6.045: Automata, Computability, and Complexity
Or, Great Ideas in Theoretical Computer Science
Spring, 2010

Class 4

Nancy Lynch

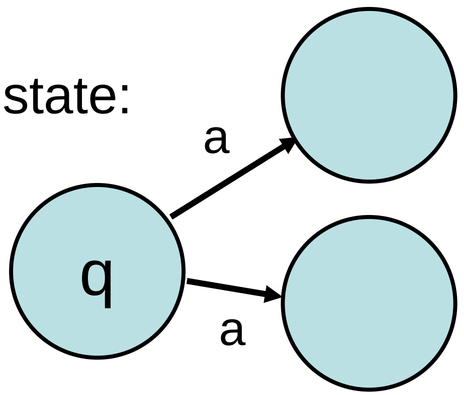
Today

- Two more models of computation:
  - Nondeterministic Finite Automata (NFAs)
    - Add a guessing capability to FAs.
    - But provably equivalent to FAs.
  - Regular expressions
    - A different sort of model---expressions rather than machines.
    - Also provably equivalent.
- Topics:
  - Nondeterministic Finite Automata and the languages they recognize
  - NFAs vs. FAs
  - Closure of FA-recognizable languages under various operations, revisited
  - Regular expressions
  - Regular expressions denote FA-recognizable languages
- Reading: Sipser, Sections 1.2, 1.3
- Next: Section 1.4

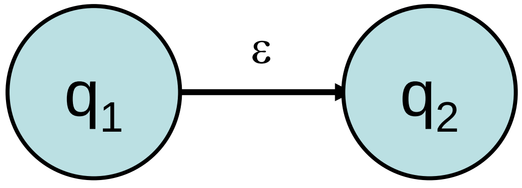
Nondeterministic Finite Automata and the languages they recognize

Nondeterministic Finite Automata

- Generalize FAs by adding nondeterminism, allowing several alternative computations on the same input string.
- Ordinary deterministic FAs follow one path on each input.
- Two changes:
  - Allow $\delta(q, a)$ to specify more than one successor state:



  - Add ε-transitions, transitions made "for free", without "consuming" any input symbols.



- Formally, combine these changes:

Formal Definition of an NFA

- An NFA is a 5-tuple (Q, Σ, δ, q_0, F), where:
  - Q is a finite set of states,
  - Σ is a finite set (alphabet) of input symbols,
  - $\delta: Q \times \Sigma_\varepsilon \rightarrow P(Q)$ is the transition function,

The arguments are a state and either an alphabet symbol or ε. Σ_ε means $\Sigma \cup \{\varepsilon\}$.

The result is a set of states.

  - $q_0 \in Q$, is the start state, and
  - $F \subseteq Q$ is the set of accepting, or final states.

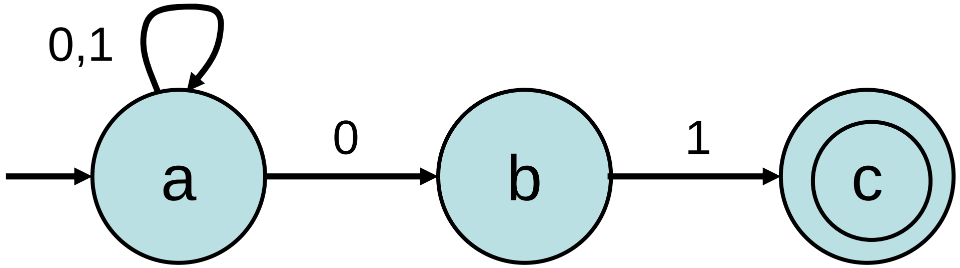
Formal Definition of an NFA

- An NFA is a 5-tuple (Q, Σ, δ, q_0, F), where:
  - Q is a finite set of states,
  - Σ is a finite set (alphabet) of input symbols,
  - $\delta: Q \times \Sigma_\varepsilon \rightarrow P(Q)$ is the transition function,
  - $q_0 \in Q$, is the start state, and
  - $F \subseteq Q$ is the set of accepting, or final states.
- How many states in P(Q)?

  $2^{|Q|}$
- Example: Q = { a, b, c }

  P(Q) = { $\emptyset$, {a}, {b}, {c}, {a,b}, {a,c}, {b,c}, {a,b,c} }

NFA Example 1



$Q = \{ a, b, c \}$

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

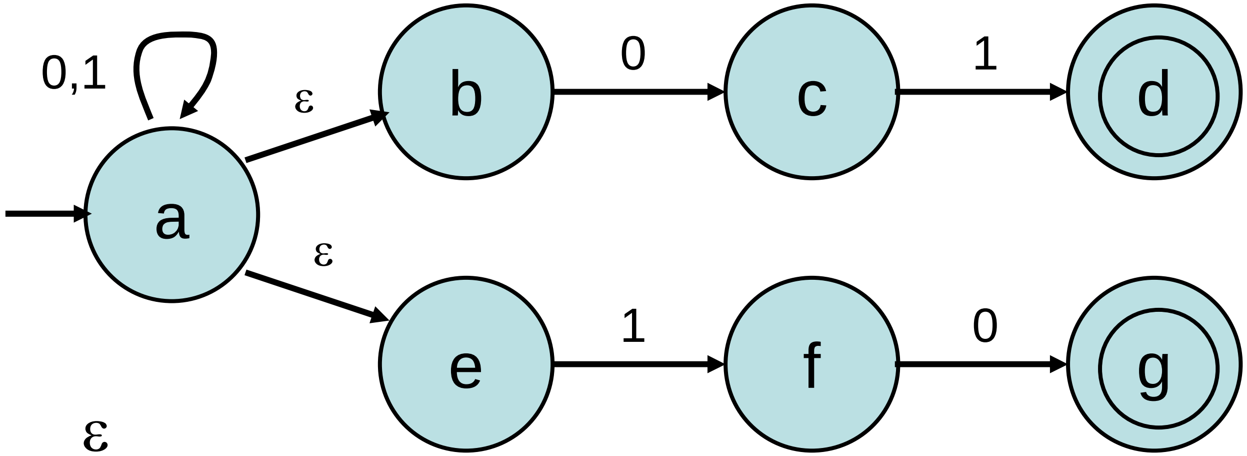
$q_0 = a$

$F = \{ c \}$

δ:

	0	1	ε
a	{a,b}	{a}	$\varnothing$
b	$\varnothing$	{c}	$\varnothing$
c	$\varnothing$	$\varnothing$	$\varnothing$

NFA Example 2



	0	1	ε
a	{a}	{a}	{b,c}
b	{c}	∅	∅
c	∅	{d}	∅
d	∅	∅	∅
e	∅	{f}	∅
f	{g}	∅	∅
g	∅	∅	∅

Nondeterministic Finite Automata

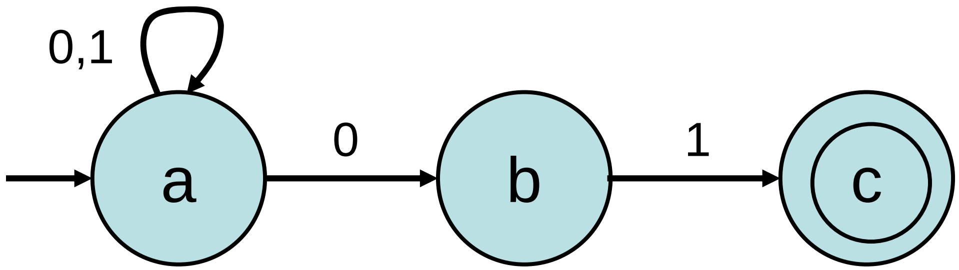
- NFAs are like DFAs with two additions:
  - Allow $\delta(q, a)$ to specify more than one successor state.
  - Add ε-transitions.
- Formally, an NFA is a 5-tuple ($Q, \Sigma, \delta, q_0, F$), where:
  - Q is a finite set of states,
  - Σ is a finite set (alphabet) of input symbols,
  - $\delta: Q \times \Sigma_\varepsilon \rightarrow P(Q)$ is the transition function,

    Σ_ε means $\Sigma \cup \{\varepsilon\}$.

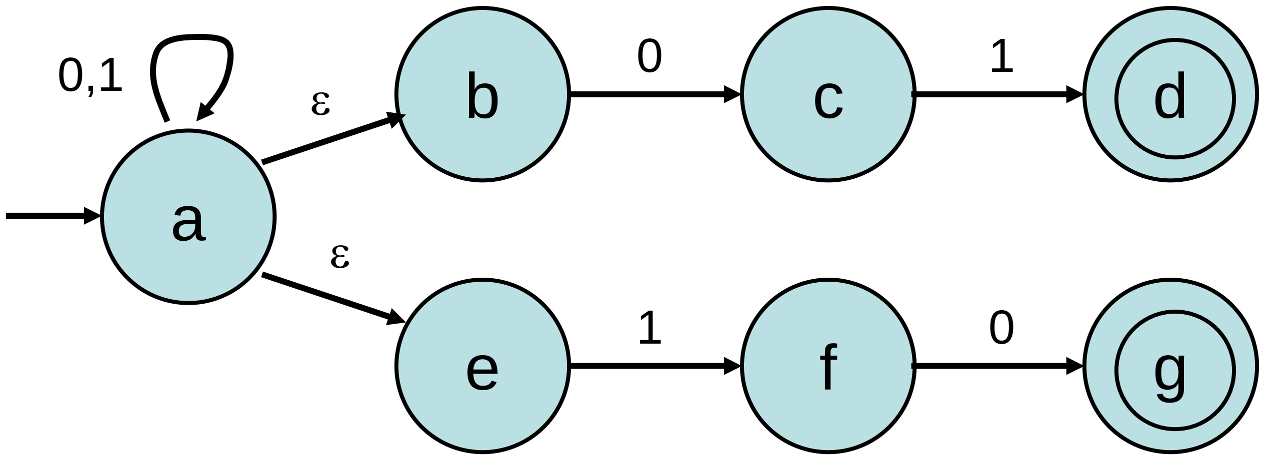
  - $q_0 \in Q$, is the start state, and
  - $F \subseteq Q$ is the set of accepting, or final states.

NFA Examples

Example 1:



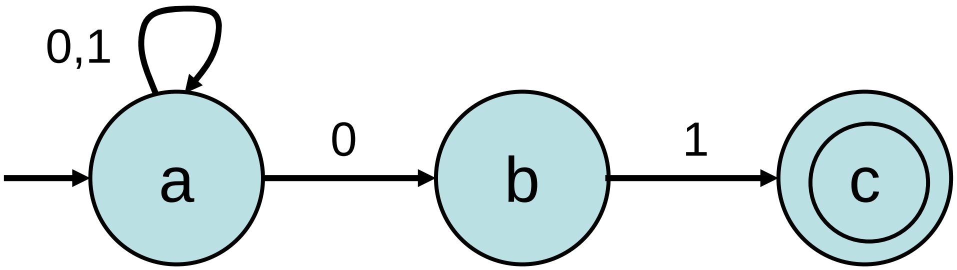
Example 2:



How NFAs compute

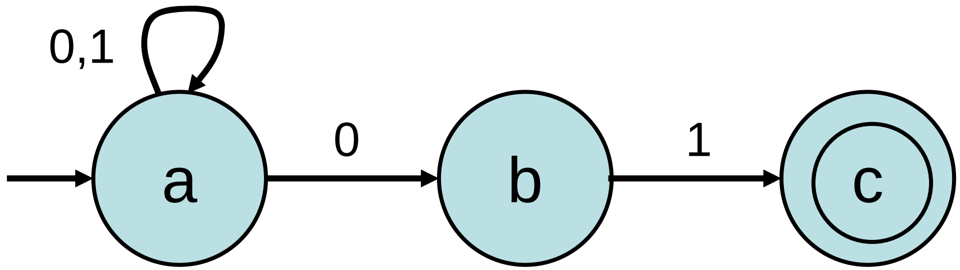
- Informally:
  - Follow allowed arrows in any possible way, while “consuming” the designated input symbols.
  - Optionally follow any ε arrow at any time, without consuming any input.
  - Accepts a string if some allowed sequence of transitions on that string leads to an accepting state.

Example 1



- L(M) = { w | w ends with 01 }
- M accepts exactly the strings in this set.
- Computations for input word w = 101:
  - Input word w: 1 0 1
  - States: a a a a
  - Or: a a b c
- Since c is an accepting state, M accepts 101

Example 1



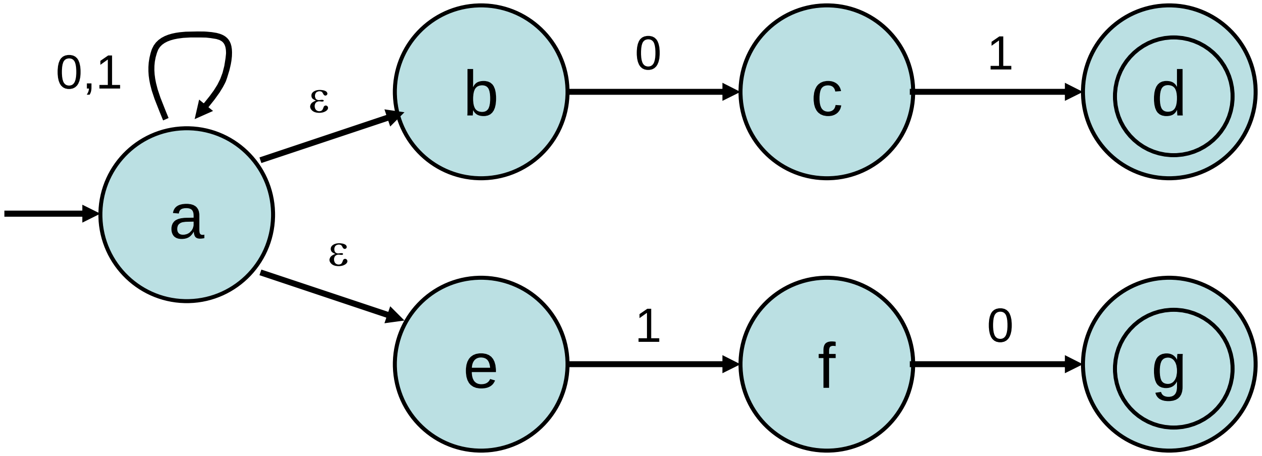
- Computations for input word w = 0010:
  - Possible states after 0: { a, b }
  - Then after another 0: { a, b }
  - After 1: { a, c }
  - After final 0: { a, b }
- Since neither a nor b is accepting, M does not accept 0010.

0 0 0

{ a } → { a, b } → { a, b } → { a, c } → { a, b }

1

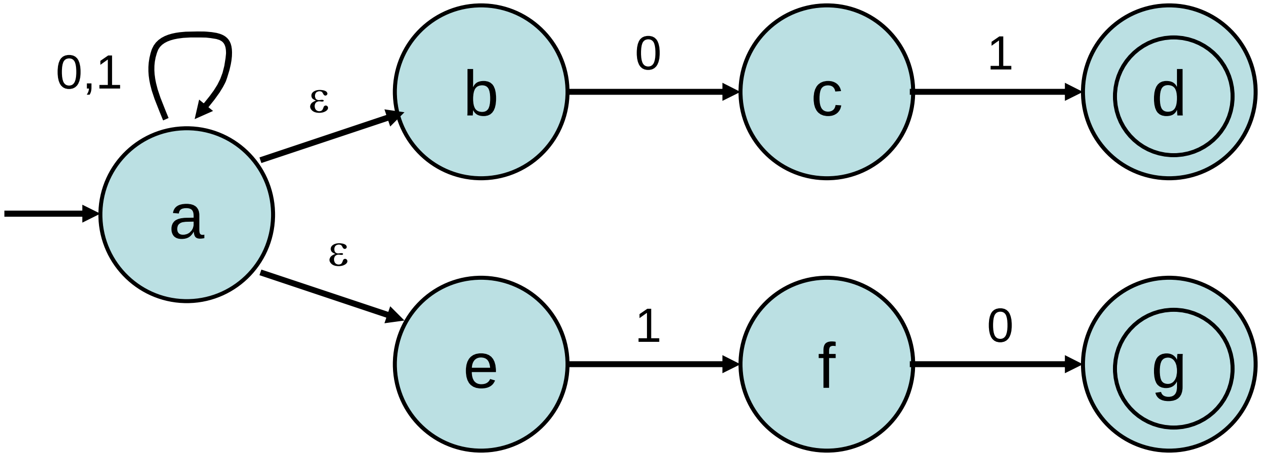
Example 2



- L(M) = { w | w ends with 01 or 10 }
- Computations for w = 0010:
  - Possible states after no input: { a, b, e }
  - After 0: { a, b, e, c }
  - After 0: { a, b, e, c }
  - After 1: { a, b, e, d, f }
  - After 0: { a, b, e, c, g }
- Since g is accepting, M accepts 0010.

$$\{a, b, e\} \xrightarrow{0} \{a, b, e, c\} \xrightarrow{0} \{a, b, e, c\} \xrightarrow{1} \{a, b, e, d, f\} \xrightarrow{0} \{a, b, e, c, g\}$$

Example 2



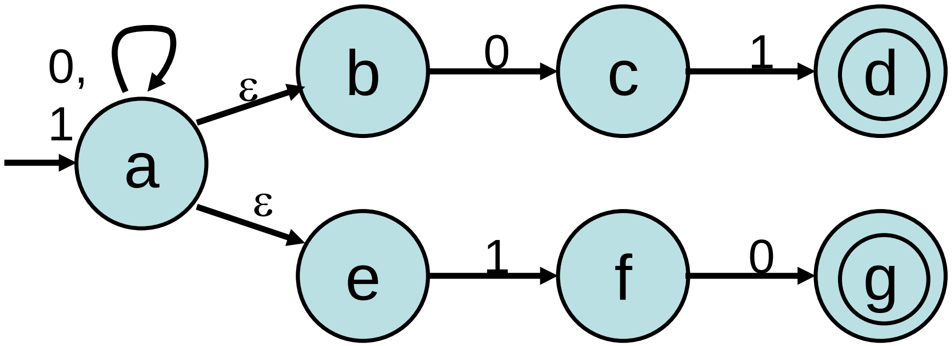
- Computations for w = 0010:

$$\{a, b, e\} \xrightarrow{0} \{a, b, e, c\} \xrightarrow{0} \{a, b, e, c\}$$
$$\xrightarrow{1} \{a, b, e, d, f\} \xrightarrow{0} \{a, b, e, c, g\}$$

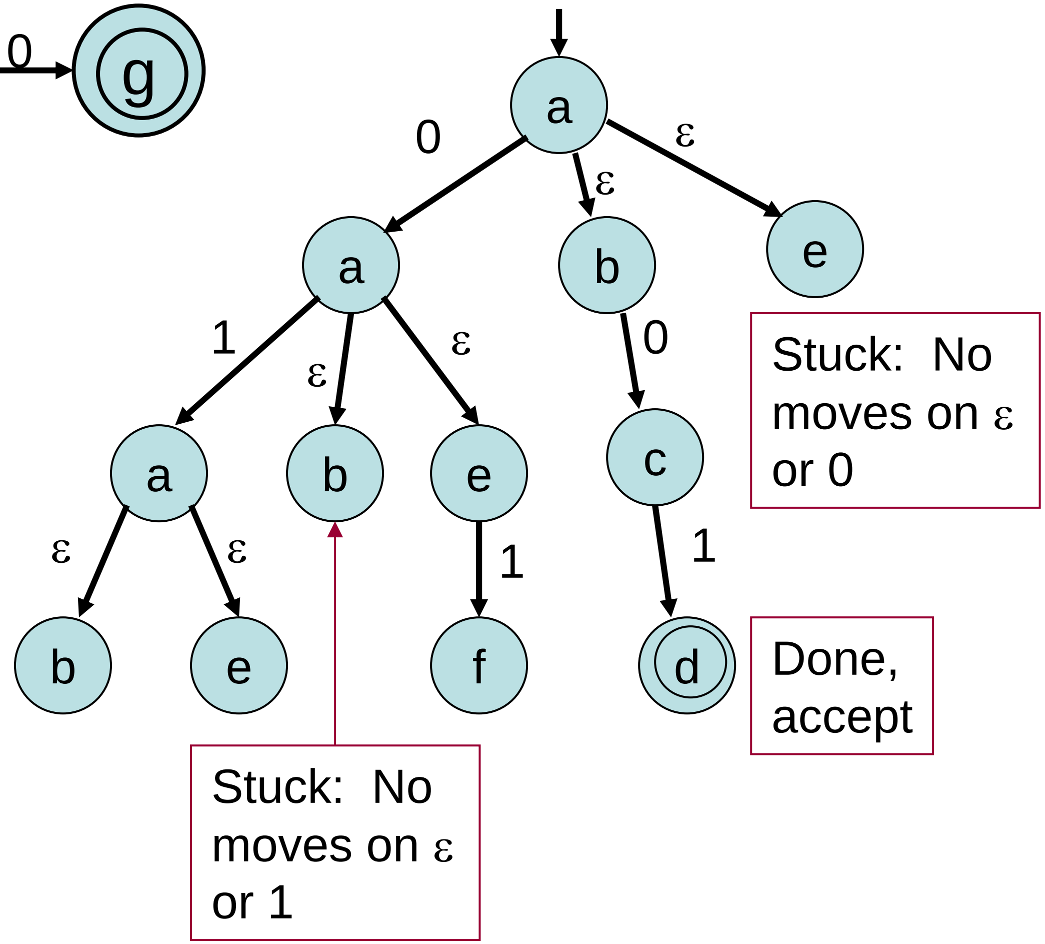
- Path to accepting state:

$$a \xrightarrow{0} a \xrightarrow{0} a \xrightarrow{\varepsilon} e \xrightarrow{1} f \xrightarrow{0} g$$

Viewing computations as a tree



Input w = 01



In general, accept if there is a path labeled by the entire input string, possibly interspersed with εs, leading to an accepting state.

Here, leads to accepting state d.

Formal definition of computation

- Define E(q) = set of states reachable from q using zero or more ε-moves (includes q itself).
- Example 2: E(a) = { a, b, e }

- Define $\delta^*: Q \times \Sigma^* \rightarrow P(Q)$, state and string yield a set of states: $\delta^*(q, w)$ = states that can be reached from q by following w.

- Defined iteratively: Compute $\delta^*(q, a_1 a_2 \ldots a_k)$ by:

  S : = E(q)

  for i = 1 to k do

  $S := \cup_{r' \in \delta(r, ai) \text{ for some } r \text{ in } S} E(r')$

- Or define recursively, LTTR.

Formal definition of computation

- $\delta^*(q, w)$ = states that can be reached from q by following w.
- String w is accepted if $\delta^*(q_0, w) \cap F \neq \varnothing$, that is, at least one of the possible end states is accepting.
- String w is rejected if it isn't accepted.
- L(M), the language recognized by NFA M, = $\{ w \mid w \text{ is accepted by } M\}$.

NFAs vs. FAs

NFAs vs. DFAs

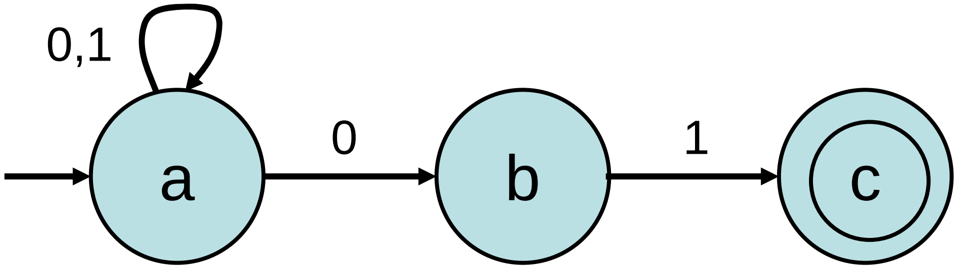
- DFA = Deterministic Finite Automaton, new name for ordinary Finite Automata (FA).
  - To emphasize the difference from NFAs.
- What languages are recognized by NFAs?
- Since DFAs are special cases of NFAs, NFAs recognize at least the DFA-recognizable (regular) languages.
- Nothing else!
- Theorem: If M is an NFA then L(M) is DFA-recognizable.
- Proof:
  - Given NFA $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, produce an equivalent DFA $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$.
    - Equivalent means they recognize the same language, $L(M_2) = L(M_1)$.
  - Each state of M_2 represents a set of states of M_1: $Q_2 = P(Q_1)$.
  - Start state of M_2 is E(start state of M_1) = all states M_1 could be in after scanning ε: $q_{02} = E(q_{01})$.

NFAs vs. DFAs

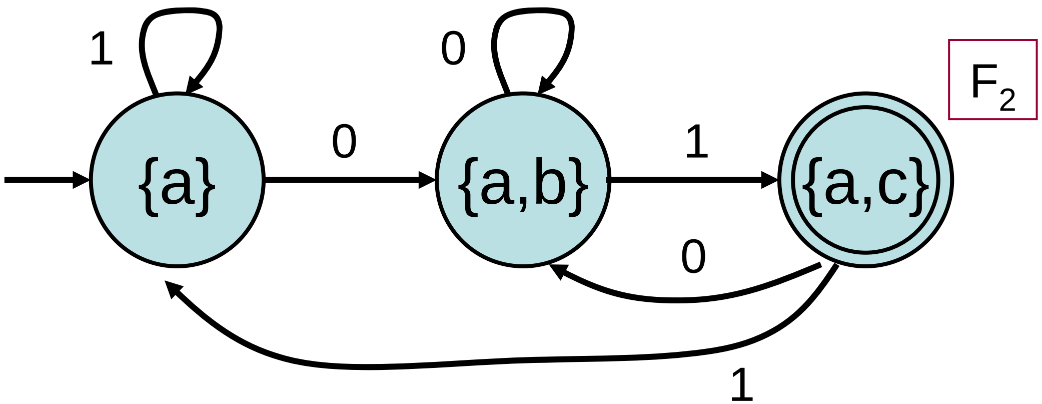
- Theorem: If M is an NFA then L(M) is DFA-recognizable.
- Proof:
  - Given NFA $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, produce an equivalent DFA $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$.
  - $Q_2 = P(Q_1)$
  - $q_{02} = E(q_{01})$
  - $F_2 = \{ S \subseteq Q_1 \mid S \cap F_1 \neq \emptyset \}$
    - Accepting states of M_2 are the sets that contain an accepting state of M_1.
  - $\delta_2(S, a) = \cup_{r \in S} E\big(\delta_1(r, a) \big)$
    - Starting from states in S, $\delta_2(S, a)$ gives all states M_1 could reach after a and possibly some ε-transitions.
  - M_2 recognizes $L(M_1)$: At any point in processing the string, the state of M_2 represents exactly the set of states that M_1 could be in.

Example: NFA → DFA

- M_1:



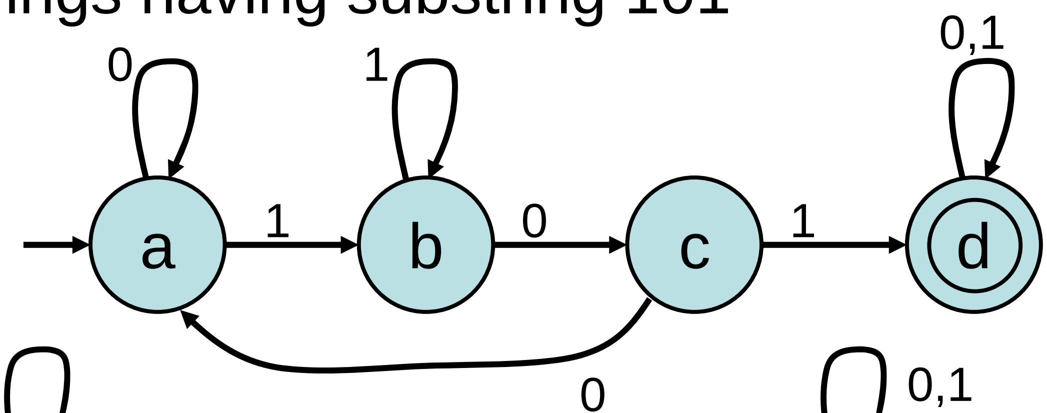
- States of M_2: ∅, {a}, {b}, {c}, {a,b}, {a,c}, {b,c}, {a,b,c}
- δ_2:



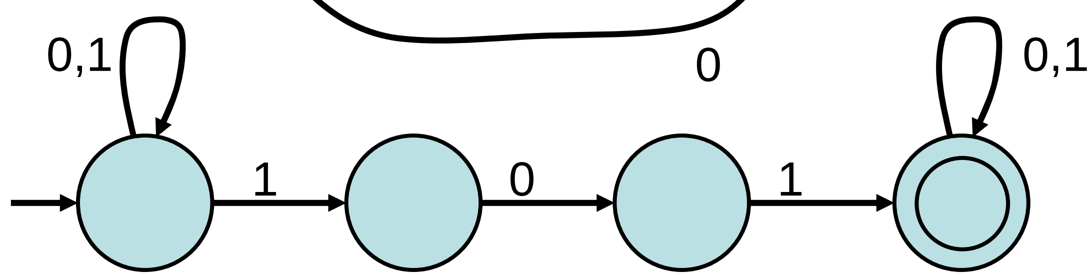
- Other 5 subsets aren't reachable from start state, don't bother drawing them.

NFAs vs. DFAs

- NFAs and DFAs have the same power.
- But sometimes NFAs are simpler than equivalent DFAs.
- Example: L = strings ending in 01 or 10
  - Simple NFA, harder DFA (LTTR)
- Example: L = strings having substring 101
  - Recall DFA:



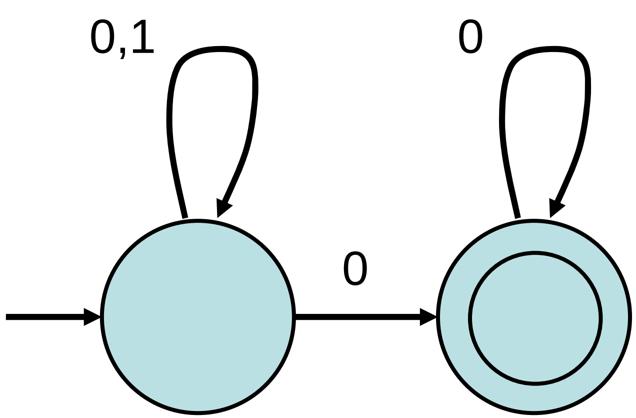
  - NFA:



  - Simpler---has the power to “guess” when to start matching.

NFAs vs. DFAs

- Which brings us back to last time.
- We got stuck in the proof of closure for DFA languages under concatenation:
- Example: L = { 0, 1 }* { 0 } { 0 }*



- NFA can guess when the critical 0 occurs.

Closure of regular (FA-recognizable) languages under various operations, revisited

Closure under operations

- The last example suggests we retry proofs of closure of FA languages under concatenation and star, this time using NFAs.
- OK since they have the same expressive power (recognize the same languages) as DFAs.
- We already proved closure under common set-theoretic operations---union, intersection, complement, difference---using DFAs.
- Got stuck on concatenation and star.

- First (warmup): Redo union proof in terms of NFAs.

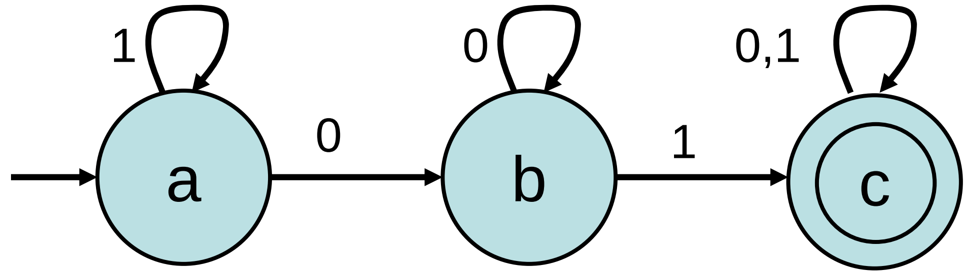
Closure under union

- Theorem: FA-recognizable languages are closed under union.
- Old Proof:
  - Start with DFAs M_1 and M_2 for the same alphabet Σ.
  - Get another DFA, M_3, with $L(M_3) = L(M_1) \cup L(M_2)$.
  - Idea: Run M_1 and M_2 "in parallel" on the same input. If either reaches an accepting state, accept.

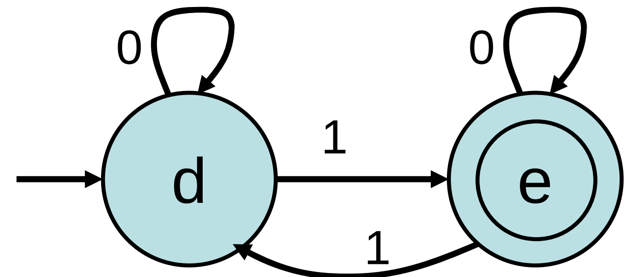
Closure under union

- Example:

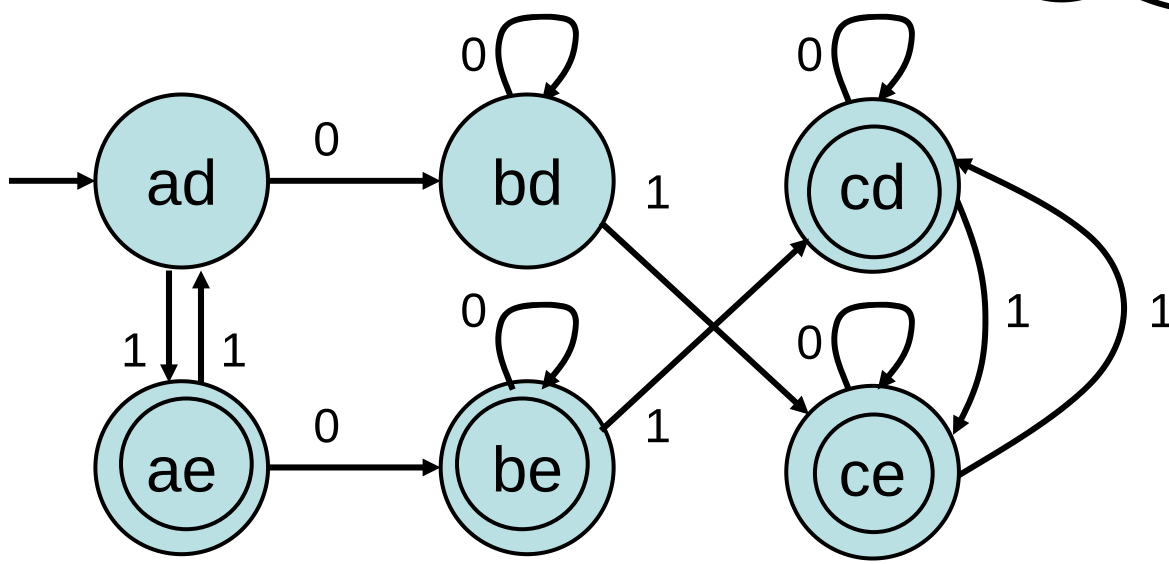
M_1: Substring 01



M_2: Odd number of 1s



M_3:

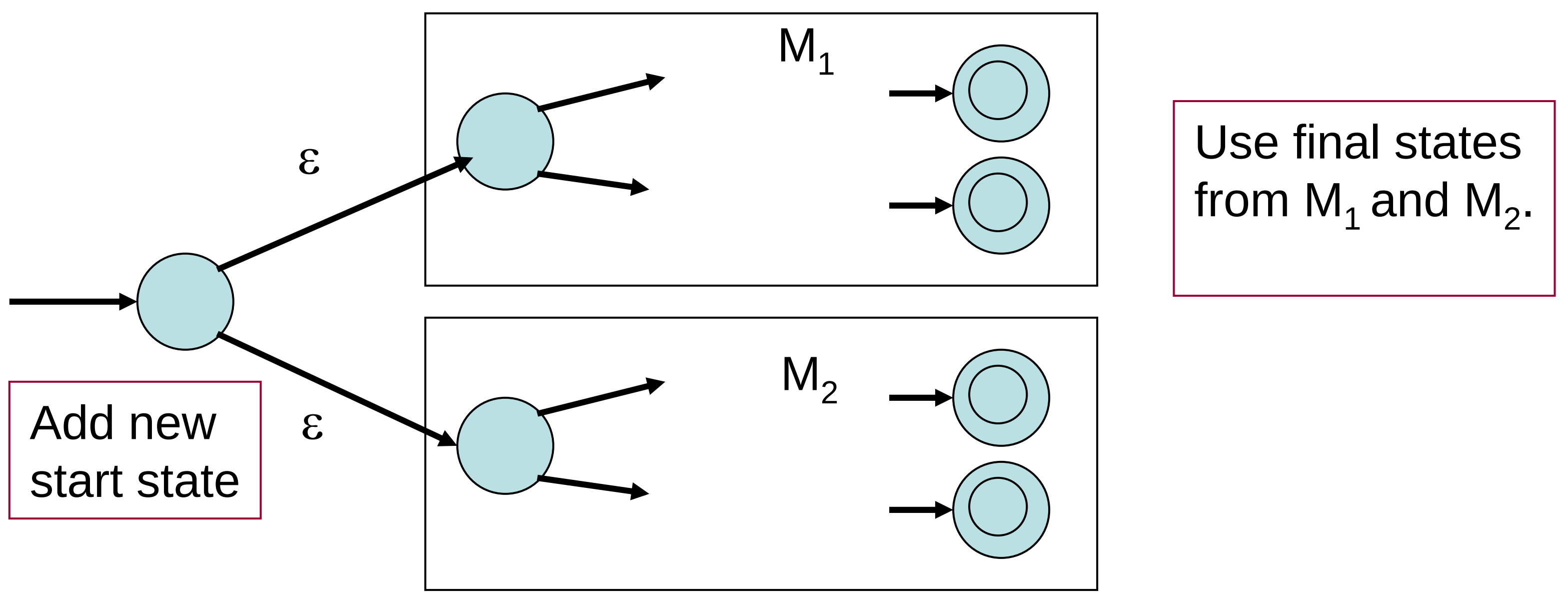


Closure under union, general rule

- Assume:
  - $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$
  - $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$
- Define $M_3 = (Q_3, \Sigma, \delta_3, q_{03}, F_3)$, where
  - $Q_3 = Q_1 \times Q_2$
    - Cartesian product, $\{(q_1,q_2) \mid q_1 \in Q_1 \text{ and } q_2 \in Q_2 \}$
  - $\delta_3 ((q_1,q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$
  - $q_{03} = (q_{01}, q_{02})$
  - $F_3 = \{ (q_1,q_2) \mid q_1 \in F_1 \text{ or } q_2 \in F_2 \}$

Closure under union

- Theorem: FA-recognizable languages are closed under union.
- New Proof:
  - Start with NFAs M_1 and M_2.
  - Get another NFA, M_3, with $L(M_3) = L(M_1) \cup L(M_2)$.



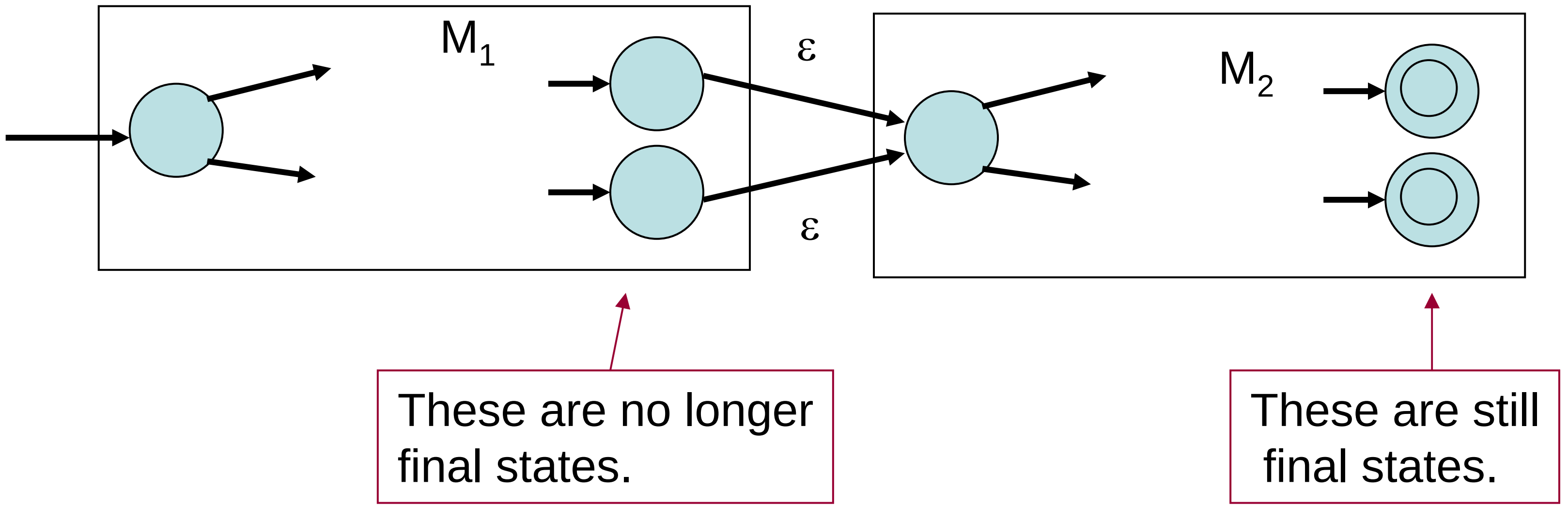
Closure under union

- Theorem: FA-recognizable languages are closed under union.
- New Proof: Simpler!

- Intersection:
  - NFAs don't seem to help.
- Concatenation, star:
  - Now try NFA-based constructions.

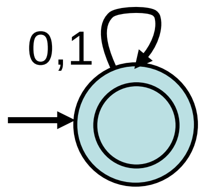
Closure under concatenation

- $L_1 \circ L_2 = \{ x\,y \mid x \in L_1 \text{ and } y \in L_2 \}$
- Theorem: FA-recognizable languages are closed under concatenation.
- Proof:
  - Start with NFAs M_1 and M_2.
  - Get another NFA, M_3, with $L(M_3) = L(M_1) \circ L(M_2)$.

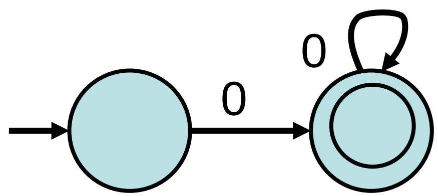


Closure under concatenation

- Example:
  - $\Sigma = \{ 0, 1\}$, $L_1 = \Sigma^*$, $L_2 = \{0\} \{0\}^*$.
  - $L_1 L_2$ = strings that end with a block of at least one 0
  - M_1:

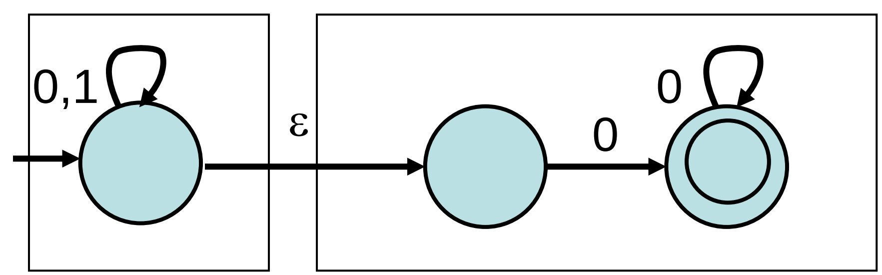


  - M_2:



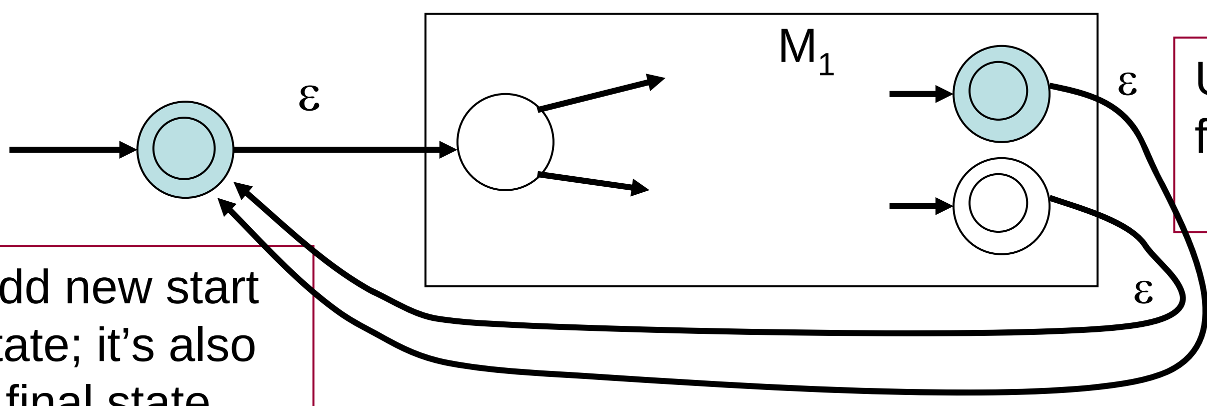
NFAs

  - Now combine:



Closure under star

- $L^* = \{ x \mid x = y_1 \, y_2 \, \ldots \, y_k$ for some $k \geq 0$, every y in L }
  $= L^0 \cup L^1 \cup L^2 \ \cup \ldots$
- Theorem: FA-recognizable languages are closed under star.
- Proof:
  - Start with FA M_1.
  - Get an NFA, M_2, with $L(M_2) = L(M_1)^*$.

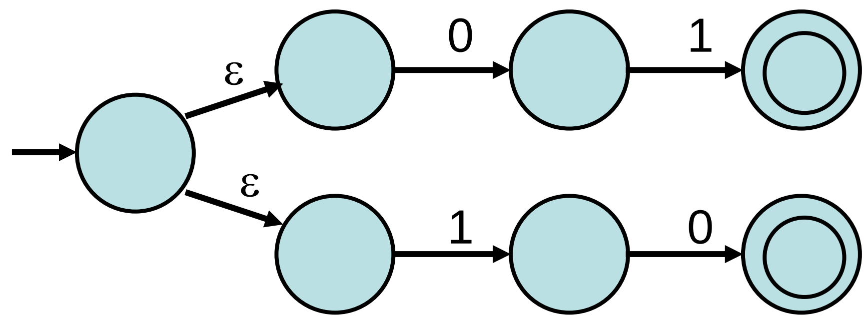


Use final states from M_1 and M_2.

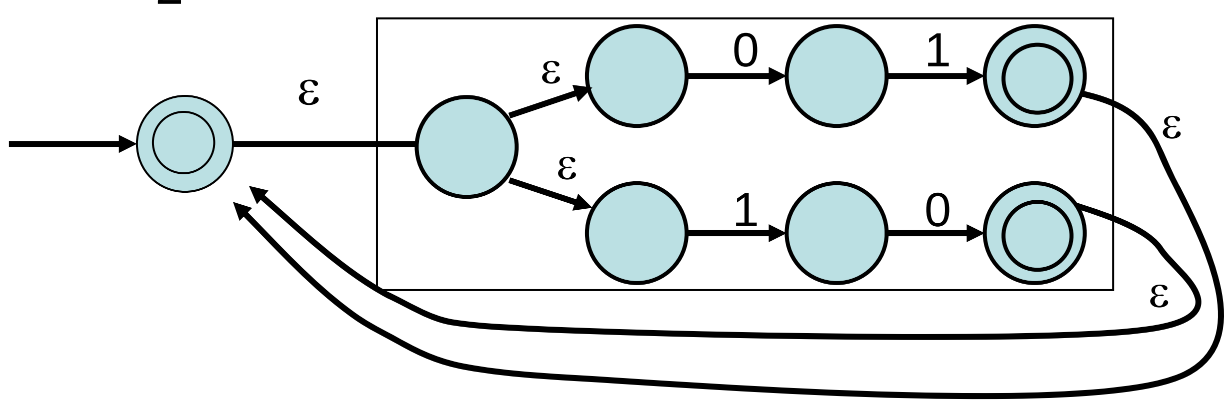
Add new start state; it's also a final state, since ε is in $L(M_1)^*$.

Closure under star

- Example:
  - $\Sigma = \{ 0, 1\}$, $L_1 = \{ 01, 10 \}$
  - $(L_1)^*$ = even-length strings where each pair consists of a 0 and a 1.
  - M_1:



  - Construct M_2:



Closure, summary

- FA-recognizable (regular) languages are closed under set operations, concatenation, and star.

- Regular operations: Union, concatenation, and star.
- Can be used to build regular expressions, which denote languages.
- E.g., regular expression $(0 \cup 1)^* \, 0 \, 0^*$ denotes the language $\{ 0, 1 \}^* \{0\} \{0\}^*$
- Study these next…

Regular Expressions

Regular expressions

- An algebraic-expression notation for describing (some) languages, rather than a machine representation.
- Languages described by regular expressions are exactly the FA-recognizable languages.
  - That's why FA-recognizable languages are called "regular".

- Definition: R is a regular expression over alphabet Σ exactly if R is one of the following:
  - a, for some a in Σ,
  - ε,
  - $\emptyset$,
  - $(R_1 \cup R_2)$, where R_1 and R_2 are smaller regular expressions,
  - $(R_1 \circ R_2)$, where R_1 and R_2 are smaller regular expressions, or
  - (R_1^*), where R_1 is a smaller regular expression.

- A recursive definition.

Regular expressions

- Definition: R is a regular expression over alphabet Σ exactly if R is one of the following:
  - a, for some a in Σ,
  - ε,
  - $\emptyset$,
  - $(R_1 \cup R_2)$, where R_1 and R_2 are smaller regular expressions,
  - $(R_1 \circ R_2)$, where R_1 and R_2 are smaller regular expressions, or
  - (R_1^*), where R_1 is a smaller regular expression.
- These are just formal expressions---we haven't said yet what they "mean".
- Example: $(((0 \cup 1) \circ \varepsilon)^* \cup 0)$
- Abbreviations:
  - Sometimes omit $\circ$, use juxtaposition.
  - Sometimes omit parens, use precedence of operations: * highest, then $\circ$, then $\cup$.
- Example: Abbreviate above as $((0 \cup 1) \varepsilon)^* \cup 0$
- Example: $(0 \cup 1)^* 111 (0 \cup 1)^*$

How regular expressions denote languages

- Define the languages recursively, based on the expression structure:
- Definition:
  - $L(a) = \{ a \}$; one string, with one symbol a.
  - $L(\varepsilon) = \{ \varepsilon \}$; one string, with no symbols.
  - $L(\emptyset) = \emptyset$; no strings.
  - $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$
  - $L(R_1 \circ R_2) = L(R_1) \circ L(R_2)$
  - $L(R_1^*) = (L(R_1))^*$

- Example: Expression $((0 \cup 1) \varepsilon)^* \cup 0$ denotes language $\{ 0, 1 \}^* \cup \{ 0 \} = \{ 0, 1 \}^*$, all strings.
- Example: $(0 \cup 1)^* 111 (0 \cup 1)^*$ denotes $\{ 0, 1 \}^* \{ 111 \} \{ 0, 1 \}^*$, all strings with substring 111.

More examples

- Definition:
  - $L(a) = \{ a \}$; one string, with one symbol a.
  - $L(\varepsilon) = \{ \varepsilon \}$; one string, with no symbols.
  - $L(\emptyset) = \emptyset$; no strings.
  - $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$
  - $L(R_1 \circ R_2) = L(R_1) \circ L(R_2)$
  - $L(R_1^*) = (L(R_1))^*$

- Example: L = strings over { 0, 1 } with odd number of 1s.

  $0^* \, 1 \, 0^* \, (0^* \, 1 \, 0^* \, 1 \, 0^*)^*$

- Example: L = strings with substring 01 or 10.

  $(0 \cup 1)^* \, 01 \, (0 \cup 1)^* \cup (0 \cup 1)^* \, 10 \, (0 \cup 1)^*$

  Abbreviate (writing Σ for $(0 \cup 1)$):

  $\Sigma^* \, 01 \, \Sigma^* \cup \Sigma^* \, 10 \, \Sigma^*$

More examples

- Example: L = strings with substring 01 or 10.

  $(0 \cup 1)^* 01 (0 \cup 1)^* \cup (0 \cup 1)^* 10 (0 \cup 1)^*$

  Abbreviate:

  $\Sigma^* 01 \Sigma^* \cup \Sigma^* 10 \Sigma^*$

- Example: L = strings with neither substring 01 or 10.
  - Can't write complement.
  - But can write: $0^* \cup 1^*$.
- Example: L = strings with no more than two consecutive 0s or two consecutive 1s
  - Would be easy if we could write complement.

    $(\varepsilon \cup 1 \cup 11) ((0 \cup 00) (1 \cup 11))^* (\varepsilon \cup 0 \cup 00)$
  - Alternate one or two of each.

More examples

- Regular expressions commonly used to specify syntax.
  - For (portions of) programming languages
  - Editors
  - Command languages like UNIX shell
- Example: Decimal numbers

$D\ D^*\ .\ D^* \ \cup\ D^*\ .\ D\ D^*$,

where D is the alphabet { 0, …, 9 }

Need a digit either before or after the decimal point.

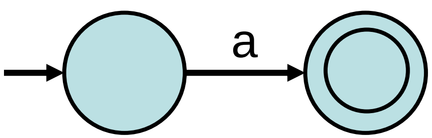
Regular Expressions Denote FA-Recognizable Languages

Languages denoted by regular expressions

- The languages denoted by regular expressions are exactly the regular (FA-recognizable) languages.
- Theorem 1: If R is a regular expression, then L(R) is a regular language (recognized by a FA).
- Proof: Easy.
- Theorem 2: If L is a regular language, then there is a regular expression R with L = L(R).
- Proof: Harder, more technical.

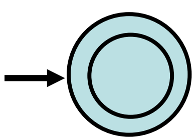
Theorem 1

- Theorem 1: If R is a regular expression, then L(R) is a regular language (recognized by a FA).
- Proof:
  - For each R, define an NFA M with L(M) = L(R).
  - Proceed by induction on the structure of R:
    - Show for the three base cases.
    - Show how to construct NFAs for more complex expressions from NFAs for their subexpressions.
  - Case 1: R = a
    - L(R) = { a }



Accepts only a.

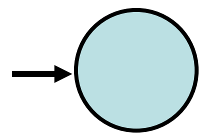
  - Case 2: R = ε
    - L(R) = { ε }



Accepts only ε.

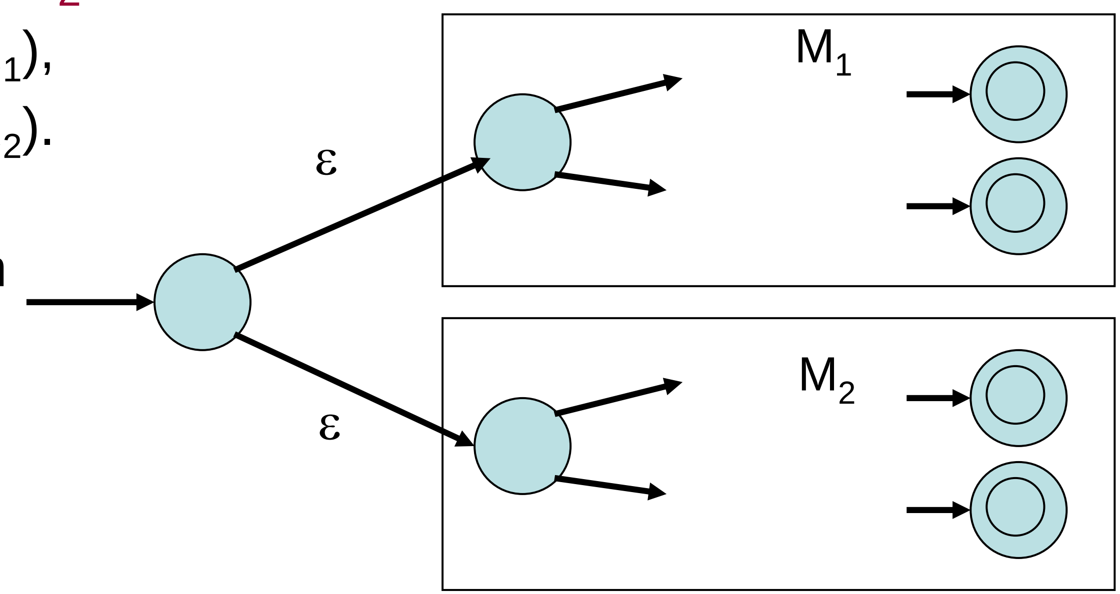
Theorem 1

- Theorem 1: If R is a regular expression, then L(R) is a regular language (recognized by a FA).
- Proof:
  - Case 3: $R = \varnothing$
    - $L(R) = \varnothing$



Accepts nothing.

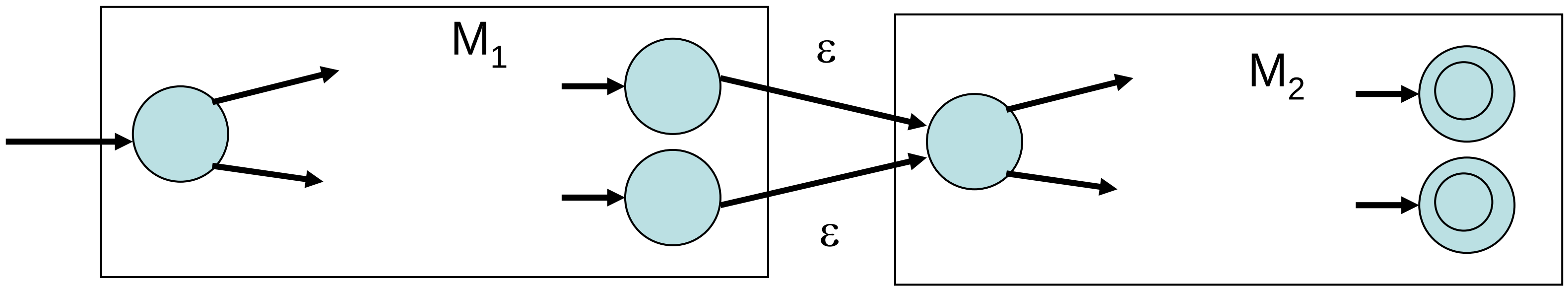
  - Case 4: $R = R_1 \cup R_2$
    - M_1 recognizes $L(R_1)$,
    - M_2 recognizes $L(R_2)$.
    - Same construction we used to show regular languages are closed under union.



Theorem 1

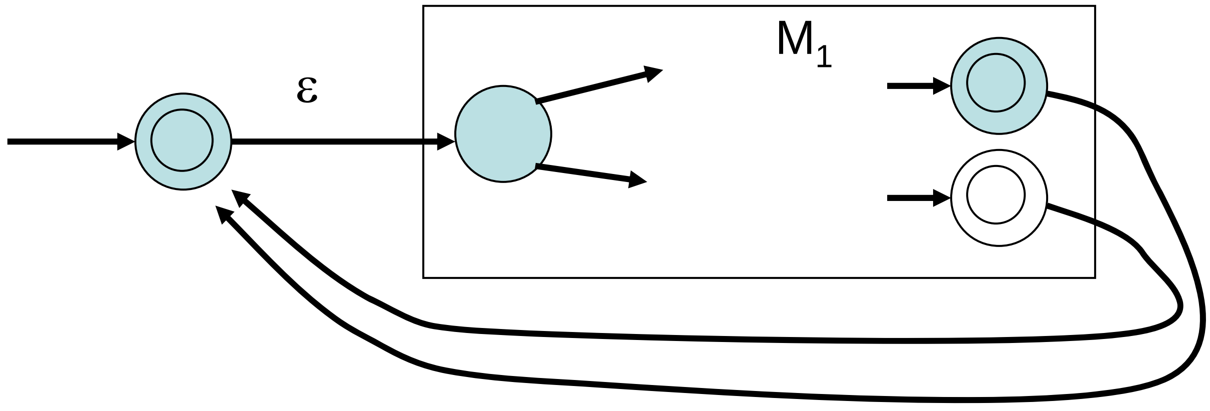
- Theorem 1: If R is a regular expression, then L(R) is a regular language (recognized by a FA).
- Proof:
  - Case 5: $R = R_1 \circ R_2$
    - M_1 recognizes $L(R_1)$,
    - M_2 recognizes $L(R_2)$.

    - Same construction we used to show regular languages are closed under concatenation.

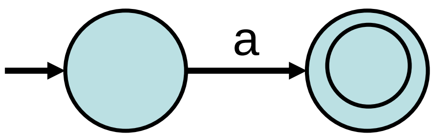


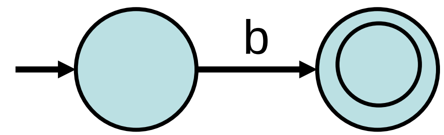
Theorem 1

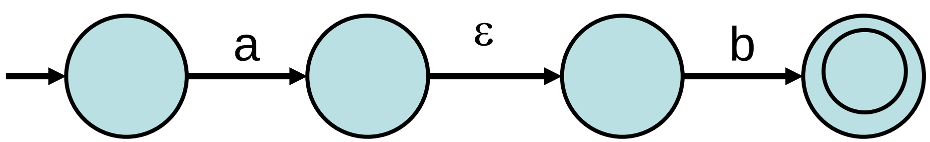
- Theorem 1: If R is a regular expression, then L(R) is a regular language (recognized by a FA).
- Proof:
  - Case 6: $R = (R_1)^*$
    - M_1 recognizes $L(R_1)$,
    - Same construction we used to show regular languages are closed under star.

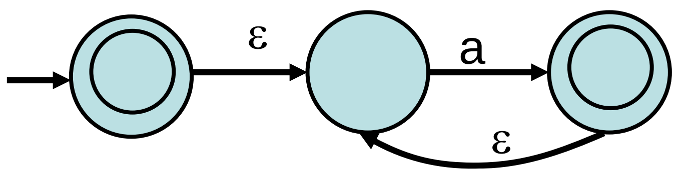


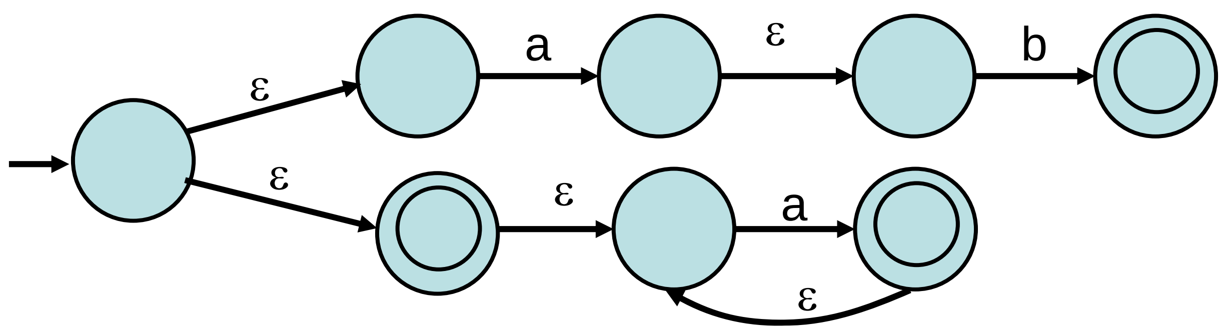
Example for Theorem 1

- L = ab ∪ a*
- Construct machines recursively:
- a: 

  b: 

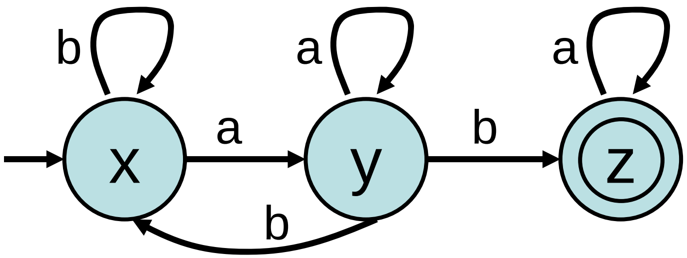
- ab: 

- a*: 

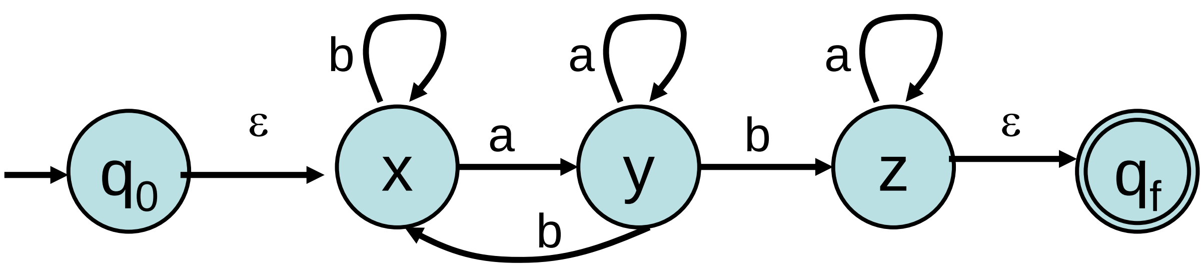
- ab ∪ a*: 

Theorem 2

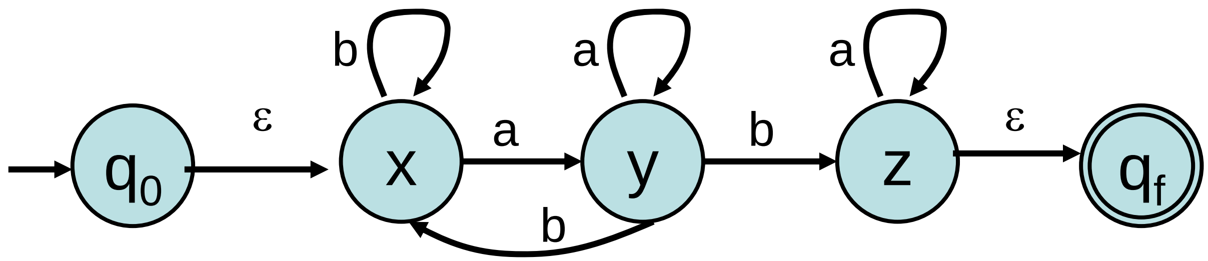
- Theorem 2: If L is a regular language, then there is a regular expression R with L = L(R).
- Proof:
  - For each NFA M, define a regular expression R with L(R) = L(M).
  - Show with an example:



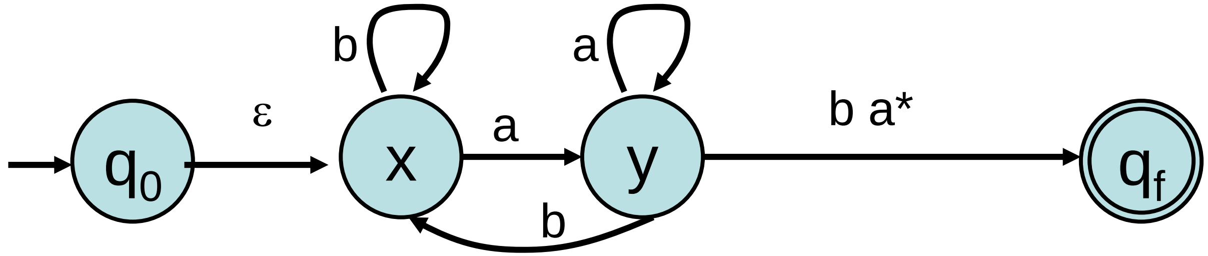
  - Convert to a special form with only one final state, no incoming arrows to start state, no outgoing arrows from final state.



Theorem 2

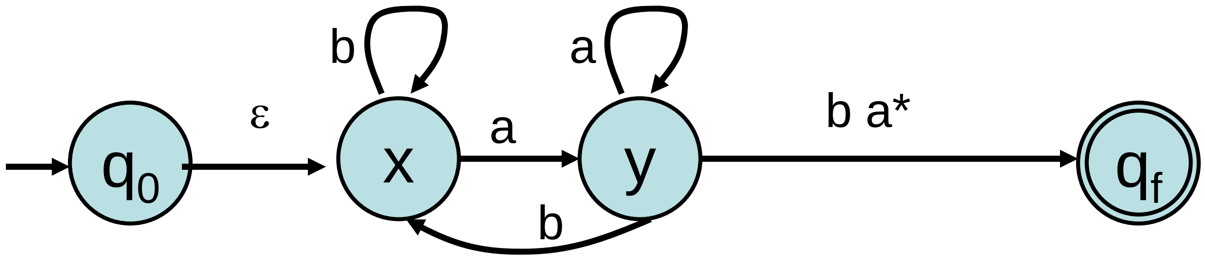


- Now remove states one at a time (any order), replacing labels of edges with more complicated regular expressions.
- First remove z:

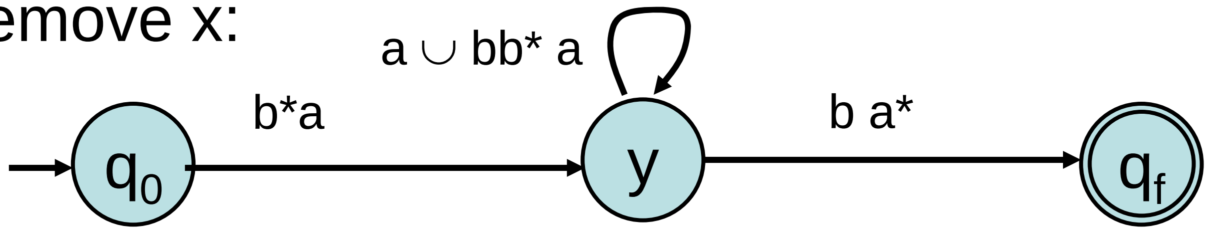


- New label b a* describes all strings that can move the machine from state y to state q_f, visiting (just) z any number of times.

Theorem 2

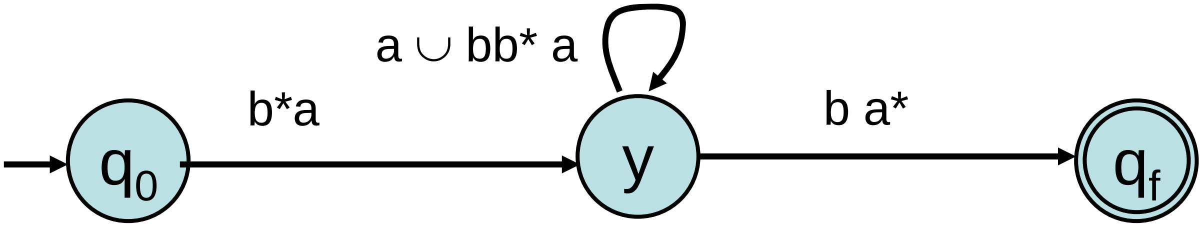


- Then remove x:

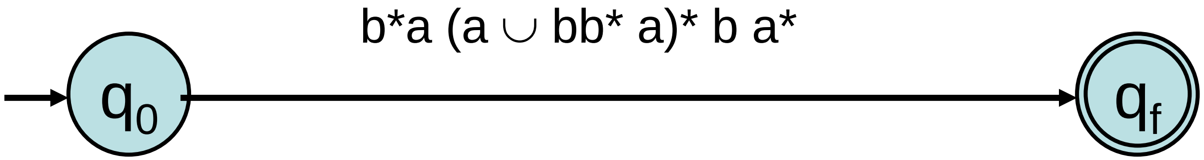


- New label b*a describes all strings that can move the machine from q_0 to y, visiting (just) x any number of times.
- New label a ∪ bb* a describes all strings that can move the machine from y to y, visiting (just) x any number of times.

Theorem 2



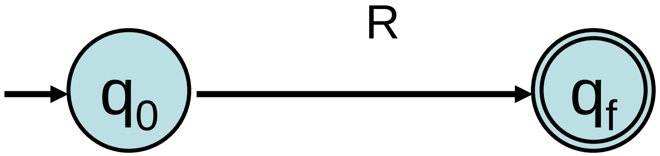
- Finally, remove y:



- New label describes all strings that can move the machine from q_0 to q_f, visiting (just) y any number of times.
- This final label is the needed regular expression.

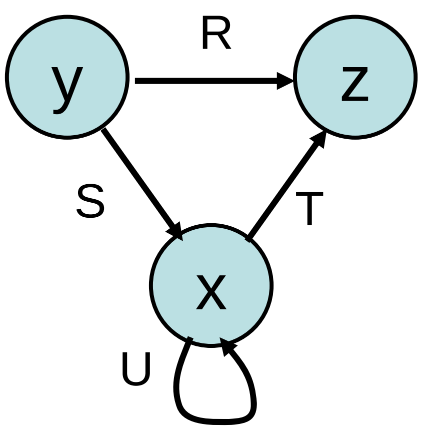
Theorem 2

- Define a generalized NFA (gNFA).
  - Same as NFA, but:
    - Only one accept state, ≠ start state.
    - Start state has no incoming arrows, accept state no outgoing arrows.
    - Arrows are labeled with regular expressions.
  - How it computes: Follow an arrow labeled with a regular expression R while consuming a block of input that is a word in the language L(R).
- Convert the original NFA M to a gNFA.
- Successively transform the gNFA to equivalent gNFAs (recognize same language), each time removing one state.
- When we have 2 states and one arrow, the regular expression R on the arrow is the final answer:

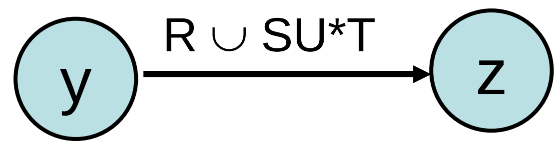


Theorem 2

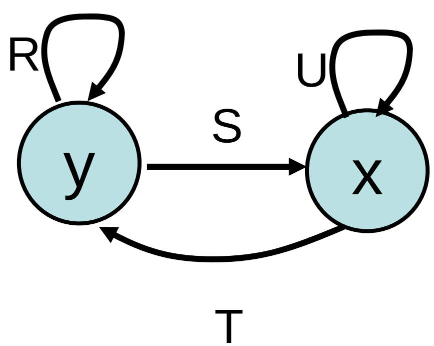
- To remove a state x, consider every pair of other states, y and z, including y = z.
- New label for edge (y, z) is the union of two expressions:
  - What was there before, and
  - One for paths through (just) x.
- If y ≠ z:

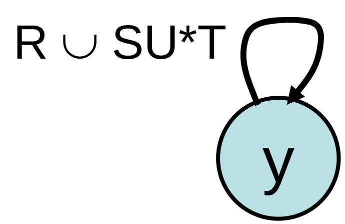


we get:



- If y = z:





Next time…

- Existence of non-regular languages
- Showing specific languages aren’t regular
- The Pumping Lemma
- Algorithms that answer questions about FAs.

- Reading: Sipser, Section 1.4; some pieces from 4.1

6.045J / 18.400J Automata, Computability, and Complexity
Spring 2011