

CS240 • Language Theory and Automata • Fall 2011

# Finite Automata

## Abstract Machines

- Modern computers are capable of performing a wide variety of computations
- An **abstract machine** reads in an input string, and depending on the input
  - outputs **true** (accept)
  - outputs **false** (reject)
  - gets stuck in an infinite loop and outputs nothing
- We say that a machine **recognizes** a particular language if it outputs true for any input string in the language it is designed to handle, and false otherwise
- The artificial restriction to such **decision problems** is purely for notational convenience

## Language Recognition Problems

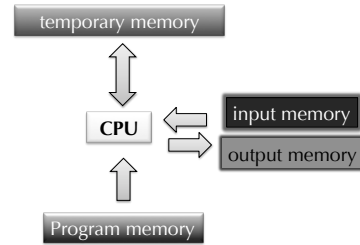
- Virtually all computational problems can be recast as **language recognition problems**
- Examples:
  - to determine whether an integer 97 is prime, ask whether 97 is in the language consisting of all primes {2, 3, 5, 7, 13, ... }
  - to determine the decimal expansion of the mathematical constant  $\pi$ , ask whether 7 is the 100th digit of  $\pi$  and so on

## Power

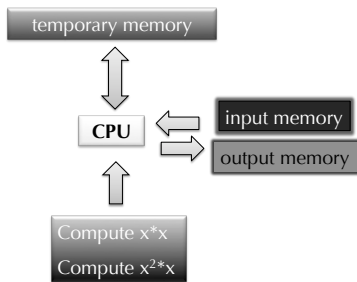
- We would like to be able to formally compare different classes of abstract machines in order to address questions like
  - Is a Mac more powerful than a PC?
  - Can Java do more things than C++?
- To accomplish this, we define a notion of **power**
  - We say that machine A is at least as powerful as machine B if machine A can be "programmed" to **recognize all of the languages** that B can.
  - Machine A is more powerful than B, if in addition, it can be programmed to recognize at least one additional language
  - Two machines are equivalent if they can be programmed to recognize precisely the same set of languages
- Using this definition of power, we will classify several fundamental machines

- We are interested in designing the most powerful computer, i.e., the one that can solve the widest range of language recognition problems
  - our notion of power does not say anything about how fast a computation can be done
  - reflects a more fundamental notion of whether or not it is even possible to perform some computation in a finite number of steps

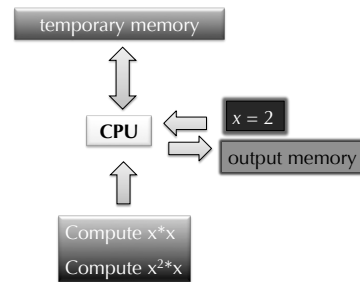
### Computing Machine

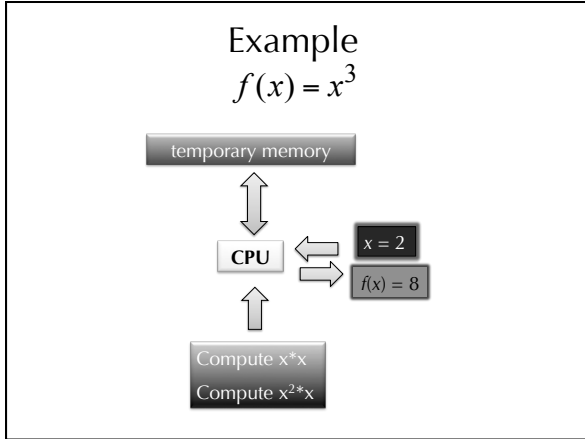


### Example $f(x) = x^3$

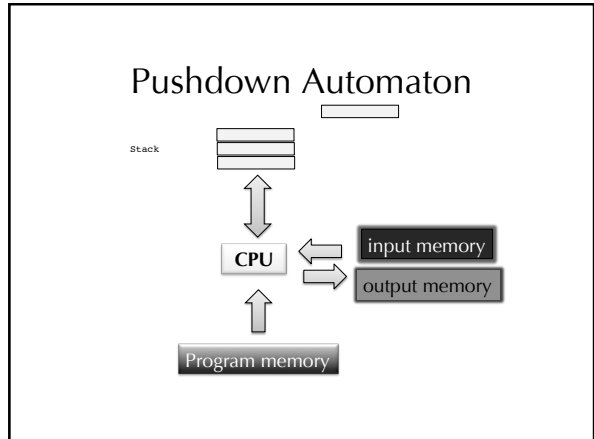
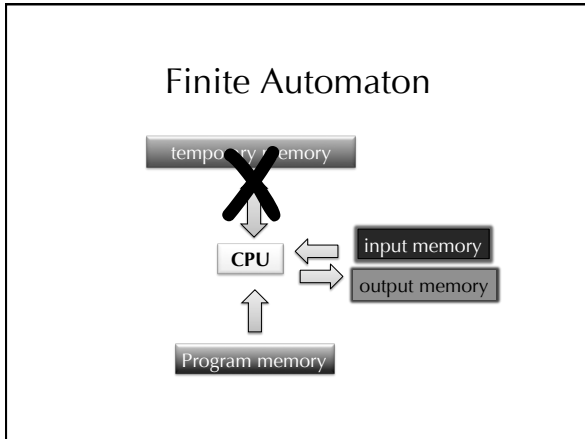


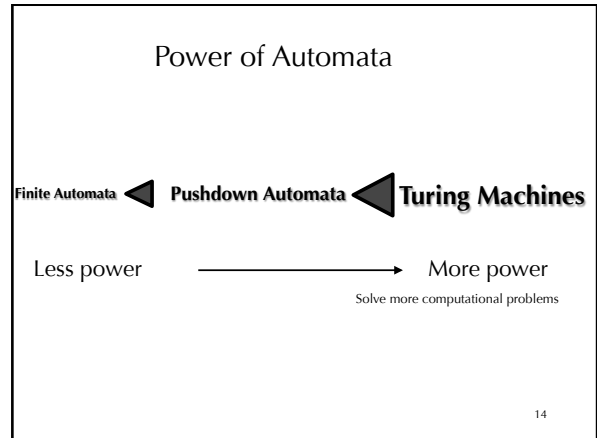
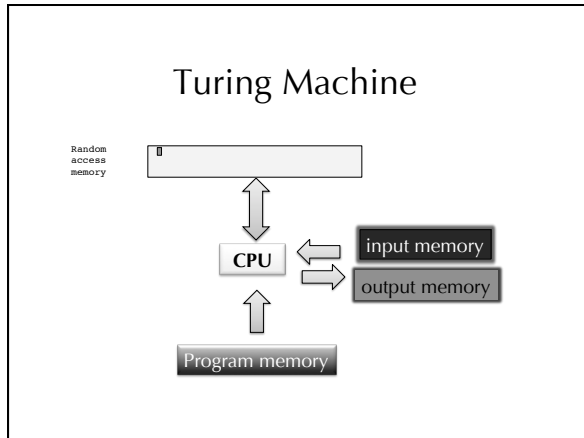
### Example $f(x) = x^3$





- ### Different Kinds of Automata
- Automata are distinguished by their **temporary memory**
    - **Finite Automata**
      - No temporary memory
    - **Pushdown Automata**
      - Stack
    - **Turing Machines**
      - Random access memory
- 10





- ### Finite Automaton
- Perhaps the simplest type of machine that is still interesting to study
    - Many of its important properties carry over to more complicated machines
    - To understand more complicated machines, we first study FAs
  - Captures the basic elements of an abstract machine
    - Reads in a string, and depending on the input and the way the machine was designed, it outputs true (yes) or false (no)
  - A useful practical abstraction because
    - FAs retain sufficient flexibility to perform interesting tasks
    - Hardware requirements for building them are relatively minimal
  - An important way to describe certain simple, but highly useful languages called **regular languages**

- ### Definition
- A **finite automaton** is a graph with a finite number of nodes, called **states**.
  - Arcs are labeled with one or more symbols from some alphabet.
  - One state is designated the **start state** or *initial state*.
  - Some states are **final states** or *accepting states*.
  - The **language of the FA** is the set of strings that label paths that go from the start state to some accepting state

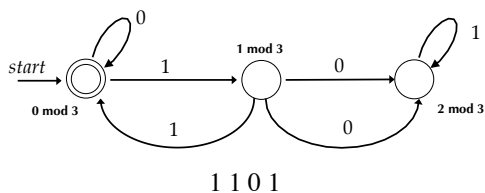
## Operation

- FA is always is one of the  $n$  states, which we name 0 through  $n-1$ 
  - Each state is labeled **true** (“yes”) or **false** (“no”)
- Begins in the start state
- As the input characters are read in one at a time, changes from one state to another in a pre-specified way
  - new state is completely determined by the current state and the character just read in
- When input is exhausted, outputs true (yes, the string is in the language) or false (no, the string is not in the language) according to the label of the state it is currently in

## Transition Diagram

- Represent visually by a graph:
  - **nodes = states**
  - **arc from  $q$  to  $p$**  is labeled by the set of input symbols  $a$  such that  $\delta(q, a) = p$
  - No arc if no such  $a$
  - **Start state** indicated by an **arrow**
  - **Accepting states** (those labeled “yes”) get **double circles**
  - Non-accepting states are implicitly labeled “no”

## Example



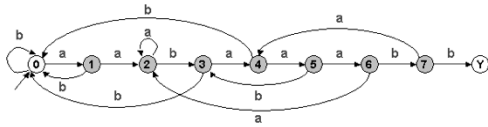
Automaton Simulator lets you create and simulate a DFA  
<http://ozark.bendrix.edu/~7Ehurch/iproj/autosim/download.html>  
<http://www.cs.vassar.edu/~cs240/AutoSim.jar>

## FAs in Action

- Used in
  - Text editors for pattern matching
  - Compilers for lexical analysis
  - Web browsers for html parsing
  - Operating systems for graphical user interfaces
- Serve as the control unit in many physical systems, including
  - Vending machines, elevators, automatic traffic signals
  - Computer microprocessors
  - Network protocol stacks and old VCR clocks
- Play a key role in natural language processing and machine learning

## String searching FAs

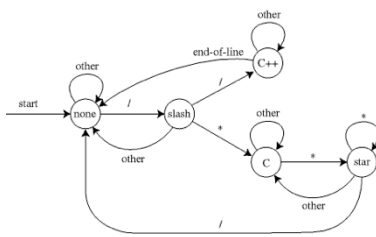
- One of the most important applications of FAs is searching for patterns in strings
  - at the heart of Web search engines like Google
- FA providing a simplified example of such a tool over the binary alphabet
  - Accepts all string inputs that contain the pattern `aabaaabb` as a substring



## Another example

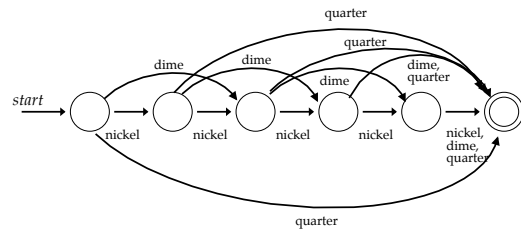
- CommentStripper.java
  - reads in a Java (or C++) program from standard input, removes all comments, and prints the result to standard output
  - removes `/** */` and `//` style comments using a 5 state finite state automaton
  - Shows DFA power, but to properly strip Java comments, you would need a few more states to handle extra cases, e.g., quoted string literals like `"/** */"`

## The FA

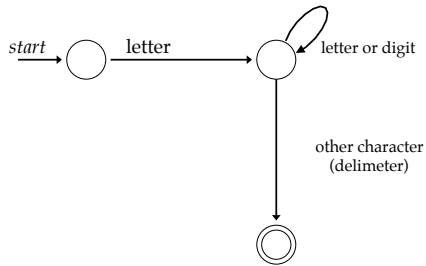


Picture courtesy of David Eppstein via <http://www.cs.princeton.edu/introcs/>

## FA for Newspaper Vending Machine



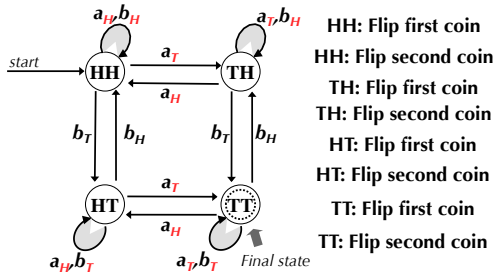
**FA that recognizes simple identifiers**



**FA for coin flipping**

- Four ways of arranging two coins, depending on which is heads (H) and which is tails (T)
  - HH, HT, TT, TH
- Two operations:
  - Flip the first coin (*a*)
  - Flip the second coin (*b*)
- Assume initially coins are laid out as HH
- What are all possible ways of applying the operations so that the configuration is TT?

**Model as an FA**

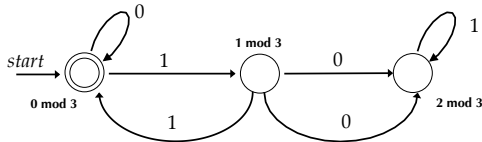


**Conventions**

- It helps if we can avoid mentioning the type of every name by following some rules:
  - Input symbols are *a*, *b*, etc., or digits.
  - Strings of input symbols are *u*, *v*, . . . , *z*.
  - States are *q*, *p*, etc.

### Determinism

- All of the FAs we have just seen are **deterministic finite automata** (DFAs)
  - Only **one choice of move** from one state to another for a given input symbol
  - A move in each state for **every input symbol**



### Formal Definition of DFA

- Finite set of **states**,  $Q$ .
- Alphabet of **input symbols**,  $\Sigma$ .
- One state is the **start/initial state**,  $q_0$ .
- Zero or more **final/accepting states**, the set is  $F$ .
- A **transition function**,  $\delta$ . This function:
  - Takes a state and input symbol as arguments.
  - Returns a state.
  - One "rule" of  $\delta$  would be written  $\delta(q, a) = p$ , where  $q$  and  $p$  are states, and  $a$  is an input symbol.
  - Intuitively: if the FA is in state  $q$ , and input  $a$  is received, then the FA goes to state  $p$  (note:  $q = p$  OK).

An FA is represented as the five-tuple:  $A = (Q, \Sigma, \delta, q_0, F)$ .

### Example: Clamping Logic

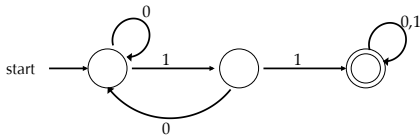
- We may think of an accepting state as representing a "1" output and non-accepting states as representing "0" output.
- A "clamping" circuit waits for a 1 input, and forever after makes a 1 output. However, to avoid clamping on spurious noise, we'll design an FA that waits for two 1's in a row, and "clamps" only then.
- In general, we may think of a state as representing a summary of the history of what has been seen on the input so far.

- The states we need are:
  - State  $q_0$ , the start state, says that the most recent input (if there was one) was not a 1, and we have never seen two 1's in a row.
  - State  $q_1$  says we have never seen 11, but the previous input was 1.
  - State  $q_2$  is the only accepting state, it says that we have at some time seen 11.
  - Thus,  $A = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$ , where  $\delta$  is given by:

	0	1
$>q_0$	$q_0$	$q_1$
$q_1$	$q_0$	$q_2$
$*q_2$	$q_2$	$q_2$

By marking the start state with  $>$  and accepting states with  $*$ , the transition table that defines  $\delta$  also specifies the entire FA

### Transition Graph



### Extension of $\delta$ to Paths

Intuitively, a FA accepts a string  $w = a_1 a_2 \dots a_n$  if there is a path in the transition diagram that:

1. Begins at the start state,
2. Ends at an accepting state, and
3. Has sequence of labels  $a_1, a_2, \dots, a_n$ .

Formally, we extend transition function  $\delta$  to  $\hat{\delta}(q, w)$ , where  $w$  can be any string of input symbols:

- **Basis:**  $\hat{\delta}(q, \epsilon) = q$  (i.e., on no input, the FA doesn't go anywhere).
- **Induction:**  $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$ , where  $w$  is a string, and  $a$  a single symbol (i.e., see where the FA goes on  $w$ , then look for the transition on the last symbol from that state).

**Important fact with a straightforward, inductive proof:**  
 $\hat{\delta}$  really represents paths. That is, if  $w = a_1 a_2 \dots a_n$ , and  $\hat{\delta}(p_i, a_i) = p_{i+1}$  for all  $i = 0, 1, \dots, n-1$ , then  $\hat{\delta}(p_0, w) = p_n$

#### • Acceptance of Strings

An FA  $A = (Q, \Sigma, \delta, q_0, F)$  accepts string  $w$  if

$$\hat{\delta}(q_0, w) \in F$$

#### • Language of a FA

- FA  $A$  accepts the language

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \in F\}$$

### Type Errors

- A major source of confusion when dealing with automata (or mathematics in general) is making "type errors."
  - Example: Don't confuse  $A$ , a FA, i.e., a program, with  $L(A)$ , which is of type "set of strings."
  - Example: the start state  $q_0$  is of type "state," but the accepting states  $F$  is of type "set of states."
  - Trickier example: Is  $a$  a symbol or a string of length 1?

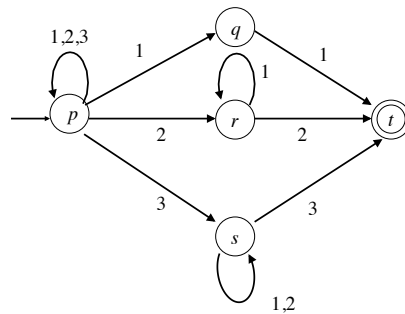
Answer: it depends on the context, e.g., is it used in  $\delta(q, a)$ , where it is a symbol, or  $\delta(q, a)$ , where it is a string?

### Non-deterministic Finite Automata

- Allow (deterministic) FA to have a choice of 0 or more next states for each state-input pair.
- Important tool for designing string processors, e.g., grep, lexical analyzers.
- But "imaginary," in the sense that it has to be implemented deterministically.

### Example

- Design an NFA to accept strings over alphabet  $\{1, 2, 3\}$  such that the last symbol appears previously, without any intervening higher symbol, e.g.,
  - ... 11
  - ... 21112
  - ... 312123
- *Trick:* use start state to mean "I guess I haven't seen the symbol that matches the ending symbol yet."
- Three other states represent a guess that the matching symbol has been seen, and remembers what that symbol is.

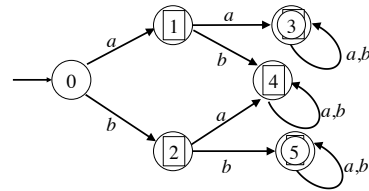


### Formal NFA

- $N = (Q, \Sigma, \delta, q_0, F)$  where all is as DFA, but:
  - $\delta(q, a)$  is a set of states, rather than a single state.
- Extension to  $\hat{\delta}$ 
  - Basis:  $\hat{\delta}(q, \epsilon) = \{q\}$ .
  - Induction: Let:
    - $\hat{\delta}(q, w) = \{p_1, p_2, \dots, p_k\}$ .
    - $\hat{\delta}(p_i, a) = S_i$ , for  $i = 1, 2, \dots, k$ .
    - Then  $\hat{\delta}(q, wa) = S_1 \cup S_2 \cup \dots \cup S_k$ .
- Language of an NFA
  - An NFA accepts  $w$  if any path from the start state to an accepting state is labeled  $w$ . Formally:
 
$$L(N) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \Phi\}.$$

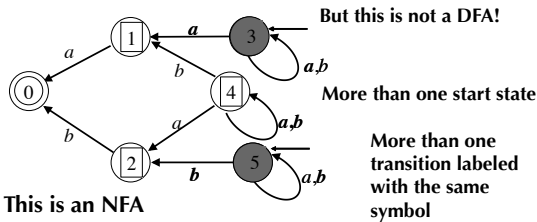
### Example

Here is a DFA that accepts a language  $L$  consisting of all strings over  $\Sigma(a,b)$  that begin with either  $aa$  or  $bb$



Suppose we want to make an automaton to recognize  $REV(L)$ , the language of all strings that end in  $aa$  or  $bb$

**Easy solution:** reverse all transitions and interchange start and final states:



This is an NFA

### NFAs and DFAs

- It might seem that because there is a degree of choice available in an NFA that it might be more powerful than a DFA
  - That is, NFAs might be able to recognize languages a DFA could not
- But this is not the case!

## Equivalence of NFAs and DFAs

- **NFAs and DFAs recognize the same class of languages**
- A bit surprising: NFAs seem more powerful
- Useful: easier to specify an NFA for a language, convert to DFA

Two machines are **equivalent** if they recognize the same language

## Theorem

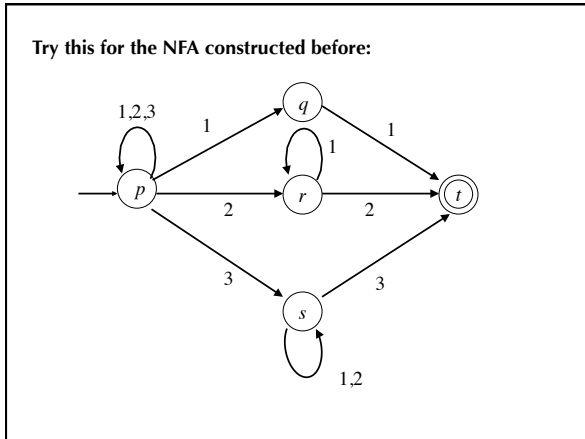
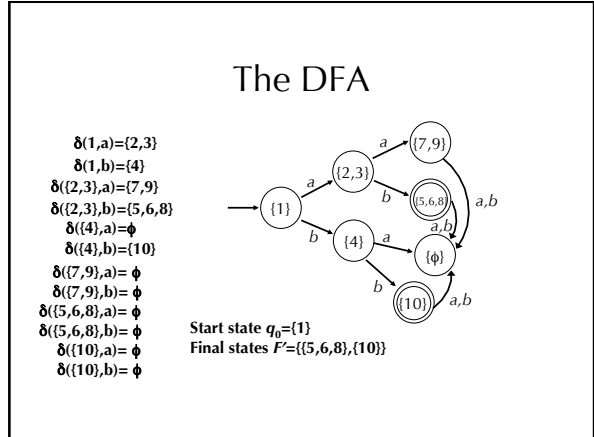
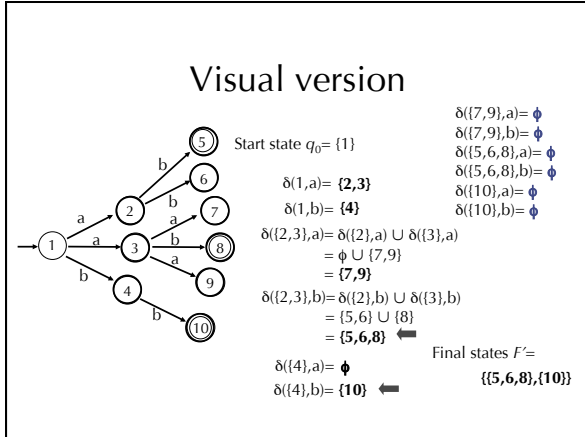
Every non-deterministic finite automaton has an equivalent deterministic finite automaton

## Proof Idea

- If a language is recognized by an NFA, show the existence of a DFA that also recognizes it
- Convert NFA to an equivalent DFA that simulates the NFA
  - **Proof by construction**
- Intuitively, can simulate the NFA by keeping track of all the states you can get to on a given input

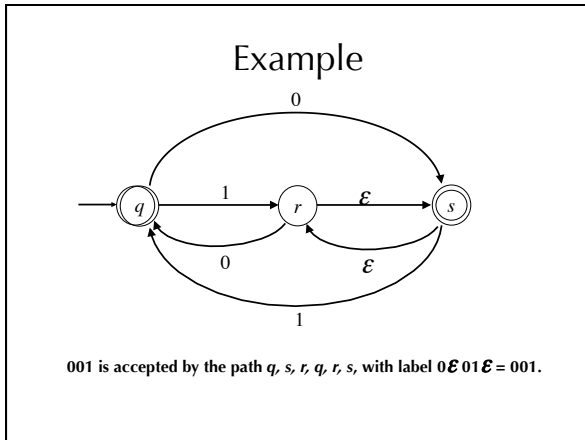
## Proof

- Let  $N = (Q, \Sigma, \delta, q_0, F)$  be an NFA recognizing some language  $A$
- Construct a DFA recognizing  $A$ 
  - $M = (Q', \Sigma, \delta', q_0', F')$
- 1.  $Q'$  = the set of subsets of  $Q$
- 2. For  $R \in Q'$  and  $a \in \Sigma$  let  $\delta'(R, a) = \{q \in Q \mid q \in \delta(r, a) \text{ for some } r \in R\}$ 
  - If  $R$  is a state of  $M$ , it is also a set of states of  $N$  (because of 1 above). When  $M$  reads a symbol  $a$  in a state  $R$ , it shows where  $a$  takes each state in  $R$ . Because each state may go to a set of states, we take the union of all these sets. This can be written as:
 
$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$$
- $q_0' = \{q_0\}$ 
  - $M$  starts in the state corresponding to the collection containing just the start state of  $N$ .
- $F' = \{R \in Q' \mid R \text{ contains an accept state of } N\}$ 
  - The machine  $M$  accepts if one of the possible states that  $N$  could be in at this point is an accept state.



### NFA With $\epsilon$ -Transitions

- Allow  $\epsilon$  to be a label on arcs
  - **Nothing else changes:** acceptance of  $w$  is still the existence of a path from the start state to an accepting state with label  $w$ .
  - But  $\epsilon$  can appear on arcs, and means the empty string (i.e., no visible contribution to  $w$ )
  - When an arc labeled  $\epsilon$  is traversed, **no input is consumed**



### DFAs and NFA- $\epsilon$ 's

- $\epsilon$ -transitions are a convenience, but do not increase the power of FA's.
- **For any NFA- $\epsilon$  there is an equivalent (i.e., accepts the same language) DFA**
- The construction is similar to the NFA-to-DFA construction

### Creating a DFA from a NFA- $\epsilon$

(or, eliminating  $\epsilon$ -transitions)

1. Compute the  $\epsilon$ -closure for each state (set of states reachable from that state on  $\epsilon$ -transitions only).
2. Start state is  $\epsilon$ -CLOSE( $q_0$ ).
3. Compute  $\delta$  for each  $a \in \Sigma$  and each set  $S$  (each of the  $\epsilon$ -CLOSE'd sets) as follows:
  - If a state  $p \in S$  can reach state  $q$  on input  $a$  (not  $\epsilon$ ), then add a transition on input  $a$  from  $S$  to  $\epsilon$ -CLOSE( $q$ ).
4. The set of final states includes those sets that contain at least one accepting state of the NFA- $\epsilon$ .

