Computer Language Theory

Chapter 1: Regular Languages

Last updated 2/26/21

Chapter 1.1: Finite Automata

What is a Computer?

- Not a simple question to answer precisely
  - Computers are quite complicated
- We start with a computational model
  - Different models will have different features and may match a real computer better in some ways and worse in others
- Our first model is the finite state machine or finite automata

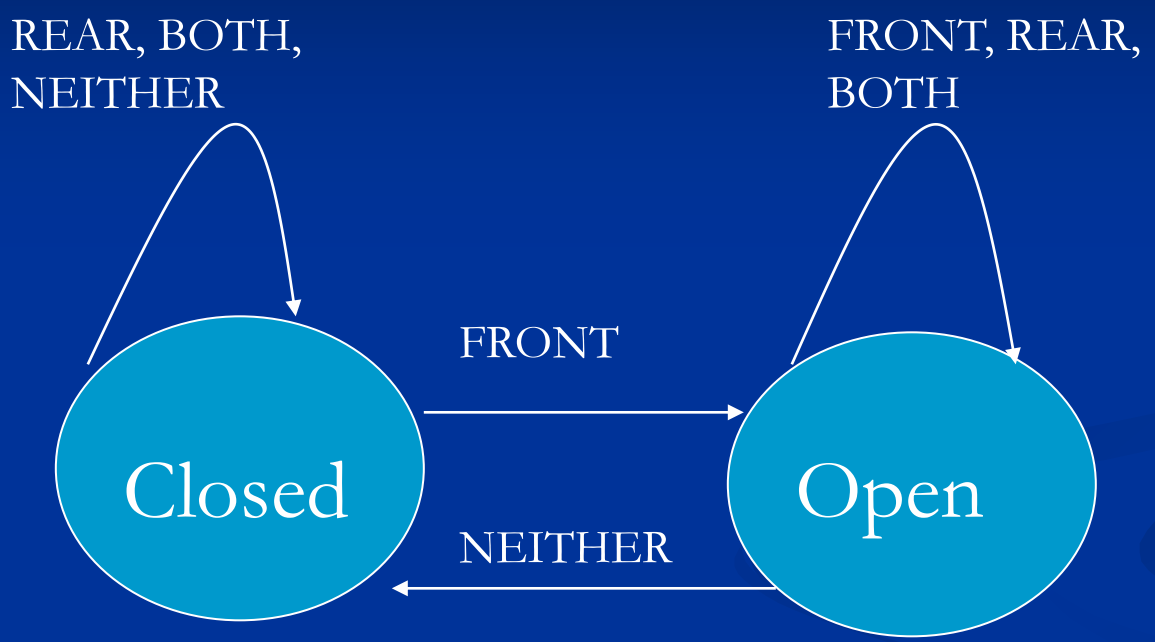
Finite Automata

- Models of computers with extremely limited memory
  - Many simple computers have extremely limited memories and are in fact finite state machines
  - Can you name any? Hint: several are in this building but have nothing specifically to do with our department
    - Vending machine
    - Elevator
    - Thermostat
    - Automatic door at supermarket

Automatic Door

- What is the desired behavior? Describe the actions and then list the states.
  - Person approaches, door should open
  - Door should stay open while person going thru
  - Door should shut if no one near doorway
  - States are open and closed
- More details about automatic door
  - Front pad   Door   Rear Pad
  - Describe behavior now
    - Hint: action depends not just on what happens, but what state you are currently in
    - If you walk thru door should stay open when you are on rear pad
    - But if door is closed and someone steps on rear pad, door does not open

Automatic Door cont.

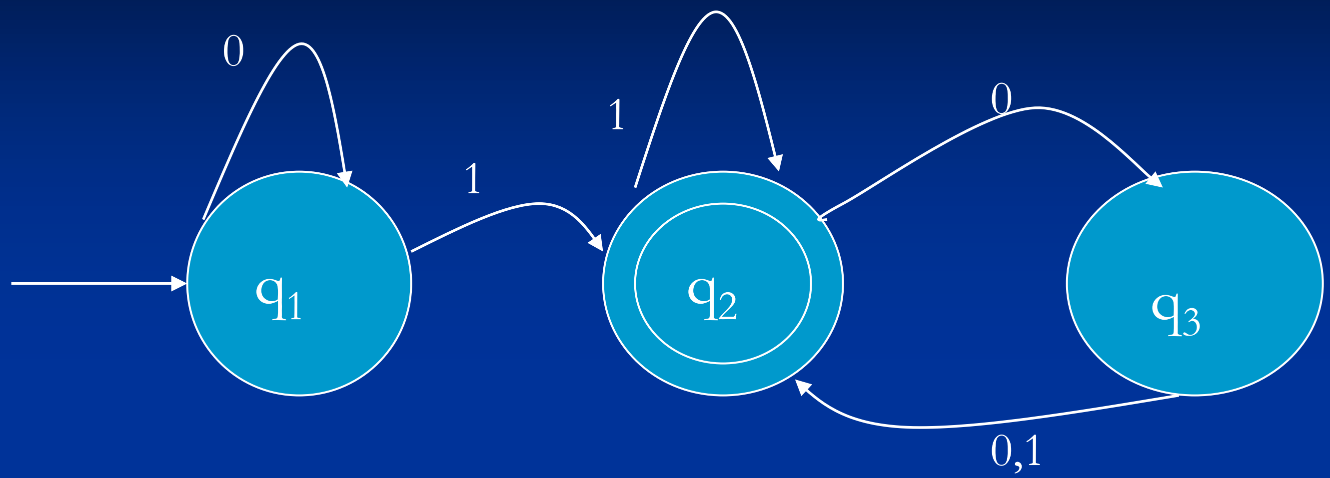


	NEITHER	FRONT	REAR	BOTH
CLOSED	CLOSED	OPEN	CLOSED	CLOSED
OPEN	CLOSED	OPEN	OPEN	OPEN

More on Finite Automata

- How many bits of data does this FSM store?
  - 1 bit: open or closed
- What about state information for elevators, thermostats, vending machines, etc?
- FSM used in speech processing, optical character recognition, etc.
- Have you implemented FSM? What?
  - I have implemented network protocols and expert systems for diagnosing telecommunication equipment problems

A finite automata M1

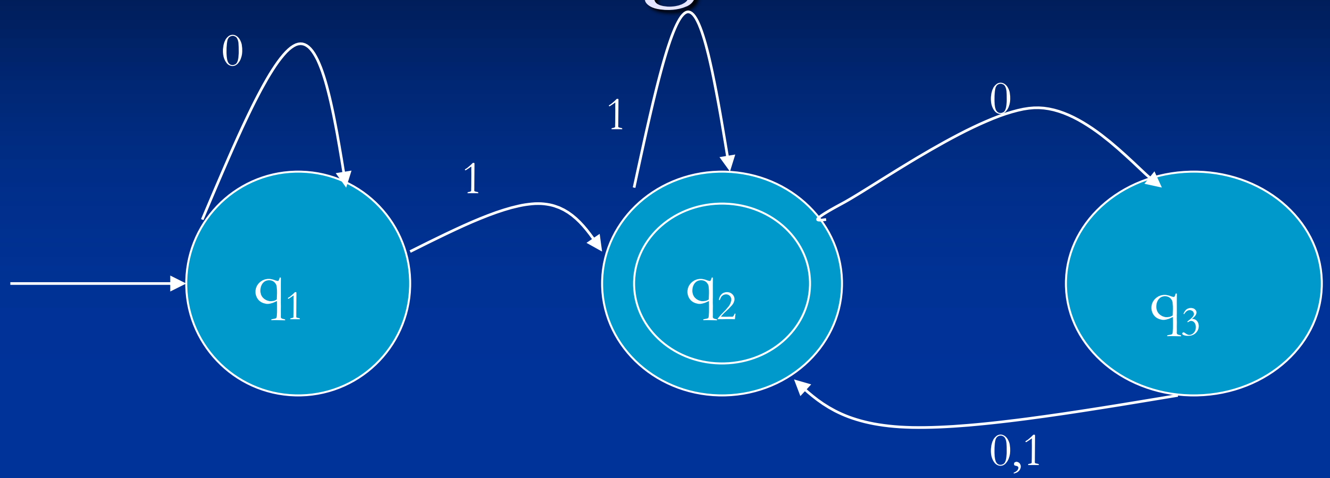


- A finite automata M1 with 3 states
  - We see the state diagram
    - Start state q1, accept state q2 (double circle), and several transitions
  - If a string like 1101 will accept if ends in accept state or else reject. What will it do?
  - Can you describe all string that this model will accept?
    - It will accept all strings ending in a 1 and any string with an even number of 0's following the last 1

Formal Definition of Finite Automata

- A finite automata is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$
  - Q is a finite set called states
  - Σ is a finite set called the alphabet
  - $\delta : Q \times \Sigma \rightarrow Q$ is the transition function
  - $q_0 \in Q$ is the start state
  - $F \subseteq Q$ is the set of accept states

Describe M1 using Formal Definition



- M1 = (Q, Σ, δ, q0, F)
  - Q = {q1, q2, q3}
  - Σ = {0,1}
  - q1 is the start state
  - F = {q2}

	0	**1**
q1	q1	q2
q2	q3	q2
q3	q2	q2

Transition function δ

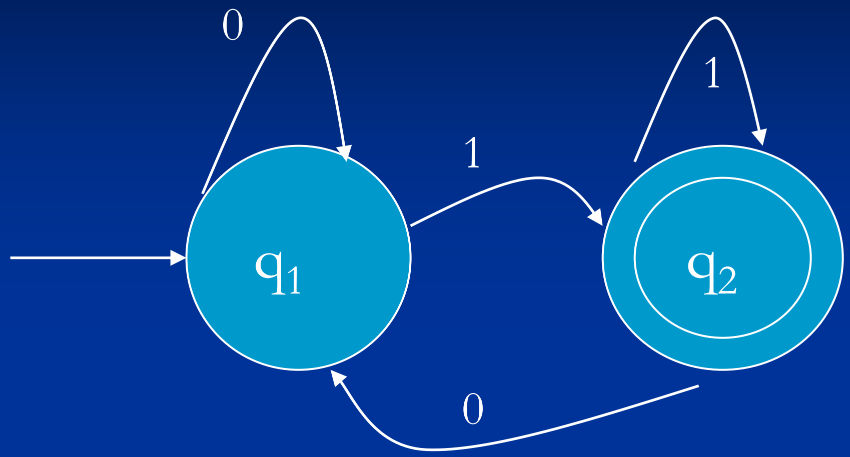
The Language of M1

- If A is the set of all strings that a machine M accepts, then A is the language of M
  - L(M) = A
  - We also say that M recognizes A or M accepts A
- A machine may accept many strings, but only one language
- Convention: M accepts string and recognizes a language

What is the Language of M1?

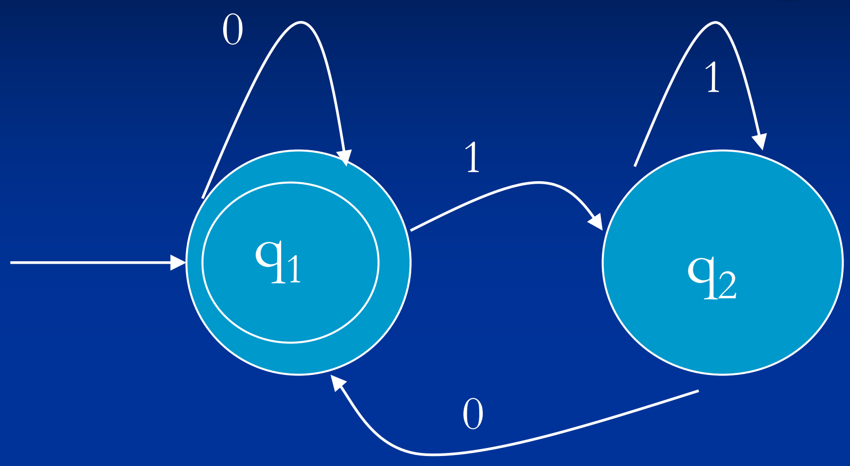
- L(M1) = A or M1 recognizes A
- What is A?
  - A = {w | …….}
  - A = {w| w contains at least one 1 and an even number of 0's follows the last 1}

What is the Language of M2?



- M2 = {{q1,q2}, {0,1}, δ, q1, {q2}}
  - I leave δ as an exercise
  - What is the language of M2?
    - L(M2) = {w| ? }
    - L(M2) = {w| w ends in a 1}

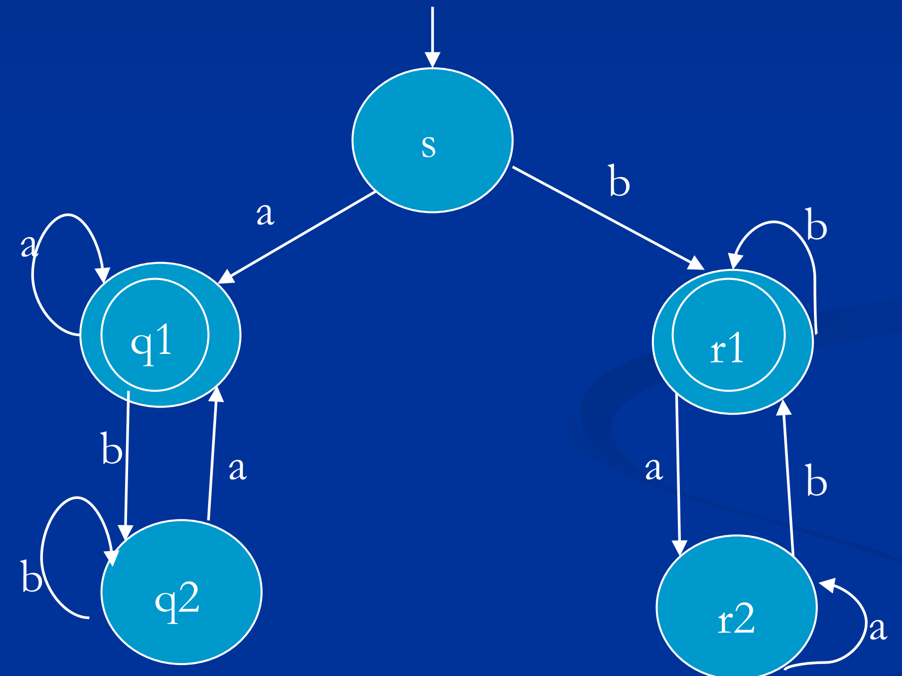
What is the Language of M3?



- M3 is M2 with different accept state
- What is the language of M3?
  - L(M3) = {w| ? }
  - L(M3) = {w| w ends in 0} [Not quite right! Why?]
  - L(M3) = {w| w is the empty string ε or ends in 0}

What is the Language of M4?

- M4 is a 5 state automata (Figure 1.12 on page 38)



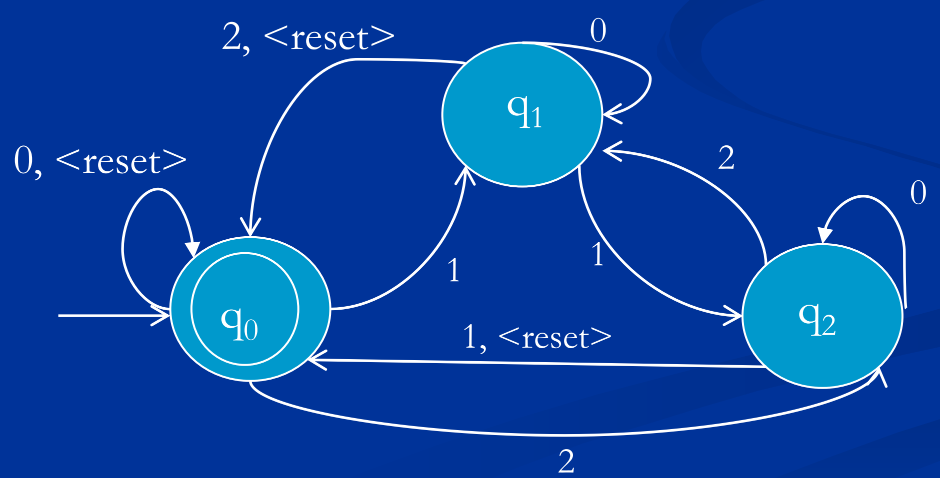
What language does M4 accept?

What is the Language of M4?

- What does M4 accept?
  - All strings that start and end with a or start and end with b
  - More simply, language is all string starting and ending with the same symbol
    - Note that length of 1 is okay

Construct M5 to do Modulo Arithmetic

- Let Σ = {RESET, 0, 1, 2}
- Construct M5 to accept a string only if the sum of each input symbol is a multiple of 3 and RESET sets the sum back to 0 (1.13, page 39)



Now Generalize M5

- Generalize M5 to accept if sum of symbols is a multiple of *i* instead of 3 (with same Σ)
  - How many states are needed? What is the start and accept state?
  - $(\{q0, q1, q2, q3, \ldots, q_{i-1}\}, \{0,1,2,RESET\}, \delta, q_0, \{q_0\})$
    - $\delta_i(q_j, 0) = q_j$
    - $\delta_i(q_j, 1) = q_k$ where k=j+1 modulo i
    - $\delta_i(q_j, 2) = q_k$ where k=j+2 modulo i
    - $\delta_i(q_j, RESET) = q_o$
- Note: as long as *i* is finite, we are okay and only need finite memory (# of states)
- Could you generalize on $\Sigma = \{1, 2, 3, \ldots k\}$?

Formal Definition of Accept

- Definition of M accepting a string:
  - Let M = (Q, Σ, δ, q0, F) be a finite automata and let $w=w_1w_2 \ldots w_n$ be a string where $w_i \in \Sigma$.
  - Then M accepts w if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with 3 conditions
    - $r_0=q_0$
    - $\delta(r_i, w_{i+1}) = r_{i+1}$, for i =0, 1, …, n-1
    - $r_n \in F$
    - We start in q_0, end in accept state, and input symbols yield path from start to accept that follows transition table

Regular Languages

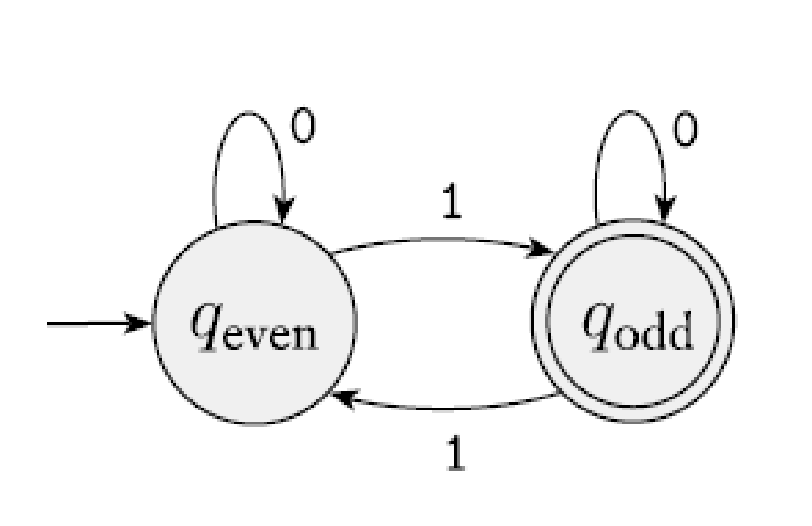
- Definition: A language is called a regular language if some finite automata recognizes it
  - That is, all strings in a regular language are accepted by some finite automata
  - Why should you expect proofs by construction coming up in your next homework?

Designing Finite Automata

- You will need to design FA's to accept a language
- Strategies
  - Determine what you need to remember (the states)
    - E.g., how many states to determine even vs. odd number of 1's?
    - What does each state represent? Use meaningful state labels.
  - Set the start and finish states based on what each state represents
  - Assign the transitions
  - Check solution: should accept $w \in L$ and not accept $w \notin L$
  - Be careful about the empty string!

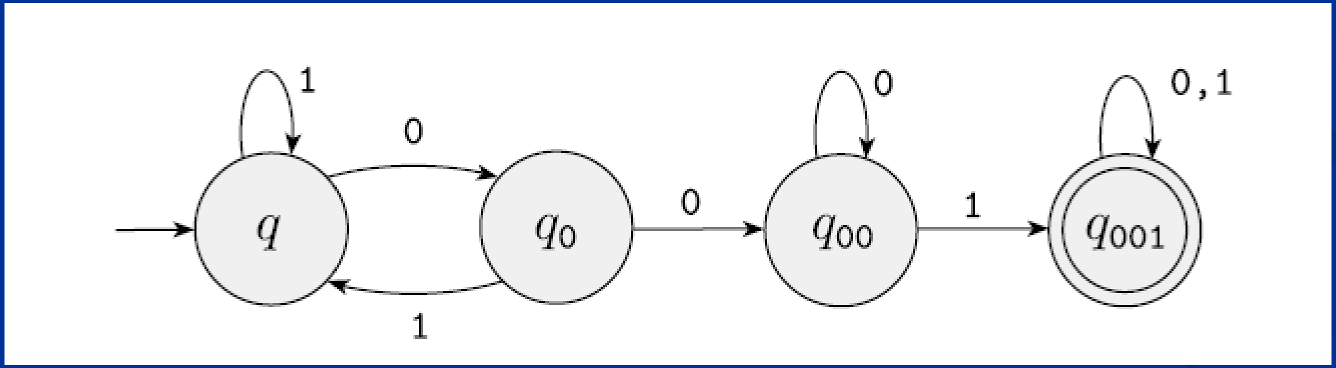
FA Design Example 1

- Design a FA to accept the language of binary strings where the number of 1's is odd (page 43)



FA Design Example 2

- Design a FA to accept all string with 001 as a substring (page 44).



Regular Operations

- Let A and B be languages. We define 3 regular operations:
  - Union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
  - Concatenation: $A \cdot B$ where $\{xy \mid x \in A \text{ and } y \in B\}$
  - Star: $A^* = \{x_1 x_2 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$
    - Star is a unary operator on a single language
    - Star repeats a string 0 or more times

Examples of Regular Operations

- Let A = {good, bad} and B = {boy, girl}
- Then what is:
  - A ∪ B
    - A ∪ B = {good, bad, boy, girl}
  - A · B
    - A · B = {goodboy, goodgirl, badboy, badgirl}
  - A*
    - A* = {ε, good, bad, goodgood, goodbad, badbad, badgood, goodgoodgood, …}

Closure

- The natural numbers is closed under addition and multiplication (but not division and subtraction)
- A collection of objects is closed under an operation if applying that operation to members of the collection returns an object in the collection

Closure for Regular Languages

- Regular languages are closed under the 3 regular operators we just introduced
- Can you look ahead to see why we care?
  - If these operators are closed, then if we can implement each operator using a FA, then we can build a FA to recognize a regular expression

Closure of Union

- Theorem 1.25: The class of regular languages is closed under the union operation
  - If A_1 and A_2 are regular languages then so is $A_1 \cup A_2$
  - How can we prove this? Use proof by construction.
    - Assume M1 accepts A1 and M2 accepts A2
    - Construct M3 using M1 and M2 to accept A1 ∪ A2
    - We need to simulate M1 and M2 running in parallel and stop if either reaches an accept state
      - This last part is feasible since we can have multiple accept states
      - You need to remember where you would be in both machines

Closure of Union II

- You need to generate a state to represent the state you would be in with M1 and M2
- Let $M1=(Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M2=(Q_2, \Sigma, \delta_2, q_2, F_2)$
- Build M3 as follows (we will do $Q, \Sigma, q_0, F, \delta$):
  - $Q=\{(r1,r2) \mid r1 \in Q_1$ and $r2 \in Q_2$ (Cartesian product)
  - Σ stays the same but could more generally be $\Sigma_1 \cup \Sigma_2$
  - q_0 is the pair (q_1, q_2)
  - $F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\}$
  - $\delta((r_1,r_2),a) = (\delta_1(r_1, a), \delta_2(r_2, a))$

Closure of Concatenation

- Theorem 1.26: The class of regular languages is closed under the concatenation operator
  - If A1 and A2 are regular languages then so is A1 · A2
  - Can you see how to do this simply?
    - Not trivial since cannot just concatenate M1 and M2, where start states of M2 become the finish states of M1
      - Because we do not accept a string as soon as it enters the finish state (wait until string is done) it can leave and come back
      - Thus we do not know when to start using M2; if we make the wrong choice will not accept a string that can be accepted
        - Example on next slides
      - This proof is easy if we have nondeterministic FA

Concatenation Example that Works

- First here is an example I think will work
  - L(M1) = A, where Σ = {0, 1} and A = binary string with exactly 2 1's
  - L(M2) = B, where Σ = {0, 1} and B = binary string with exactly 3 1's
    - M1 will enter accept state as soon as sees 2 1's. It can then loop back on any 0's or move to M2 without issue. It can move immediately to M2 on a 1, and not have an issue since it cannot loop back, since A accepts only exactly 2 1's. Once in M2 everything will work okay.

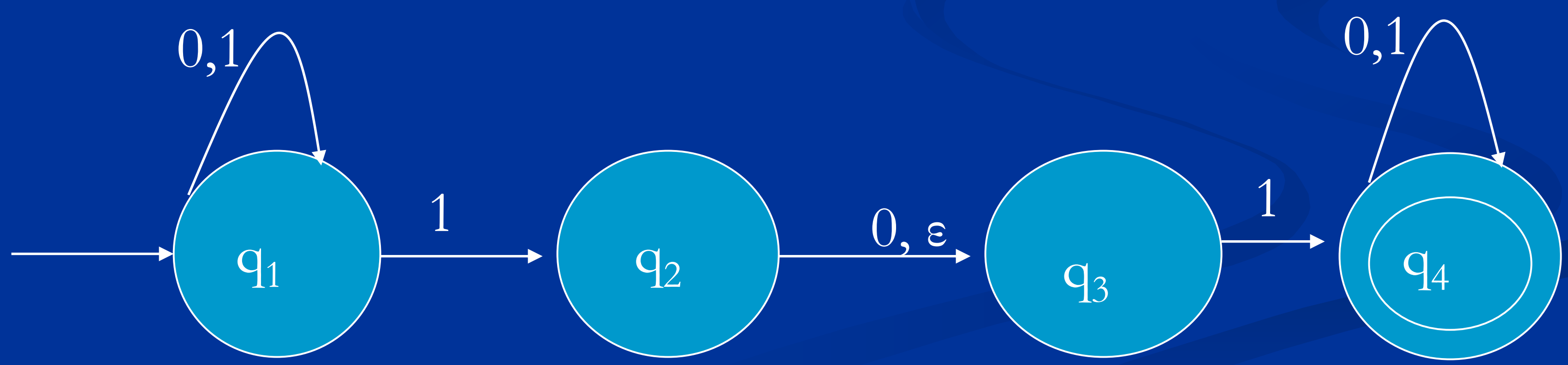
Concatenation Example that Fails

- L(M1) = A, where $\Sigma = \{0, 1\}$ and A = binary string with *at least* 2 1's
- L(M2) = B, where $\Sigma = \{0, 1\}$ and B = binary string with exactly 2 1's
- This does not work (but easy with NFA or more complicated DFA)
  - If in M1 and see 2 1's, enter accept state. When see another 1, have choice to loop back into accept state in M1, or start moving into M2, to the state that represents saw first 1 for string in B.
    - If the concatenated string has exactly 4 1's total, then will only accept if move into M2 as early as possible (after seeing the first 2 1's)
    - If the concatenated string has more than 4 1's, then will only accept if loop in M1 accept state until only 2 1's left.
- Note that the general procedure for putting M1 and M2 together involves superimposing the start state for M2 onto accept state of M1 and removing the original arcs in M1 for that state.

Chapter 1.2: Nondeterminism

Nondeterminism

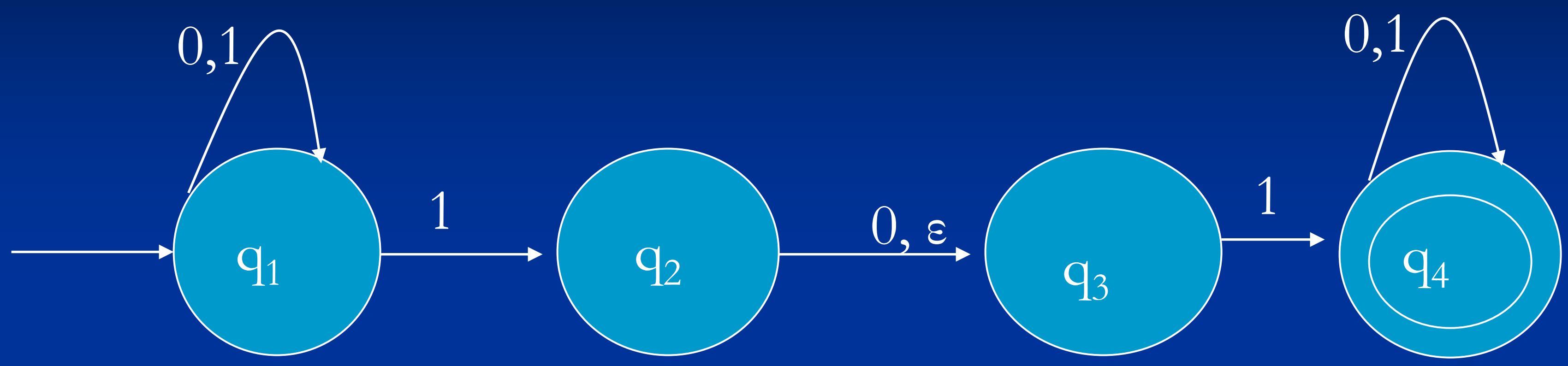
- So far our FA is deterministic in that the state and next symbol determines the next state
- In a nondeterministic machine, several choices may exist
- DFA's have one transition arrow per alphabet symbol, while NFAs have 0 or more for each and ε



How does an NFA Compute?

- When there is a choice, all paths are followed
  - Think of it as cloning a process and continuing
  - If there is no arrow, the path terminates and the clone dies (it does not accept if at an accept state when that happens)
  - An NFA may have the empty string cause a transition
  - The NFA accepts if any path is in the accept state
  - Can also be modeled as a tree of possibilities
- An alternative way of thinking of this
  - At each choice you make one guess of which way to go
  - You magically always guess the right way to go

Try Computing This!



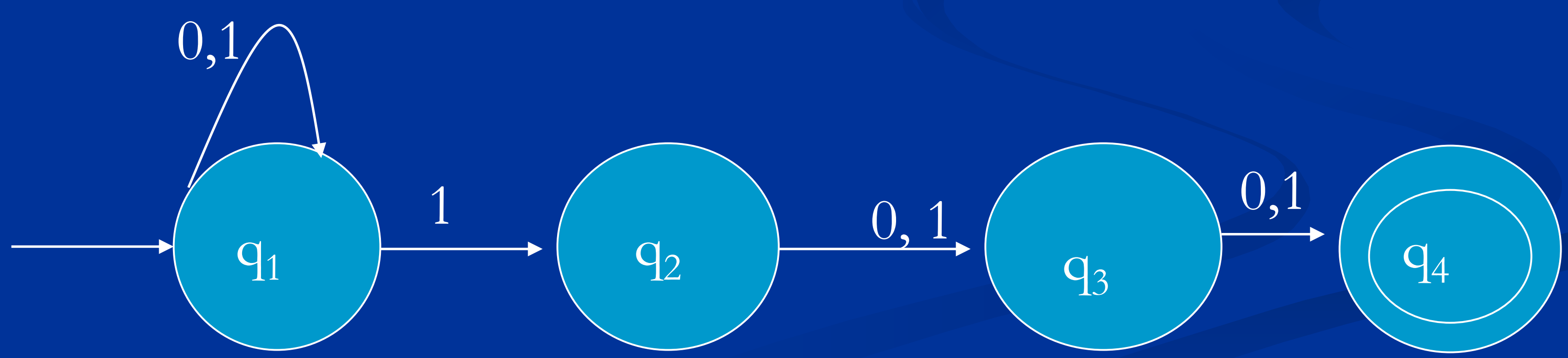
- Try out 010110
  - Is it accepted?
    - Yes
- What is the language?
  - Strings containing a substring of 101 or 11

Construct an NFA

- Construct an NFA that accepts all string over {0,1} with a 1 in the third position from the end
  - Hint: the NFA stays in the start state until it guesses that it is three places from the end

Construct an NFA

- Construct an NFA that accepts all string over {0,1} with a 1 in the third position from the end
  - Hint: the NFA stays in the start state until it guesses that it is three places from the end



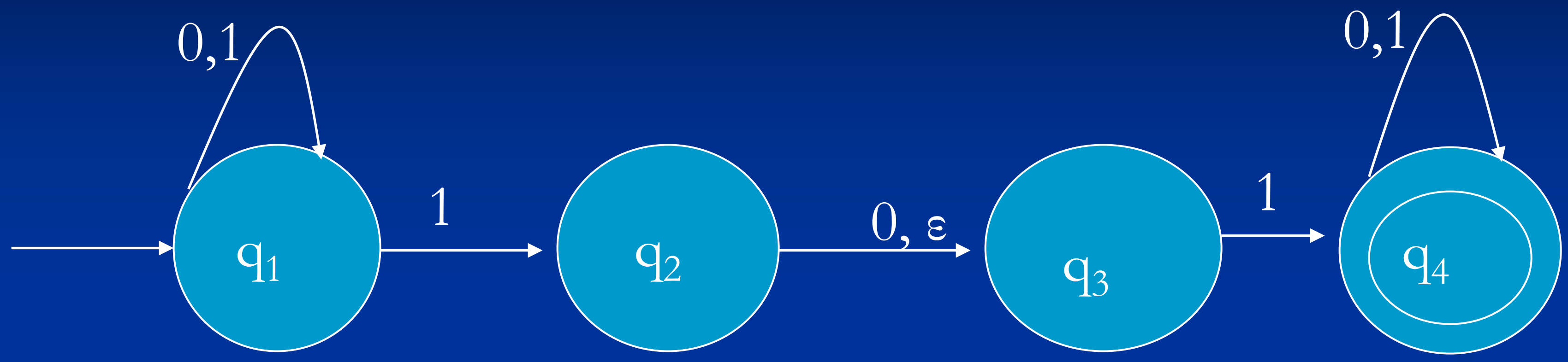
Can we generate a DFA for this?

- Yes, but it is more complicated and has 8 states
  - See book Figure 1.32 page 51
  - Each state represents the last 3 symbols seen, where we assume we start with 000
  - So, states 000, 001, 010, 011, …, 111
  - What is the transition from 010
    - On a 1 we go to 101
    - On a 0 we go to 100

Formal Definition of Nondeterministic Finite Automata

- Similar to DFA except Σ includes ε and next state is not a state but a set of possible states
- A nondeterministic finite automata is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where
  - Q is a finite set called states
  - Σ is a finite set called the alphabet
  - $\delta : Q \times \Sigma_\varepsilon \rightarrow P(Q)$ is the transition function
  - $q0 \in Q$ is the start state
  - $F \subseteq Q$ is the set of accept states

Example of Formal Definition of NFA



- NFA N_1 is $(Q, \Sigma, \delta, q_1, F)$
  - $Q = \{q_1, q_2, q_3, q_4\}$
  - $\Sigma = \{0,1\}$
  - q_1 is the start state
  - $F = \{q_4\}$

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

Equivalence of NFAs and DFAs

- NFAs and DFAs recognize same class of languages
- What does this mean? What is the implication?
  - NFAs have no more power than DFAs
    - With respect to what can be expressed
    - Every NFA has an equivalent DFA
    - But NFAs may make it easier to describe some languages
  - Terminology: Two machines are equivalent if they recognize the same language

Similar Idea you are More Familiar with

- C, C++, Python, Pascal, Fortran, …
- Are these languages equivalent?
  - Some are more suited to some tasks, but with enough effort any of these languages can compute anything the others can
  - If necessary, you can even write a compiler for one language using another

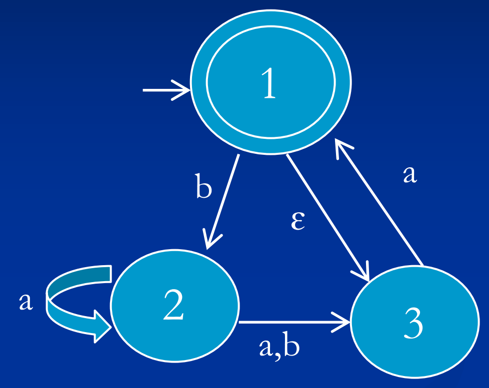
Proof of Equivalence of NFA & DFA

- Proof idea
  - Need to simulate an NFA with a DFA
  - With NFA's, given an input we follow all possible branches and keep a finger on the state for each
  - That is what we need to keep track of– the states we would be in for each branch
  - If the NFA has k states then it has 2^k possible subsets
    - Each subset corresponds to one of the possibilities that the DFA needs to remember
    - The DFA will have 2^k states

Proof by Construction

- Let $N=(Q, \Sigma, \delta, q_0, F)$ be the NFA recognizing A
- Construct DFA $M = (Q', \Sigma, \delta', q_0', F')$
  - Lets do the easy ones first (skip δ' for now)
  - $Q' = P(Q)$
  - $q_0' = \{q_0\}$
  - $F' = \{R \in Q' \mid R$ contains an accept state of $N\}$
  - Transition function
    - The state R in M corresponds to a set of states in N
    - When M reads symbol *a* in state R, it shows where *a* takes each state
    - $\delta'(R,a)$ = Union of $r \in R$ of $\delta(r,a)$
  - I ignore ε, but taking that into account does not fundamentally change the proof– we just need to keep track of more states

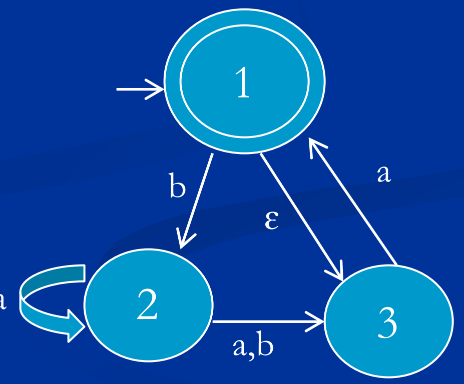
Example: Convert an NFA to a DFA



- See example 1.41 (pg. 57 2nd ed.)
  - For now don't look at solution DFA
  - The NFA has 3 states: Q = {1, 2, 3}
  - What are the states in the DFA?
    - {∅, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}}
  - What are the start states of the DFA?
    - Start states of NFA include those reachable by ε-moves
    - {1, 3}
      - 3 is included because if we start in 1 we can immediately move to 3
  - What are the accept states?
    - {{1}, {1,2}, {1,3}, {1,2,3}}

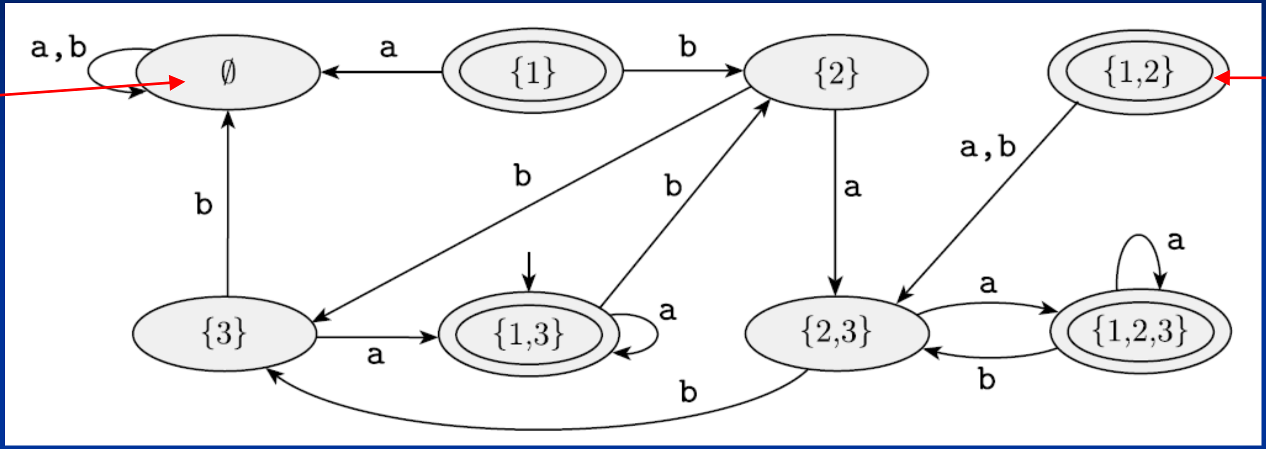
Example: Convert an NFA to a DFA

- Now let's work on some of the transitions
  - Let's look at state 2 in NFA and complete the transitions for state 2 in the DFA
    - Where do we go from state 2 on an "a" and "b"
      - On "a" to state 2 and 3 and on "b" to state 3
    - So, what state does {2} in DFA go to for a and b?
      - Answer: on a to {2,3} and {3} for b
  - Now let's do state {3}
    - On "a" goes to {1,3} and on b goes to ∅
      - Why {1, 3}? Because first goes to 1 then ε permits a move back to 3!
  - DFA equivalent to NFA on next slide (and Fig. 1.43, pg. 58 2^{nd} ed)
    - Any questions? Could you do it on a HW, exam, or <u>quiz</u>?



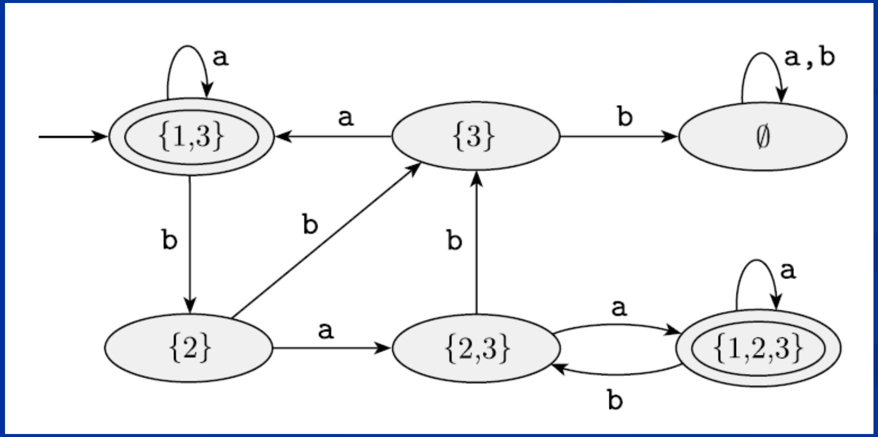
DFAs Equivalent to 3-State NFA

Represents "Dead State"



Clearly not reachable

Can be simplified to DFA below since some states not reachable from start state.
On HW or exam I would want to see the unsimplified version (can also show simplified)



Closure under Regular Operations

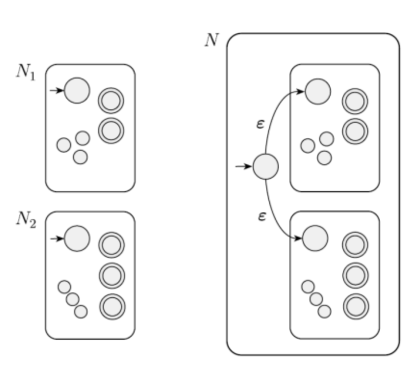
- We started this before and did it for Union only
  - Union much simpler using NFA
- Concatenation and Star much easier using NFA
- Since DFAs equivalent to NFAs, we can now just use NFAs
- Fewer states to keep track of because we can act as if we always "guess" correctly

Why do we care about closure?

- We need to look ahead
  - A regular language is what a DFA/NFA accepts
  - We are now introducing regular operators and then will generate regular expressions from them (Ch 1.3)
  - We will want to show that the language of regular expressions is equivalent to the language accepted by NFAs/DFAs (i.e., a regular language)
  - How do we show this?
    - Basic terms in regular expression can generated by a FA
    - We can implement each operator using a FA and the combination is still able to be represented using a FA

Closure Under Union

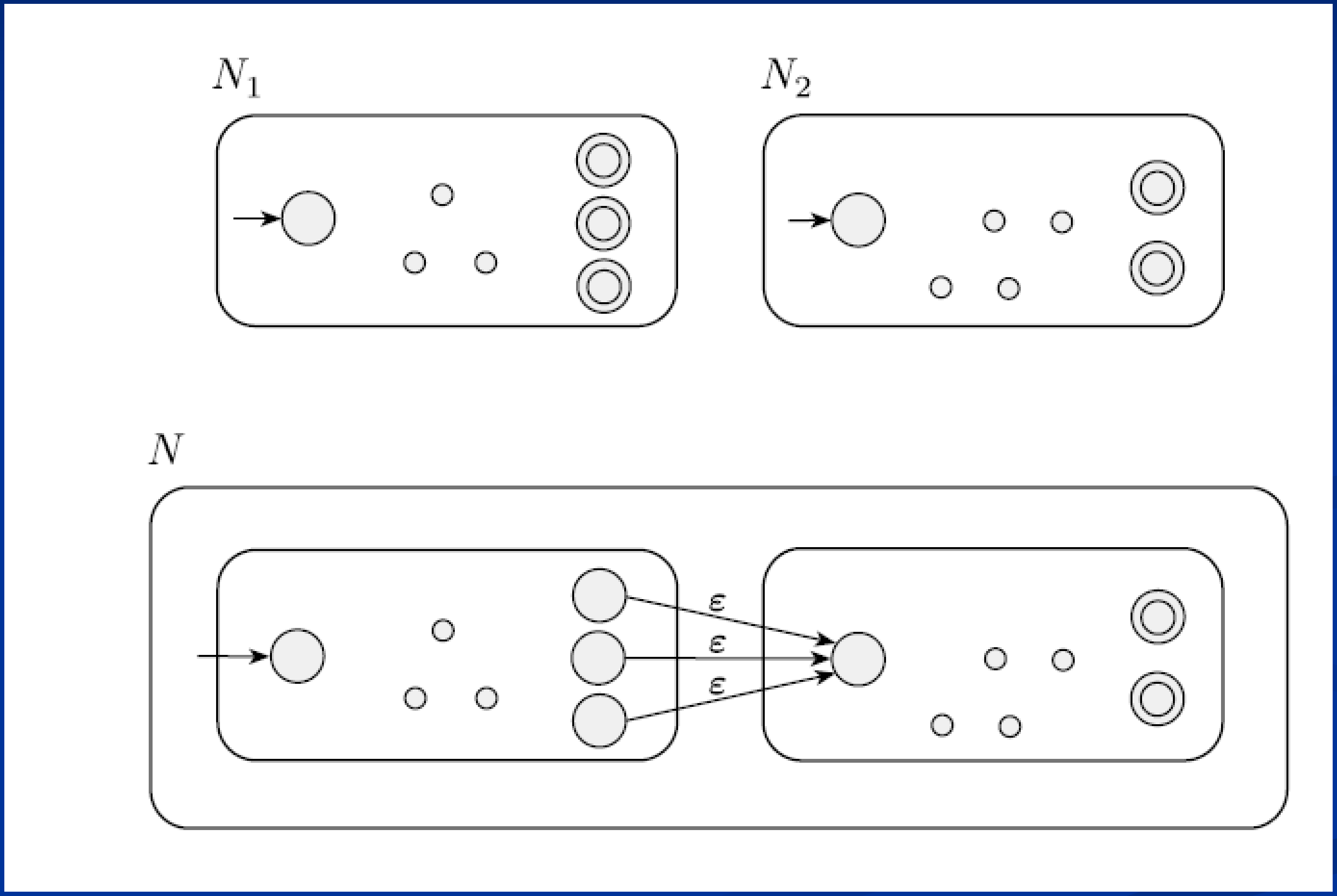
- Given two regular languages A_1 and A_2 recognized by two NFAs N_1 and N_2, construct N to recognize $A_1 \cup A_2$
- How do we construct N? Think!
  - Start by writing down N_1 and N_2. Now what?
  - Add a new start state and then have it take ε branches to the start states of N_1 and N_2
    - Isn't that easy!



Closure under Concatenation

- Given two regular languages A_1 and A_2 recognized by two NFAs N_1 and N_2, construct N to recognize $A_1 \cdot A_2$
- How do we do this?
  - The complication is that we did not know when to switch from handling A_1 to A_2 since can loop thru an accept state
  - Solution with NFA:
    - Connect every accept state in N_1 to every start state in N_2 using an ε transition
      - don't remove transitions from accept state in N_1 back to N_1

NFA Proof of Concatenation Closure



Closure under Concatenation II

- Given:
  - $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizes A_1
  - $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognizes A_2
- Construct $N=(Q_1 \cup Q_2, \Sigma, \delta, q_1, F_2)$ so that it recognizes $A_1 \cdot A_2$

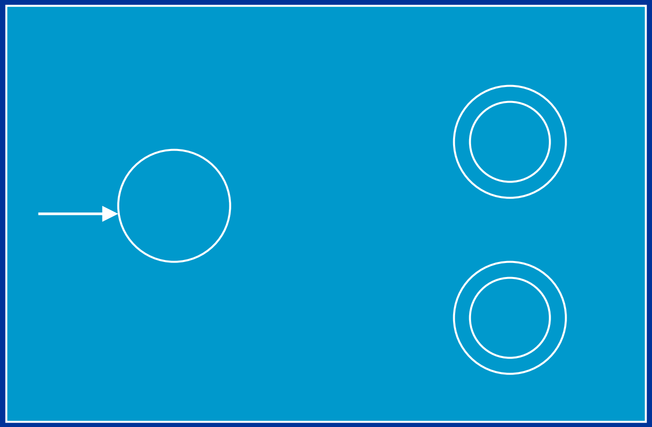
$\delta(q,a) =$

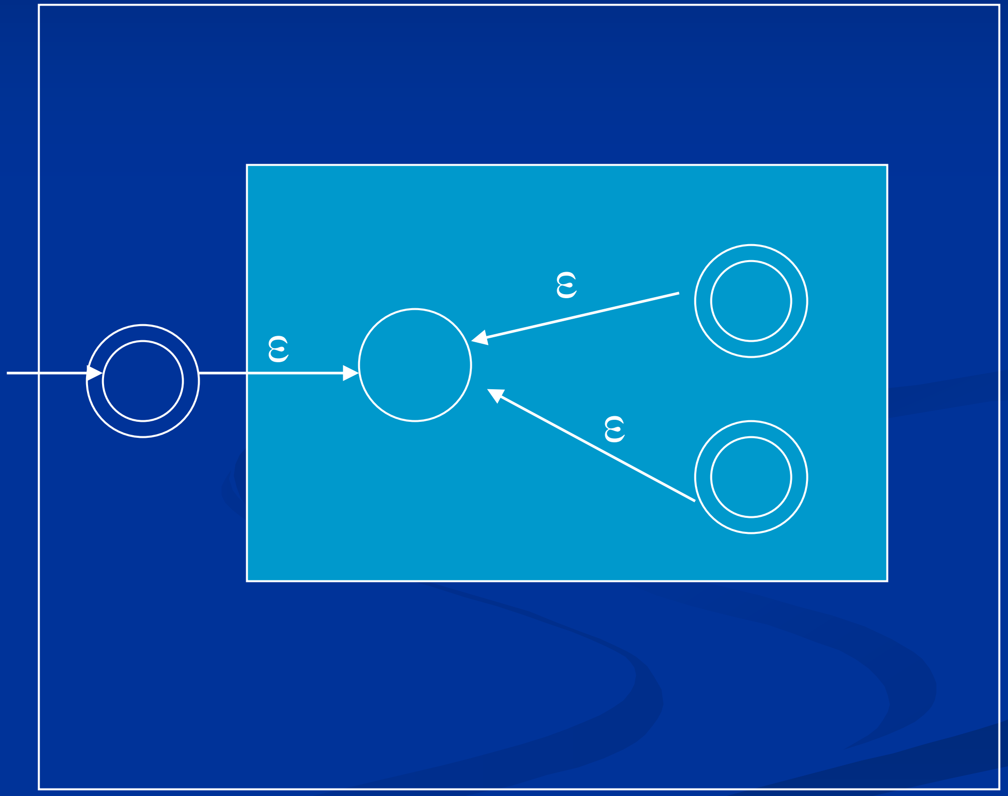
$\delta_1(q,a)$	$q \in Q_1$ and $q \notin F_1$
$\delta_1(q,a)$	$q \in F_1$ and $a \neq \varepsilon$
$\delta_1(q,a) \cup \{q_2\}$	$q \in F_1$ and $a = \varepsilon$
$\delta_2(q,a)$	$q \in Q_2$

Closure under Star

- Given regular language A_1 prove A_1* is also regular
  - Note (ab)* = {∅, ab, abab, ababab, ...}
- Proof by construction
  - Take NFA N_1 that recognizes A_1 and construct N from it that recognizes A_1*
  - How do we do this?
    - Add new ε-transition from accept states to start state
    - Then make the start state the accept state so that ∅ is accepted
      - This almost works, but not quite. What is the problem?
        - May have transition from intermediate state to start state and should not accept on this loop-back
    - Solution: add a *new* start state that is accept state, with an ε-transition to the original start state and have ε-transitions from accept states to old start state

Closure under Star





Chapter 1.3: Regular Expressions

Regular Expressions

- Based on the regular operators
- Examples:
  - $(0 \cup 1)0^*$
    - A 0 or 1 followed by any number of 0's
    - Concatenation operator implied
  - What does $(0 \cup 1)^*$ mean?
    - All possible strings of 0 and 1
      - Not 0^* or 1^* so does not require that commit to 0 or 1 before applying * operator
    - Assuming $\Sigma = \{0,1\}$, then equivalent to Σ^*

Definition of Regular Expression

- R is a regular expression if R is
  1. a, for some a in alphabet Σ
  2. ε
  3. $\emptyset$
  4. $(R1 \cup R2)$, where R1 and R2 are regular expressions
  5. $(R1 \cdot R2)$, where R1 and R2 are regular expressions
  6. $(R1^*)$, where R1 is a regular expression
- Note:
  - This is a recursive definition, common in computer science
    - R1and R2 always smaller than R, so no issue of infinite recursion
  - $\emptyset$ means language does not include any strings and ε means it includes the empty string

Examples of Regular Expressions

- $0^*10^* =$
  - {w| w contains a single 1}
- $\Sigma^*1\Sigma^*=$
  - {w| w has at least one 1}
- $01 \cup 10 =$
  - {01, 10}
- $(0 \cup \varepsilon)(1 \cup \varepsilon) =$
  - {ε, 0, 1, 01}

Equivalence of Regular Expressions and Finite Automata

- Theorem: A language is regular if and only if some regular expression describes it
  - This has two directions so we need to prove:
    - If a language is described by a regular expression then it is regular
    - If a language is regular then it is described by a regular expression
    - We will only do the first direction

Proof: Regular Expression → Regular Language

- Proof idea: Given a regular expression R describing a language L, we should …
  - Show that some FA recognizes it
  - Use NFA since may be easier and equivalent to DFA
- How do we do this?
  - We will use definition of a regular expression and show that we can build a FA covering each step.
    - We will do quickly with two parts:
      - Steps 1,2 and 3 of definition (handle a, ε, and ∅)
      - Steps 4,5 and 6 (handle union, concatenation, and star)

Proof Continued

- For steps 1-3 we construct the FA below. As a reminder:
  1. a, for some a in alphabet ∑
  2. ε
  3. ∅

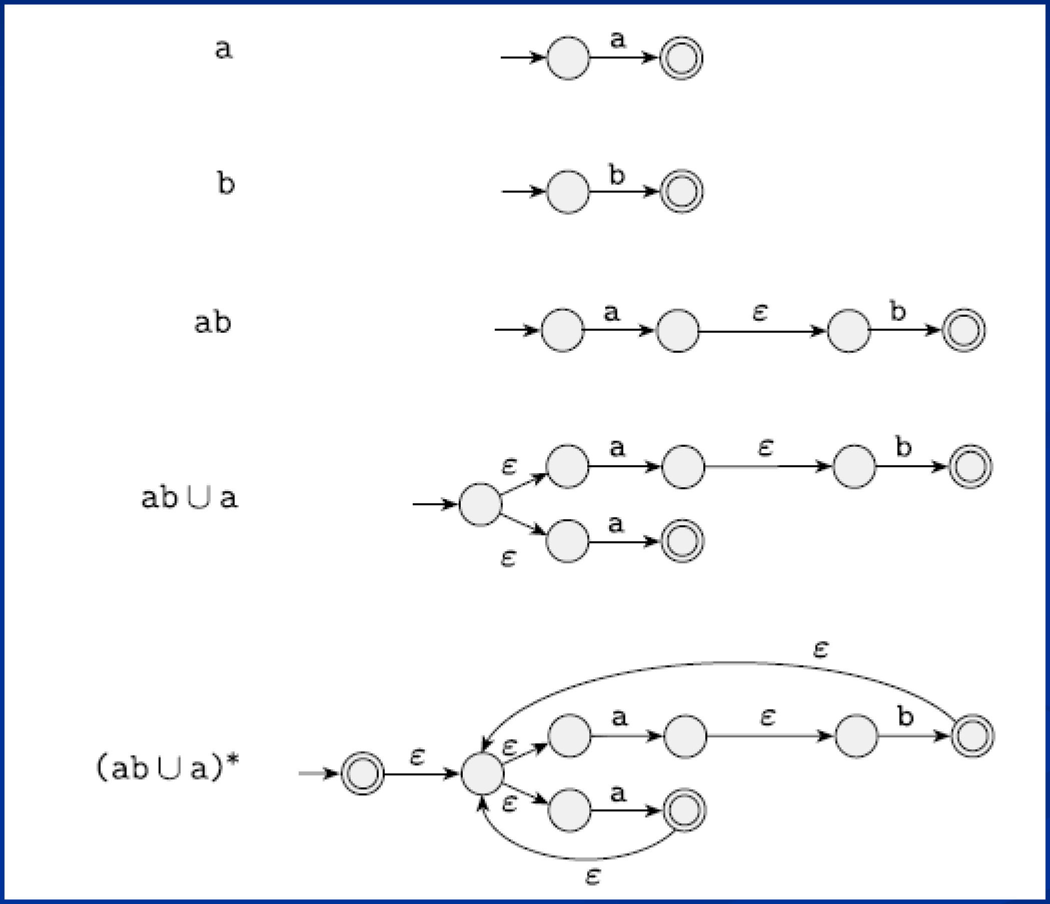
Proof Continued

- For steps 4-6 (union, concatenation and star) we use the proofs we already constructed to show that FA are closed under union, concatenation, and star
- So we are done with the proof in one direction!
- Now let's try an example

Example: Regular Expression → NFA

- Convert $(ab \cup a)^*$ to an NFA
  - Let's describe the outline of what we need to do
    - Handle a
    - Handle ab
    - Handle $ab \cup a$
    - Handle $(ab \cup a)^*$
  - In the solution on next page, which is from the book (Fig 1.57 page 68), there are states for ε-transitions. They seem unnecessary and may confuse you. They are unnecessary in this case.

Conversion of (ab ∪ a)* to an NFA

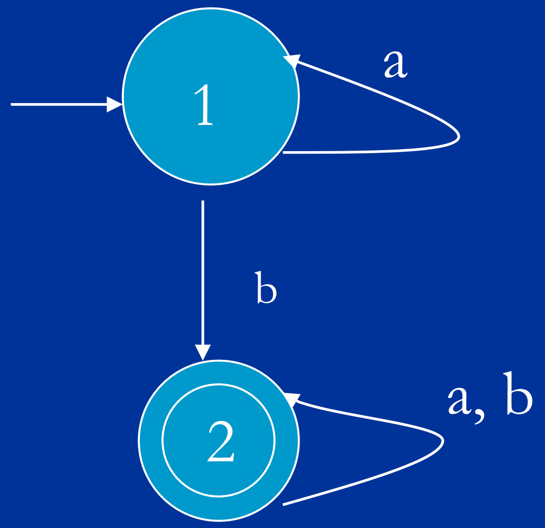


Proof: Regular Language → Regular Expression

- This is the proof in the other direction
  - Need to show can convert any DFA to regular expression
  - The book goes through several pages (Lemma 1.60 page 69 – 74) that does not really add much insight
    - You can skip the proof. For the most part, if you understand the ideas for going in the previous direction, you also understand this direction.
    - But you should be able to handle an example. But you need not use the formal method they have in the book. Just be able to handle simple examples.

Example: DFA → Regular Expression

- This example is on page 75, Figure 1.67
  - Do not worry about doing it the way they do it
- For the DFA below, what is the equivalent regular expression?



Answer: $a^*b(a \cup b)^*$

Chapter 1.4: Nonregular Languages

… and not the real fun begins

Never has something so simple confused so many – Dr. Weiss

Non-Regular Languages

- Do you think every language is regular?
  - That would mean that every language can be described by a FA
- What might make a language non-regular? Think about the properties of a finite automata.
  - Answer: finite memory
  - So a language is not regular if you need infinite memory

Some Example Questions

- Consider the following languages
  - L1 = {w| w has an equal number of 0's and 1's}
    {ε, 01, 10, 1100, 0011, 0101, 1010, 0110, …}
  - L2 = {w| w has at least 100 1's}
    {1^{100}, 01^{100}, $(01)^{201}$, …}
  - L3 = {w| w is of the form 0^n1^n, $n \geq 0$}
    {ε, 01, 0011, 000111, 00001111, …}
- Now determine if you think they are regular
  - Is L1 regular? Is L2 regular? Is L3 regular?
  - Take a minute to think about it

Answers

- Answers:
  - L1 (equal # 1's and 0's) is:
    - Not regular. It requires infinite memory
  - L2 (at least 100 1's) is:
    - Regular
    - We can design the DFA easily. Each state represents the number of 1's seen and once we get to 100 we loop in the accept state.
  - L3 (0^n1^n) is:
    - Not regular. It requires infinite memory

Languages We Will Consider

- We will only consider infinite languages
  - This only means that the number of strings belonging to the language is infinite, not that it requires infinite memory
  - L1, L2, and L3 are infinite languages
  - This language is not:
    - L5 = {a, ab, abb}
  - Are finite languages always regular?
    - Yes, we can create a path and accept state for each element in the language

More Examples (I)

- For $L = (01)^n$, is L regular? Why or why not?
  - Note L = {ε, 01, 0101, 010101, 01010101, …}
- L is regular because there is a finite pattern
  - The pattern is "01". We want to accept $(01)^*$
  - We can build a DFA for this quite easily using the construction for concatenation and *.

More Examples (2)?

- Let $B_n = \{a^k | \text{where k is a multiple of n}\}$ for $n \geq 1$
  - $B_3 = \{\varepsilon, aaa, aaaaaa, aaaaaaaaa, \ldots\}$
- Is this regular?
  - This language is regular!
  - How is this question different from the ones before?
    - Each language has a specific value of n, so n is not a free variable, but k is a free variable. The number of states is bounded by n, not k.
    - That is B_n is not a language, but a family of languages, where each one is defined for a value of n.
    - The DFA counts the number of a's modulo k. This requires k states. Since k is a multiple of n, we need n states. For B_3 we need 3 states, for B_6 we need 6 states.

More on Regular Languages

- Regular languages can be infinite but must be described using finite number of states
  - Thus there are restrictions on the structure of regular languages
  - For a FA to generate an infinite set of string, clearly there must be a _______ between some states
    - loop
  - This leads to the (in)famous pumping lemma

Pumping Lemma for Regular Languages

- The pumping lemma states that all regular languages have a special property
- If a language does not have this property it is not regular
  - So can use to prove a language non-regular
  - Note: the pumping lemma can hold and a language still not be regular. This is not usually highlighted.

Pumping Lemma II

- Pumping lemma property
  -
    1. For each $i \geq 0$, $x y^i z \in L$
    2. $|y| > 0$, and
    3. $|xy| \leq p$
  - This means every string $s \in L$ contains a section that can be repeated any number of times (via a loop)

Pumping Lemma Conditions

- Condition 1: for each $i \geq 0$, $x y^i z \in L$
  - This just says that there is a loop
- Condition 2: $|y| > 0$
  - Without this condition, then there really would be no loop
- Condition 3: $|xy| \leq p$
  - We don't allow more states than the pumping length, since we want to bound the amount of memory

Pumping Lemma Proof Idea

- Set the pumping lemma length *p* to number of states of the FA
  - If length of s ≤ pumping lemma trivially holds, so ignore these strings
  - Consider the states that the FA goes through for s
    - Since there are only p states and length s > p, by pigeonhole property one state much be repeated
      - This means there is a cycle

Partitioning a String s into xyz

- We mean breaking s into 3 pieces x, y, and z, so when we concatenate them, they equal s
  - Example 1: s = 1101
    - Solution 1: x=1, y=1, z=01
      - $xyyz = xy^2z = 1\ 11\ 01 = 11101$ (result of “pumping on” y)
    - Solution 2: x=110, y=1, z = ε (only y cannot be ε)
      - $xyyz = xy^2z = 110\ 11\ \varepsilon = 11011$
  - Example: $s = 0^p1^p$
    - Solution 1: $x = 0$, $y = 0$, $z = 0^{p-2}1^p$ (xyz= p0’s then p 1’s)
      - $xy^2z = 0\ 0^2\ 0^{p-2}1^p = 0^{p+1}1^p$ (note now one more 0 than 1)

Using Condition 3

- Condition 3 says that $|xy| \leq p$
- Example: $s = 0^p 1^p$
  - What constraints does condition 3 put on y?
    - y can only contain 0's since $\text{length}(xy) \leq p$ and s starts with 0 p's.
    - Let's try to have y include a 1 as follows:
      - $x = 0^p$ $y=1$ $z=1^{p-1}$
      - Now xyz still equal s and y has a 1. Also, $|xy|$ is the shortest it can be and still contain a 1
      - $|xy| = |x| + |y| = p + 1 \geq p$ so violates condition 3

Example 1

- Let B be the language $\{0^n1^n \mid n \geq 0\}$ (Ex 1.73)
  - Prove B is not regular using proof by contradiction.
    - Assume B is regular. Pick a string that will cause a problem.
  - What string? Use the pumping length *p* in the string.
  - Try 0^p1^p
    - We need $x\ y^i\ z$, let's focus on y. What may y contain? What then happens when we pump on *y*? Is the pumped string in B?
      - If *y* all 0's or all 1's, then if $xyz \in L$ then $xyyz \notin L$ (number 0's $\neq$ 1's)
      - If *y* a mixture of 0 and 1, then 0's and 1's in s not separated

Example 1 Continued

- We can simplify the proof by using condition 3, since then instead of 3 possibilities for y, we have one
  - Since condition 3 says that $|xy| \leq p$, and string s starts with 0^p, then the strings x and y can only contain 0's.
    - We showed why this was a few slides earlier with an example
    - But let's try again anyway:
      - $x = 0^p$ $y = 1$ $z = 1^{p-1}$ so that $xyz = 0^p1^p$
      - But $|xy| = |x| + |y| = p + 1 \geq p$ so condition 3 is violated
      - Demonstrates that y cannot have 0's while satisfying condition 3
    - Since y must be all 0's, pumping up adds only 0's to a string that was in the language. But if we add only 0's to a string that had equal #'s of 0's and 1's, then it must now have more 0's than 1's.

Common Sense Idea Behind Proof

- The key idea is that we can have any number of 0's at the start and we must then have an equal number of 1's
  - But we cannot keep track of an arbitrary number of 0's. That is, the number of 0's is unbounded
    - For any string, the number of 0's is not infinite, but it is not bounded by any value
- So if you tell me that the FA has *p* states, I will tell you that it cannot recognize a string with more than *p* 0's.
- Alternatively, if you build a FA to handle 1 million 0's, I will then give you a string with more than that number of 0's.

Example 2

- Let C = {w | w has equal number of 0's and 1's} (Ex 1.74)
  - Can you do this with finite memory?
    - No
  - Prove C is not regular, using proof by contradiction
  - Assume C is regular. Pick a problematic string.
    - Let's try 0^p1^p
    - This string is in C since equal 0's and 1's but note the language does not require them to be separated.
    - If we pick y =01, can we pump it and have pumped string $\in$ C?
      - Yes! Each time we pump (i.e., loop) we add one 0 and 1. So works!
      - Note however that pumped string not in 0^n1^n, but that is okay since in C
      - But by condition 3, y must be only 0's given the string that we picked
        - So y can only have 0s and pumping break equality
        - Unlike the last case, in this case we must use condition 3

Example 2 (failing proof)

- I believe we often learn more by failing. Let's retry the problem.
- Let C = {w | w has equal number of 0's and 1's}
  - Prove C is not regular, using proof by contradiction
  - Assume C is regular
    - Let's try $s = (01)^p$, which is a valid string in C
    - Can we pump this?
    - Yes! Specify x, y, and z
      - Let $x = \varepsilon$, $y = 01$, $z = (01)^{p-1}$
    - What does this prove?
      - Nothing! Proof by contradiction failed but does not mean it is regular. It means we failed to prove it is not regular.
      - We failed because we picked an "easy" string. We need to pick a hard string, that reflects the full complexity of the problem.

Common-Sense Interpretation

- FA can only use finite memory. If infinite strings, then the loop must handle this
  - If there are two parts that can generate infinite sequences and there is a condition that ties them together, then we must find a way to link them in the loop
    - If they are connected and we cannot link them, then it is not regular
    - Examples: 0^n1^n equal numbers of 0s and 1s

Example 3

- Let $F = \{ww \mid w \in \{0,1\}^*\}$ (Ex 1.75)
  - $F = \{\varepsilon, 00, 11, 0101, 1010, 0000, 1111, 101101, \ldots\}$
  - Can you do this with finite memory?
    - No, you need to remember all of w
  - Use proof by contradiction. Pick $s \in F$ that will be problematic
    - $s = 0^p10^p1$
      - Since $|xy| < p$, y must be all 0's (0's from the start of string)
      - If we pump y, then only adding 0's. That will be a problem in this case since the number of 0's separated by the 1 must be equal
  - Need to be careful. Sometimes when you add to the first half, the halfway points changes. But use of 1 as delimiter still prevents successful pumping.

Example 4

- Let $D = \{1^{n^2} \mid n \geq 0\}$
  - $D = \{\varnothing, 1, 1111, 111111111, \ldots\}$
  - Proof by contradiction
  - Choose 1^{p^2}
    - Assume we have an $xyz \in D$
    - What about xyyz? The # of 1's differs from xyz by $|y|$
      - Since $|xy| \leq p$ then $|y| \leq p$
      - Thus xyyz has at most p more 1's than xyz
      - So if xyz has length $\leq p^2$ then $xyyz \leq p^2 + p$
      - But $(p+1)^2 = p^2 + 2p + 1$ and $p^2 + p$ is less than this
      - Thus length of xyyz lies between consecutive perfect squares and hence $xyyz \notin D$

An Intuitive Explanation

- As you create larger and larger consecutive perfect squares, the difference between them grows and is not bounded.
  - Thus if you have p states, at some point the difference between perfect squares will exceed the p states in the FA.

Example 5

- Let $E = \{0^i1^j \mid i > j\}$
- Assume E is regular and let $s = 0^{p+1}1^p$
- By condition 3, y must be all 0's
  - What can we say about xyyz?
    - Adding the extra y increases number of 0's, which appears to be okay since i > j is okay
  - But we can pump down. What about $xy^0z = xz$?
    - Since s has one more 0 than 1, removing at least one 0 leads to a contradiction. So not regular.

What you need to be able to do

- You should be able to handle examples like 1-3.
- Example 5 is not really any more difficult, just one more thing to think about
- Example 4 was tough, so I would not expect everyone to get an example like that (although I could still ask it)
- Everyone should be able to handle the easy examples
- Try to reason about the problem using "common sense" and then use that to drive your proof
- The homework problems will give you more practice