process α_1.

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

$\alpha = \alpha_1\alpha_2$ such that $(q_{loop}, \alpha_2\$) \in \widehat{\delta}(q_{loop}, \alpha_1, \$) \implies S \overset{*}{\to} \alpha$

Proof by induction on the number of steps used by P to process α_1.

- A single step: $\alpha_1 = \varepsilon$ and $\alpha_2 = S\$$, and the proof follows since $S \overset{*}{\to} S$.
- Assume α_1 was processed in $k > 1$ steps, and let α_1' and α_2' be the input string read and the stack value *before* the last step

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

Proof by induction on the number of steps used by P to process α_1.

- A single step: $\alpha_1 = \varepsilon$ and $\alpha_2 = S\$$, and the proof follows since $S \xrightarrow{*} S$.
- Assume α_1 was processed in $k > 1$ steps, and let α_1' and α_2' be the input string read and the stack value *before* the last step
- Note that $(q_{loop}, \alpha_2'\$) \in \widehat{\delta}(q_{loop}, \alpha_1', \$)$.

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

Proof by induction on the number of steps used by P to process α_1.

- A single step: $\alpha_1 = \varepsilon$ and $\alpha_2 = S\$$, and the proof follows since $S \xrightarrow{*} S$.
- Assume α_1 was processed in $k > 1$ steps, and let α_1' and α_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 $S \xrightarrow{*} \alpha' = \alpha_1'\alpha_2'$.

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

Proof by induction on the number of steps used by P to process α_1.

- A single step: $\alpha_1 = \varepsilon$ and $\alpha_2 = S\$$, and the proof follows since $S \xrightarrow{*} S$.
- Assume α_1 was processed in $k > 1$ steps, and let α_1' and α_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 $S \xrightarrow{*} \alpha' = \alpha_1'\alpha_2'$.
- In case the k'th move of P is reading an input character, then $\alpha_1\alpha_2 = \alpha_1'\alpha_2'$, and therefore $S \xrightarrow{*} \alpha_1\alpha_2$

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

Proof by induction on the number of steps used by P to process α_1.

- A single step: $\alpha_1 = \varepsilon$ and $\alpha_2 = S\$$, and the proof follows since $S \xrightarrow{*} S$.
- Assume α_1 was processed in $k > 1$ steps, and let α_1' and α_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 $S \xrightarrow{*} \alpha' = \alpha_1'\alpha_2'$.
- 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 $A \rightarrow t \in R$

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

Proof by induction on the number of steps used by P to process α_1.

- A single step: $\alpha_1 = \varepsilon$ and $\alpha_2 = S\$$, and the proof follows since $S \xrightarrow{*} S$.
- Assume α_1 was processed in $k > 1$ steps, and let α_1' and α_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 $S \xrightarrow{*} \alpha' = \alpha_1'\alpha_2'$.
- 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 $A \to t \in R$
- Hence $S \xrightarrow{*} \alpha_1\alpha_2$

PDA $\implies$ CFL

Lemma 17

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

PDA $\Longrightarrow$ CFL

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

PDA $\Longrightarrow$ CFL

Lemma 17

$\mathcal{L}_{\mathsf{PDA}} \subseteq \mathcal{L}_{\mathsf{CFG}}$: *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 wlg. that:

PDA $\Longrightarrow$ CFL

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 wlg. that:

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

PDA $\Longrightarrow$ CFL

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 wlg. that:

- A single accepting state $q_a \in F$.
- P empties the stack before accepting

PDA $\implies$ CFL

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 wlg. that:

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

PDA $\Longrightarrow$ CFL

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 wlg. that:

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

PDA $\implies$ CFL

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 wlg. 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 $A_{pq} \to aA_{rs}b$, 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 $A_{pq} \to aA_{rs}b$, 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 $A_{pq} \to A_{pr}A_{rq}$ where r is intermediate state and P has empty stack.

Defining $G = (V, \Sigma, R, S)$

- $V = \{A_{pq} \colon p, q \in Q\}$

Defining $G = (V, \Sigma, R, S)$

- $V = \{A_{pq} \colon p, q \in Q\}$

  Idea: A_{pq} will generate all strings that take P from p with an empty stack, to q with an empty stack

Defining $G = (V, \Sigma, R, S)$

- $V = \{A_{pq} \colon p, q \in Q\}$

  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 $G = (V, \Sigma, R, S)$

- $V = \{A_{pq} \colon p, q \in Q\}$

  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 $G = (V, \Sigma, R, S)$

- $V = \{A_{pq} \colon p, q \in Q\}$

  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 $\{A_{pq} \to A_{p,r}A_{r,q} \colon p, q, r \in Q\}$ to R

Defining $G = (V, \Sigma, R, S)$

- $V = \{A_{pq} \colon p, q \in Q\}$

  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 $\{A_{pq} \to A_{p,r}A_{r,q} \colon p, q, r \in Q\}$ to R
  2. Add $\{A_{qq} \to \varepsilon \colon q \in Q\}$ to R

Defining $G = (V, \Sigma, R, S)$

- $V = \{A_{pq} \colon p, q \in Q\}$

  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 $\{A_{pq} \to A_{p,r}A_{r,q} \colon p, q, r \in Q\}$ to R
  2. Add $\{A_{qq} \to \varepsilon \colon q \in Q\}$ to R
  3. For all $p, r, s, q \in Q$, $a, b \in \Sigma_\varepsilon$ and $t \in \Gamma$ such that

Defining $G = (V, \Sigma, R, S)$

- $V = \{A_{pq} \colon p, q \in Q\}$

  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 $\{A_{pq} \to A_{p,r} A_{r,q} \colon p, q, r \in Q\}$ to R
  2. Add $\{A_{qq} \to \varepsilon \colon q \in Q\}$ 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 $G = (V, \Sigma, R, S)$

- $V = \{A_{pq} \colon p, q \in Q\}$

  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 $\{A_{pq} \to A_{p,r}A_{r,q} \colon p, q, r \in Q\}$ to R
  2. Add $\{A_{qq} \to \varepsilon \colon q \in Q\}$ 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 $G = (V, \Sigma, R, S)$

- $V = \{A_{pq} \colon p, q \in Q\}$

  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 $\{A_{pq} \to A_{p,r}A_{r,q} \colon p, q, r \in Q\}$ to R
  2. Add $\{A_{qq} \to \varepsilon \colon q \in Q\}$ 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** $(q, \varepsilon) \in \delta(s, b, t)$

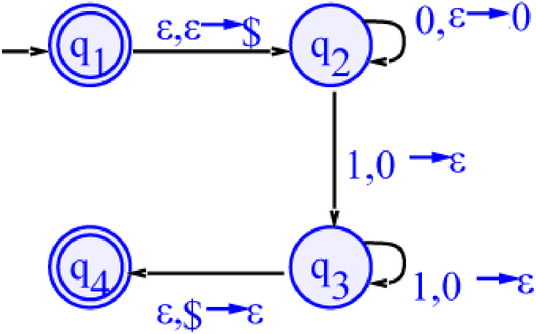
Defining $G = (V, \Sigma, R, S)$

- $V = \{A_{pq} \colon p, q \in Q\}$

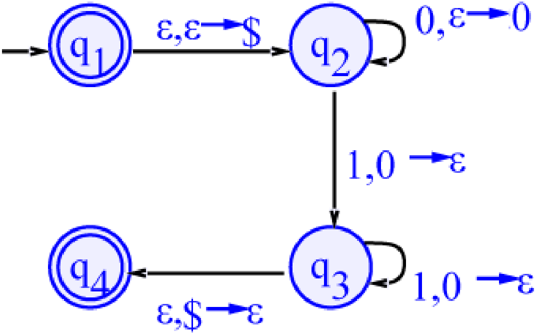
  Idea: A_{pq} will generate all strings that take P from p with an empty stack, to q with an empty stack
- $S = A_{q_0, q_a}$
- Initially $R = \emptyset$ and
  1. Add $\{A_{pq} \to A_{p,r} A_{r,q} \colon p, q, r \in Q\}$ to R
  2. Add $\{A_{qq} \to \varepsilon \colon q \in Q\}$ to R
  3. For all $p, r, s, q \in Q$, $a, b \in \Sigma_\varepsilon$ 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 $A_{pq} \to a A_{r,s} b$ to R

Example PDA to CFG

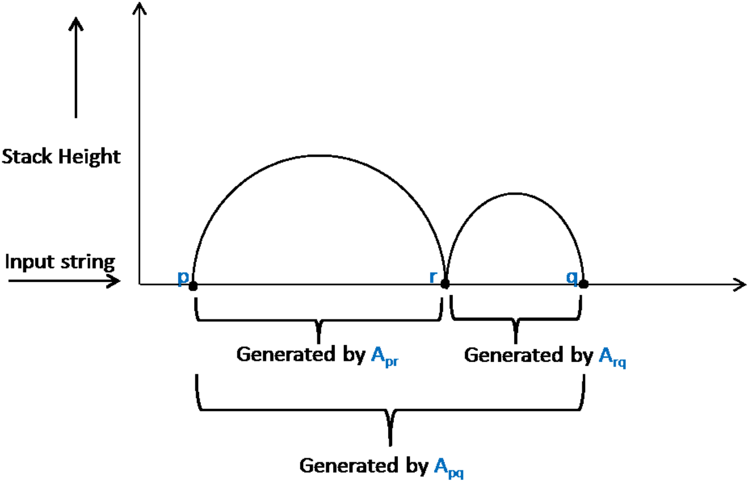


Example PDA to CFG

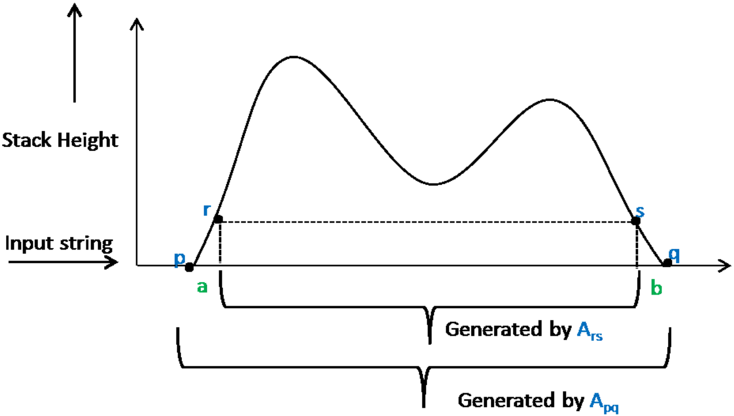


$A_{q_1,q_4} \to A_{q_2,q_3}$
$A_{q_2,q_3} \to 0A_{q_2,q_3}1$
$A_{q_2,q_3} \to 0A_{q_2,q_2}1$.
$A_{q_2,q_2} \to \varepsilon$.

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 $(q, \varepsilon) \in \widehat{\delta}(p, w, \varepsilon)$

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

Claim 18

$A_{pq} \overset{*}{\rightarrow} w \in \Sigma^*$ iff $(q, \varepsilon) \in \widehat{\delta}(p, w, \varepsilon)$

Proof by induction on the number of derivation rules/ transitions

A Short Summary

- Regular Languages ≡ Finite Automata.
- Context Free Languages ≡ 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