CS311—Computational Structures

Nondeterministic Finite State Automata

Lecture 3

Andrew P. Black
Andrew Tolmach

Portland State University

Review: Closure Under Union

- Theorem: Suppose $L_1 = L(M_1)$ and $L_2 = L(M_2)$ for DFAs M_1 and M_2. Then there is a machine M such that $L(M) = L_1 \cup L_2$.
- Proof by product construction
- We say "the set of regular languages is **closed** under union."

Portland State UNIVERSITY

Regular Operations

- Let A and B be languages. We define the following **regular operations**:
  - Union: $A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$
  - Concatenation: $A \cdot B = \{ xy \mid x \in A \text{ and } y \in B\}$
  - Star: $A^* = \{x_1 x_2 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A \}$
- Claim: the set of regular languages is **closed** under all the regular operations (that's where the name comes from!)

DFAs by concatenation?

- Design DFAs that recognize each of these languages over {a,b} :
  - $\{uv \mid u \in L_{2a}$ and $v \in L_{b\ldots}\}$
    - easy, but what's the general method?
  - $\{uv \mid u \in L_{2a}$ and $v \in L_{2b}\}$
    - not so easy!

DFAs by *-closure?

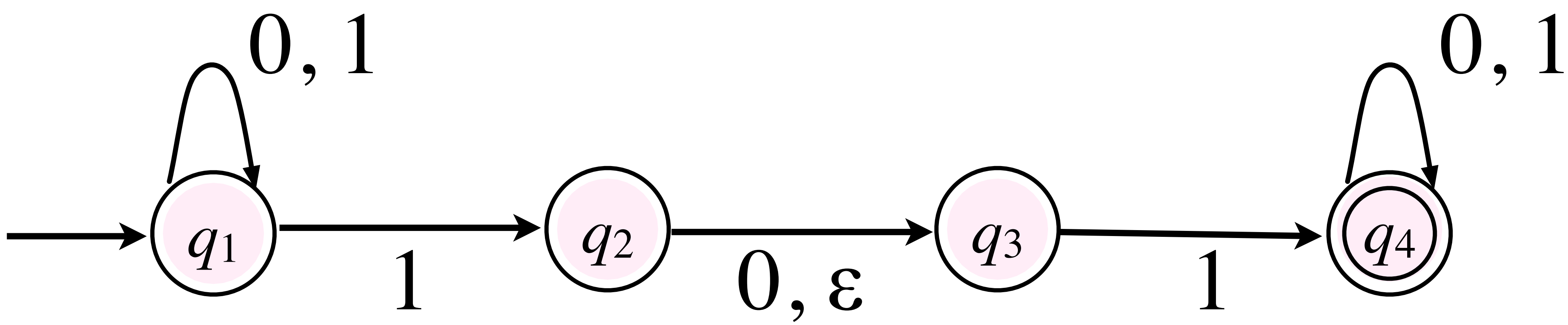
- Design DFAs that recognize each of these languages over {a,b} :
  - {w | w consists of 0 or more copies of ab}
    - Easy, but what's the general method?
  - L* where L = {w | w starts with aa or bb}
    - Does the method work here?

Nondeterminism

- In the FSAs that we have seen so far, there is exactly one action to be taken on each input symbol.
  – that's what it means for δ to be a function!
- In a nondeterministic FSA, *a set* of choices exist at each step.
  - zero, one or several possible transitions

Portland State University

Example of NFA

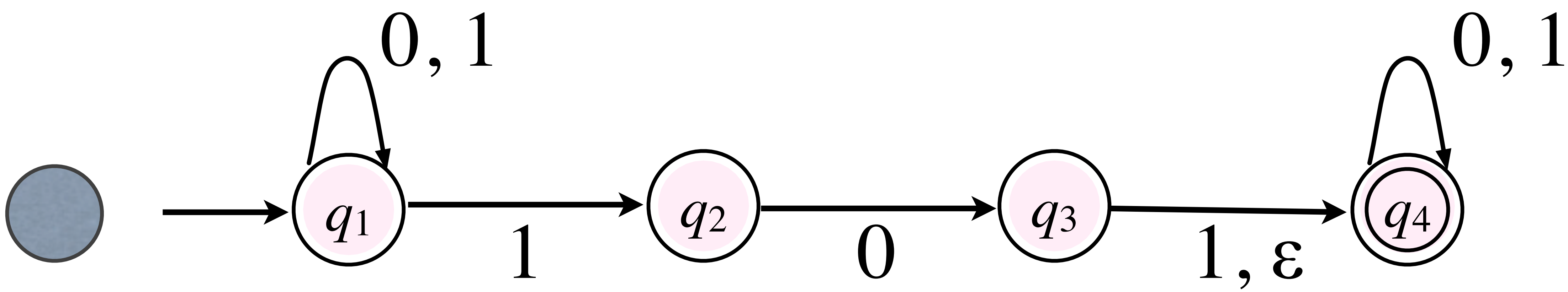


- How does it differ from a DFA?
  1. some states have multiple transitions on a given input
  2. some states have *no* transition on an input
  3. some transitions have label ε

Nondeterministic Computation

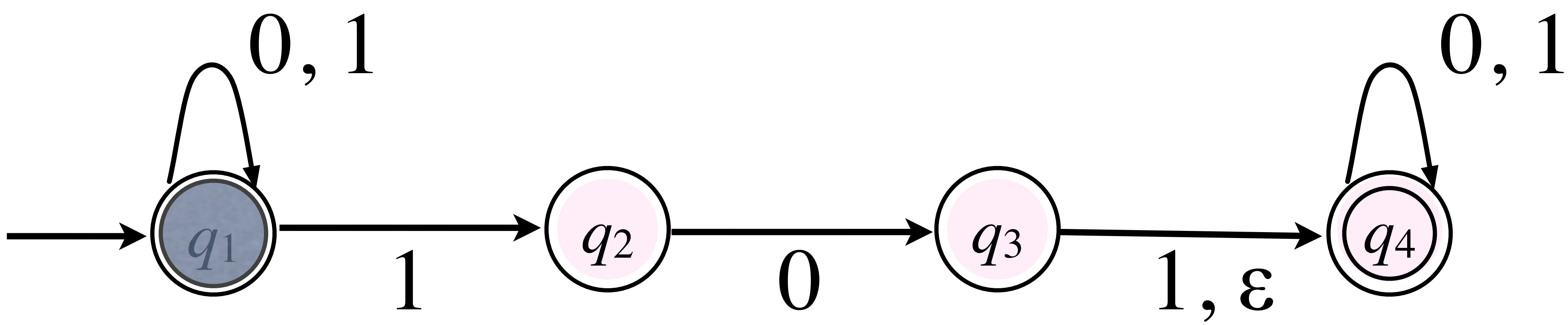
- What does it mean for an NFA to take a "step" when there are multiple possibilities at each step? What about ε?
  - the NFA makes all possible transitions in parallel; or, equivalently,
  - the NFA clones itself and one clone explores each possibility.
- an NFA can reach a "dead end" (gets stuck)
- an NFA accepts its input if **any** of the clones reaches a final state.

Example: 01100



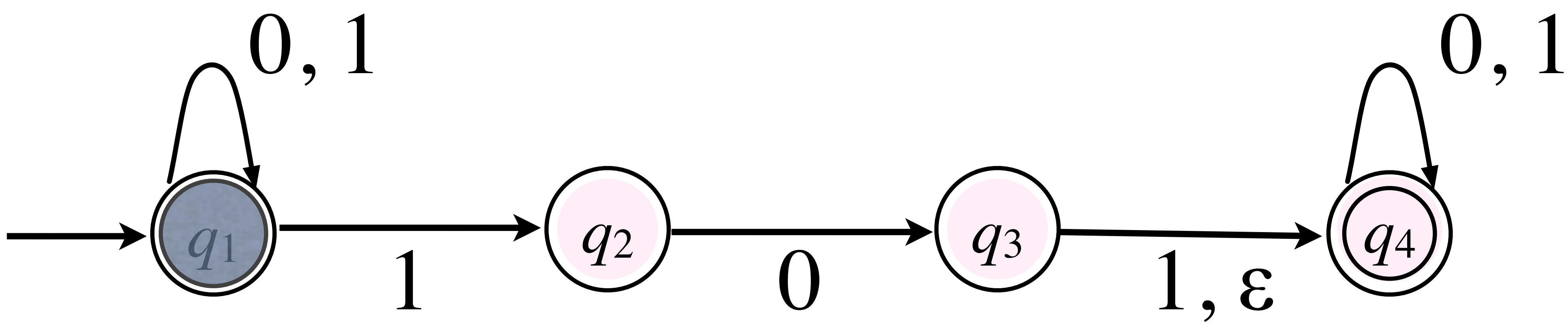
Portland State University

Example: 01100



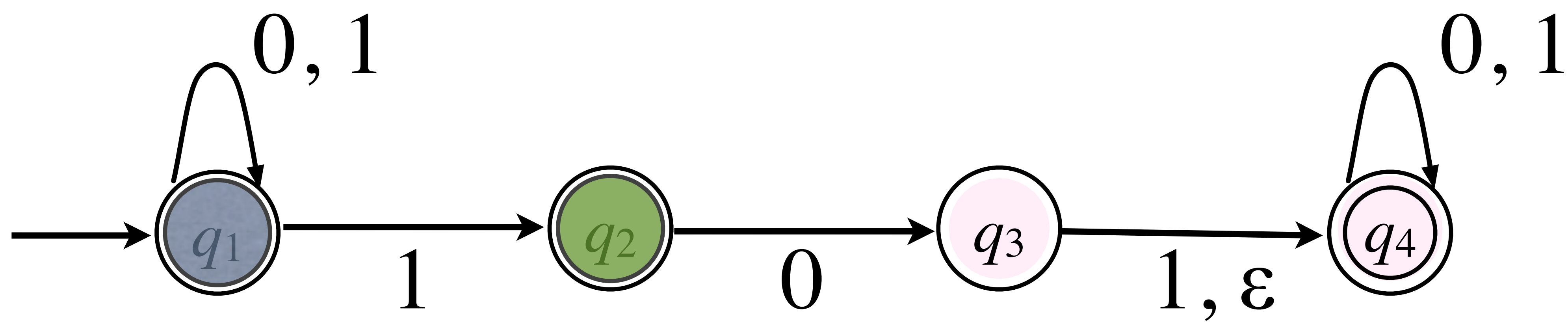
Computation always begins in start state, with one "thread"

Example: 01100



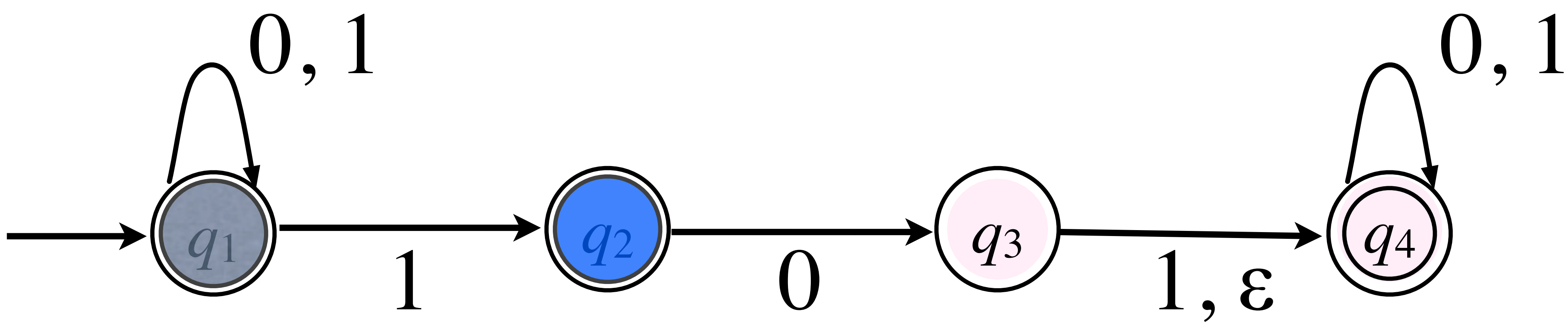
Only one transition possible on 0.

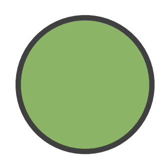
Example: 01100



- *Two* transitions are possible on 1, so machine enters either of two states.
  - Can think of two "threads" each representing a *possible* path through the machine.
  - Or: two machines, one grey, one green, each in its own state.

Example: 01100

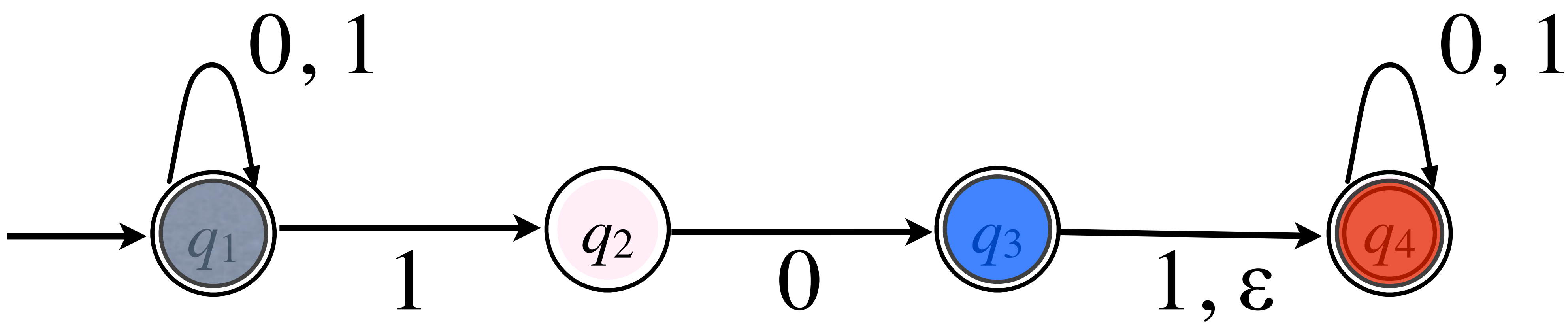


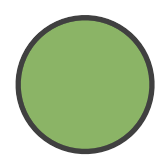


- Green token cannot move on 1, so the "green machine" aborts.
- But grey token can make either of two moves, so we get a new token (say, blue)

Portland State University

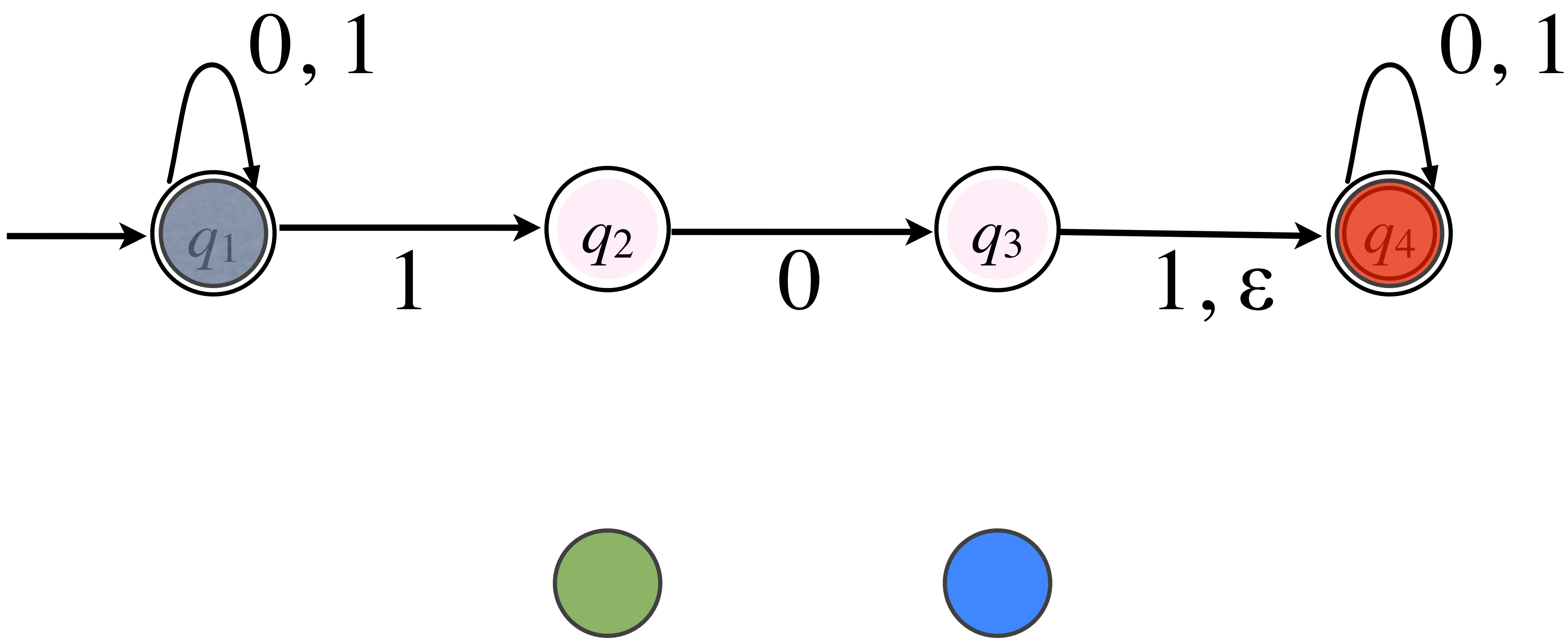
Example: 01100





- Blue token moves on 0, and then has the option of moving *again*, along the ε-edge.
- This makes a new token — say, red.
- So now we have three "threads"

Example: 01100



- Red token follows self-edge. Since there is a token in an accepting state, machine *accepts*.
- Blue token is stuck, so its "thread" dies.
- Grey "thread" is still alive in initial state.

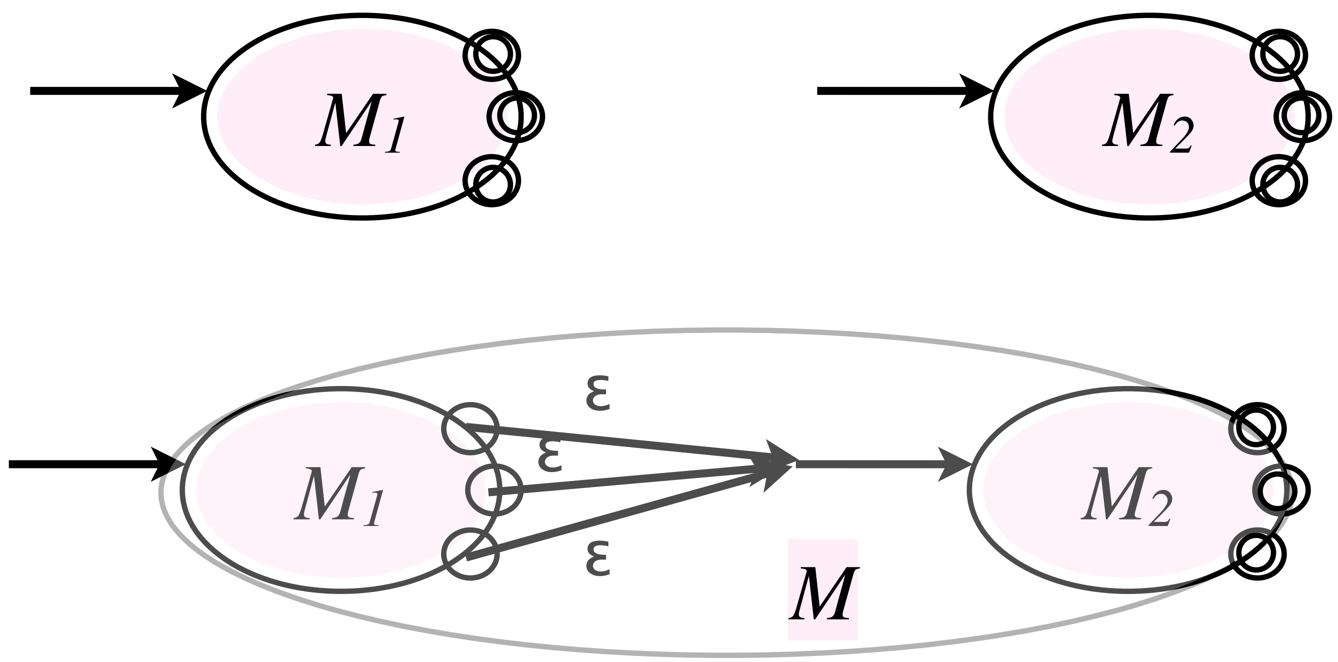
More NFA examples

- Construct NFAs recognizing these languages over {a,b}:
  - $\{uv \mid u \in L_{b\ldots} \text{ and } v \in L_{2a}\}$
  - $\{uv \mid u \in L_{baa} \text{ and } v \in L_{2b}\}$
  - L^* where $L = \{w \mid w \text{ starts with aa or bb}\}$
  - $L_{2a} \cup L_{bba}$

Portland State University

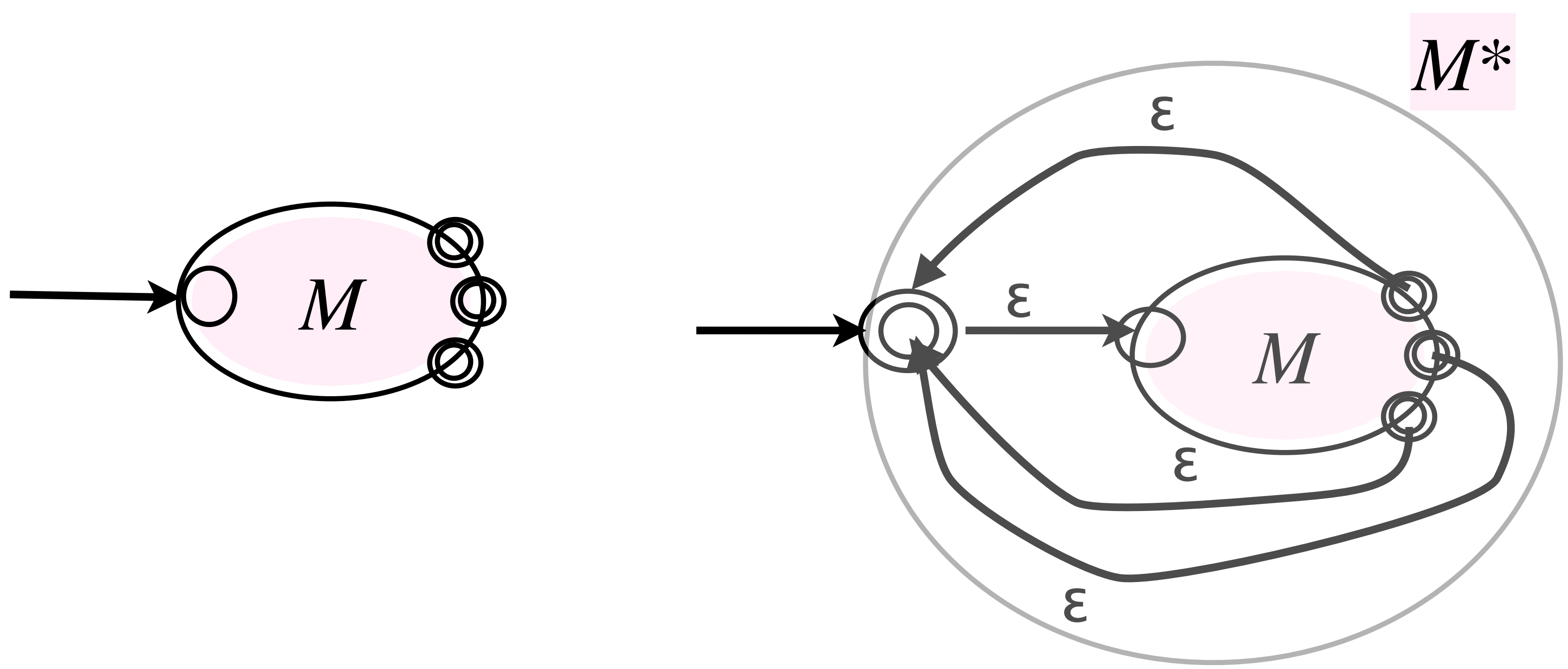
Concatenation with NFAs

- Suppose M_1 recognizes L_1 and M_2 recognizes L_2. Can easily construct M recognizing $L_1 \cdot L_2$:



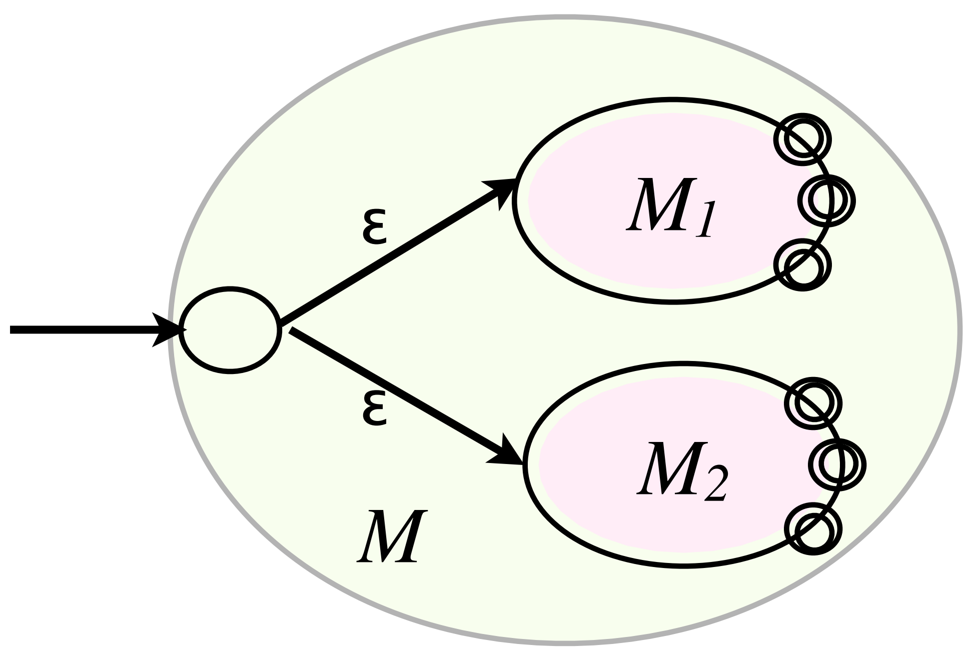
Building *-closures with NFAs

- Suppose M recognizes L. Can construct M* recognizing L*:



Building Unions with NFAs

- If M_1 recognizes L_1 and M_2 recognizes $L_{2,}$ can construct M recognizing $L_1 \cup L_2$.



Formal Definition

- A nondeterministic finite automaton (with ε-transitions) 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. $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \wp(Q)$ is the **transition function**,
  4. q_0 is the **start state**, and
  5. $F \subseteq Q$ is the set of **final states**

Portland State University

Formal Definition of NFA acceptance

- Let M = (Q, Σ, δ, q_0, F) be an NFA and let $w = a_1 a_2 \ldots a_n$ be a string, where each $a_i \in \Sigma \cup \{\varepsilon\}$
- M accepts w iff there is a sequence of states $r_0, r_1, r_2, \ldots r_n \in$ Q such that:

1. $r_0 = q_0$
2. $r_i \in \delta(r_{i-1}, a_i)$ for $i = 1, 2, \ldots n$
3. $r_n \in F$

Portland State University

Recap

- Two kinds of FA:
  - Deterministic: for each input, there is exactly one “next state”.
  - Nondeterministic:
    - for each input there may be zero, one or many “next states”.
    - NFA can also make ε-transitions at any time.
- Both are formally defined by a 5-tuple
  - the details of the transition function differ

- NFA can be used to define languages just like DFA:
  - If M is an NFA with alphabet Σ
    $L(M) = \{ w \in \Sigma^* \mid w \text{ is accepted by } M \}$
- Surprising fact: NFA and DFA are of equivalent power!
  - that is, for any NFA, there is an equivalent DFA, *i.e.*, a DFA that recognizes the same language
  - and vice versa, but that's obvious
    - why is it obvious?

Portland State University

Converting an NFA to a DFA

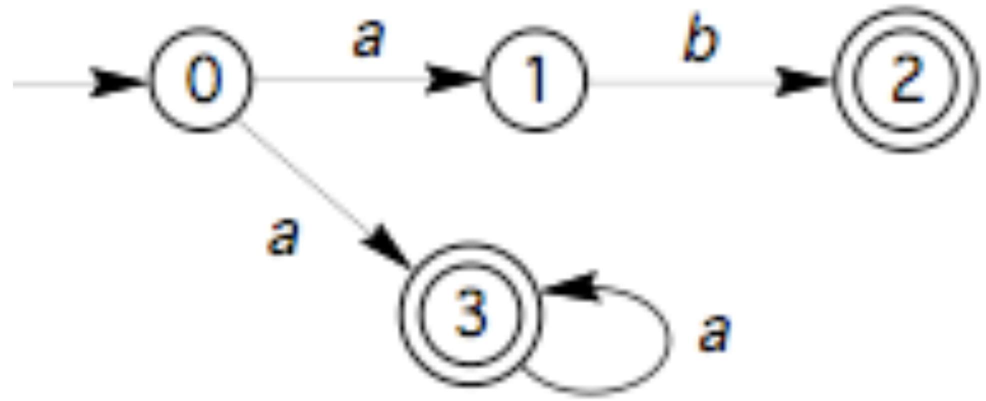
- When an NFA computes, at any time it is in a *set* of states
  - or, equivalently, there are a set of machines each of which is in one state;
  - the set grows and shrinks as the computation continues
- Key idea: build a DFA whose states represent *sets* of states in the NFA

Subset Construction (v1)

- Let N = $(Q, \Sigma, \delta, q_0, F)$ be an NFA **with no ε-transitions** that recognizes L.
- Then the DFA M = $(Q', \Sigma, \delta', q_0', F')$ recognizes L, where:
  - $Q' = \wp(Q)$, *i.e.*, the set of subsets of Q.
  - For each $R \in Q'$ and $a \in \Sigma$,
    $\delta'(R,a) = \{q \in Q \mid q \in \delta(r,a) \text{ for some } r \in R\}$
  - $q_0' = \{q_0\}$
  - $F' = \{R \in Q' \mid R \text{ contains } r \text{ for some } r \in F\}$

Portland State University

Subset construction example



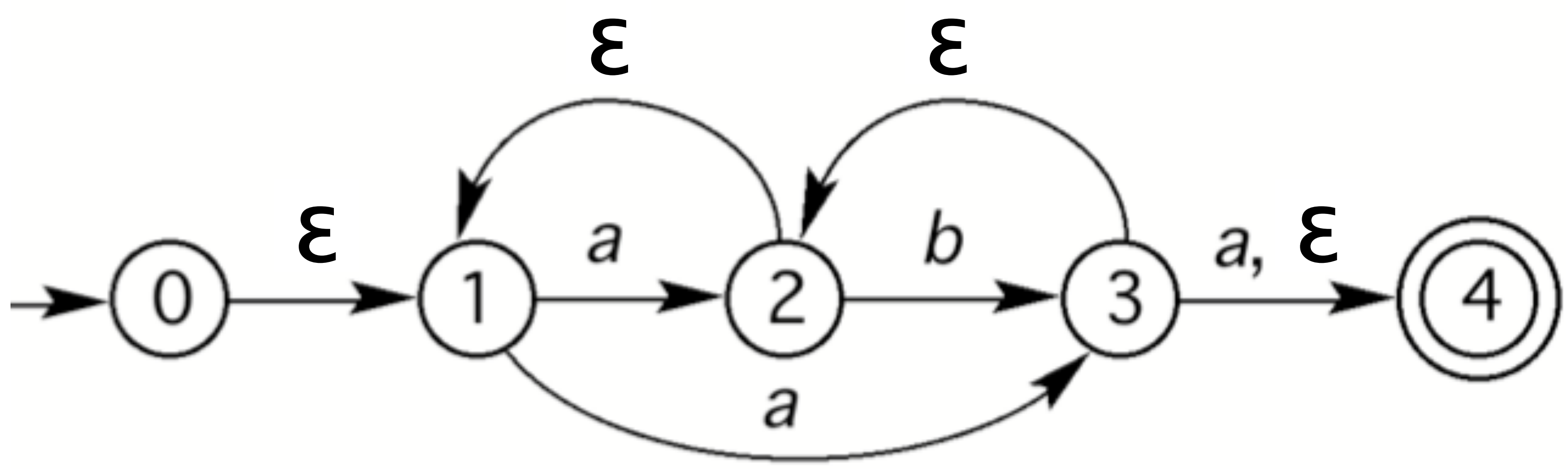
- In practice, construct only **reachable** states in Q' as they are discovered

Q' state	a	b
→{0}	{1,3}	{}
* {1,3}	{3}	{2}
* {3}	{3}	{}
* {2}	{}	{}
{}	{}	{}

ε–Closure

- Needed to handle subset construction for NFAs containing ε-transitions
- for all NFA states s, the ε closure of s, written $\varepsilon(s)$, is defined by:
  1. $s \in \varepsilon(s)$
  2. if $p \in \varepsilon(s)$, and $\delta(p, \varepsilon) = q$, then $q \in \varepsilon(s)$
- $\varepsilon(s)$ is the set of states reachable from s by traveling along 0 or more ε-edges

ε–Closure Computation Example



	T_N	a	b	ε
start	0	∅	∅	{1}
	1	{2, 3}	∅	∅
	2	∅	{3}	{1}
	3	{4}	∅	{2, 4}
final	4	∅	∅	∅

ε (0) =

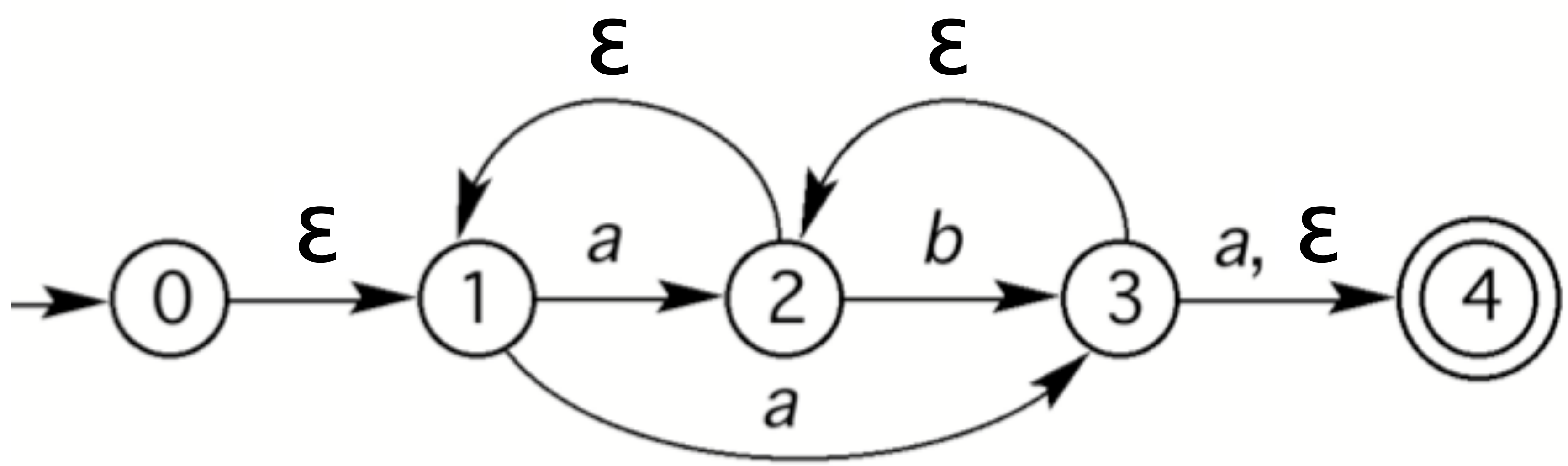
ε (1) =

ε (2) =

ε (3) =

ε (4) =

ε–Closure Computation Example



	T_N	a	b	ε
start	0	∅	∅	{1}
	1	{2, 3}	∅	∅
	2	∅	{3}	{1}
	3	{4}	∅	{2, 4}
final	4	∅	∅	∅

$$\begin{aligned}
\varepsilon(0) &= \{0, 1\},\\
\varepsilon(1) &= \{1\},\\
\varepsilon(2) &= \{1, 2\},\\
\varepsilon(3) &= \{1, 2, 3, 4\},\\
\varepsilon(4) &= \{4\}.
\end{aligned}$$

Subset Construction (v2)

- Let $N = (Q, \Sigma, \delta, q_0, F)$ be an arbitrary NFA that recognizes L

- Then the DFA $M = (Q', \Sigma, \delta', q_0', F')$ recognizes L, where:

  - $Q' = \wp(Q)$, i.e., the set of subsets of Q.

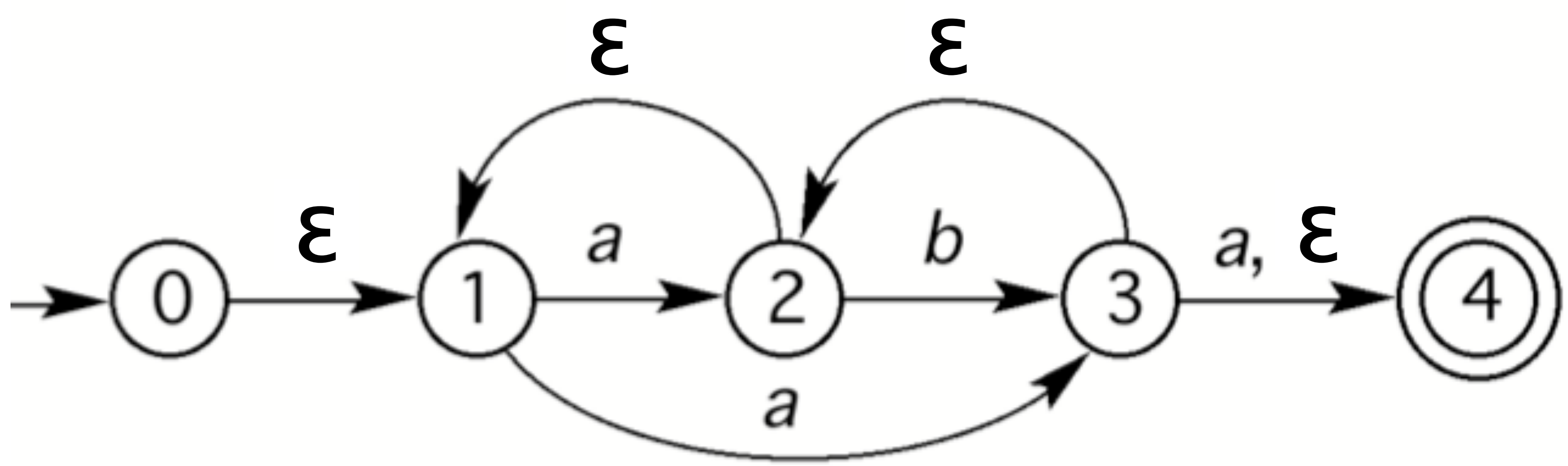
  - For each $R \in Q'$ and $a \in \Sigma$,
    $\delta'(R,a) = \{q \mid q \in \varepsilon(s)$ for some $s \in \delta(r,a)$ for some $r \in R\}$

  - $q_0' = \varepsilon(q_0)$

  - $F' = \{R \in Q' \mid R$ contains r for some $r \in F\}$

Portland State UNIVERSITY

Subset construction example



- ε(0) = {0,1} ε(1) = {1} ε(2) = {1,2}
  ε(3) = {1,2,3,4} ε(4) = {4}

Q' state	a	b
→{0,1}	{1,2,3,4}	{}
* {1,2,3,4}	{1,2,3,4}	{1,2,3,4}
{}	{}	{}

Table-driven NFA simulator

```
/* TABLE-DRIVEN NFA SIMULATOR */

#include "stdio.h"

/* Machine-specific data follows. It must be
adjusted for each different NFA to be simulated. */

/* Here we specify the NFA for language L2a.L2b */

/* number of states (must be <= 32 !!) */
#define STATES 4

/* number of symbols */
#define SYMBOLS 2

/* convert ASCII character to symbol number
   0,1,2,...,SYMBOLS-1 */
#define SYMBOL_OF_CHAR(c) (c-'a')

/* these are just defined to increase legibility in
the remainder of the machine description */
#define Sevena 0
#define Sodda 1
#define Sevenb 2
#define Soddb 3
```

```
/* Sets of states are represented by 32-bit bit
vectors. */

/* Convert a state number to a bit position. */
#define B(s) (1<<s)

/* Test whether a state is in a state set. */
#define in(set,s) ((set & B(s))!= 0)

int initial_state = Sevena;

int next_states[STATES][SYMBOLS] =
  { /* from Sevena */ {B(Sodda),B(Sevena)},
    /* from Sodda */  {B(Sevena),B(Sodda)},
    /* from Sevenb */ {B(Sevenb),B(Soddb)},
    /* from Soddb */  {B(Soddb),B(Sevenb)} };

/* Transitions from each state on epsilon. */
int eps_transitions[STATES] =
  { /* from Sevena */ B(Sevenb),
    /* from Sodda *   0,
    /* from Sevenb */ 0,
    /* from Soddb */  0 };

/* Set of accepting states */
int accepting_states = B(Sevena) | B(Sevenb);
```

Representing sets by bitmaps

- We represent a set of states by a bitmap
  - bit i is 1 ⇔ state i is in the set
  - bitmap fits in a word if # of states ≤ 32
  - B(i) = 1 << i   converts state # to bit position
- Common set operations:
  - member(set,state): set & B(state) != 0
  - add(set,state): set | B(state)
  - union(set1,set2): set1 | set2
  - is_empty(set): set == 0

Simulator: driver code

```
/* The simulation code is identical for
   every NFA */


int eps_closure(int states) {
  int i;
  int old_states;
  do {
    old_states = states;
    for (i = 0; i < STATES; i++)
      if (in(states,i))
         states |= eps_transitions[i];
  } while (states != old_states);
  return states;
}
```

```
int main (int argc, char **argv) {
  int i;
  char *input = *++argv;

  int current_states =
        eps_closure (B(initial_state));
  char c;
  while (c = *input++) {
    int symbol = SYMBOL_OF_CHAR(c);
    if (symbol >=0 && symbol < SYMBOLS) {
      int new_states = 0;
      for (i = 0; i < STATES; i++)
        if (in(current_states,i))
          new_states |= next_states[i][symbol];
      current_states = eps_closure(new_states);
    } else {
      printf("invalid symbol in input\n");
      return 1;
    }
  }
  if ((current_states & accepting_states) != 0)
    printf("accept\n");
  else
    printf("reject\n");
  return 0;
}
```