Chapter 1
Regular Languages

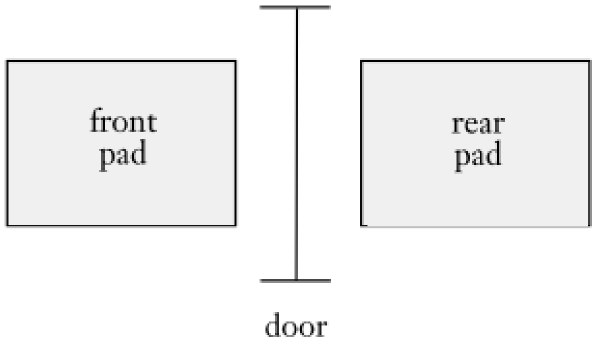
Overview

- computation theory begins with the question: what is a computer?
  - real computers are overly complicated for our uses
  - instead, we use an idealized computer, or computational model
  - we will use several different models with varying features
    - the first is the finite state machine, or finite automaton

Finite Automata

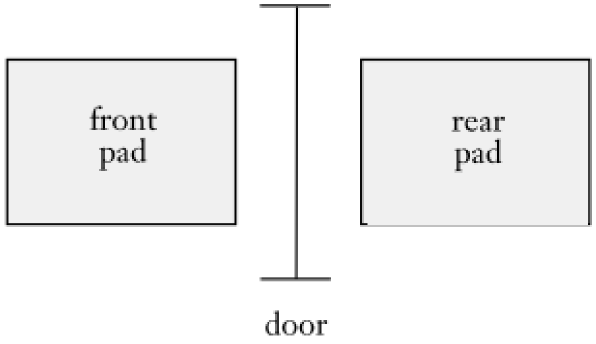
- finite automata
  - useful
  - limited memory
  - common in everyday life

  - example: automatic door controller with ground pads
    - front pad: detect person about to walk through door
    - rear pad: detect how long to hold the door, and to keep the door shut if someone is standing there



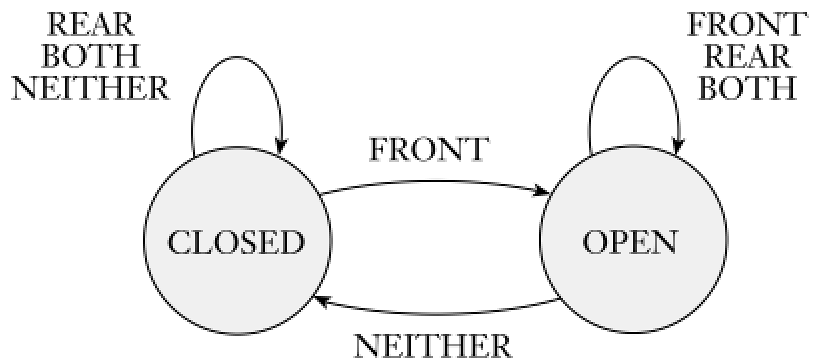
Finite Automata

- example: automatic door with ground pads (cont.)
  - controller in one of two states: OPEN or CLOSED
  - four possible input conditions
    - FRONT: person standing on front pad
    - REAR: person is standing on rear pad
    - BOTH: people are standing on both pads
    - NEITHER: no one is standing on either pad



Finite Automata

- example: automatic door with ground pads (cont.)
  - controller moves between states OPEN and CLOSED depending on input
  - state diagram



Finite Automata

- example: automatic door with ground pads (cont.)
  - state transition table

		input signal			
		NEITHER	FRONT	REAR	BOTH
state	CLOSED	CLOSED	OPEN	CLOSED	CLOSED
	OPEN	CLOSED	OPEN	OPEN	OPEN

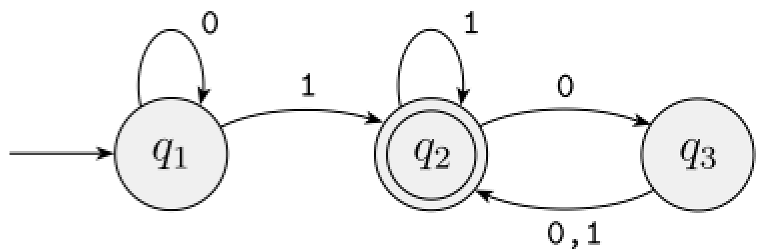
  - if controller is CLOSED and receives input:
    - FRONT, REAR, NEITHER, FRONT, BOTH, NEITHER, REAR, and NEITHER
    - it would go through states:
      - CLOSED (starting), OPEN, OPEN, CLOSED, OPEN, OPEN, CLOSED, CLOSED, CLOSED

Finite Automata

- automatic door controller as finite automaton
  - controller: computer with single bit of memory to hold state
- other controllers might need larger memories
  - elevator controller
    - state for current floor
    - inputs from buttons
  - dishwashers, thermostats, digital watches, calculators
  - Markov chains: useful for recognizing patterns in data
    - speech processing, optical character recognition
    - employ probabilistic state chains

Finite Automata

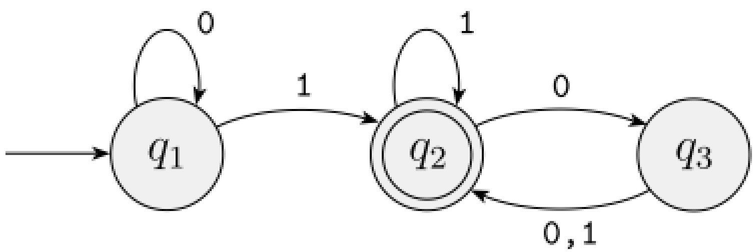
- sample finite automaton M_1



  - state diagram
  - three states: q_1, q_2, q_3
  - start state: q_1 indicated by arrow pointing from nowhere
  - accept state: q_2 with double circle
  - transitions: other arrows

Finite Automata

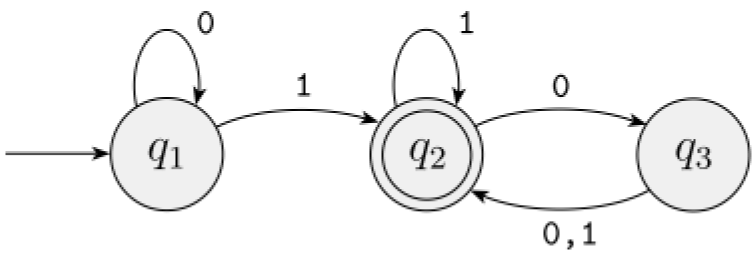
- sample finite automaton M_1



- when input string is received, e.g. 1101, the FA processes it and produces an output: accept or reject
  - begins at start state of M_1
  - input string symbols processed one by one from left to right
  - after reading each symbol, M_1 moves from one state to another according to the symbol
  - when the last symbol is read, M_1 produces output *accept* if it is in the accept state; otherwise *reject*

Finite Automata

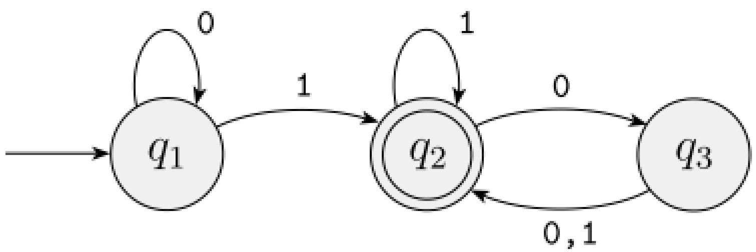
- sample finite automaton M_1



  - e.g. 1101
    - start at state q_1
    - read 1, follow transition from q_1 to q_2
    - read 1, follow transition from q_2 to q_2
    - read 0, follow transition from q_2 to q_3
    - read 1, follow transition from q_3 to q_2
    - accept because M_1 is in an accept state q_2 at end of input

Finite Automata

- sample finite automaton M_1



- other strings accepted
  - 1, 01, 11, 01010101
    - any string that ends with 1
  - 100, 0100, 110000, 0101010000
    - any string that ends with an even number of 0s
- rejected strings
  - 0, 10, 101000

Finite Automata

- formal definition
  - diagrams easier to understand, but formal definition needed because it is
    - precise
      - resolves uncertainties as to what is allowed
    - notation
      - helps express thoughts clearly

Finite Automata

- formal definition
  - requires multiple parts (5-tuple)
    - set of states
    - rules for transitions between states depending on input
    - input alphabet of allowable input symbols
    - start state
    - set of accept states (or final states)

Finite Automata

- concerning rules for transitions between states
  - use transition functions, denoted by δ
  - if FA has an arrow from state x to state y when it reads a 1, it will move from x to y when 1 is read
    - $\delta(x, 1) = y$

Finite Automata

- formal definition

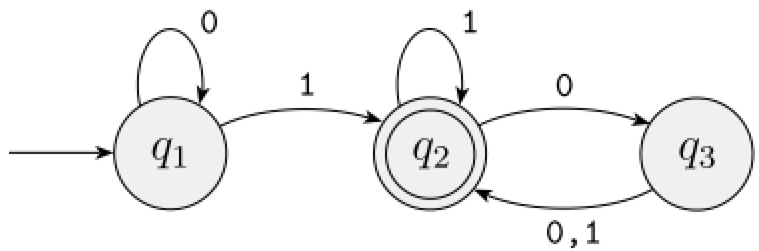
A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***

- with this definition we see
  - 0 accept states is allowable
  - δ specifies exactly one next state for each state/input value

Finite Automata

- for example



- M_1 can be described as $M_1 = (Q, \Sigma, \delta, q_1, F)$ where
  - $Q = \{q_1, q_2, q_3\}$
  - $\Sigma = \{0, 1\}$
  - δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

  - q_1 is the start state
  - $F = \{q_2\}$

Finite Automata

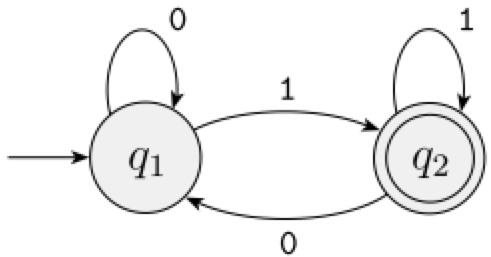
- if A is the set of all strings that M accepts
  - A is the language of M
  - $L(M) = A$
  - M recognizes A
  - M accepts A

- a machine may accept multiple strings, but it only recognizes one language
  - if it accepts no strings, it recognizes the empty language $\emptyset$

- M_1 recognizes A where $A = \{w \mid w \text{ has at least one 1 and an even number of 0s follow the last 1}\}$

Finite Automata

- example



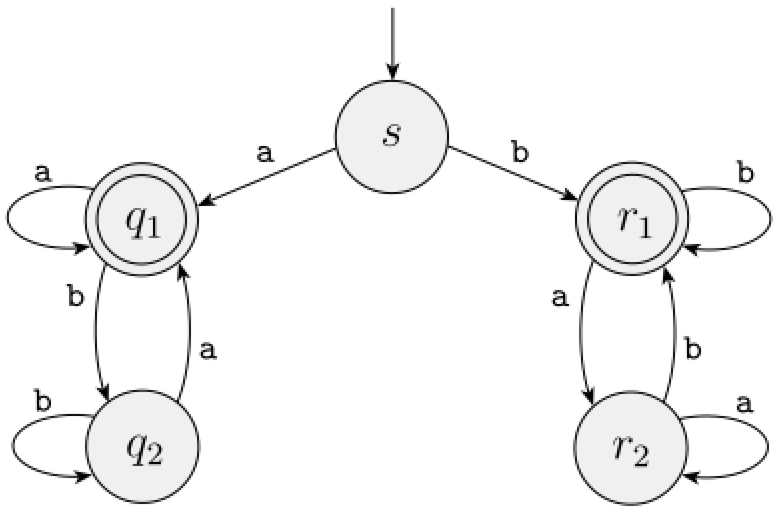
- $M_2 = (\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_2\})$ where
  - δ is described as

	0	1
q_1	q_1	q_2
q_2	q_1	q_2.

- try sample strings
- What language does M_2 recognize?
  - all binary strings ending with a 1

Finite Automata

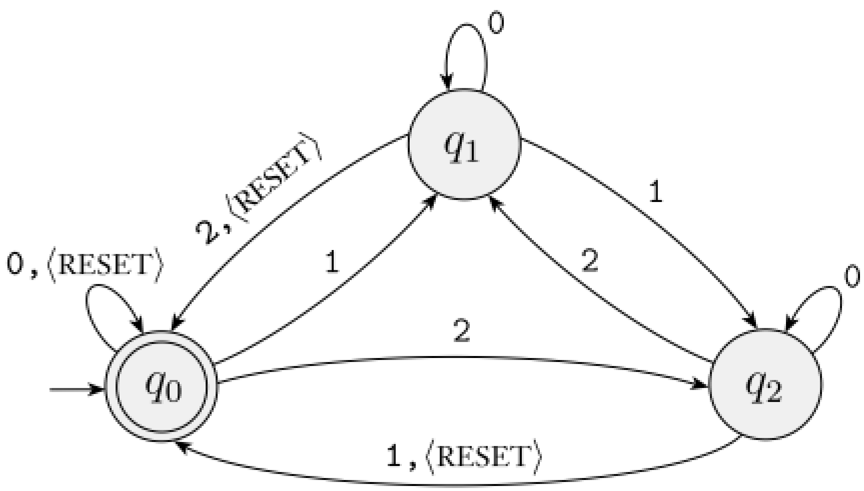
- example



- What language does M_4 recognize?
  - all strings of {a,b} beginning and ending with the same letter

Finite Automata

- example



- think about a counter or accumulator where RESET sets it back to 0
- What language does M_5 recognize?
  - accepts all strings with digits summing to 0 mod 3

Finite Automata

- for some FAs, a state diagram is not possible
  - it may be too large to draw (but not infinite)
  - description depends on an unspecified parameter
  - a formal definition must then be used to specify the machine

- e.g., a generalization of the previous example

Finite Automata

- formal definition of computation
  - let $M = (Q, \Sigma, \delta, q_1, F)$ be a FA and $w = w_1 w_2 \ldots w_n$ be a string where each w_i is a member of the alphabet
  - M accepts w if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with three conditions:
    - $r_0 = q_0$
      - machine starts at start state
    - $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, \ldots, n-1$
      - machine goes from state to state according to transition function
    - $r_n \in F$
      - machine accepts its input if it ends up in an accept state

Finite Automata

- formal definition of computation (cont.)
  - M recognizes language A if
    - $A = \{w \mid M \text{ accepts } w\}$
  - a language is called a regular language if some finite automaton recognizes it

Finite Automata

- designing finite automata
  - cannot be prescribed easily
  - put yourself in the place of the machine
    - you receive a string an input string and must determine whether it is a member of the language the automaton is supposed to recognize
    - process the symbols in the string one by one
      - decide whether the string seen so far is in the language since you don't know when the string will end

Finite Automata

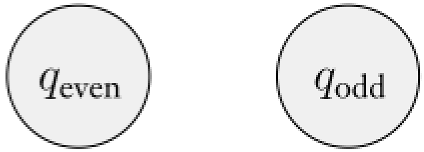
- designing finite automata (cont.)
  - determine what you need to remember about the string as you are reading it
    - input could be very long, but you probably don't need to remember the entire input string
    - you have finite memory, e.g., a single sheet of paper
    - what is the crucial information to remember?

Finite Automata

- designing finite automata
  - example: construct a FA that recognizes the language of all bit strings with an odd number of 1s
    - as you traverse the string, you don't need to remember the entire string
    - simply remember whether the number of 1's seen so far is odd or even
      - if you read a 1, flip the answer
      - if you read a 0, leave the answer as is

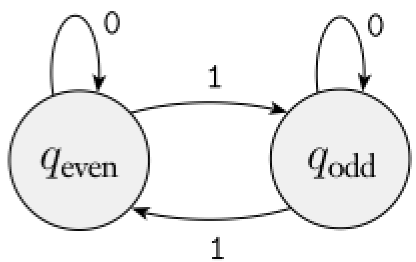
Finite Automata

- designing finite automata
  - example: construct a FA that recognizes the language of all bit strings with an odd number of 1s (cont.)
    - once you have the necessary information to remember, make a finite list of possibilities
      - even so far
      - odd so far
    - assign a state to each of the possibilities

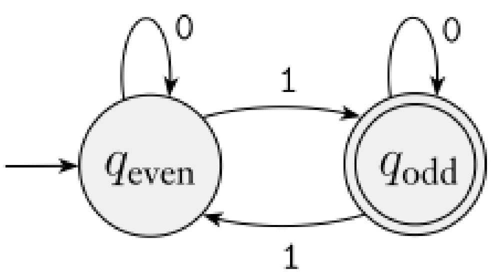


Finite Automata

- designing finite automata
  - example: construct a FA that recognizes the language of all bit strings with an odd number of 1s (cont.)
    - assign transitions to go from one possibility to another



    - set the start state to q_{even} since 0 is an even number
    - set q_{odd} to be the accept state



Finite Automata

- designing finite automata
  - example: construct a FA that recognizes the regular language of all bit strings that contain 001
    - e.g., 0010, 1001, 001, 11111001111, but not 11 and 000
    - if you were the automaton, you would read symbols from the beginning, skipping over all 1s
      - if you read a 0, you may be seeing the start of 001
        - if you read a 1 next, there are too few 0s, so go back to skipping over 1s
        - if you read a 0 next, you need to remember that you have now seen two symbols of the pattern
        - continue scanning until you see a 1 – if so, remember that you have found the pattern, and keep reading to the end of the string

Finite Automata

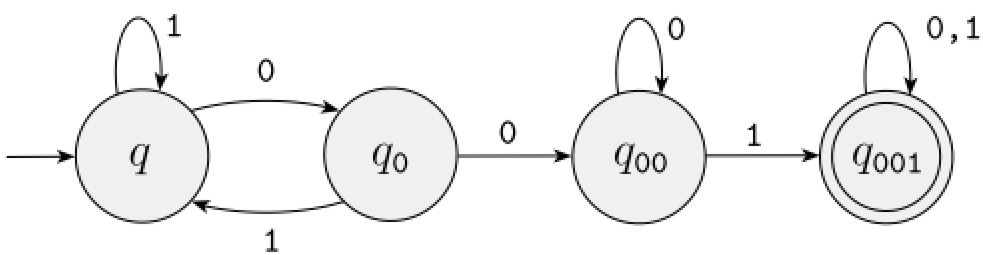
- designing finite automata
  - example: construct a FA that recognizes the regular language of all bit strings that contain 001 (cont.)
    - four different possibilities
      - you haven't seen any symbols of the pattern
      - you have seen just one 0
      - you have seen 00
      - you have seen the entire pattern 001
    - assign states q, q_0, q_{00}, and q_{001} to these possibilities

Finite Automata

- designing finite automata
  - example: construct a FA that recognizes the regular language of all bit strings that contain 001 (cont.)
    - assign the transitions
      - from q
        - if you read a 1, stay in q
        - if you read a 0, go to q_0
      - from q_0
        - if you read a 1, return to q
        - if you read a 0, go to q_{00}
      - from q_{00}
        - if you read a 1, go to q_{001}
        - if you read a 0, stay in q_{00}
      - from q_{001}
        - if you read a 0 or 1, stay in q_{001}

Finite Automata

- designing finite automata
  - example: construct a FA that recognizes the regular language of all bit strings that contain 001 (cont.)
    - start state is q
    - accept state is q_{001}



Finite Automata

- regular operations
  - properties for finite automata
    - help us design FA to recognize particular languages
    - help us determine other languages are nonregular
  - three regular operations
    - union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
    - concatenation: $A \circ B = \{ xy \mid x \in A \text{ and } y \in B\}$
    - star: $A^* = \{x_1x_2 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

    where A and B are regular languages

Finite Automata

- regular operations
  - regular operation notes
    - union: takes all strings in A and B and puts them into one language
    - concatenation: attaches a string from A in front of a string from B in all possible ways to get the new language
    - star: unary rather than binary
      - attaches any number (0 or more) of strings in A to get a string in the new language
      - empty string ε is always a member of A^*

Finite Automata

- regular operations
  - example: $\Sigma = \{a, b, \ldots, z\}$, $A = \{good, bad\}$, $B = \{boy, girl\}$
    - $A \cup B = \{good, bad, boy, girl\}$
    - $A \circ B = \{goodboy, goodgirl, badboy, badgirl\}$
    - $A^* = \{\varepsilon, good, bad, goodgood, goodbad, badgood,$
      $badbad, goodgoodgood, goodgoodbad,$
      $goodbadgood, goodbadbad, \ldots\}$

Finite Automata

- closure
  - consider N = {1, 2, 3, …}
    - N is closed under multiplication means that when we multiply any two numbers from N, we get a product that is also in N
    - N is not closed under division (why?)
  - in general, a collection of objects is closed under some operation if the result of that operation is still in the collection

Finite Automata

- closure
  - regular languages are closed under union
    - if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$
    - proof idea: construct a FA M that recognizes $A_1 \cup A_2$
    - if M_1 recognizes A_1 and M_2 recognizes A_2, then M will simulate both M_1 and M_2, accepting if either M_1 or M_2 accepts
      - cannot simulate M_1 and then M_2
      - cannot rewind the input

Finite Automata

- closure
  - regular languages are closed under union (cont.)
    - instead, simulate M_1 and M_2 simultaneously
      - remember state each machine would be in if it had read the input up to this point
      - if M_1 has k_1 states and M_2 has k_2 states, the number of pairs of states is $k_1 \times k_2$
      - each state in M will be a pair
      - transitions go from pair to pair, updating the current state of both M_1 and M_2
      - accept states are those pairs where either M_1 or M_2 is in an accept state

Finite Automata

- closure
  - proof: regular languages are closed under union (cont.)
    - let M_1 recognize A_1 where $M_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1)$
    - let M_2 recognize A_2 where $M_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2)$

Finite Automata

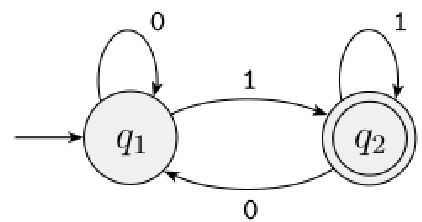
- closure
  - proof: regular languages are closed under union (cont.)
    - construct $M = \{Q, \Sigma, \delta, q_0, F)$ to recognize $A_1 \cup A_2$
      - $Q = \{(r_1, r_2) \mid r_1 \in Q_1$ and $r_2 \in Q_2\}$
        - cartesian product for all pairs of states $Q_1 \times Q_2$
      - Σ alphabet for both
      - δ transition function for each $(r_1, r_2) \in Q$ and $a \in \Sigma$
        - $\delta\,((r_1, r_2), a) = (\delta_1\,(r_1, a), \delta_2\,(r_2, a))$
        - moves from state pair to state pair based on a
      - q_0 is the pair (q_1, q_2)
      - F is set of pairs where M_1 or M_2 is in an accept state
        - $F = \{(r_1, r_2) \mid r_1 \in F_1$ or $r_2 \in F_2\}$ not and

Finite Automata

- closure
  - regular languages are closed under union example
    - let M_1 recognize A_1 where $M_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1\}$

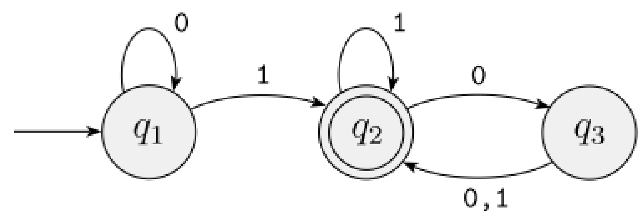
$M_1 = \{\{q_1, q_2\}, \{0,1\}, \delta_1, q_1, \{q_2\}\}$

binary strings ending in 1



- let M_2 recognize A_2 where $M_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2)$

$M_2 = \{\{q_1, q_2, q_3\}, \{0,1\}, \delta_2, q_1, \{q_2\}\}$



binary strings with 1 followed by even number of 0s

Finite Automata

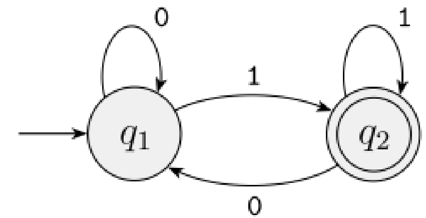
- closure
  - regular languages are closed under union example
    - let M_1 recognize A_1 where $M_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1\}$
    - let M_2 recognize A_2 where $M_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2)$

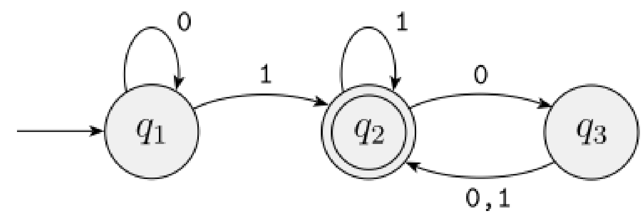
new states: $q_{11}, q_{12}, q_{13}, q_{21}, q_{22}, q_{23}$

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

start state: q_{11}

accept states: $\{q_{12}, q_{21}, q_{22}, q_{23}\}$

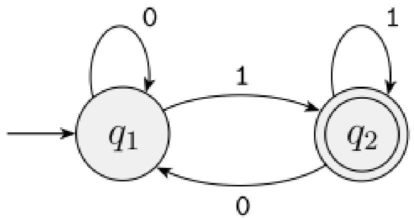


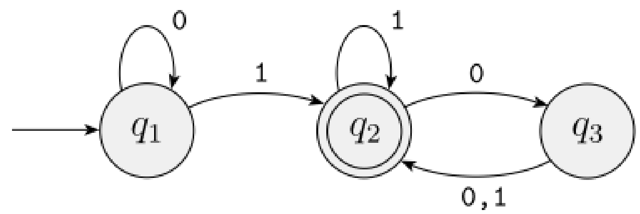


accepts binary strings ending with 1
or containing a 1 followed by an even # of 0s

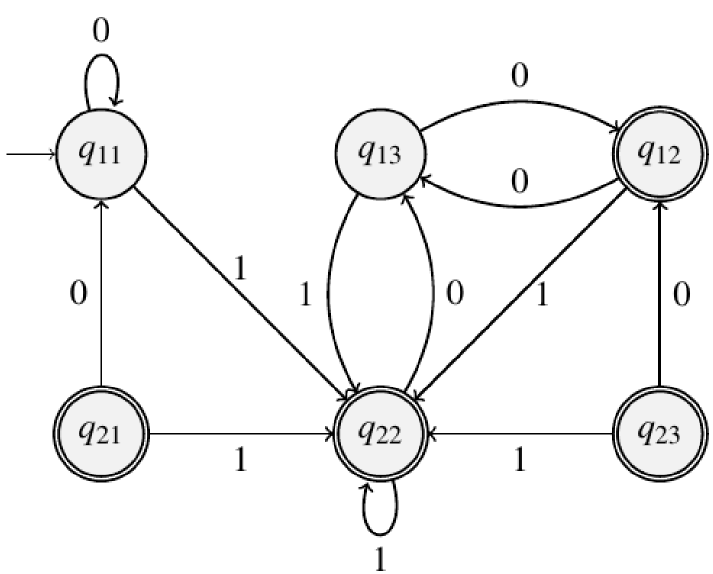
Finite Automata

- closure

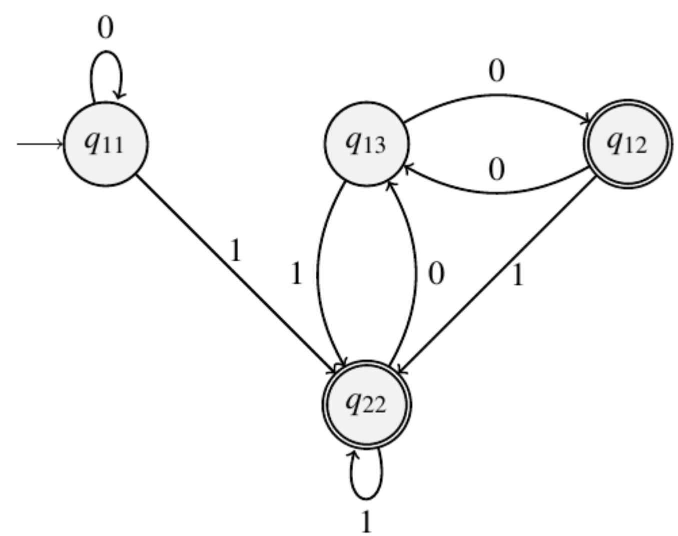




union



simplified



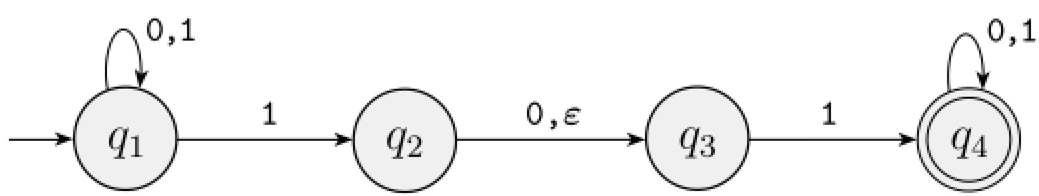
Finite Automata

- closure
  - regular languages are closed under concatenation
    - let M_1 recognize A_1 where $M_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1)$
    - let M_2 recognize A_2 where $M_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2)$
    - construct M to accept input first for M_1, then for M_2
    - BUT, M doesn't know where to break its input
      - where the first part ends and the second part begins

  - we need to introduce a new technique called nondeterminism

Nondeterminism

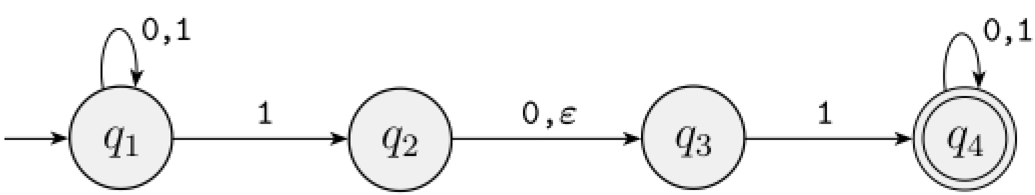
- so far, we have considered only deterministic finite automata (DFA)
  - i.e., when a machine is in a given state and reads the next input symbol, there is only one state that can be the next state
- in a nondeterministic machine, several choices may exist for the next state
  - nondeterminism is a generalization of determinism



  - what do you notice that is different in this NFA?

Nondeterminism

- differences between DFAs and NFAs
  - DFAs: states may have exactly one exiting arrow for each symbol
    NFAs: a state may have zero, one, or many exiting arrows for each symbol
  - DFAs: labels on transition arrows are symbols from the alphabet
    NFAs: labels on transition arrows are symbols from the alphabet or ε; zero, one, or many arrows may exit from each state with label ε



Nondeterminism

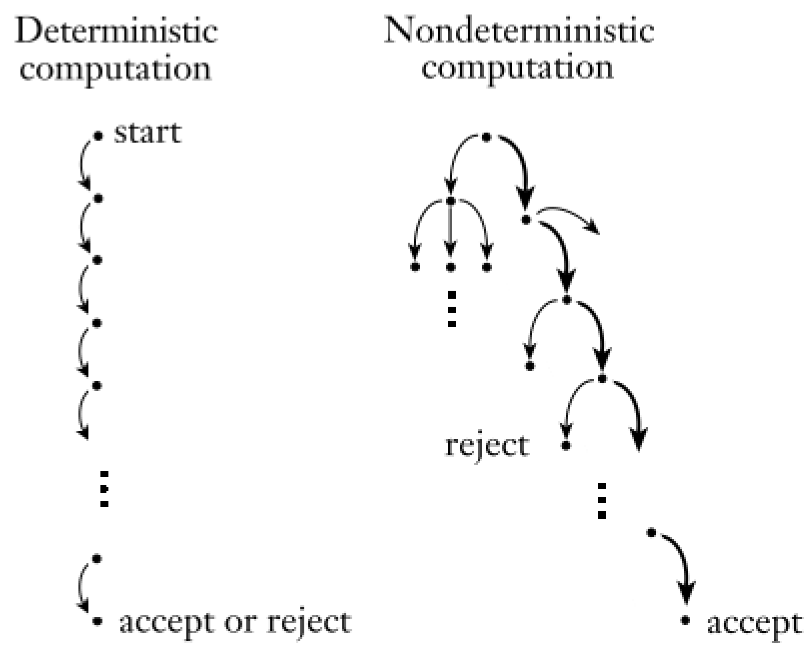
- how does an NFA compute?
  - if multiple ways to proceed exist after reading a symbol, the machine splits into multiple copies of itself and follows all possibilities in parallel
  - machine also splits for all ε branches that can be taken
  - each copy takes one of the possible ways to proceed and continues as before
  - each machine continues to split as needed
  - if the next input symbol does not match an exiting arrow for a machine's current state, that copy of the machine dies, along with its branch of computation
  - if any one of the copies reaches an accept state at the end of the input, the NFA accepts the input string

Nondeterminism

- nondeterminism can be viewed as a parallel computation
  - multiple independent “processes” or “threads” can be running concurrently
  - each split corresponds to a process forking into multiple children, with each proceeding separately
  - if at least one of these processes accepts, then the entire computation accepts

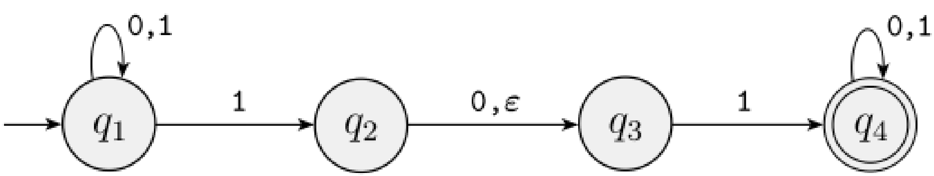
Nondeterminism

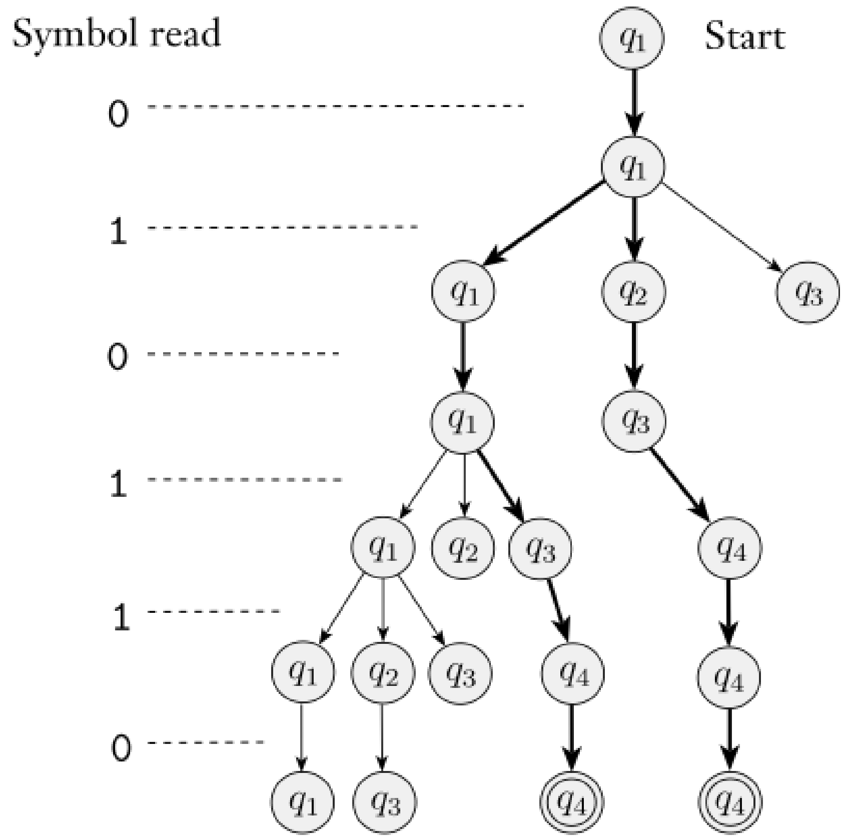
- nondeterminism can be viewed as a tree of possibilities
  - root is the start of the computation
  - branches signify the machine splitting across multiple choices
  - machine accepts if at least one branch ends in an accept state



Nondeterminism

- example: NFA N_1 on 010110
  - keep track of possibilities by placing fingers over each state where a machine could be





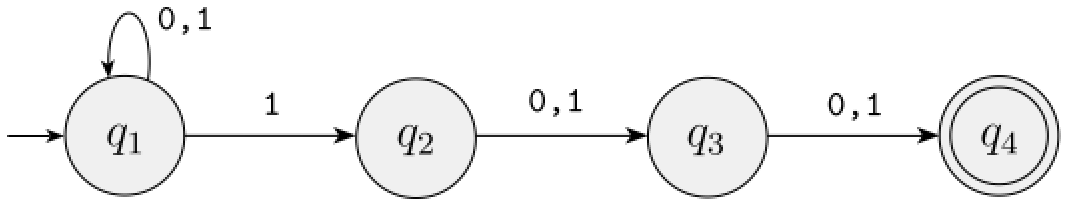
- what about 010?
- what language does this accept?
  - all strings with 101 or 11

Nondeterminism

- NFAs are useful in several ways
  - every NFA can be converted directly into a DFA
  - constructing NFAs is sometimes easier than directly constructing DFAs
  - an NFA may be much smaller or easier to understand than its corresponding DFA

Nondeterminism

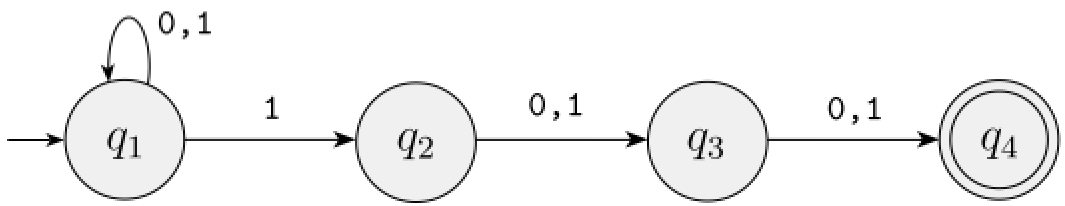
- example NFA N_2



  - what language does it accept?
    - all binary strings with 1 in third-to-last position

Nondeterminism

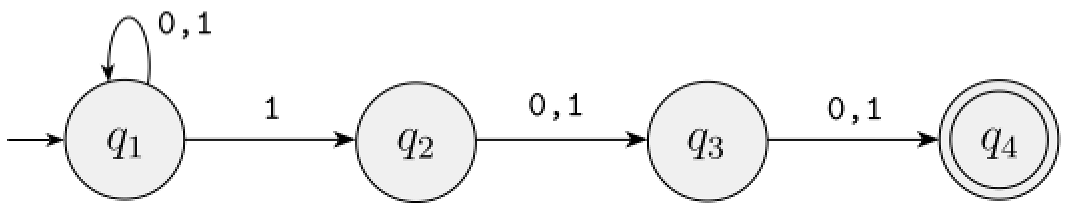
- example NFA N_2



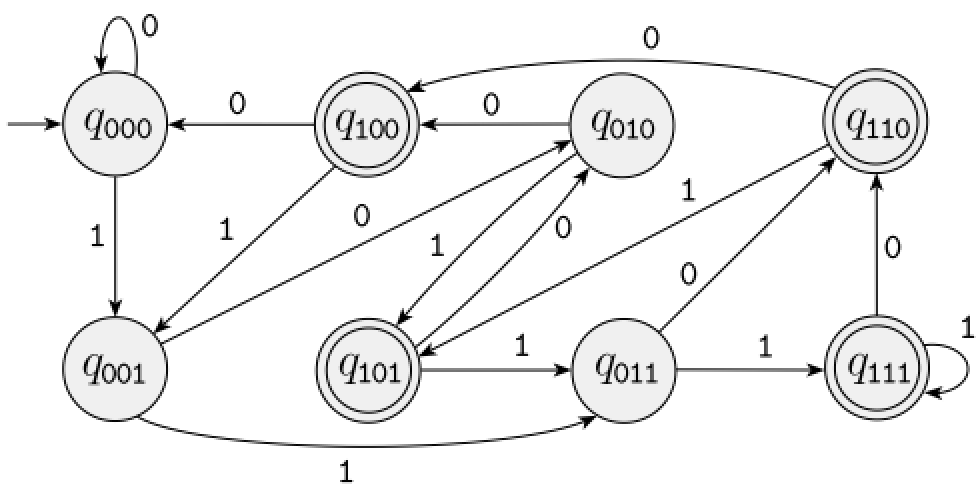
  - can think of it as staying in the start state until it guesses that it is three places from the end
  - at that point, if the next symbol is 1, it branches to q_2 and uses q_3 and q_4 to check its guess

Nondeterminism

- example NFA N_2

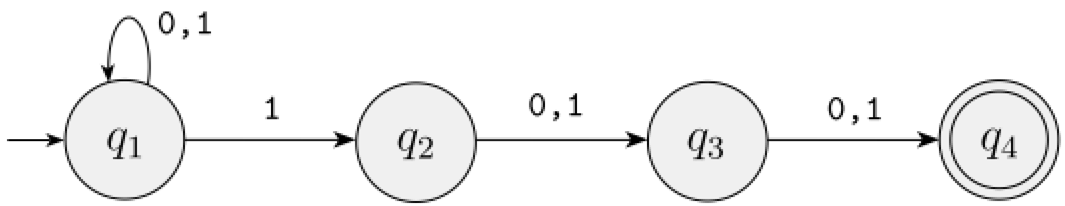


  - this NFA can be converted to an equivalent DFA, but with more states and transitions
  - smallest equivalent DFA



Nondeterminism

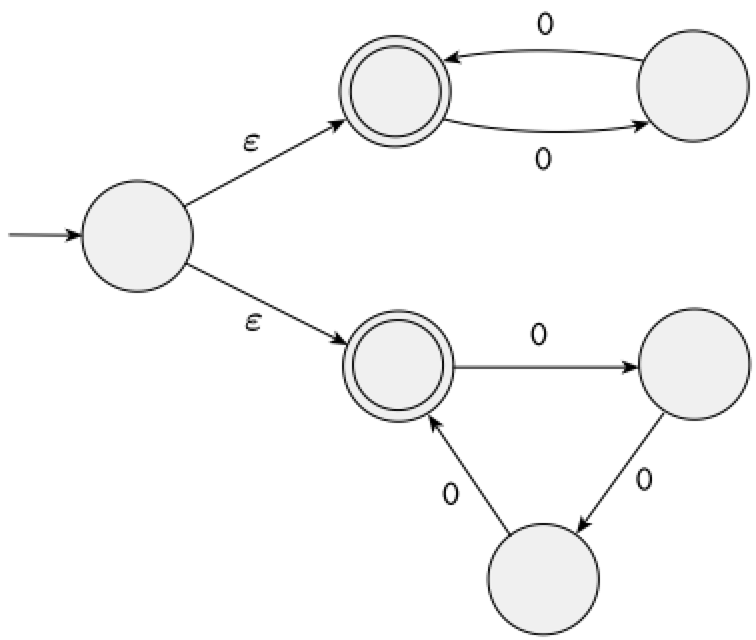
- example NFA N_2



  - what language would N_2 recognize if edges with labels ε were added from q_2 to q_3 and from q_3 to q_4?
    - all binary strings containing a 1 in any of the last three positions
  - how would the corresponding DFA change?

Nondeterminism

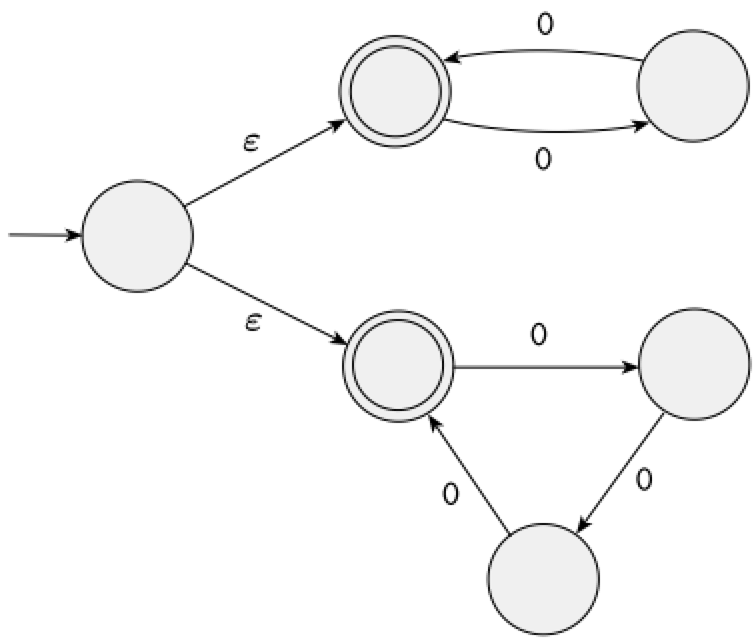
- example NFA N_3



- unary alphabet {0}
- what language does this accept?

Nondeterminism

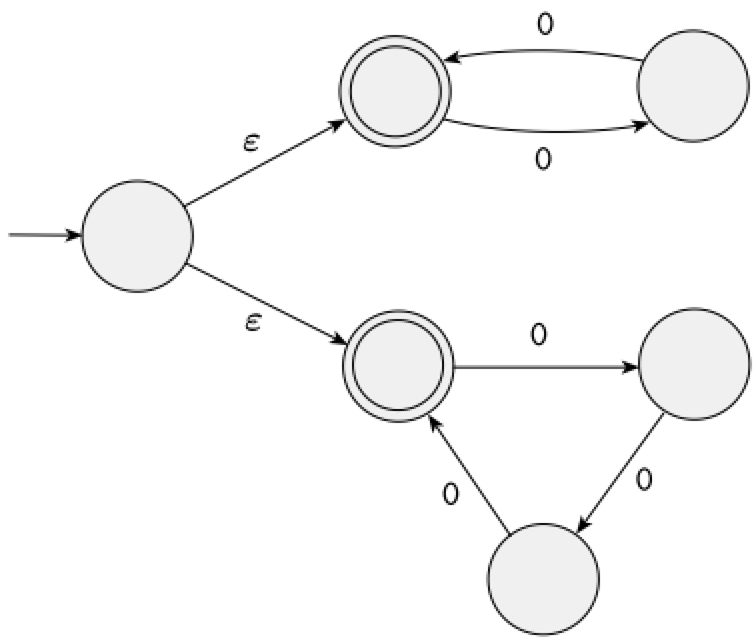
- example NFA N_3



- accepts ε, 00, 000, 0000, 000000 but not 0, 00000
- accepts all strings 0^k where k is 0 or a multiple of 2 or 3

Nondeterminism

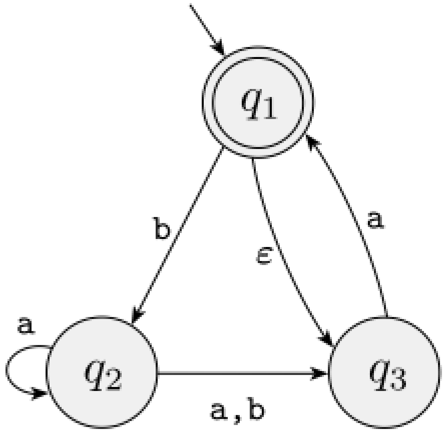
- example NFA N_3



- think of the machine as guessing whether to test for multiples of 2 or 3
- could use a DFA instead, but N_3 is easiest to understand

Nondeterminism

- example NFA N_4



- accepts ε, a, baba, baa
- does not accept b, bb, babba
- so, the language consists of ε and strings composed of a's and b's, but always ending in a
  - more limitations, but this language cannot be easily and succinctly described

Nondeterminism

- formal definition of NFA
  - similar to DFA, but transition functions are different
    - in NFA, transition function takes a state and an input symbol *or the empty string* and produces a *set* of possible next states
  - recall P(Q) is the power set (set of all subsets)
  - alphabet must add ε
    - $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$

Nondeterminism

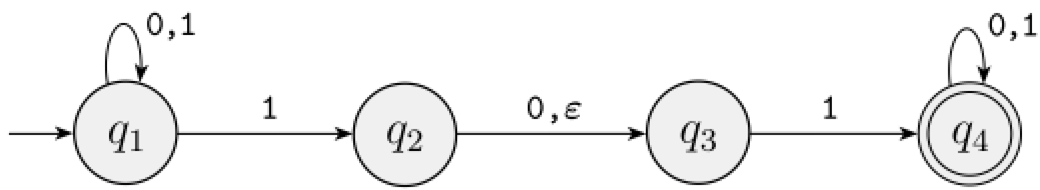
- formal definition of NFA

A ***nondeterministic finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta \colon Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

Nondeterminism

- example N_1



- formal definition
  - $Q = \{q_1, q_2, q_3, q_4\}$
  - $\Sigma = \{0, 1\}$
  - δ is given as

	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	$\{q_4\}$	$\{q_4\}$	$\emptyset$,

  - q_1 is the start state
  - $F = \{q_4\}$

Nondeterminism

- formal definition of computation
  - similar to DFA
  - let $N = (Q, \Sigma, \delta, q_0, F)$ be a NFA and w a string over alphabet Σ
  - N accepts w if we can write w as $w = y_1 y_2 \ldots y_n$ where each y_i is a member of Σ_ε and the sequence of states $r_0, r_1, \ldots r_n$ in Q exists with three conditions:
    - $r_0 = q_0$
      - machine starts at start state
    - $r_{i+1} \in \delta\,(r_i, y_{i+1})$ for $i = 0, \ldots, m-1$
      - machine goes from state r_i to r_{i+1} which is a member of the set of allowable next states according to transition function
    - $r_m \in F$
      - machine accepts its input if it ends up in an accept state

Nondeterminism

- equivalence of NFAs and DFAs
  - deterministic and nondeterministic FAs recognize the same class of languages
  - surprising since NFAs seem more powerful
  - useful because NFAs are often easier to construct and understand
  - two machines are equivalent if they recognize the same language

Nondeterminism

- Theorem: every nondeterministic finite automaton has an equivalent deterministic finite automaton
  - proof idea
    - convert NFA to equivalent DFA that simulates it
    - consider what happens as input is read
    - what do you need to keep track of?
      - various branches of computation by placing fingers over active states
    - if the NFA has k states, there are 2^k subsets of states
    - each subset corresponds to one state the DFA will need to keep track of, so the DFA will have 2^k states
    - set start and accept states for DFA

Nondeterminism

- proof
  - let $N = (Q, \Sigma, \delta, q_0, F)$ be the NFA recognizing A
    - construct DFA $M = (Q', \Sigma, \delta', q_0', F')$ recognizing A
  - first consider case where N has no ε edges
    - $Q' = P(Q)$
      - every state of M is a set of states of N
    - let $\delta'(R, a) = \{q \in Q \mid q \in \delta(r, a) \text{ for some } r \in R\}$ where $R \in Q$
      - if R is a state of M, it is also a set of states of N; when M reads a symbol a in R, it goes to one or more states in R, so $\delta'(R, a) = \cup_{r \in R} \delta(r, a)$
    - $q_0' = \{q_0\}$
      - M starts in the state corresponding to the collection containing just the start state of N
    - $F' = \{R \in Q' \mid R \text{ contains an accept state of } N\}$
      - machine accepts if one of the possible states that N could be in at this point is an accept state

Nondeterminism

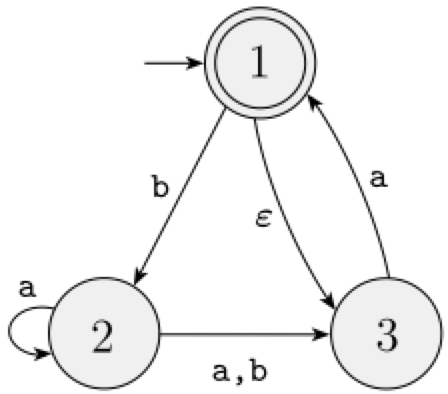
- proof (cont.)
  - now consider ε edges
    - for any state R of M, E(R) is the collection of states that can be reached from members of R by following ε arrows, including the members of R themselves
      - $E(R) = \{q \mid q$ can be reached from R by 0 or more ε arrows$\}$
    - modify transition function to include states reached by ε arrows
      - $\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a))$ for some $r \in R\}$
    - modify start state $q_0' = E(\{q_0\})$

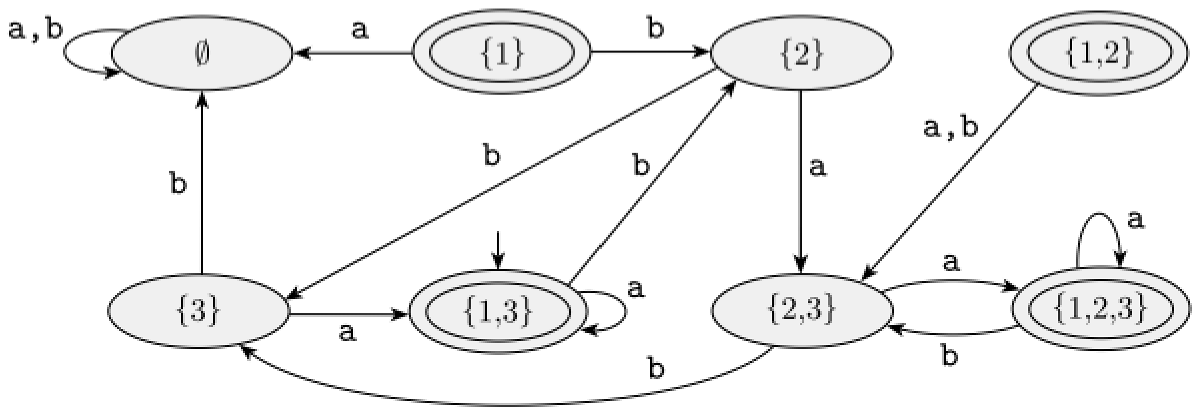
Nondeterminism

- corollary
  - a language is regular if and only if some nondeterministic finite automaton recognizes it

Nondeterminism

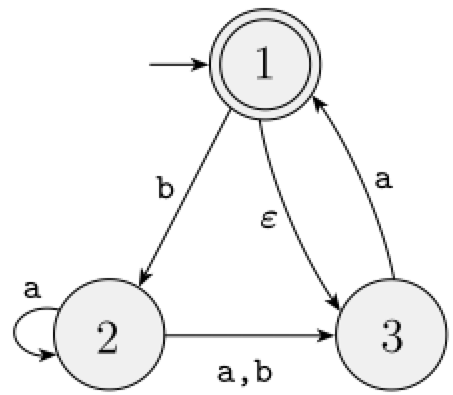
- convert NFA N_4 to a DFA





Nondeterminism

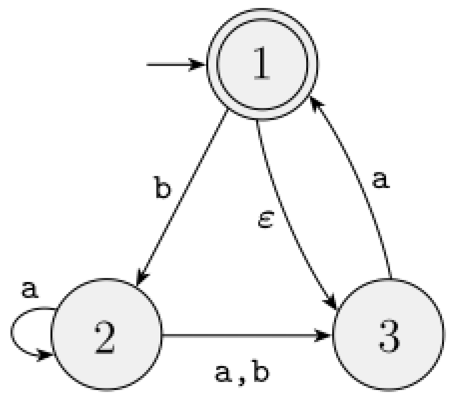
- convert NFA N_4 to a DFA



- $N_4 = (Q, \{a, b\}, \delta, 1, \{1\})$ where $Q = \{1, 2, 3\}$
- DFA D's states will be
  - $P(Q) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$
- D's start state = $E(\{1\}) = \{1, 3\}$
- D's accept states = $\{\{1\}, \{1,2\}, \{1,3\}, \{1,2,3\}\}$
  - anything containing the N_4's accept states

Nondeterminism

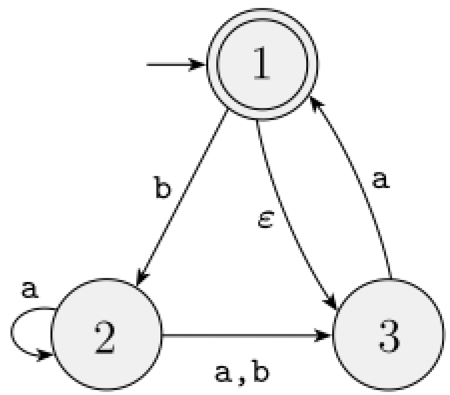
- convert NFA N_4 to a DFA (cont.)



  - D's transition function
    - each of D's states must go to one place on input a and one place on input b
      - state {2} goes to {2,3} on a and {3} on b
      - state {1} goes to Ø on a and {2} on b
        - note: follow ε arrows as a new state is entered (start state or state reached by input symbol)

Nondeterminism

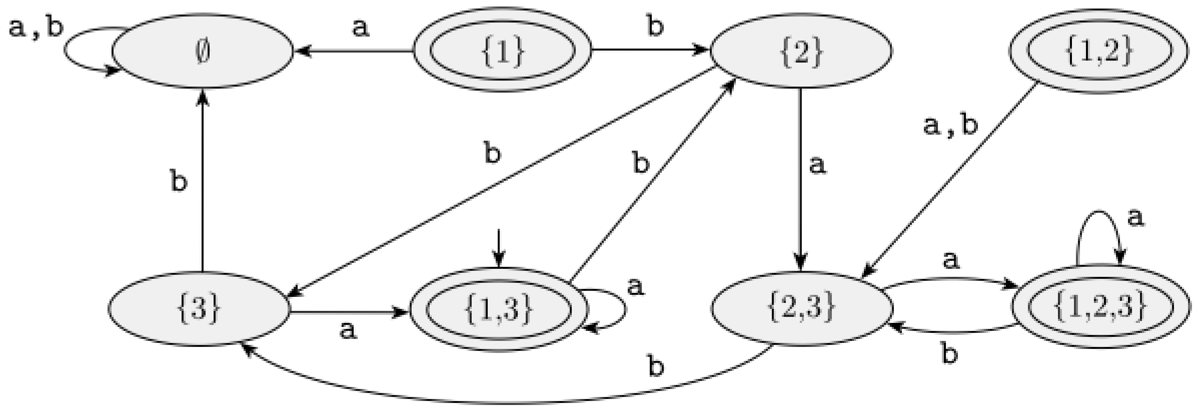
- convert NFA N_4 to a DFA (cont.)



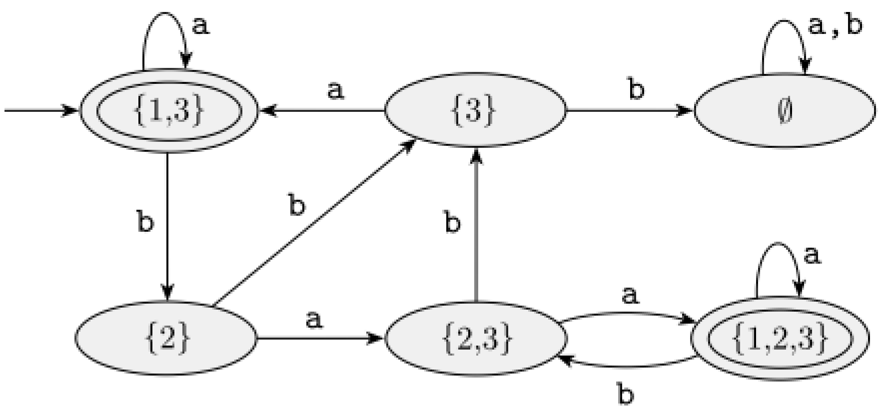
- D's transition function (cont.)
  - state {3} goes to {1,3} on a and Ø on b
  - state {1,2} goes to {2,3} on a and {2,3} on b
  - etc.

Nondeterminism

- convert NFA N_4 to a DFA (cont.)
  - DFA D



- DFA D simplified
  - remove states that cannot be reached



Nondeterminism

- closure under the regular operations
  - remember that we started this topic on nondeterminism because we needed NFA to prove regular operations were closed under
    - union
    - concatenation
    - star

Nondeterminism

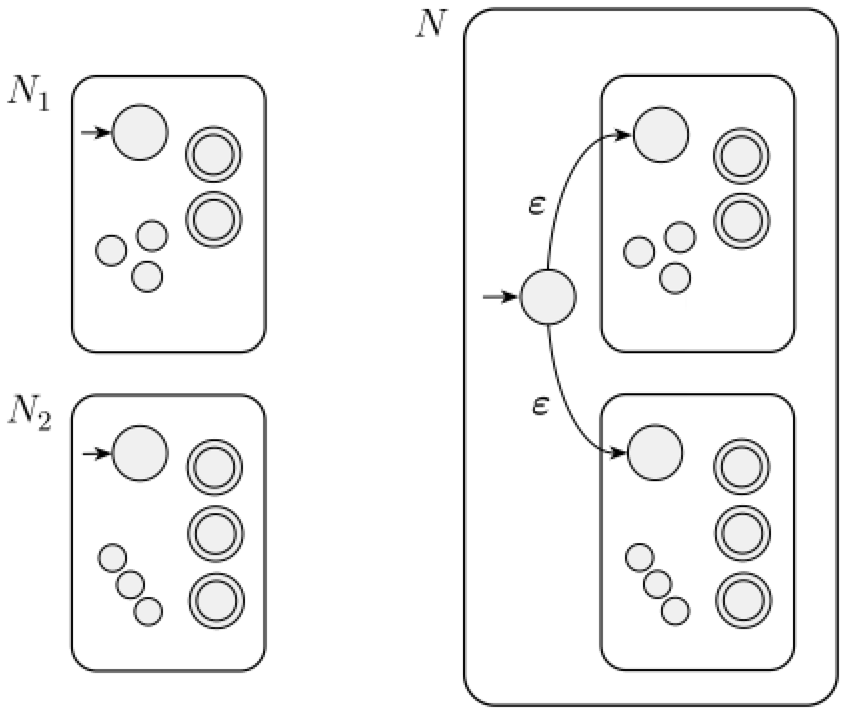
- closure under union
  - we proved closure under union before by simulating both machines simultaneously
    - the new proof using nondeterminism is easier

Nondeterminism

- closure under union (cont.)
  - if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$
  - proof idea: construct NFA N that recognizes $A_1 \cup A_2$
  - if N_1 recognizes A_1 and N_2 recognizes A_2, then N will combine N_1 and N_2, accepting if either N_1 or N_2 accepts
    - N has new start state that branches to the start states of N_1 and N_2 with ε arrows
    - N nondeterministically guesses which machine accepts the input
    - if either N_1 or N_2 accepts, N will accept, too

Nondeterminism

- closure under union (cont.)



Nondeterminism

- closure under union (cont.)
  - proof: regular languages are closed under union
    - let N_1 recognize A_1 where $N_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1)$
    - let N_2 recognize A_2 where $N_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2)$
    - construct $N = \{Q, \Sigma, \delta, q_0, F)$ to recognize $A_1 \cup A_2$
      - $Q = \{q_0\} \cup Q_1 \cup Q_2$
        - the states of N are all states of N_1 and N_2 with new start state q_0
      - Σ alphabet for both
      - δ transition function for each $q \in Q$ and $a \in \Sigma_\varepsilon$
        - $\delta(q, a) = \delta_1(q, a) \quad q \in Q_1$
        - $\delta(q, a) = \delta_2(q, a) \quad q \in Q_2$
        - $\delta(q, a) = \{q_1, q_2\} \quad q = q_0$ and $a = \varepsilon$
        - $\delta(q, a) = \varnothing \quad q = q_0$ and $a \neq \varepsilon$
      - q_0 is the start state of N
      - $F = F_1 \cup F_2$
        - the accept states of N are all the accept states of N_1 and N_2 so that N accepts if either N_1 or N_2 accepts

Nondeterminism

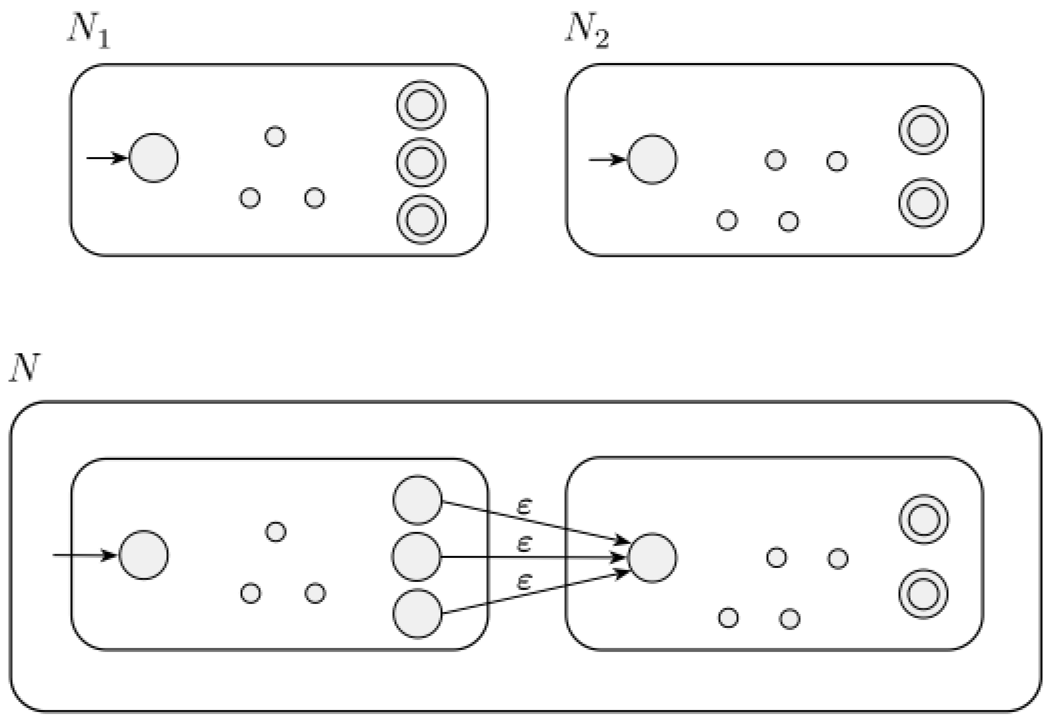
- closure under concatenation
  - we tried earlier to prove closure under concatenation, but we didn't finish because it was too difficult
    - the new proof using nondeterminism is easier

Nondeterminism

- closure under concatenation (cont.)
  - if A_1 and A_2 are regular languages, so is $A_1 \circ A_2$
  - proof idea: construct NFA N that recognizes $A_1 \circ A_2$
  - if N_1 recognizes A_1 and N_2 recognizes A_2, then N will combine N_1 and N_2
    - start state of N is assigned to the start state of N_1
    - the accept states of N_1 have additional ε arrows that nondeterministically allow branching to N_2 whenever N_1 is in an accept state
      - i.e., the first part of the concatenation has been found
    - accept states of N are the accept states of N_2 only
      - accepts when input split into two parts: N_1 and N_2
      - nondeterministically guesses where to make split

Nondeterminism

- closure under concatenation (cont.)



Nondeterminism

- closure under concatenation (cont.)
  - proof: regular languages are closed under concatenation
    - let N_1 recognize A_1 where $N_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1)$
    - let N_2 recognize A_2 where $N_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2)$
    - construct $N = \{Q, \Sigma, \delta, q_0, F)$ to recognize $A_1 \circ A_2$
      - $Q = Q_1 \cup Q_2$
        - the states of N are all states of N_1 and N_2
      - Σ alphabet for both
      - δ transition function for any $q \in Q$ and any $a \in \Sigma_\varepsilon$
        - $\delta(q, a) = \delta_1(q, a)$ $q \in Q_1$ and $q \notin F_1$
        - $\delta(q, a) = \delta_1(q, a)$ $q \in F_1$ and $a \neq \varepsilon$
        - $\delta(q, a) = \delta_1(q, a) \cup \{q_2\}$ $q \in F_1$ and $a = \varepsilon$
        - $\delta(q, a) = \delta_2(q, a)$ $q \in Q_2$
      - q_1 is the start state of N
      - $F = F_2$
        - the accept states of N are all the accept states of N_2

Nondeterminism

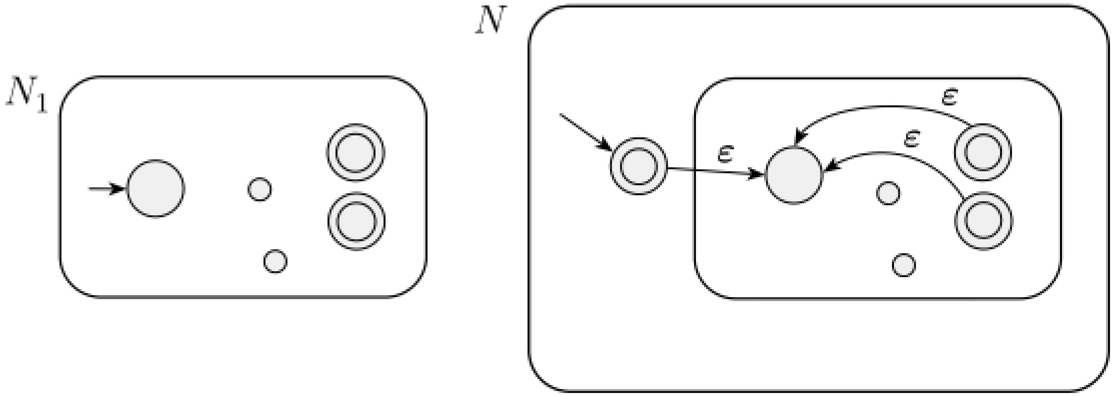
- closure under star
  - if A_1 is a regular languages, so is A_1*
  - proof idea: construct NFA N that recognizes A_1*
  - modify N_1 that recognizes A_1 to produce N
    - N will accept its input whenever it can be broken into several pieced and N_1 accepts each piece

Nondeterminism

- closure under star (cont.)
  - proof idea: construct NFA N that recognizes A_1*
  - modify N_1 that recognizes A_1 to produce N
    - N will be similar to N_1, but with additional ε arrows returning to the start state from the accept states
      - when processing gets to the end of a piece that N_1 accepts, you can jump back to the start state to try to read another piece that N_1 accepts
    - N must also accept ε, which is always a member of A_1*
      - could add start state to set of accept states, but may cause other bad strings to be accepted
      - instead, add a new start state that is also an accept state and that has an ε arrow to the old start state

Nondeterminism

- closure under star (cont.)



Nondeterminism

- closure under star (cont.)
  - proof: regular languages are closed under star
    - let N_1 recognize A_1 where $N_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1)$
    - construct $N = \{Q, \Sigma, \delta, q_0, F)$ to recognize $A_1{}^*$
      - $Q = \{q_0\} \cup Q_1$
        - the states of N are states of N_1 plus new start state
      - Σ alphabet
      - δ transition function for each $q \in Q$ and $a \in \Sigma_\varepsilon$
        - $\delta(q, a) = \delta_1(q, a)$ $\quad q \in Q_1$ and $q \notin F_1$
        - $\delta(q, a) = \delta_1(q, a)$ $\quad q \in F_1$ and $a \neq \varepsilon$
        - $\delta(q, a) = \delta_1(q, a) \cup \{q_1\}$ $\quad q \in F_1$ and $a = \varepsilon$
        - $\delta(q, a) = \{q_1\}$ $\quad q = q_0$ and $a = \varepsilon$
        - $\delta(q, a) = \emptyset$ $\quad q = q_0$ and $a \neq \varepsilon$
      - q_0 is the new start state of N
      - $F = \{q_0\} \cup F_1$
        - the accept states are old accept states plus new start state

Regular Expressions

- in arithmetic, we can use operations + and x to build expressions
  - (5 + 3) x 4
  - value?
- similarly, we use regular expression operations to build up regular expressions
  - $(0 \cup 1)0^*$
  - value: a language consisting of all strings starting with 0 or 1 followed by any number of 0s

Regular Expressions

- similarly, we use regular expression operations to build up regular expressions (cont.)
  - $(0 \cup 1)0^*$
  - in this example
    - $(0 \cup 1)$ is short for $(\{0\} \cup \{1\})$
      - value is language $\{0, 1\}$
    - 0^* means $\{0\}^*$
      - value is language of all strings containing any number of 0s
    - concatenation symbol can be implicit
      - instead of $(0 \cup 1) \circ 0^*$, it's just $(0 \cup 1)0^*$
        - like multiplication

Regular Expressions

- regular expressions are important in computer science applications
  - e.g., search for strings with specific patterns
  - regular expressions are used in
    - awk and grep in Unix/Linux
    - Perl
      - e.g., $myfilesearch =~ s/"//g;
    - text editors

Regular Expressions

- e.g., $(0 \cup 1)^*$
  - value is language of all possible strings of 0s and 1s
- if $\Sigma = \{0, 1\}$
  - Σ is shorthand for $(0 \cup 1)$
  - Σ describes language consisting of all strings of length 1 over this alphabet
  - Σ^* describes language consisting of all strings over this alphabet
  - Σ^*1 is all strings that end in 1
  - $(0\Sigma^*) \cup (\Sigma^*1)$ is all strings that start with 0 or end with 1

Regular Expressions

- in arithmetic, x has precedence over +
  - 2 + 3 x 4
  - value?
- to change the precedence, must use parentheses
  - (2 + 3) x 4

- precedence in regular expressions
  - ()
  - *
  - concatenation
  - union

Regular Expressions

- R is a regular expression if R is
  - a for some a in Σ
  - ε
  - $\emptyset$
  - $(R_1 \cup R_2)$ where R_1 and R_2 are regular expressions
  - $(R_1 \circ R_2)$ where R_1 and R_2 are regular expressions
  - $(R_1{}^*)$ where R_1 is a regular expressions

- careful with ε and $\emptyset$
  - ε – the language containing one string: the empty string
  - $\emptyset$ - the language containing no strings
- using R_1 and R_2 in definition not circular, but inductive

Regular Expressions

- R^+ - shorthand for RR^*
  - R^* - 0 or more concatenations from R
  - R^+ - 1 or more concatenations from R
    - $R^+ \cup \varepsilon = R^*$
  - R^k – k concatenations of R
- L(R) – language of R

Regular Expressions

- regular expression exercises
  - 0^*10^* =
    - {w | w contains a single 1}
  - $\Sigma^*1\Sigma^*$ =
    - {w | w contains at least one 1}
  - $\Sigma^*001\Sigma^*$ =
    - {w | w contains the substring 001}
  - $1^*(01^+)^*$ =
    - {w | every 0 in w is followed by at least one 1}
  - $(\Sigma\Sigma)^*$ =
    - {w | w is a string of even length}
  - $(\Sigma\Sigma\Sigma)^*$ =
    - {w | the length of w is a multiple of 3}

Regular Expressions

- regular expression exercises (cont.)
  - $01 \cup 10 =$
    - $\{01, 10\}$
  - $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 =$
    - $\{w \mid w$ starts and end with the same symbol$\}$
  - $(0 \cup \varepsilon)1^* =$
    - $01^* \cup 1^*$
  - $(0 \cup \varepsilon)(1 \cup \varepsilon) =$
    - $\{\varepsilon, 0, 1, 01\}$
  - $1^*\varnothing =$
    - $\varnothing$
  - $\varnothing^* =$
    - $\{\varepsilon\}$

Regular Expressions

- regular expression identities
  - $R \cup \varnothing = R$
  - $R \circ \varepsilon = R$
  - $R \cup \varepsilon$ may not $= R$
    - if $R = 0$ then $L(R) = \{0\}$ but $L(R \cup \varepsilon) = \{0, \varepsilon\}$
  - $R \circ \varnothing$ may not $= R$
    - if $R = 0$ then $L(R) = \{0\}$ but $L(R \circ \varnothing) = \varnothing$

Regular Expressions

- regular expressions are useful for designing compilers for programming languages
  - tokens, such as constants or variable names, may be described using regular expressions
  - e.g., numerical constant that may include a fractional part and/or a sign can be described as

$$(+ \cup - \cup \varepsilon)\,(D^{+} \cup D^{+}.D^{*} \cup D^{*}.D^{+})$$

    - examples: 72, 3.14159, +7., and -.01
  - once syntax has been described with regular expressions in terms of its tokens, a lexical analyzer that processes the program can be generated

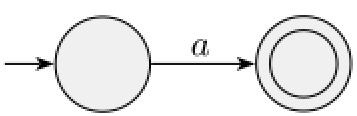
Regular Expressions

- regular expressions are equivalent to finite automata
  - surprising since they appear to be quite different
  - a regular expression that describes a language can be converted into a FA that recognizes that language, and vice versa

- Theorem: *A language is regular if and only if some regular expression describes it.*
  - iff requires proof in each direction

Regular Expressions

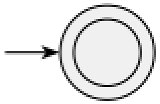
- Proof:
  - if a language is described by a regular expression, it is regular
    - Proof idea: convert R describing A into an NFA recognizing A
    - Proof: consider 6 cases
      1. $R = a$ for some $a \in \Sigma$, so $L(R) = \{a\}$ that can be recognized by the following NFA (easier than DFA)



- note that this is an NFA (why?)
- $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$ where δ is shown above

Regular Expressions

- Proof: (cont.)
  - Proof: consider 6 cases
    2. $R = \varepsilon$, so $L(R) = \{\varepsilon\}$ that can be recognized by the following NFA (easier than DFA)

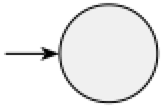


- $N = \{\{q_1\}, \Sigma, \delta, q_1, \{q_1\}\}$ where
  - $\delta(r, b) = \varnothing$ for any r and b

Regular Expressions

- Proof: (cont.)
  - Proof: consider 6 cases

3. $R = \emptyset$, so $L(R) = \emptyset$ that can be recognized by the following NFA



- $N = (\{q\}, \Sigma, \delta, q, \emptyset)$ where
  - $\delta(r, b) = \emptyset$ for any r and b

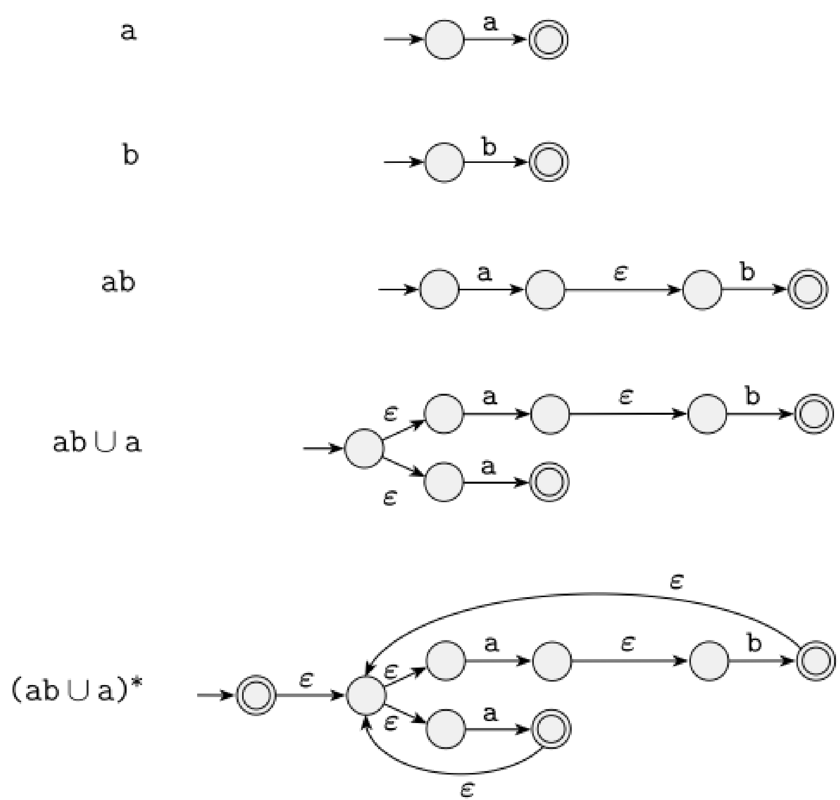
Regular Expressions

- Proof: (cont.)
  - Proof: consider 6 cases

    4. $R = R_1 \cup R_2$
    5. $R = R_1 \circ R_2$
    6. $R = R_1{}^*$

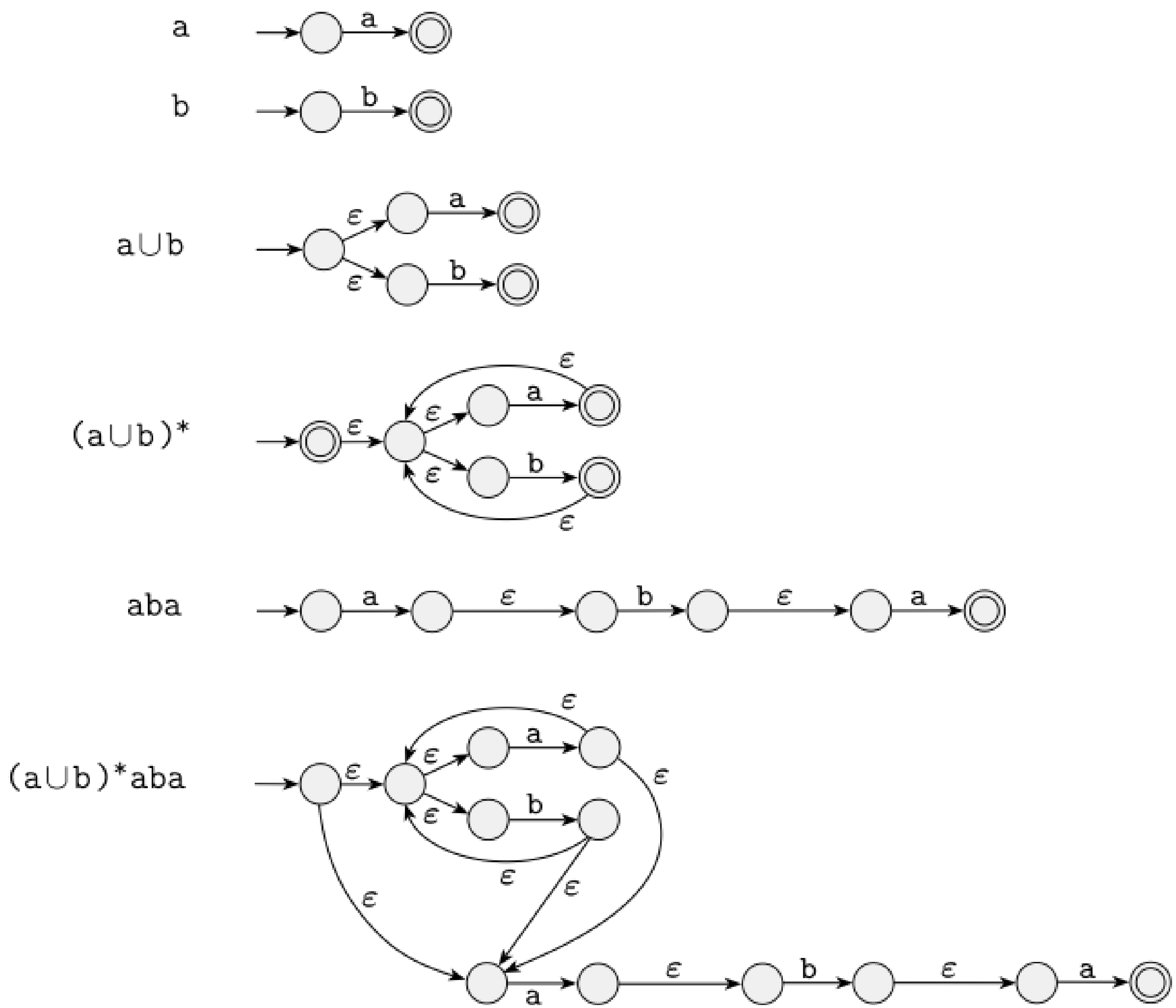
    - for these last three cases, we use constructions given in the proofs of regular languages closed under these operations

- example: build an NFA from the RE (ab ∪ a)*
  - start with smallest and build up
    - this technique generally does not result in an NFA with the fewest states (2 states for this NFA)



Regular Expressions

- example: build an NFA from the RE (a ∪ b)*aba



follows from
def of concat

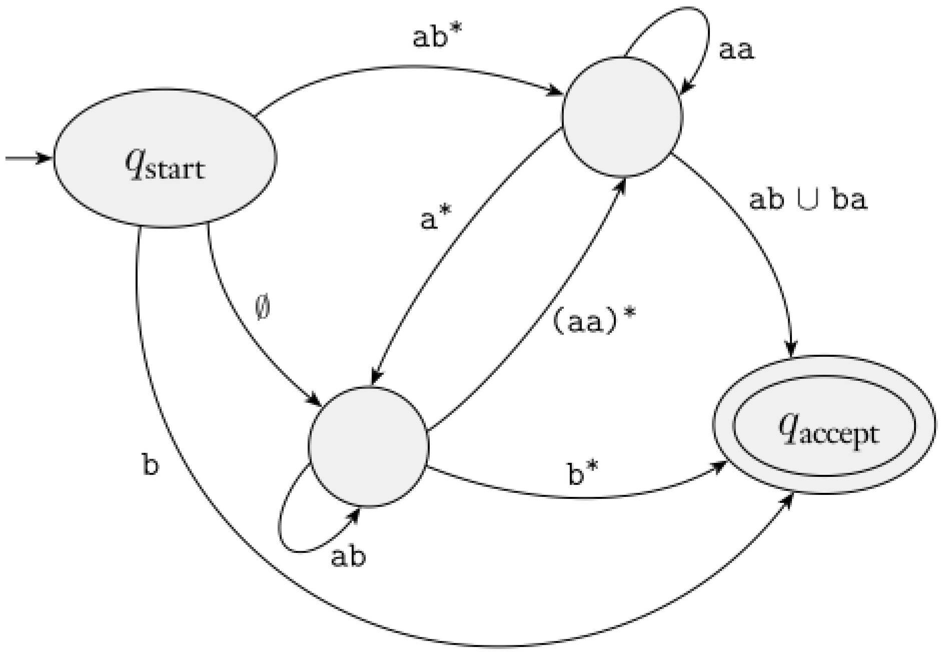
Regular Expressions

- Proof: (cont.)
  - if a language is regular, then it is described by a regular expression
    - Proof idea: if *A* is regular, it is accepted by a DFA; convert the DFA into an equivalent regular expression
      - break procedure into two parts using a GNFA (generalized nondeterministic finite automaton)
        - convert DFA to GNFA
        - GNFA to regular expression

Regular Expressions

- Proof: (cont.)
  - GNFA (generalized nondeterministic finite automaton)
    - NFA with transition arrows that may have regular expressions as labels
    - can read blocks of symbols instead of just one at a time
    - moves along transition arrow by reading a block of symbols representing a string described by the RE on that arrow
    - nondeterministic so may have different ways to process the same input string
    - accepts if in an accept state at end of input

- Proof: (cont.)
  - example: GNFA



Regular Expressions

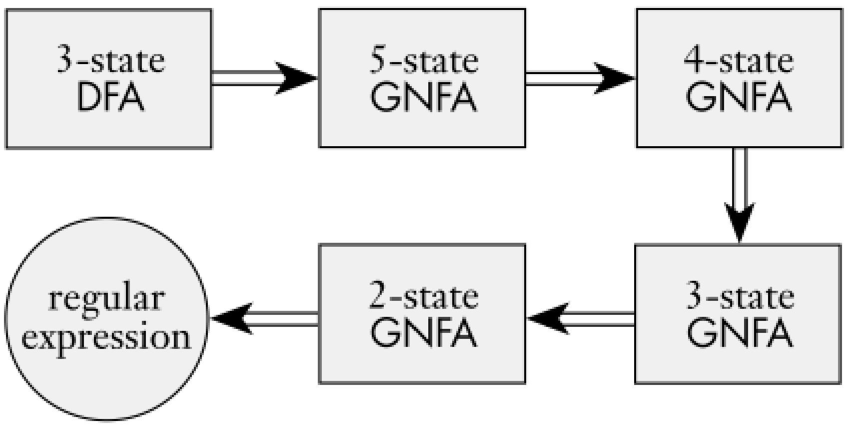
- Proof: (cont.)
  - for convenience, we will require GNFAs to have a special form
    - the start state has transition arrows going to every other state but no arrows coming in from any other state
    - only one accept state with arrows coming in from every other state but no arrows going to any other states; cannot be the same as the start state
    - except for the start and accept states, one arrow goes from every state to every other state and to itself

Regular Expressions

- Proof: (cont.)
  - easy to convert a GNFA into a RE
    - if GNFA has k states, $k \geq 2$ since at least a start and accept state
      - if $k > 2$, we can construct an equivalent GNFA with $k - 1$ states
      - this step can be repeated on a GNFA until it is reduced to just 2 states
      - if $k = 2$, the GNFA has a single arrow from start to accept state with the label being the equivalent of the RE

Regular Expressions

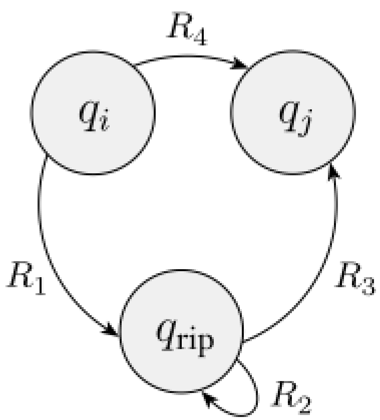
- Proof: (cont.)
  - stages to convert a GNFA into a RE



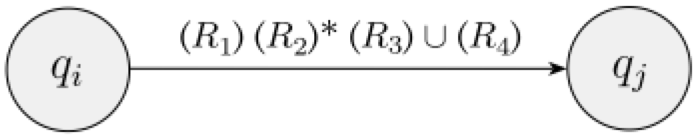
Regular Expressions

- Proof: (cont.)
  - constructing an equivalent GNFA with one fewer state when k > 2
    - select a state, rip it out of the machine, and repair the remaining machine so the language is still recognized
      - any state can be ripped out except the start or accept states
      - ripped state termed q_{rip}
      - after removing q_{rip}, repair the machine by altering the RE on the labels of the remaining arrows
        - compensate for absence of q_{rip} by adding back lost computations

- Proof: (cont.)
  - constructing an equivalent GNFA with one fewer state



before



after

Regular Expressions

- Proof: (cont.)
  - in the old machine, if
    - q_i goes to q_{rip} with an arrow labeled R_1,
    - q_{rip} goes to itself with an arrow labeled R_2,
    - q_{rip} goes to q_j with an arrow labeled R_3, and
    - q_i goes to q_j with an arrow labeled R_4

    - then in the new machine, the arrow from q_i to q_j gets the label

$$(R_1)(R_2)^*(R_3) \cup (R_4)$$

      - make this change for any arrow from q_i to q_j, even when $q_i = q_j$
      - the new machine recognizes the original language

Regular Expressions

- Proof: (cont.)
  - formal definition of a GNFA (similar to NFA but diff δ)

$$\delta: (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \rightarrow R$$

- R: all regular expressions over alphabet Σ
- if $\delta(q_i, q_j) = R$, the arrow from q_i to q_j has RE R as its label
- an arrow connects every state to every other state
  - no arrows coming from q_{accept} or going to q_{start}

Regular Expressions

- Proof: (cont.)
  - formal definition of a GNFA

A ***generalized nondeterministic finite automaton*** is a 5-tuple, $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, where

1. Q is the finite set of states,
2. Σ is the input alphabet,
3. $\delta\colon (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \longrightarrow \mathcal{R}$ is the transition function,
4. q_{start} is the start state, and
5. q_{accept} is the accept state.

Regular Expressions

- Proof: (cont.)
  - a GNFA accepts a string w in Σ^* if
    - $w = w_1w_2\ldots w_k$
    - each w_i is in Σ^*
    - a sequence of states $q_0, q_1, \ldots, q_k$ exists
  - such that
    - $q_0 = q_{start}$ is the start state
    - $q_k = q_{accept}$ is the accept state
    - for each i, we have $w_i \in L(R_i)$ where
      - $R_i = \delta(q_{i-1}, q_i)$
      - i.e., R is the RE on the arrow from q_{i-1} to q_i

Regular Expressions

- Proof: (cont.)
  - returning to the lemma proof: if a language is regular, then it is described by a regular expression
    - let M be the DFA for language A
    - convert M to GNFA G
      - add new start state (with ε arc to old start state)
      - add new accept state (with ε arcs from old accept states)
      - add all other missing arcs and label with Ø
      - use new procedure CONVERT(G)
        - takes GNFA and returns equivalent RE
        - recursive, but only calls itself with a GNFA with one fewer state (to avoid infinite recursion)

Regular Expressions

- Proof: (cont.)
  - CONVERT(G)

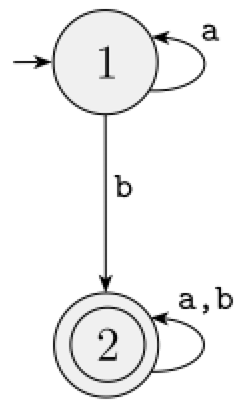
1. k is number of states of G
2. if $k = 2$, G has start state, accept state, and one arrow connecting them labeled with RE R
3. if $k > 2$, select any state $q_{rip} \in Q$ (other than q_{start} and q_{accept})
   - let $G' = (Q', \Sigma, \delta', q_{start}, q_{accept})$
   - $Q' = Q - \{q_{rip}\}$
   - for any $q_i \in Q' - \{q_{accept}\}$ and any $q_j \in Q' - \{q_{start}\}$ let

   $$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4)$$

   for
   - $R_1 = \delta(q_i, q_{rip})$, $R_2 = \delta(q_{rip}, q_{rip})$, $R_3 = \delta(q_{rip}, q_j)$, and $R_4 = \delta(q_i, q_j)$
   - if $\delta(q_i, q_j) = R$, the arrow from q_i to q_j has RE R as its label
4. compute CONVERT(G′) and return this value

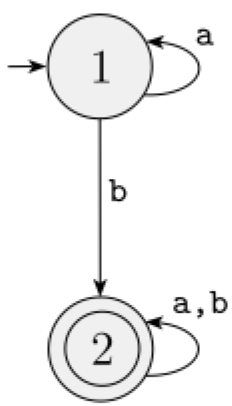
Regular Expressions

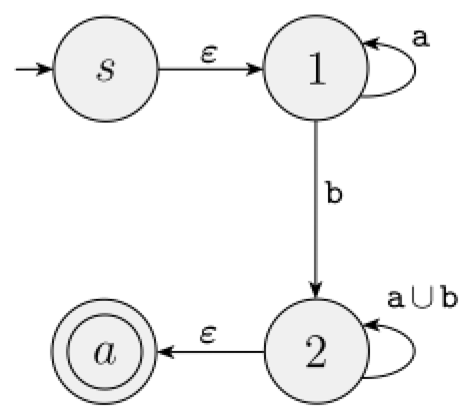
- Proof: (cont.)
  - example: convert two-state DFA to a regular expression



Regular Expressions

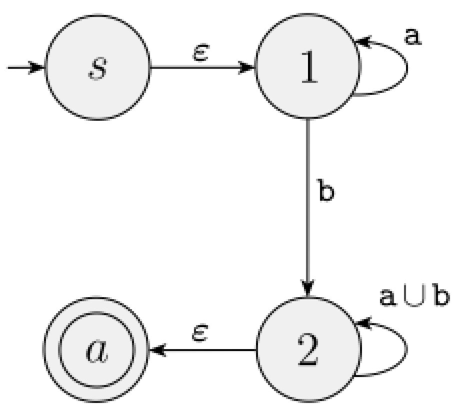
- Proof: (cont.)
  - example: convert two-state DFA to a regular expression
    - create 4-state GNFA by adding new start and accept states
      - labeled s and a for diagram clarity
      - do not draw arcs labeled Ø (i.e., s→2, s→a, 1→a, 2→1)
      - replace label a,b with a ∪ b since only one transition allowed per arc in GNFA

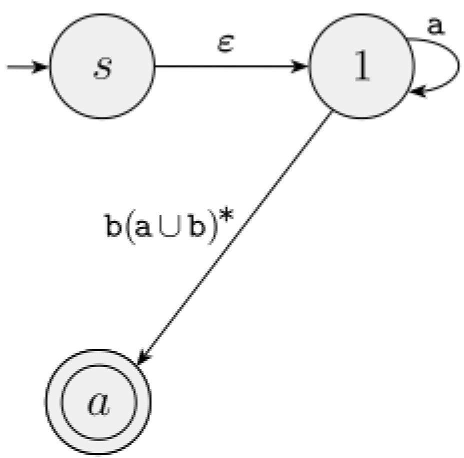




Regular Expressions

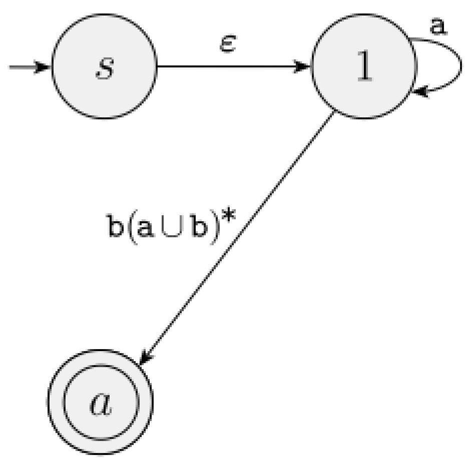
- Proof: (cont.)
  - example: convert two-state DFA to a regular expression
    - remove state 2 and update arc labels
      - only arc that changes is 1 to a (step 3 in CONVERT)
        - $q_i = 1$, $q_j = a$, $q_{rip} = 2$
        - $R_1 = b$, $R_2 = a \cup b$, $R_3 = \varepsilon$, and $R_4 = \emptyset$
        - new label: $(b)(a \cup b)^*(\varepsilon) \cup \emptyset$, or just $b(a \cup b)^*$

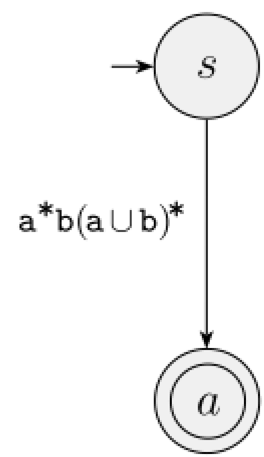




Regular Expressions

- Proof: (cont.)
  - example: convert two-state DFA to a regular expression
    - remove state 1 and update arc labels
      - only arc that changes is s to a (step 3 in CONVERT)
        - $q_i = s$, $q_j = a$, $q_{rip} = 1$
        - $R_1 = \varepsilon$, $R_2 = a^*$, $R_3 = b(a \cup b)^*$, and $R_4 = \emptyset$
        - new label: $(\varepsilon)(a^*)b(a \cup b)^* \cup \emptyset$, or just $a^*b(a \cup b)^*$



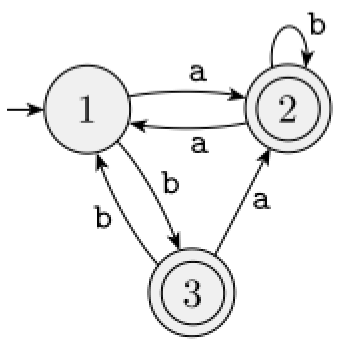


final RE:

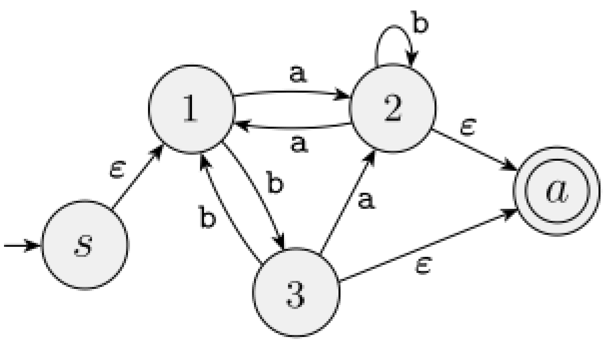
$a^*b(a \cup b)^*$

Regular Expressions

- Proof: (cont.)
  - example: convert three-state DFA to a regular expression
    - steps are similar

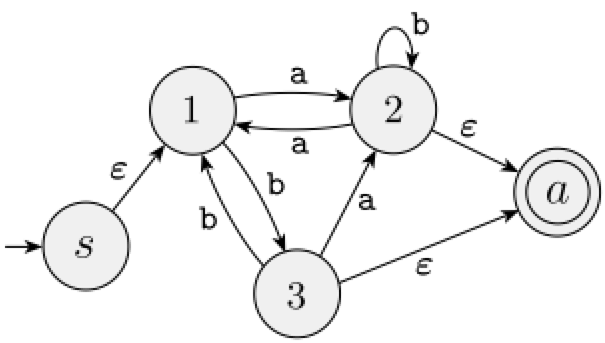


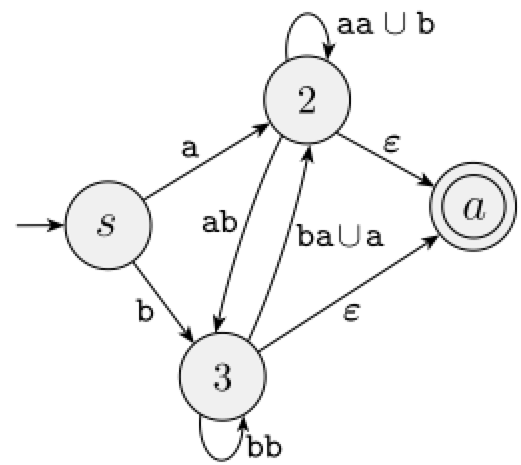
DFA



Regular Expressions

- Proof: (cont.)
  - example: convert three-state DFA to a regular expression
    - steps are similar

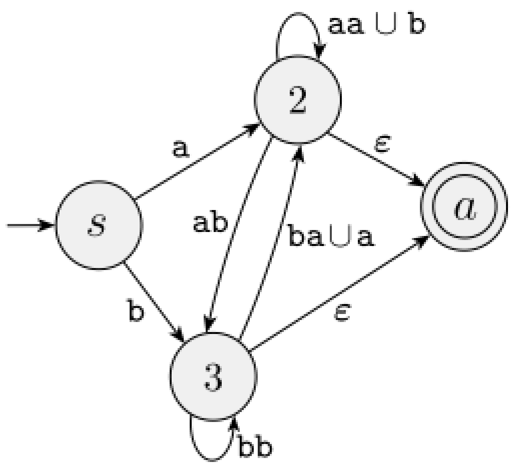


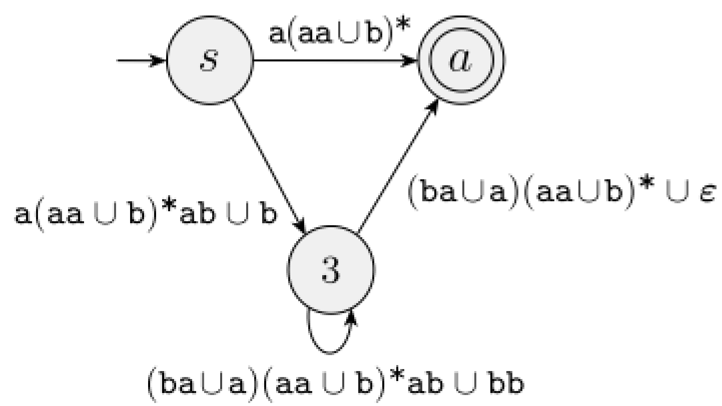


$q_{rip} = 1$

Regular Expressions

- Proof: (cont.)
  - example: convert three-state DFA to a regular expression
    - steps are similar

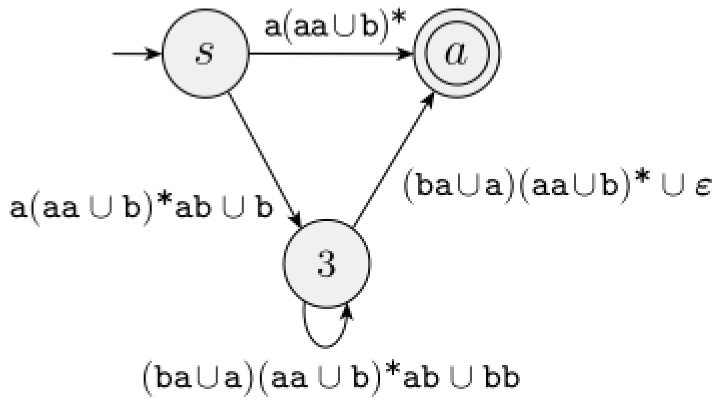




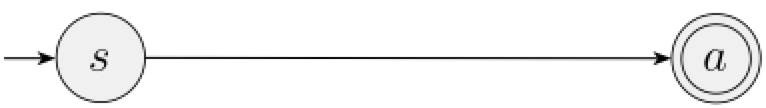
$q_{rip} = 2$

Regular Expressions

- Proof: (cont.)
  - example: convert three-state DFA to a regular expression
    - steps are similar



$q_{rip} = 3$



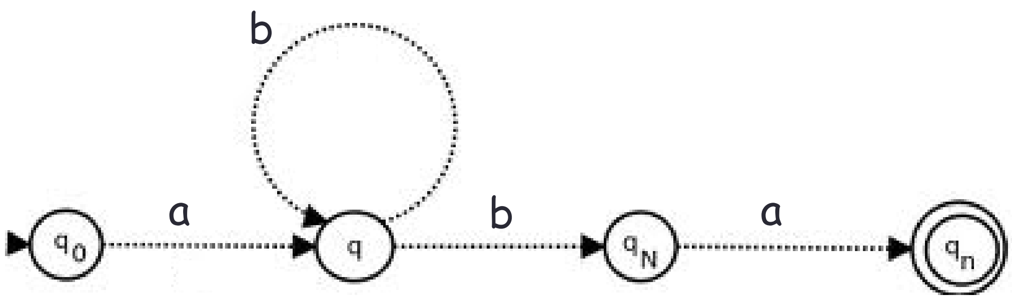
RE: $(a(aa \cup b)^*ab \cup b)((ba \cup a)(aa \cup b)^*ab \cup bb)^*((ba \cup a)(aa \cup b)^* \cup \varepsilon) \cup a(aa \cup b)^*$

Nonregular Languages

- some languages cannot be recognized by FA
  - ex.: $B = \{0^n1^n \mid n \geq 0\}$
    - machine would need to be able to remember how many 0s were seen as it reads the input
      - could be unlimited, so could not be done with a finite number of states
  - need a proof method to show a language is nonregular
    - cannot use example above because though a language appears to require unlimited memory does not mean that it actually does
      - examples
        - $C = \{w \mid w$ has an equal number of 0s and 1s$\}$
        - $D = \{w \mid w$ has an equal number of 01 and 10 substrings$\}$
      - C is not regular, but D is

Nonregular Languages

- pumping lemma preliminary
  - consider the NFA for the language A

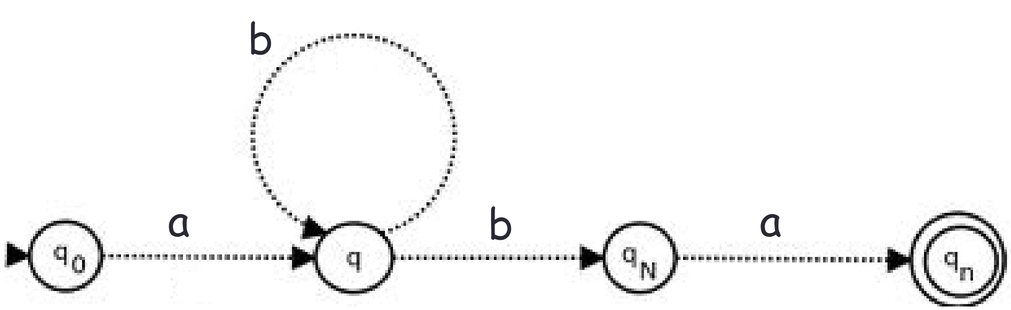


  - what is the smallest string in this language?
  - is there a relationship between the number of symbols in the smallest string and the number of states?
  - what can you say about a string in the language whose length is greater than or equal to the number of states?

Nonregular Languages

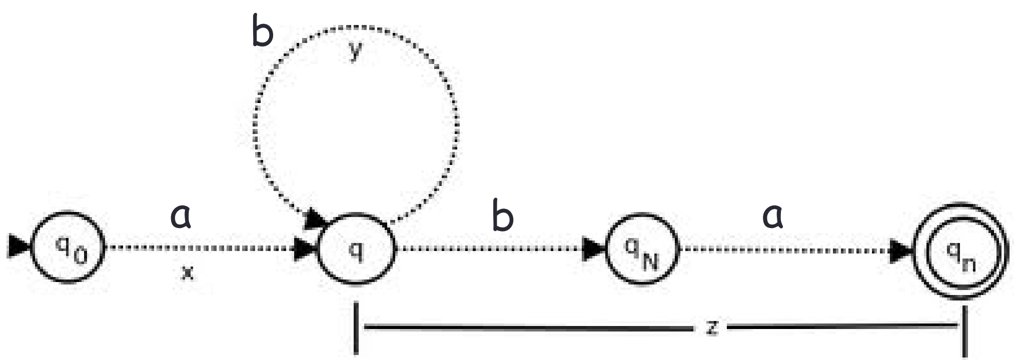
- the pumping lemma
  - all regular languages can be pumped if they are at least as long as a special value termed the pumping length
    - each such string contains a section that can be repeated any number of times with the resulting string remaining in the language
    - if a language does not have this property, it is nonregular

pumping length = 4



Nonregular Languages

- pumping lemma
  - if A is a regular language, there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, s = xyz, satisfying the following conditions
    - for each $i \geq 0$, $xy^iz \in A$
    - $|y| > 0$
    - $|xy| \leq p$



Nonregular Languages

- pumping lemma (cont.)
  - for each $i \geq 0$, $xy^iz \in A$
  - $|y| > 0$
  - $|xy| \leq p$

- note that
  - $|s|$ is the length of string s
  - y^i means i copies of y are concatenated together
  - $y^0 = \varepsilon$

  - x or z may be ε, but $y \neq \varepsilon$
  - x and y together have length at most p

Nonregular Languages

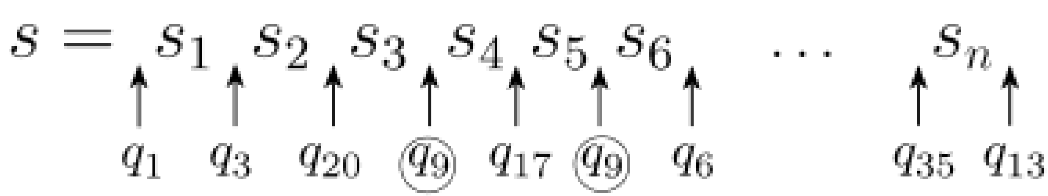
- pumping lemma
  - proof idea:
    - let $M = \{Q, \Sigma, \delta, q_1, F)$ is a DFA that recognizes A
    - let pumping length p = number of states of M
    - show that any string s in A can be broken into pieces xyz satisfying the three conditions
      - if no strings in A are of length at least p, the theorem is vacuously true
      - otherwise, three conditions hold

Nonregular Languages

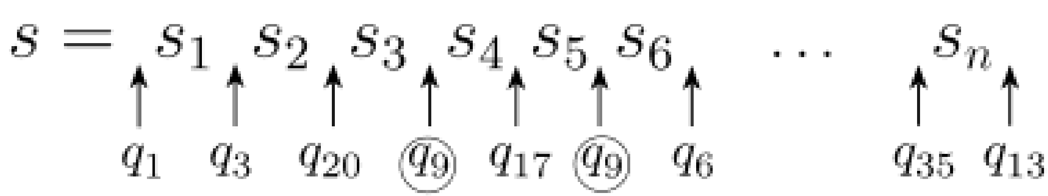
- pumping lemma
  - proof idea: (cont.)
    - if s in A has length at least p, consider the sequence of states M goes through with input s
      - e.g., let's say it starts with q_1 (start state), then goes on to q_3, q_{20}, q_9, ... until it reaches the end of s in q_{13}
      - if $s \in A$, M must accept s, so q_{13} is an accept state

Nonregular Languages

- pumping lemma
  - proof idea: (cont.)
    - let n = |s| - therefore, the sequence of states q_1, q_3, q_{20}, q_9,…,q_{13} has length n + 1
    - because n is at least p, n + 1 > p (or |Q|)
    - therefore, the sequence must contain a repeated state due to the pigeonhole principle
      - e.g.,

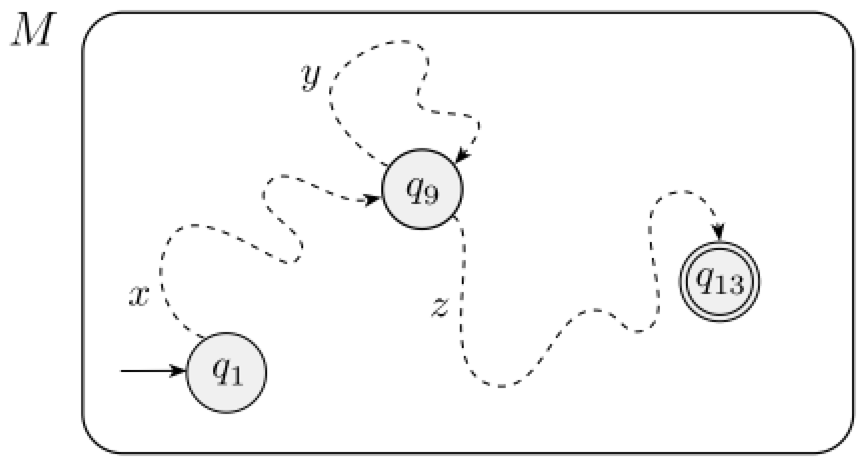


- pumping lemma
  - proof idea: (cont.)
    - divide s into three pieces x, y, and z
      - x appears before q_9
      - y is the part between the two q_9's
      - z is the remaining part of s



Nonregular Languages

- pumping lemma
  - proof idea: (cont.)
    - so,
      - x takes M from q_1 to q_9
      - y takes M from q_9 back to q_9
      - z takes M from q_9 to the accept state q_{13}



Nonregular Languages

- pumping lemma
  - proof idea: (cont.)
    - this division of xyz satisfies the three conditions on input $xyyz$:
      - for each $i \geq 0$, $xy^iz \in A$
        - x takes M from q_1 to q_9
        - y takes M from q_9 back to q_9, as does the second y
        - z takes M to q_{13}, the accept state, so M accepts $xyyz$
        - similarly, it accepts xy^iz for any $i > 0$
        - for $i = 0$, $xy^iz = xz$, which is also accepted
      - $|y| > 0$
        - since it was the part of s that occurred between two different occurrences of state q_9
      - $|xy| \leq p$
        - make sure q_9 is the first repetition in the sequence
        - $p+1$ states must contain a repetition (pigeonhole principle)

Nonregular Languages

- pumping lemma (cont.)
  - proof is similar to proof idea
  - use the pumping lemma to show that a language B is nonregular
    - assume B is regular and show a contradiction
    - use the pumping lemma where all strings of B with at least length p can be pumped
    - find string s in B with length > p that can't be pumped
    - show s cannot be pumped by considering all ways of dividing s into x, y, and z, and for each division, finding an i where xy^iz is not in B
    - s contradicts pumping lemma, so B is nonregular

Nonregular Languages

- pumping lemma (cont.)
  - finding s may take creative thinking
    - try members of B that seem to exhibit B's nonregularity
    - see following examples

Nonregular Languages

- example: let $B = \{0^n1^n \mid n \geq 0\}$ – use the pumping lemma to prove by contradiction that B is not regular
  - assume B is regular with p pumping length
  - let $s = 0^p1^p$
    - because of our assumption, $s = xyz$ where xy^iz is in B for any $i > 0$
    - three cases to show how this is impossible
      - y consists of only 0s
        - now xyyz has more 0s than 1s and is not in B, violating condition 1 of the pumping lemma
      - y consists of only 1s
        - also a contradiction
      - y consists of 0s and 1s
        - xyyz may have same number of 0s and 1s, but they will be out of order with some 1s before 0s (not in B)

Nonregular Languages

- let B = $\{0^n1^n \mid n \geq 0\}$ – use the pumping lemma to prove by contradiction that B is not regular (cont.)
  - in any of the cases, a contradiction is unavoidable
  - can simplify argument by applying condition 3 of the pumping lemma to eliminate cases 2 and 3
  - in this example, finding s was easy

Nonregular Languages

- example: let C = {w | w has an equal number of 0s and 1s} – show that C is not regular
  - assume C is regular with p pumping length
  - let $s = 0^p1^p$
    - because of our assumption, $s = xyz$ where xy^iz is in C for any $i > 0$
    - seems possible since if x and z are empty, and $y = 0^p1^p$, then xy^iz always has an equal number of 0s and 1s
    - but condition 3 of the pumping lemma states that $|xy| \leq p$, so s cannot be pumped in this way
      - if $|xy| \leq p$, our only choice is y consists of all 0s, so xyyz is not in C, which leads to the contradiction

Nonregular Languages

- example: let C = {w | w has an equal number of 0s and 1s} – show that C is not regular (cont.)
  - finding s was a bit harder here
  - if we had let $s = (01)^p$, it would not have worked since it can be pumped
  - keep trying different values of s until you find one that cannot be pumped
  - another way to prove C is nonregular is to use another language that we already know is nonregular, like B
  - if C were regular, $C \cap 0^*1^*$ would also be regular due to closure under intersection (proved in the textbook)
    - but $C \cap 0^*1^* = B$, which is not regular

Nonregular Languages

- example: let $F = \{ww \mid w \in \{0,1\}^*\}$ - show that F is not regular
  - assume F is regular with p pumping length
  - let $s = 0^p10^p1$
  - so s can be split into three pieces s = xyz satisfying the three conditions of the lemma
    - could let x and z be ε, but y must consist of only 0s, so xyyz not in F
    - we chose s to be a string that exhibits a nonregular language instead of say, 0^p0^p, even though it is a member since it can be pumped and fails the contradiction

Nonregular Languages

- example: let $D = \{1^{n^2} \mid n \geq 0\}$ – show that D is not regular
  - assume D is regular with p pumping length
  - let $s = 1^{p^2}$
  - so s can be split into three pieces s = xyz satisfying the three conditions of the lemma
    - perfect squares: 0, 1, 4, 9, 16, 25, 36, 49, …
      - gap between values gets greater as n increases

Nonregular Languages

- example: let $D = \{1^{n^2} \mid n \geq 0\}$ – show that D is not regular
  - assume D is regular with p pumping length (cont.)
    - consider strings xyz and xy^2z
      - differ by one repetition of y so lengths differ by $|y|$
      - by condition 3, $|xy| \leq p$ so $|y| \leq p$
      - but $|xyz| = p^2$ so $|xy^2z| \leq p^2 + p$
      - $p^2 + p < p^2 + 2p + 1 = (p + 1)^2$
      - y cannot be ε, so $|xy^2z| > p^2$
      - thus $|xy^2z|$ lies between consecutive perfect squares p^2 and $(p + 1)^2$
      - so length is not a perfect square (contradiction)
      - thus xy^2z not in D, and D is not regular

Nonregular Languages

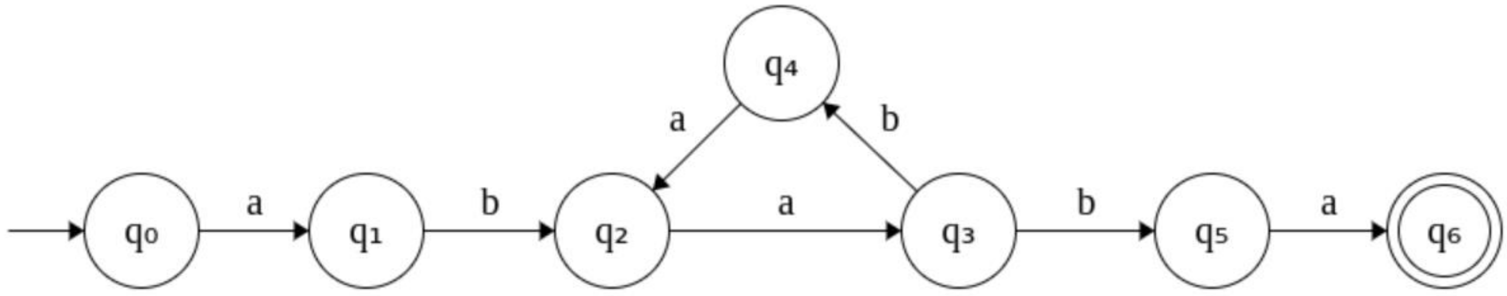
- example: let $E = \{0^i 1^j \mid i > j\}$ – use the pumping lemma to prove by contradiction that E is not regular
  - use pumping lemma to pump down
  - assume E is regular with p pumping length
  - let $s = 0^{p+1}1^p$
    - because of our assumption, $s = xyz$ where xy^iz is in E for any $i \geq 0$
    - by condition 3, y consists of only 0s
      - now xyyz has even more 0s, which is in E, so we need to try another string
    - try $xy^0z = xz$ (pumping down)
      - since s had just one more 0 than 1s, xz cannot have more 0s than 1s -> contradiction

Nonregular Languages

- pumping lemma (cont.)
  - additional notes
    - we cannot use the pumping lemma to show that a language is regular
      - some languages will pass the pumping lemma test, but still be nonregular
      - the pumping lemma, therefore, is a necessary test, but not a sufficient test, to show that a language is regular
      - we have other ways to show a language is regular
      - no language that fails the pumping lemma test is regular

Nonregular Languages

- pumping lemma (cont.)
  - additional notes
    - consider the following DFA

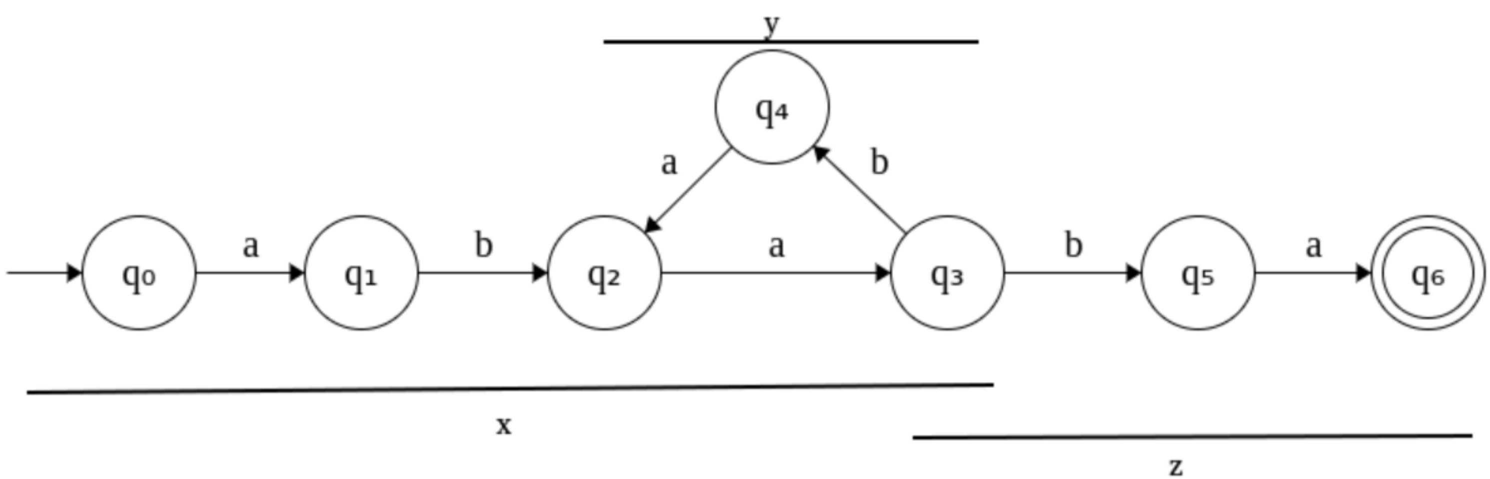


    - what strings can be accepted?
    - divide the strings into three parts
    - provide a regular expression

https://swaminathanj.github.io/fsm/pumpinglemma.html

Nonregular Languages

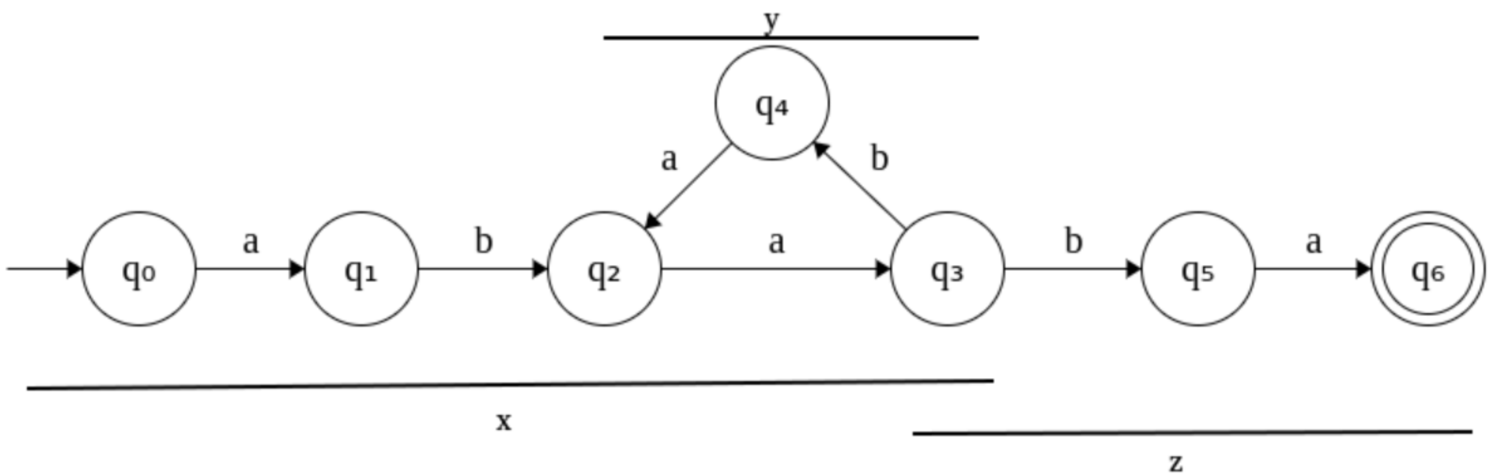
- pumping lemma (cont.)
  - additional notes



- if loop at beginning $x = \varepsilon$ and $w = yz$
- if loop at end $z = \varepsilon$ and $w = xy$
- y cannot be ε (but y^0 can!)

Nonregular Languages

- pumping lemma (cont.)
  - additional notes



- shortest string accepted? if q_2 is also accept state?
- longest string accepted without looping?
- longest string accepted by looping once (p)?

Nonregular Languages

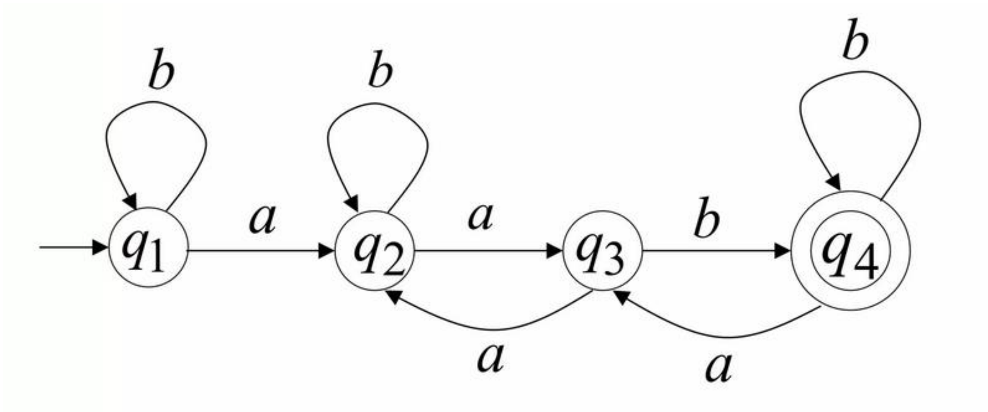
- pumping lemma (cont.)
  - additional notes
    - what is the pumping length for the following languages?
      1. 1*
      2. 01
      3. 01*0
      4. 11*

1. 1
2. 3
3. 3
4. 2

Nonregular Languages

- pumping lemma (cont.)
  - additional notes
    - remember that s is only one type of string found in the language
    - try to choose s to be a string pattern we already know is nonregular
      - use p strategically to limit the number of parts of the string that y can be assigned
    - for xy^iz, string must be in the language for all $i \geq 0$
      - only one assignment to x, y, and z must work
        - but for that assignment, it must work for all $i \geq 0$
      - so you must try them all and explain why none of them work when considering all $i \geq 0$

- pumping lemma (cont.)
  - additional notes
    - what about DFAs with multiple circuits?
      - the pumping lemma seems too limited



*Hermant Chetwani

- it still works since we can break down the strings into different cases of s where each s has only one circuit, e.g., a b^i abb and babbabab b^i

Nonregular Languages

- pumping lemma (cont.)
  - proof requirements for proving *A* is nonregular
    - Assume *A* is regular and therefore must pass the pumping lemma test
    - let s = some string using p, such as 0^p1^p
    - explain xyz assignment, such as x and y must consist entirely of 0s (from the limitations imposed by s)
    - explain how xy^iz would allow other strings to be generated with i = 0 or 2 that are not in *A*
    - explain how there are no other options, or every other option would result in the same or similar condition
    - state that this is a contradiction and therefore the pumping lemma does not hold; therefore, *A* is nonreg