CS 374A Midterm 1 Review

ACM @ UIUC

September 28, 2024

Disclaimers and Logistics

- **Disclaimer:** Some of us are CAs, but we have not seen the exam. We have no idea what the questions are. However, we've taken the course and reviewed Sariel's previous exams, so we have **suspicions** as to what the questions will be like.
- This review session is being recorded. Recordings and slides will be distributed on EdStem after the end.
- **Agenda:** We'll quickly review all topics likely to be covered, then go through a practice exam, then review individual topics by request.
  - Questions are designed to be written in the same style as Sariel's previous exams but to be *slightly* harder, so don't worry if you don't get everything right away!
- Please let us know if we're going too fast/slow, not speaking loud enough/speaking too loud, etc.
- If you have a question anytime during the review session, please ask! Someone else almost surely has a similar question.
- We'll provide a feedback form at the end of the session.

Induction

Template

Let x be an *arbitrary* <OBJECT>. Assume for all k s.t. k is smaller than x (by <ORDERING PROPERTY>), that $P(k)$ holds.

If $x =$ <MINIMAL OBJECT>, then . . . , so $P(x)$ holds ← base case

If $x \neq$ <MINIMAL OBJECT>, then . . . , so by IH, . . . , so $P(x)$ holds.

Thus, in all cases, $P(x)$ holds.

Induction

k → k+1 → weak

k<n → n → strong

Template

Let x be an *arbitrary* <OBJECT>. Assume for all k s.t. k is smaller than x (by <ORDERING PROPERTY>), that $P(k)$ holds. IH

If $x =$ <MINIMAL OBJECT>, then . . . , so $P(x)$ holds BC

If $x \neq$ <MINIMAL OBJECT>, then . . . , so by IH, . . . , so $P(x)$ holds. IS

Thus, in all cases, $P(x)$ holds.

Some tips:

- **Always use strong induction**. All weak inductive proofs can be re-written to use strong induction with minimal changes, and the extra assumption can make your life significantly easier.
- **Write out your IH, base case, and inductive step out explicitly.** Doing so will help you avoid getting confused, and will help you avoid losing points.
- If you're performing induction on a recursive definition (strings, CFLs, etc.), generally, your inductive step will consist of one step of the recursion, and then will use IH.

Regular Languages/Expressions

- Built inductively on 3 operations:

Regular Languages/Expressions

- Built inductively on 3 operations:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$

Regular Languages/Expressions

- Built inductively on 3 operations:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$

Regular Languages/Expressions

- Built inductively on 3 operations:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$
  - () are used to group expressions

Regular Languages/Expressions

- Built inductively on 3 operations:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$
  - () are used to group expressions
  - (implicit) concatenation operator: $L(r_1 r_2) = \{xy : x \in L_1, y \in L_2\}$

Regular Languages/Expressions

- Built inductively on 3 operations:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$
  - () are used to group expressions
  - (implicit) concatenation operator: $L(r_1 r_2) = \{xy : x \in L_1, y \in L_2\}$
- Closed under Union ($\cup$), intersection ($\cap$), concatenation ($\cdot$), kleene star (*), complement (^{C}), set difference ($\setminus$), and reverse (^{R})

Regular Languages/Expressions

- Built inductively on 3 operations:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$
  - () are used to group expressions
  - (implicit) concatenation operator: $L(r_1 r_2) = \{xy : x \in L_1, y \in L_2\}$
- Closed under Union ($\cup$), intersection ($\cap$), concatenation ($\cdot$), kleene star (*), complement (^{C}), set difference ($\setminus$), and reverse (^{R})
  - ... but only finitely many applications of these operations

$0^n 1^n$

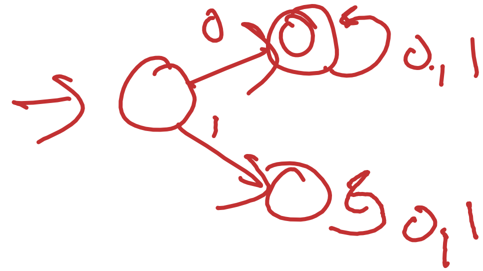
Regular Languages/Expressions

- Built inductively on 3 operations:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$
  - () are used to group expressions
  - (implicit) concatenation operator: $L(r_1 r_2) = \{xy : x \in L_1, y \in L_2\}$
- Closed under Union ($\cup$), intersection ($\cap$), concatenation ($\cdot$), kleene star (*), complement (^{C}), set difference ($\setminus$), and reverse (^{R})
  - ... but only finitely many applications of these operations
- If trying to guess whether or not a language is regular, think about memory. When processing a string through a DFA, you only need to know which state you're currently in, and do not need to look forwards/backwards in the string.
  - Implementing a DFA/NFA in code only requires $O(1)$ memory
  - If your checker program needs to count something without bound, the language you're checking isn't regular.

Regular Languages/Expressions

- Built inductively on 3 operations:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$
  - () are used to group expressions
  - (implicit) concatenation operator: $L(r_1 r_2) = \{xy : x \in L_1, y \in L_2\}$
- Closed under Union ($\cup$), intersection ($\cap$), concatenation ($\cdot$), kleene star (*), complement (^{C}), set difference ($\setminus$), and reverse (^{R})
  - ... but only finitely many applications of these operations
- If trying to guess whether or not a language is regular, think about memory. When processing a string through a DFA, you only need to know which state you're currently in, and do not need to look forwards/backwards in the string.
  - Implementing a DFA/NFA in code only requires $O(1)$ memory
  - If your checker program needs to count something without bound, the language you're checking isn't regular.
- **Regex Design Tips:** If you don't know where to start, try giving examples for strings that are in the language and strings that aren't. Look for patterns and try to build components around those patterns, then combine into something that represents the full language. Make sure to test and modify for edge cases. Explain, in English, each part of your regular expression with a short sentence. Does the explanation match the language?

DFAs/NFAs



- DFAs defined by *state set* Q, *accepting set* $A \subseteq Q$, *input alphabet* Σ, *start state* $s \in Q$, and *transition function* $\delta : Q \times \Sigma \to Q$
- NFAs allow for "trying" multiple transitions at the same time or transitioning without reading in (ϵ-transitions), accepts if there is a path to an accepting state. Transition function thereby changes to $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$
  - Power-set construction to convert from NFA to DFA- in theory exponential-time but used in practice.
- **Tips for creating DFA/NFAs**: Break down your language into smaller patterns, and figure out what you need to store as state for each part. Make sure you clearly define all components. A drawing or transition table is just as valid as a $(Q, A, \Sigma, s, \delta)$ definition.

DFAs/NFAs

- DFAs defined by *state set* Q, *accepting set* $A \subseteq Q$, *input alphabet* Σ, *start state* $s \in Q$, and *transition function* $\delta : Q \times \Sigma \to Q$
- NFAs allow for "trying" multiple transitions at the same time or transitioning without reading in (ϵ-transitions), accepts if there is a path to an accepting state. Transition function thereby changes to $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$
  - Power-set construction to convert from NFA to DFA- in theory exponential-time but used in practice.
- **Tips for creating DFA/NFAs**: Break down your language into smaller patterns, and figure out what you need to store as state for each part. Make sure you clearly define all components. A drawing or transition table is just as valid as a $(Q, A, \Sigma, s, \delta)$ definition.

Product Constructions

Given some languages $L_1, \ldots, L_n$ we want a DFA that accepts strings w satisfying $f(w \in L_1, \ldots, w \in L_n)$ where f is some logical function. Create a DFA/NFA for L using the following *rough* format:

- $Q = Q_1 \times \cdots \times Q_n$
- $\delta'(q_1, \ldots, q_n) = (\delta_1(q_1), \ldots, \delta_2(q_2))$
- $s = (s_1, \ldots, s_n)$
- $A' = \{\text{convert } f \text{ into a set expression}\}$

3 DFAs
M1, M2, M3
acc only if 1 is true
A' = {(x, y, z) : One of x ∈ A1, y ∈ A2, z ∈ A3

Fooling Sets

- DFAs only care about which state you’re in, and not how you got there
    - If two strings result in the same DFA state, any additional suffix added to both will also result in both strings being in the same state.

Fooling Sets

- DFAs only care about which state you're in, and not how you got there
  - If two strings result in the same DFA state, any additional suffix added to both will also result in both strings being in the same state.
  - Thus, if we have w, w', and we know that there exists a **distinguishing suffix** z s.t. $wz \in L, w'z \notin L$, then w, w' must be in *different* states for *any* DFA that accepts L

Fooling Sets

- DFAs only care about which state you're in, and not how you got there
  - If two strings result in the same DFA state, any additional suffix added to both will also result in both strings being in the same state.
  - Thus, if we have w, w', and we know that there exists a **distinguishing suffix** z s.t. $wz \in L, w'z \notin L$, then w, w' must be in *different* states for *any* DFA that accepts L
- A **fooling set** is a set of strings where there exists a distinguishing suffix between every pair of strings
- Myhill-Nerode: min DFA size = max fooling set size

Fooling Sets

- DFAs only care about which state you're in, and not how you got there
  - If two strings result in the same DFA state, any additional suffix added to both will also result in both strings being in the same state.
  - Thus, if we have w, w', and we know that there exists a **distinguishing suffix** z s.t. $wz \in L, w'z \notin L$, then w, w' must be in *different* states for *any* DFA that accepts L
- A **fooling set** is a set of strings where there exists a distinguishing suffix between every pair of strings
- Myhill-Nerode: min DFA size = max fooling set size
  - Thus, languages with infinite fooling sets are *not* regular

Fooling Sets

- DFAs only care about which state you're in, and not how you got there
  - If two strings result in the same DFA state, any additional suffix added to both will also result in both strings being in the same state.
  - Thus, if we have w, w', and we know that there exists a **distinguishing suffix** z s.t. $wz \in L, w'z \notin L$, then w, w' must be in *different* states for *any* DFA that accepts L
- A **fooling set** is a set of strings where there exists a distinguishing suffix between every pair of strings
- Myhill-Nerode: min DFA size = max fooling set size
  - Thus, languages with infinite fooling sets are *not* regular
- If you see the need to keep track of something without bound, you can create a fooling set around the part where you count up.

$0^n 1^n \rightarrow F = \{0^n\}$

Fooling Sets

- DFAs only care about which state you're in, and not how you got there
  - If two strings result in the same DFA state, any additional suffix added to both will also result in both strings being in the same state.
  - Thus, if we have w, w', and we know that there exists a **distinguishing suffix** z s.t. $wz \in L, w'z \notin L$, then w, w' must be in *different* states for *any* DFA that accepts L
- A **fooling set** is a set of strings where there exists a distinguishing suffix between every pair of strings
- Myhill-Nerode: min DFA size = max fooling set size
  - Thus, languages with infinite fooling sets are *not* regular
- If you see the need to keep track of something without bound, you can create a fooling set around the part where you count up.
- **If you see divisibility, think primes!** All primes are coprime, so primality provides for an infinite set with easier construction of distinguishing suffixes.

$0^a 1^b \quad a | b \to 0^p$

Fooling Sets

- DFAs only care about which state you're in, and not how you got there
  - If two strings result in the same DFA state, any additional suffix added to both will also result in both strings being in the same state.
  - Thus, if we have w, w', and we know that there exists a **distinguishing suffix** z s.t. $wz \in L, w'z \notin L$, then w, w' must be in *different* states for *any* DFA that accepts L
- A **fooling set** is a set of strings where there exists a distinguishing suffix between every pair of strings
- Myhill-Nerode: min DFA size = max fooling set size
  - Thus, languages with infinite fooling sets are *not* regular
- If you see the need to keep track of something without bound, you can create a fooling set around the part where you count up.
- **If you see divisibility, think primes!** All primes are coprime, so primality provides for an infinite set with easier construction of distinguishing suffixes.
- If you're using strings of the form $1^k, 0^p$, etc. when sampling elements of your fooling set a^i, a^j, it's completely fine to assume WLOG that $j > i$, but nothing about the underlying structure of i and j. If you want to put in such a restriction, you should instead restrict your fooling set further.

Language Transforms

- Used to prove that regularity is closed under some function f (if L is regular, then $f(L)$ is regular).
- **General Format:** Given a DFA M that accepts L, create an NFA M' that accepts $f(L)$.

Language Transforms

- Used to prove that regularity is closed under some function f (if L is regular, then $f(L)$ is regular).
- **General Format:** Given a DFA M that accepts L, create an NFA M' that accepts $f(L)$.
- **General Strategy:** Apply non-determinism to guess the future behavior of the DFA that you want to simulate.

$\text{skipevens}(L) : \{a_1 a_3 a_5 \cdots : a_1 a_2 a_3 a_4 a_5 \cdots \in L\}$

NFA

$Q' = Q \cup \{1\,\text{past}\}$

$A' = A \cup \{1\,\text{past}\}$

$s' = s$

$\delta'(x, a) = \{\delta(\delta(x, a), a') \mid a' \in \Sigma\} \cup \{1\,\text{past if } \delta(x, a) \in A\}$

Language Transforms

- Used to prove that regularity is closed under some function f (if L is regular, then $f(L)$ is regular).
- **General Format:** Given a DFA M that accepts L, create an NFA M' that accepts $f(L)$.
- **General Strategy:** Apply non-determinism to guess the future behavior of the DFA that you want to simulate.

Make sure you're going in the right direction!

If you see the format $f(L) = \{k(w) : w \in L\}$, your modified NFA should be trying to *undo* k, while if you see the format $f(L) = \{w : k(w) \in L\}$, your modified NFA should be trying to *apply* k. Mixing these up is the most common mistake we see on homeworks/exams.

In some cases, only one direction is possible. For example, $\texttt{un-palin}(L) : \{w : ww^R \in L\}$ has a transformation construction, but $\texttt{palin}(L) = \{ww^R : w \in L\}$ is irregular for some L.

Context-Free Languages/Grammars

- Formally, a context-free grammar is defined by *nonterminals/variables* V, *terminals/symbols* T, *productions* P, and the *start symbol* S. Each production rule in P looks like $A \rightarrow \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$.

Context-Free Languages/Grammars

- Formally, a context-free grammar is defined by *nonterminals/variables* V, *terminals/symbols* T, *productions* P, and the *start symbol* S. Each production rule in P looks like $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$.
- For example, consider $V = \{S\}$, $T = \{0, 1\}$, $P = \{S \to \epsilon, S \to 0S1\}$. (You can abbreviate this to $P = \{S \to \epsilon \mid 0S1\}$.) What language is this?

Intuition

CFGs "build" strings, going from the outside in; you can choose rules to add characters on the left/right.
Alternatively, CFGs "peel back" strings, removing characters from the left/right until nothing is left.

111000

1S0 → 11S00 → 111S000 → 111ε000

11000

Context-Free Languages/Grammars

- Formally, a context-free grammar is defined by *nonterminals/variables* V, *terminals/symbols* T, *productions* P, and the *start symbol* S. Each production rule in P looks like $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$.
- For example, consider $V = \{S\}$, $T = \{0, 1\}$, $P = \{S \to \epsilon, S \to 0S1\}$. (You can abbreviate this to $P = \{S \to \epsilon \mid 0S1\}$.) What language is this?

Intuition

CFGs "build" strings, going from the outside in; you can choose rules to add characters on the left/right.
Alternatively, CFGs "peel back" strings, removing characters from the left/right until nothing is left.

- CFLs only closed under union, kleene star, and concatenation. CFLs are *not* closed under intersection or complement.

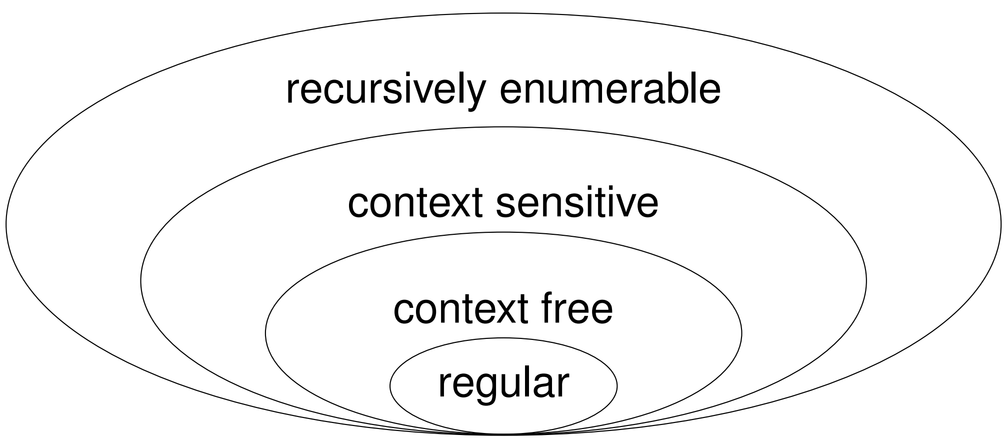
Context-Free Languages/Grammars

- Formally, a context-free grammar is defined by *nonterminals/variables* V, *terminals/symbols* T, *productions* P, and the *start symbol* S. Each production rule in P looks like $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$.
- For example, consider $V = \{S\}$, $T = \{0, 1\}$, $P = \{S \to \epsilon, S \to 0S1\}$. (You can abbreviate this to $P = \{S \to \epsilon \mid 0S1\}$.) What language is this?

Intuition

CFGs "build" strings, going from the outside in; you can choose rules to add characters on the left/right.
Alternatively, CFGs "peel back" strings, removing characters from the left/right until nothing is left.

- CFLs only closed under union, kleene star, and concatenation. CFLs are *not* closed under intersection or complement.



Short Answer T/F I

For each of the following, either mark true or false and give a one-sentence explanation of your answer. (These are intentionally tricky)

(a) For all languages L, if L is irregular, then L has a finite fooling set.

True, $\emptyset$

(b) If M is a minimal DFA that decides a language L, and running M on strings x and y result in states q and q', respectively, where $q \neq q'$, then there exists a distinguishing suffix between x and y in L.

True, otherwise combine q, q'

(c) Consider a language $L \subsetneq 0^*$. If L contains two strings i, j s.t. $\gcd(|i|, |j|) = 1$, then L^* is regular.

True, is 0^* - finite # of strs.

(d) The language $L = \{0^i 1^j 0^k : i = j \wedge k \equiv i \pmod{374}\}$ is context-free.

True, 374 prod rules

(e) For context-free languages L_1, L_2, the language $L = (L_1^* \cdot L_2) \cup (L_1 \cdot L_2^*)$ is context-free.

True, fin. many closure props.

Short Answer T/F II

For each of the following, either mark true or false and give a one-sentence explanation of your answer. (These are intentionally tricky)

(f) The language $\{xx^R y : x, y \in \{0,1\}^*\}$ is regular.

True $= (0+1)^*$

(g) If L is regular, then $\texttt{self-fold}(L) = \{a_1 a_n a_2 a_{n-1} \cdots a_{\lceil \frac{n}{2} \rceil} : a_1 a_2 \cdots a_n \in L\}$ is regular.

True, trans back/forth

(h) Consider the language $L = \{1^x 2^y 3^z : y = x + z\}$. There exists a distinguishing suffix between the strings 1112222223 and 2223.

False → #1 bal = #2 bal, so adding more 3s unch.

(i) Let M_1, M_2 be arbitrary NFAs with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then $L(M_1) \cap L(M_2) = \varnothing$.

False →

(j) Consider an infinite set of regular languages $L_1, L_2, \ldots$ s.t. $L_{i-1} \subseteq L_i$. The language $\cup_{i=1}^{\infty} L_i$ is context free.

False. 0 10, 001100, 000111000, ... → $0^n 1^n 0^n$

Regular or Not?

For each of the following languages, either *prove* that the language is regular, or *prove* that it is not regular (**Hint:** exactly one of the two languages is regular)

- $\{1xyx : x, y \in \{0,1\}^*\}$
- $\{x1xy : x, y \in \{0,1\}^*\}$

Language Transformations

Let $\Sigma = \{0, 1\}$.

(a) Given a DFA $M = (Q, A, \Sigma, s, \delta)$ that decides a regular language L, provide an NFA for the language $L' = \{w \cdot 0^n : w \in L \wedge n \geq 0\}$

$Q' = Q \cup \{z\}$

$A' = \{z\}$

$s' = s$

$\delta'(q, \varepsilon) = \{z\}$ if $q \in A$

$\delta'(z, 0) = \{z\}$

$\delta'(q, c) = \{\delta(q, c)\}$

$A' = A \cup \{z\}$

$\delta'(q, 0) = \{\delta(q, 0), z\}$ if $q \in A$

$\delta'(z, 1) = \{\}$

Language Transformations

Let $\Sigma = \{0, 1\}$.

(a) Given a DFA $M = (Q, A, \Sigma, s, \delta)$ that decides a regular language L, provide an NFA for the language $L' = \{w \cdot 0^n : w \in L \wedge n \geq 0\}$

(b) Given DFAs $M_1 = (Q_1, A_1, \Sigma, s_1, \delta_1)$, $M_2 = (Q_2, A_2, \Sigma, s_2, \delta_2)$ that decide regular languages L_1, L_2, respectively, describe an NFA to decide $L = \{w_1 \otimes w_2 : w_1 \in L_1 \wedge w_2 \in L_2 \wedge |w_1| = |w_2|\}$, where $\otimes$ is the bitwise XOR operator.

$$Q = Q_1 \times Q_2 \qquad A = A_1 \times A_2$$

$$s = (s_1, s_2)$$

$$\delta((q_1, q_2), 0) = \{(\delta_1(q_1, 0), \delta_2(q_2, 0)), (\delta_1(q_1, 1), \delta_2(q_2, 1))\}$$

$$\delta((q_1, q_2), 1) = \{(\delta_1(q_1, 0), \delta_2(q_2, 1)), (\delta_1(q_1, 1), \delta_2(q_2, 0))\}$$

Language Transformations

Let $\Sigma = \{0, 1\}$.

(a) Given a DFA $M = (Q, A, \Sigma, s, \delta)$ that decides a regular language L, provide an NFA for the language $L' = \{w \cdot 0^n : w \in L \wedge n \geq 0\}$ = ZEROPAD(L)

(b) Given DFAs $M_1 = (Q_1, A_1, \Sigma, s_1, \delta_1)$, $M_2 = (Q_2, A_2, \Sigma, s_2, \delta_2)$ that decide regular languages L_1, L_2, respectively, describe an NFA to decide $L = \{w_1 \otimes w_2 : w_1 \in L_1 \wedge w_2 \in L_2 \wedge |w_1| = |w_2|\}$, where $\otimes$ is the bitwise XOR operator. = XOR(L_1, L_2)

(c) Use your answers from parts (a) and (b) to prove that the bitwise XOR of two regular languages (zero-padding the shorter string on the right) is regular.

XOR(L_1, ZEROPAD(L_2)) ∪ XOR(ZEROPAD(L_1), L_2) — by closure property

$Q = (Q_1 \cup \{z\}) \times (Q_2 \cup \{z\})$

$s = (s_1, s_2)$

$A = \{(q_1, q_2) : q_1 \in A_1, q_2 \in A_2\}$ $\{(z, q_2) : q_2 \in A_2$, $(q_1, z) : q_1 \in A_1\}$

$\delta((q_1, q_2), 0) = \{(\delta_1(q_1, 0), \delta_2(q_2, 0)), (\delta_1(q_1, 1), \delta_2(q_2, 1))\}$

$\delta((q_1, q_2), \varepsilon) = \{(z, q_2)\}$ for $q_1 \in A$

$\delta((z, q_2), 0) = \{(z, \delta(\ldots))\}$

DFAs/NFAs/Regexes

With $\Sigma = \{0, 1\}$,

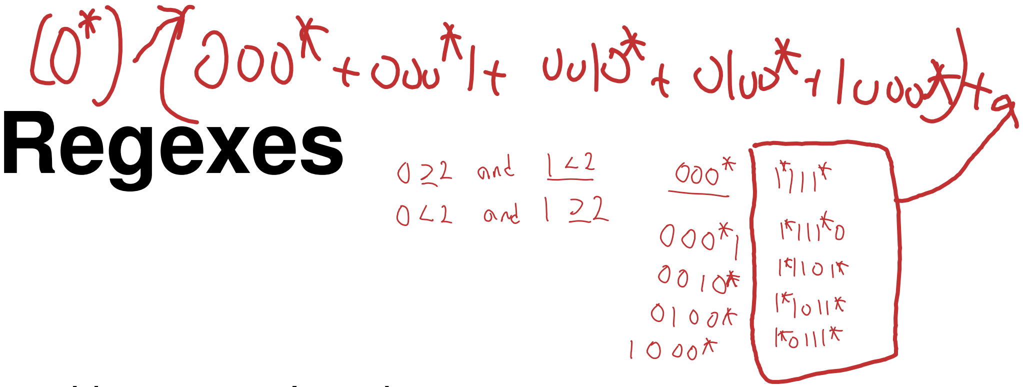
(a) Write a regex for strings with no even-length runs.

1 0 1110

$(1(11)^* \; 0(00)^*)^*$

$(0(00)^* + \epsilon)$

$(1(11)^* + \epsilon)$

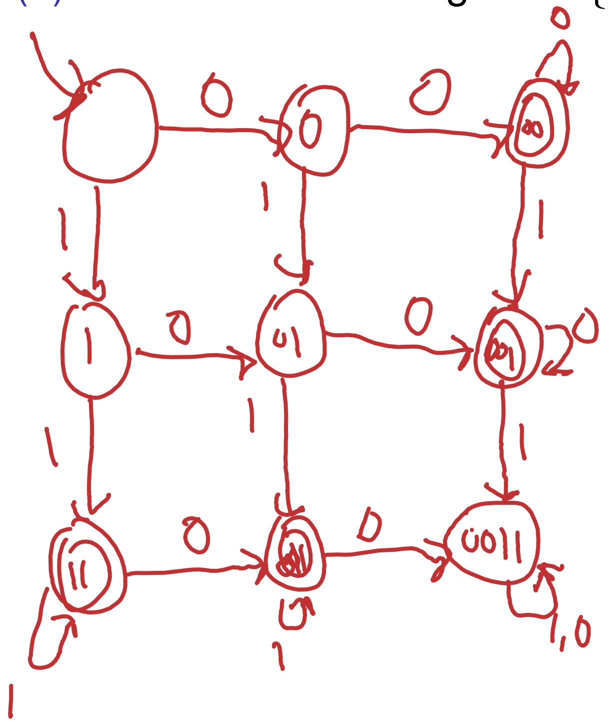


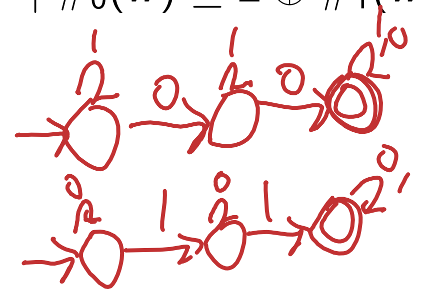
DFAs/NFAs/Regexes

With $\Sigma = \{0, 1\}$,

(a) Write a regex for strings with no even-length runs.

(b) Write a DFA *and* regex for $\{w \in \Sigma^* \mid \#_0(w) \geq 2 \oplus \#_1(w) \geq 2\}$.





CFLs

palindrome

$S \to 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$

Show that the following languages are context-free by providing grammars.

(a) $\{ww^R : w \in \{0,1\}^* \wedge |ww^R| \equiv 1 \pmod 3\}$

$S \to 0A0 \mid 1A1 \mid 0 \mid 1$: palindrome length 1 mod 3

$A \to 0B0 \mid 1B1$: 2 mod 3

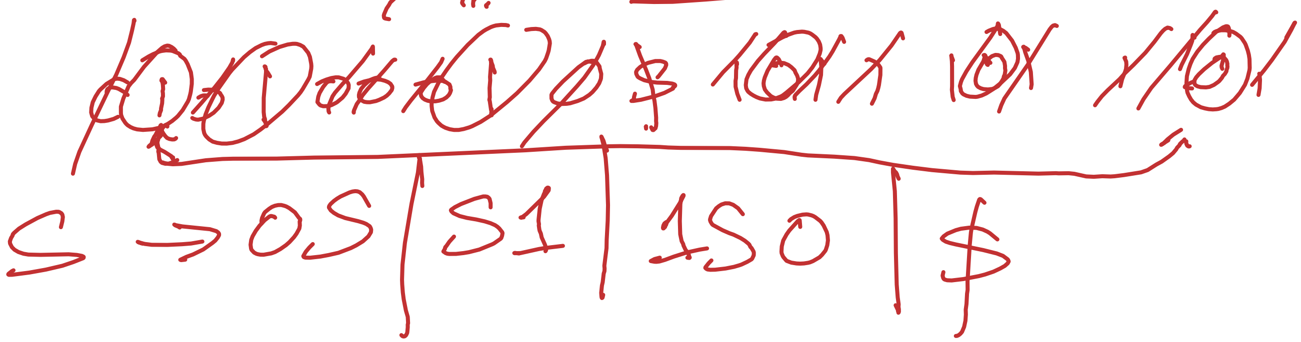
$B \to 0S0 \mid 1S1 \mid \varepsilon$: 0 mod 3

CFLs

Show that the following languages are context-free by providing grammars.

(a) $\{ww^R : w \in \{0,1\}^* \wedge |ww^R| \equiv 1 \pmod 3\}$

(b) $\{x\$y : x, y \in \{0,1\}^* \wedge \#(1,x) = \#(0,y)\}$



CFLs

Show that the following languages are context-free by providing grammars.

(a) $\{ww^R : w \in \{0,1\}^* \wedge |ww^R| \equiv 1 \pmod 3\}$

(b) $\{x\$y : x, y \in \{0,1\}^* \wedge \#(1,x) = \#(0,y)\}$

(c) $\{0^x 1^y 2^z : x - y = z\}$ $x = y + z$

$0^{y+z} 1^y 2^z$

$S \to 0S2 \mid A$

$A \to 0A1 \mid \varepsilon$

Feedback

go.acm.illinois.edu/374A_feedback