Lecture 4
Nondeterministic Finite Accepters

COT 4420

Theory of Computation

Section 2.2, 2.3

Nondeterminism

- A nondeterministic finite automaton can go to several states at once.

- Transitions from one state on an input symbol can be to a SET of states.

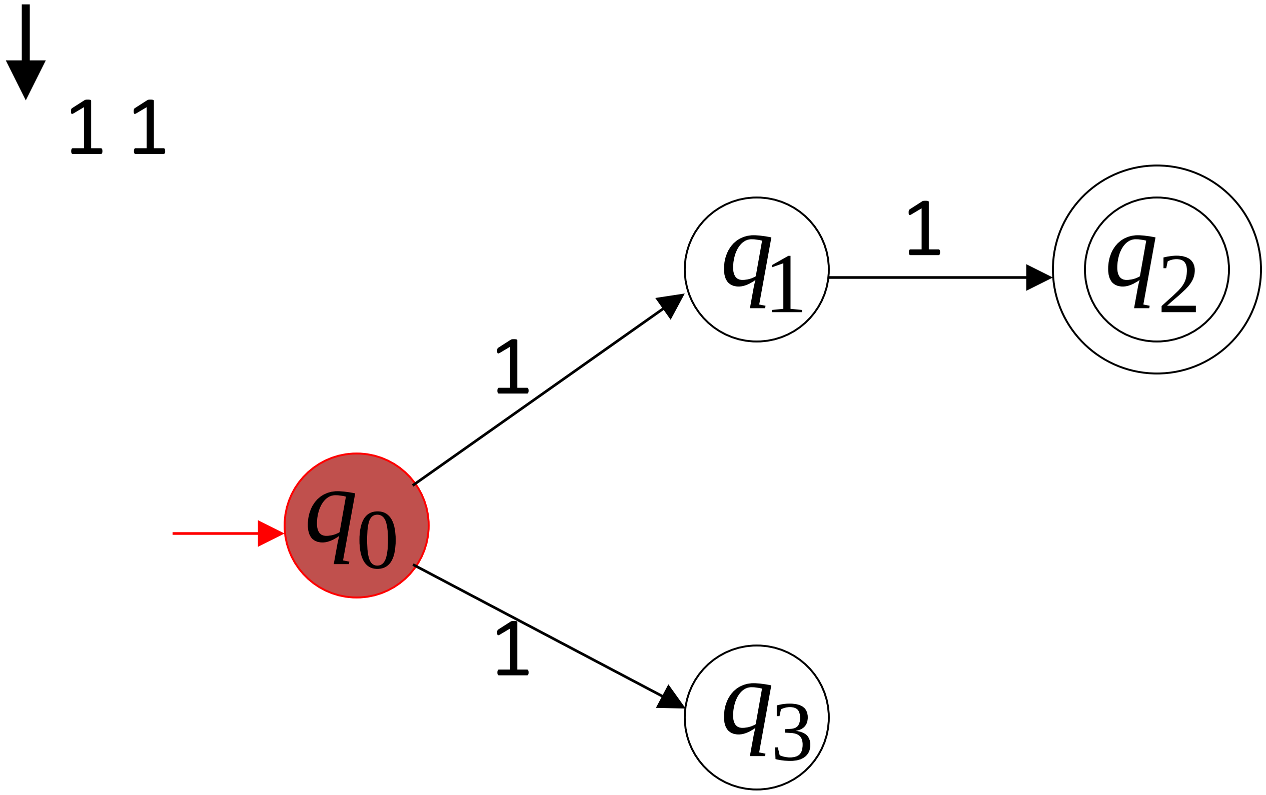
Nondeterministic Finite Accepter

- The main difference with DFA is that

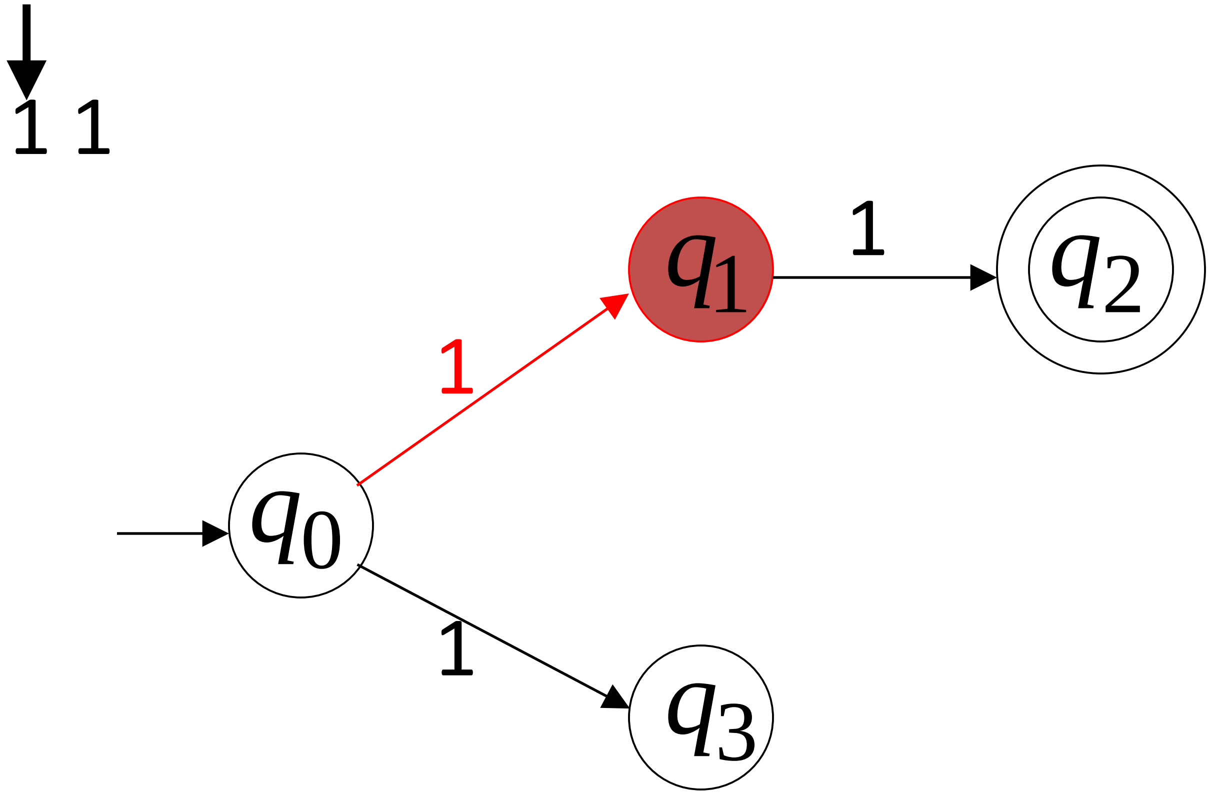
1. From one state with an input symbol there might be more than one choice in the transition function.

2. From a state there might be no transition with an input symbol (The transition function is not total). In that case the automaton halts.

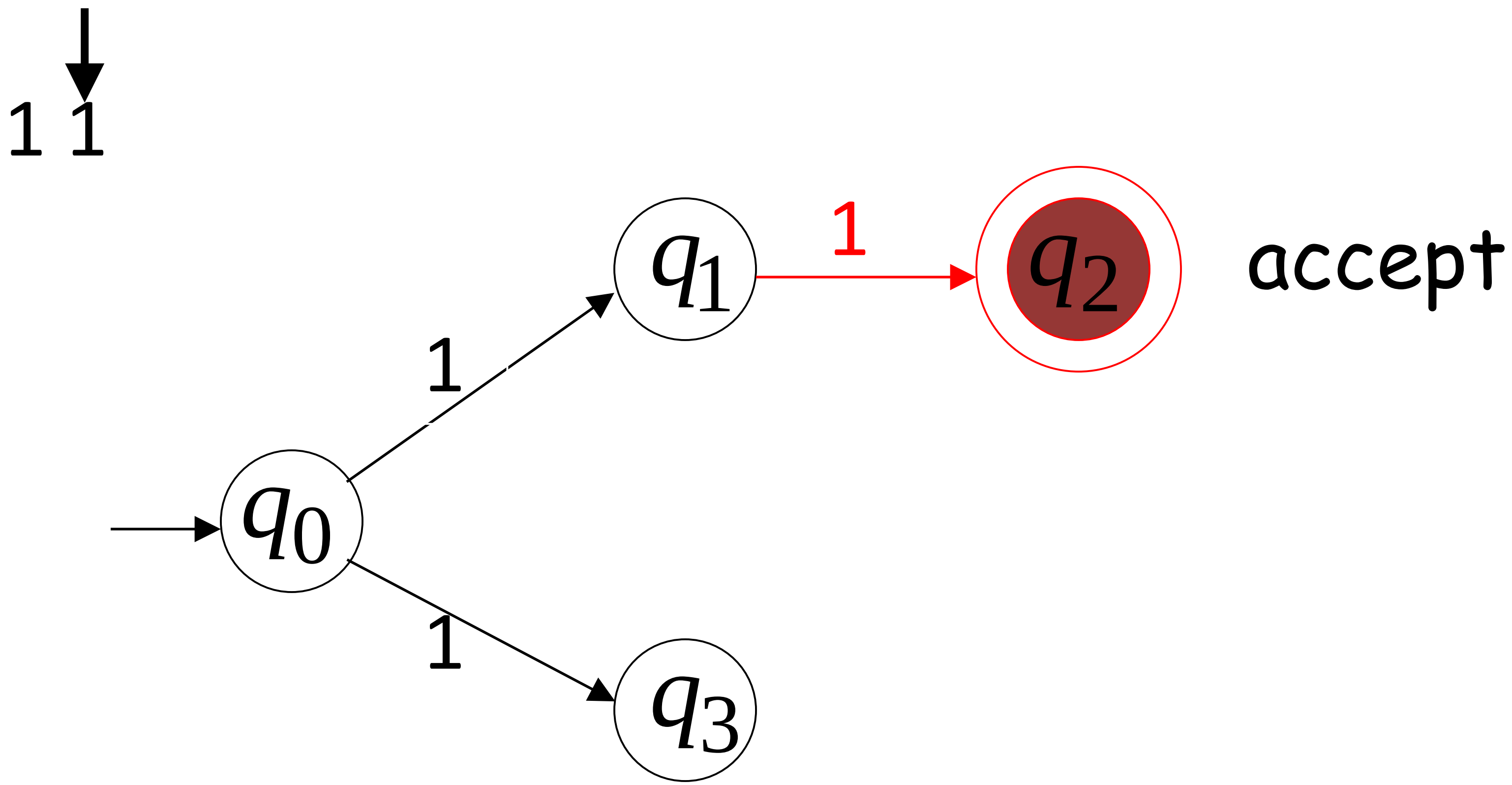
First Choice



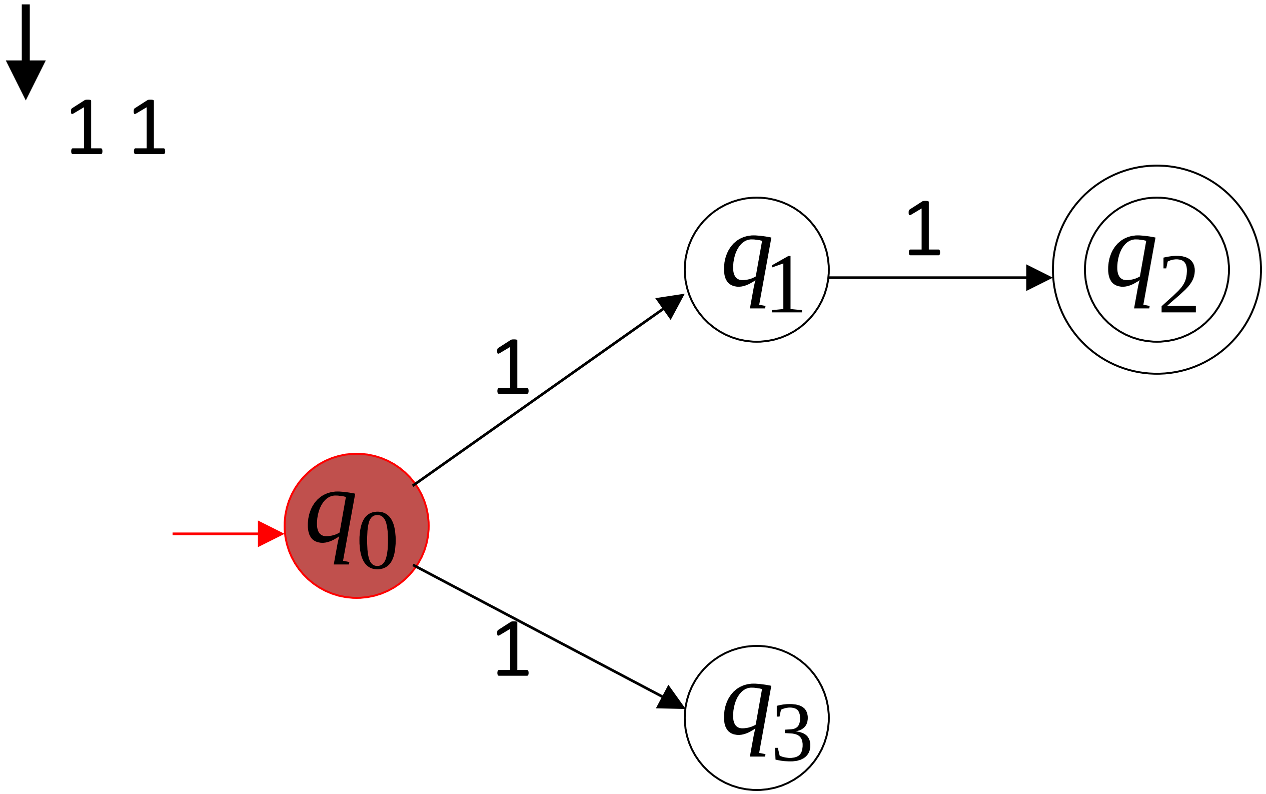
First Choice



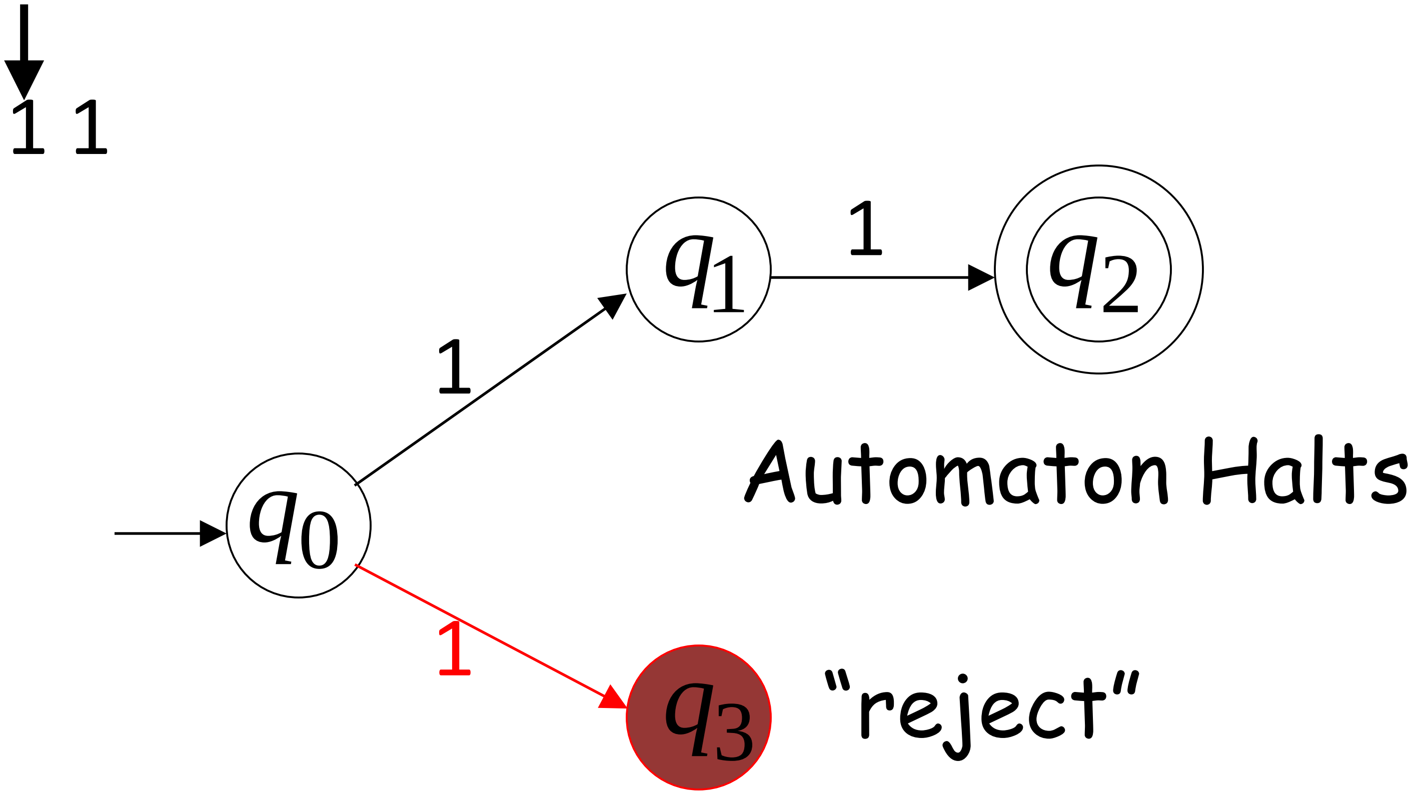
First Choice



Second Choice



Second Choice



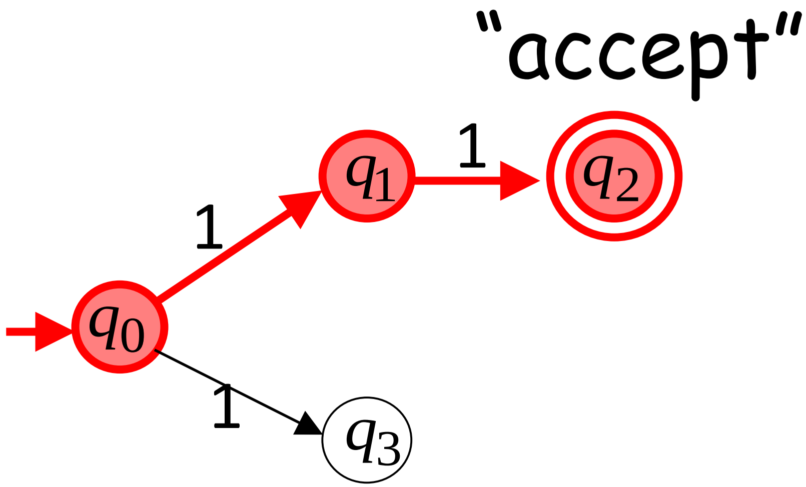
Accepting a String

- An NFA accepts a string

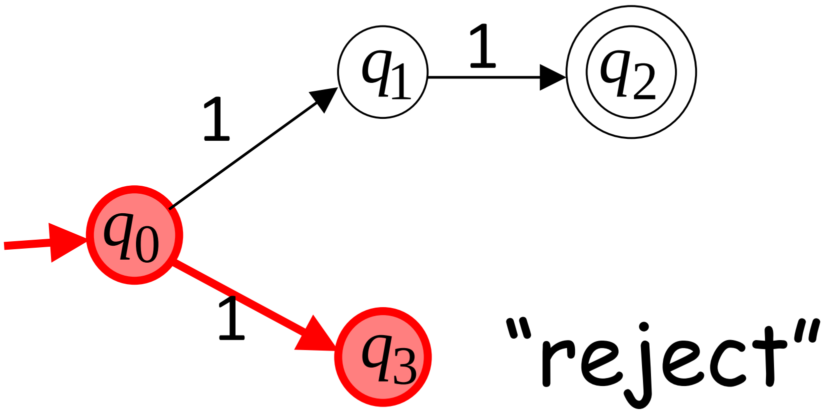
  when there is a computation of the NFA that accepts the string

❖ All the input is consumed and the automaton is in a final state

11 is accepted by the NFA:

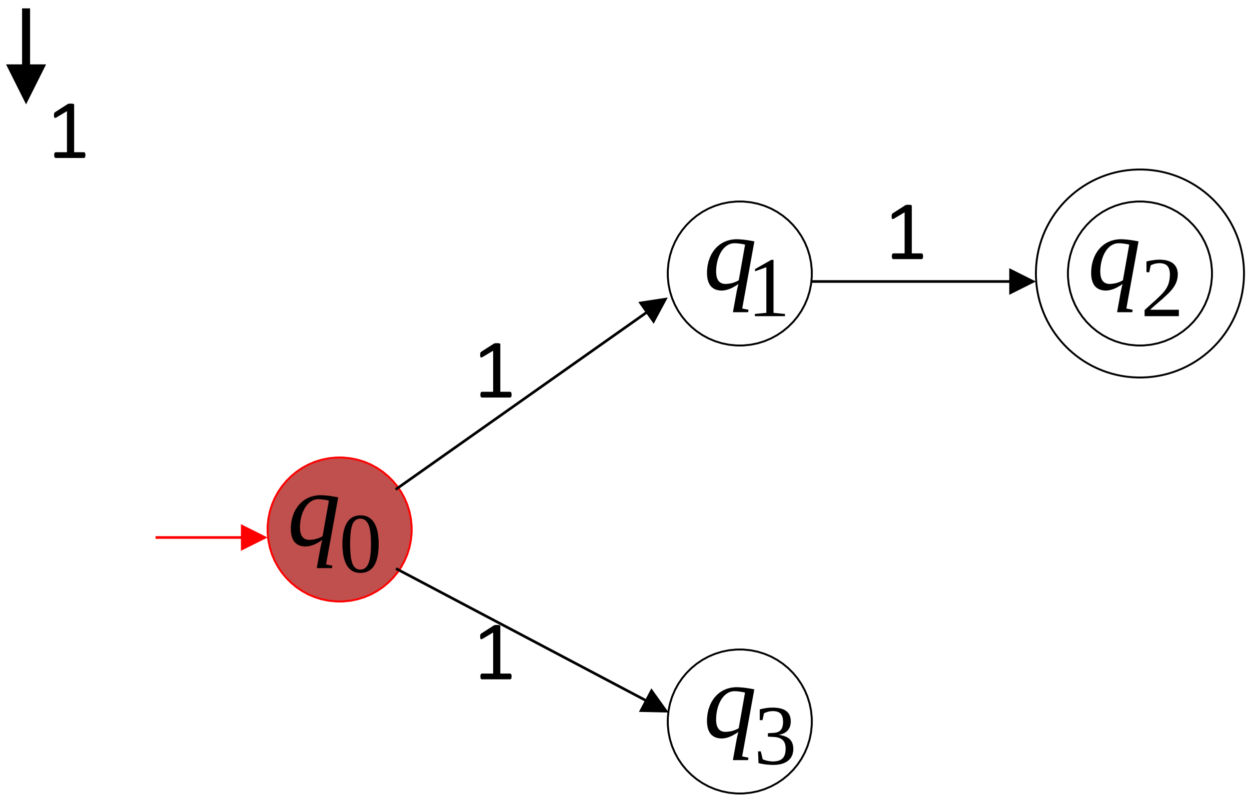


because this
Computation
accepts 11

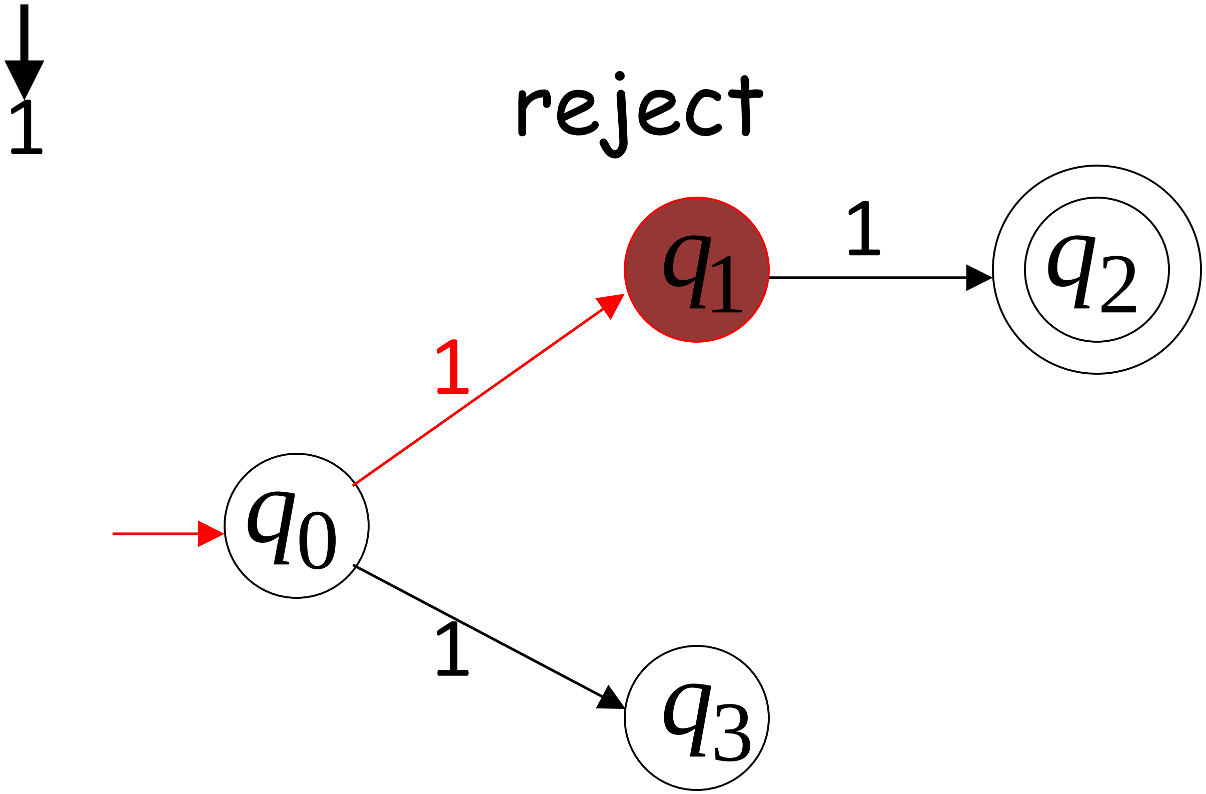


this computation
is ignored

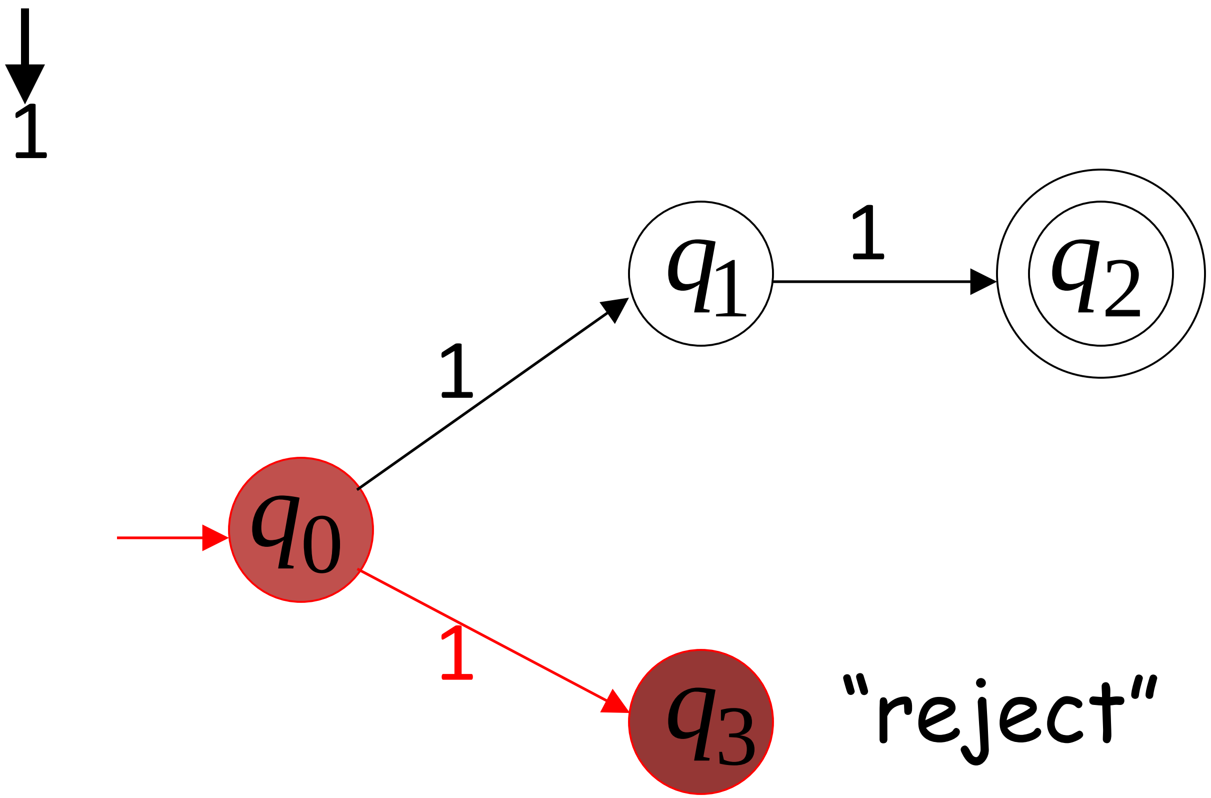
Rejection example



Rejection example
First choice



Rejection example
Second choice



An NFA rejects a string:

If there is no computation of the NFA that accepts the string.

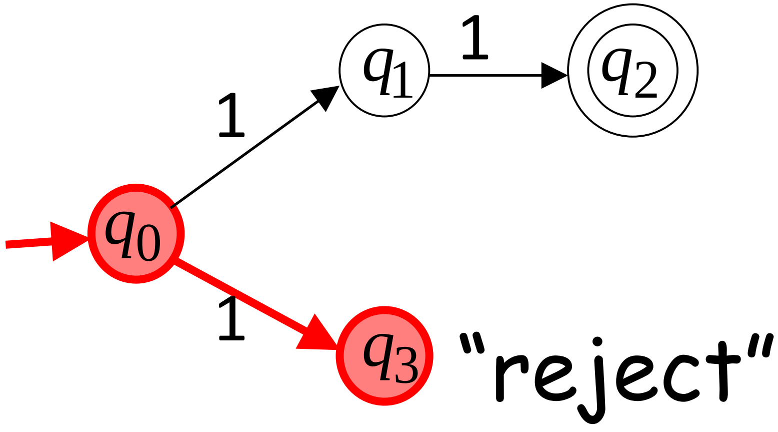
Either:

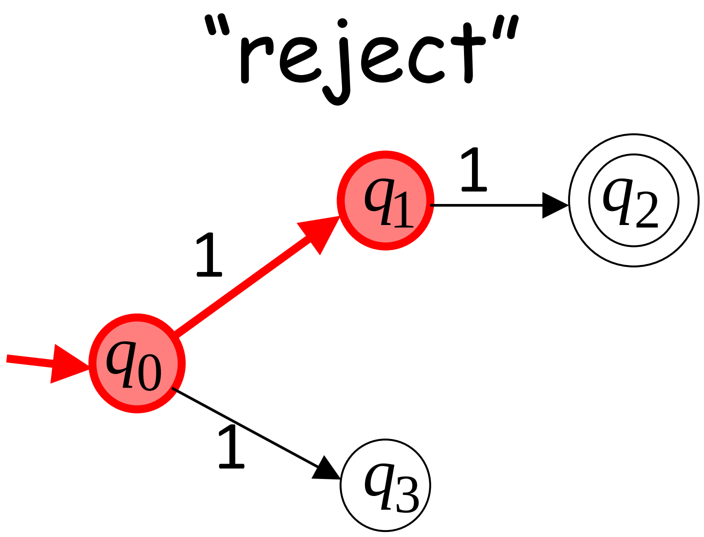
- All the input is consumed and NFA is in a non accepting state

OR

- The input cannot be consumed

1 is rejected by the NFA:





All possible computations lead to rejection

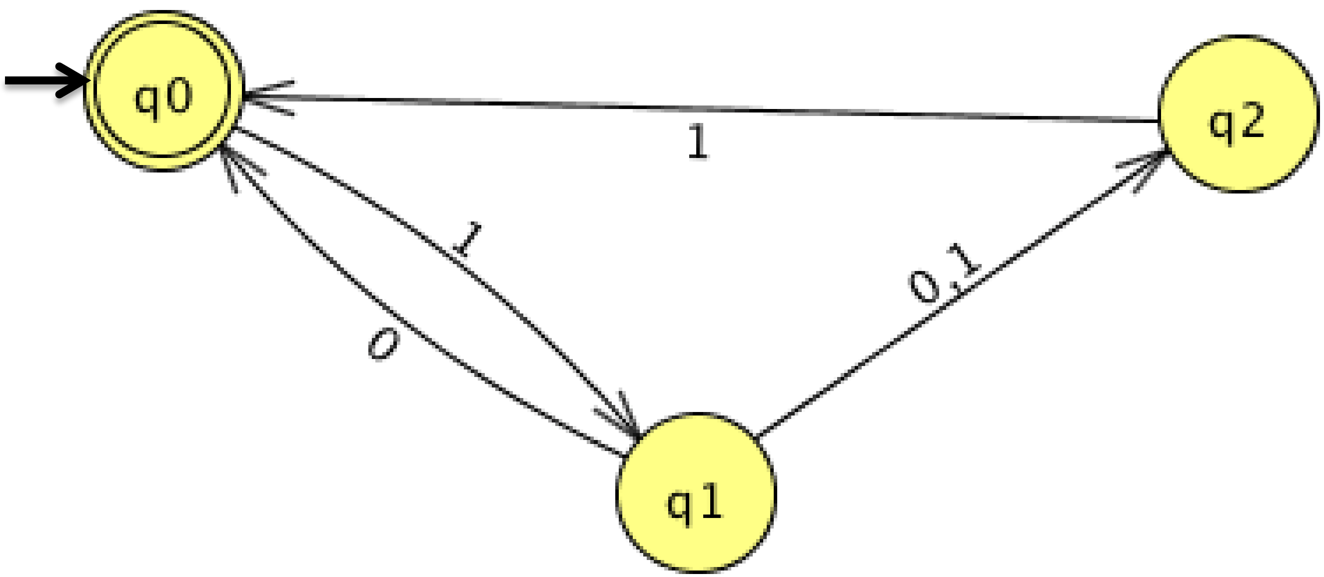
Nondeterministic Finite Accepter (NFA)

- We have one start state. Starting from start state, an input is accepted if any sequence of choices leads to some final state.

1010 ?

110 ?

10 ?



Nondeterministic Finite Accepter (NFA)

- A nondeterministic finite accepter is defined by 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

where Q, Σ, q_0, and F are defined as DFA, but

$$\delta: Q \times \Sigma \rightarrow 2^Q$$

Extended Transition Function

δ^* is defined recursively by:

$\delta^*(q, \lambda) = \{q\}$

Let S be $\delta^*(q, w)$ then:

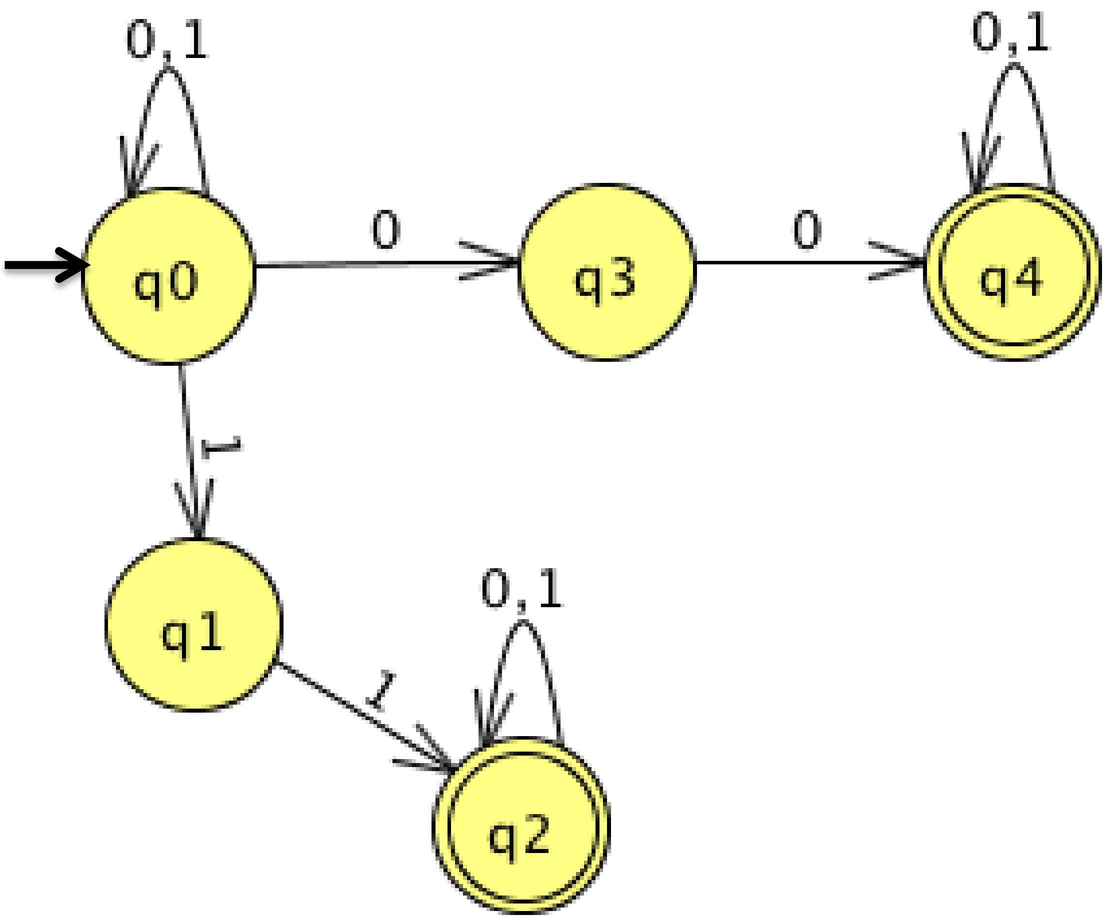
$$\delta^*(q, wa) = \bigcup_{p \in S} \delta(p, a)$$

Language of an NFA

- The language of an nfa M is defined as the set of all strings accepted by M.

$$L(M) = \{ w \in \Sigma^*: \delta^*(q_0, w) \cap F \neq \varnothing \}$$

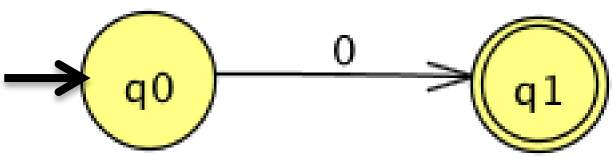
NFA - Example



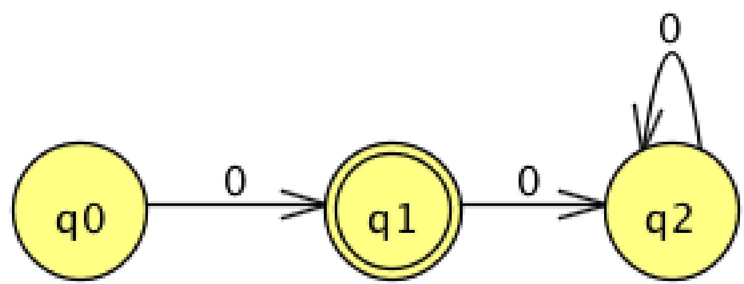
$\delta(q_0,0)=\{q_0,q_3\}$
$\delta(q_0,1)=\{q_0,q_1\}$
$\delta(q_1,0)=\emptyset$
$\delta(q_1,1)=\{q_2\}$
$\delta(q_2,0)=\{q_2\}$
$\delta(q_2,1)=\{q_2\}$
$\delta(q_3,0)=\{q_4\}$
$\delta(q_3,1)=\emptyset$
$\delta(q_4,0)=\{q_4\}$
$\delta(q_4,1)=\{q_4\}$

- It is easier to express languages with NFAs than with DFAs

NFA M_1



DFA M_2



$L(M_1) = L(M_2) = \{0\}$

NFA's and DFA's

- Is NFA more powerful than DFA?

- We can show that the classes of DFA's and NFA's are equally powerful.

What does equivalence mean?

- Two finite accepters M_1 and M_2 are said to be equivalent if they both accept the same language,

$$L(M_1) = L(M_2)$$

Equivalence of NFA's and DFA's

The set of languages accepted by NFAs = The set of languages accepted by DFAs OR Regular languages

Step1) The set of languages accepted by DFAs is a subset of the set of languages accepted by NFAs.

This is trivially true since every DFA is an NFA.

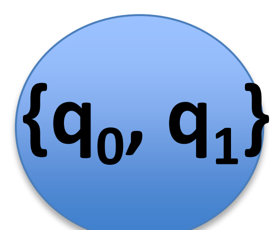
Equivalence of NFA's and DFA's

Step2) The set of languages accepted by NFAs is a subset of the set of languages accepted by DFAs.

- For any NFA there is a DFA that accepts the same language.

Equivalence of DFA’s and NFA’s

- After an NFA reads a string *w*, we know that it must be in one state of a possible set of states, e.g. $\{q_i, q_j, \ldots, q_k\}$
- In the equivalent DFA after reading *w* we will be in a state labeled $\{q_i, q_j, \ldots, q_k\}$
  - The name of the states in our DFA will be sets of states!



Equivalence of DFA's and NFA's

- If our NFA has |Q| states, the equivalent DFA will have $2^{|Q|}$ states.

Theorem: Let L be the language accepted by NFA $M_N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Then there exists a DFA $M_D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $L = L(M_D)$.

NFA to DFA

1. Our NFA has a start symbol q_0. The start state of DFA will be $\{q_0\}$

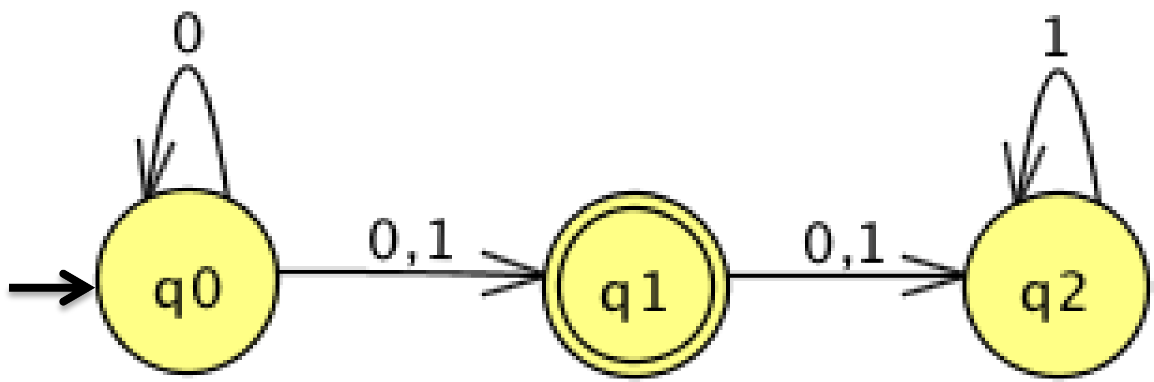
2. Repeat these steps until no more edges are missing:
   - For every DFA state $\{q_i, q_j, \ldots q_k\}$ that has no outgoing edge for some $a \in \Sigma$
   - $\delta_N(q_i, a) \cup \delta_N(q_j, a) \ldots \cup \delta_N(q_k, a) = \{q_l, \ldots q_n\}$
   - Create a vertex labeled $\{q_l, \ldots q_n\}$ if it does not exist
   - Add an edge from $\{q_i, q_j, \ldots q_k\}$ to $\{q_l, \ldots q_n\}$ with label a

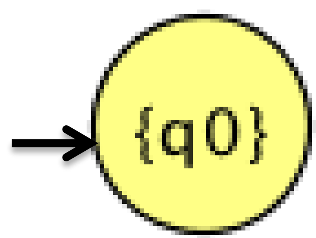
NFA to DFA

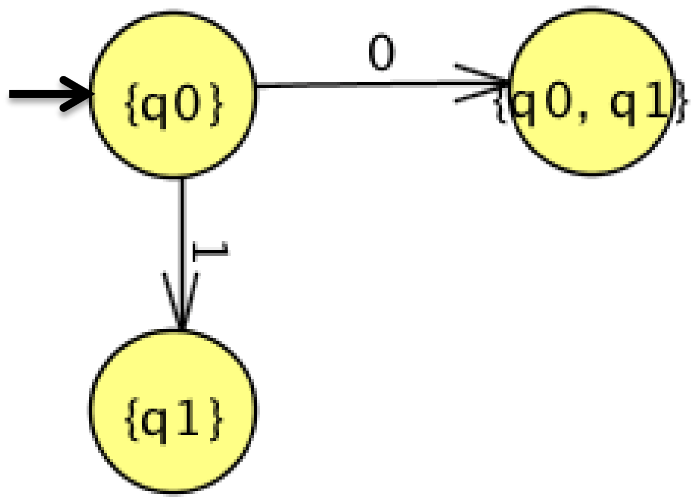
3. Every state of DFA whose label contains a final state from NFA is identified as a final state.

NFA to DFA Example

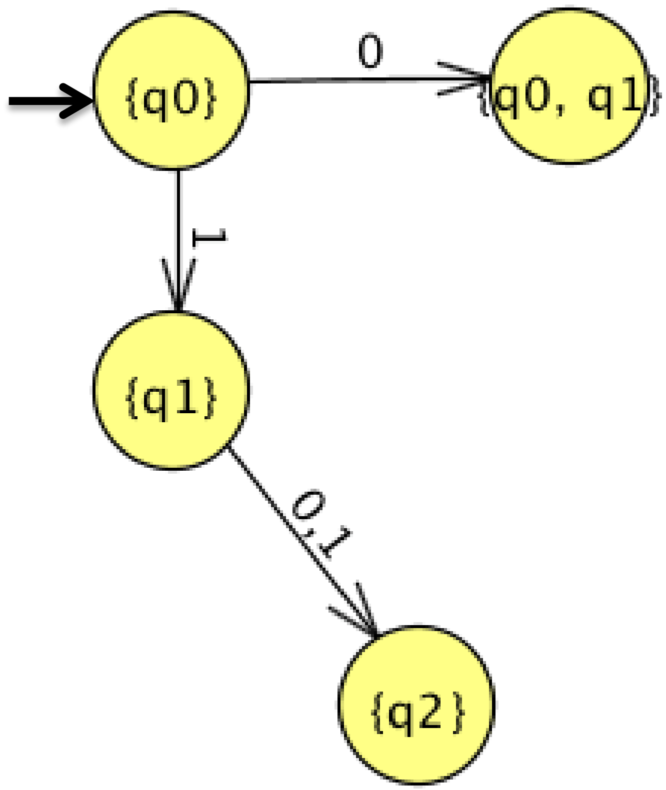
NFA:

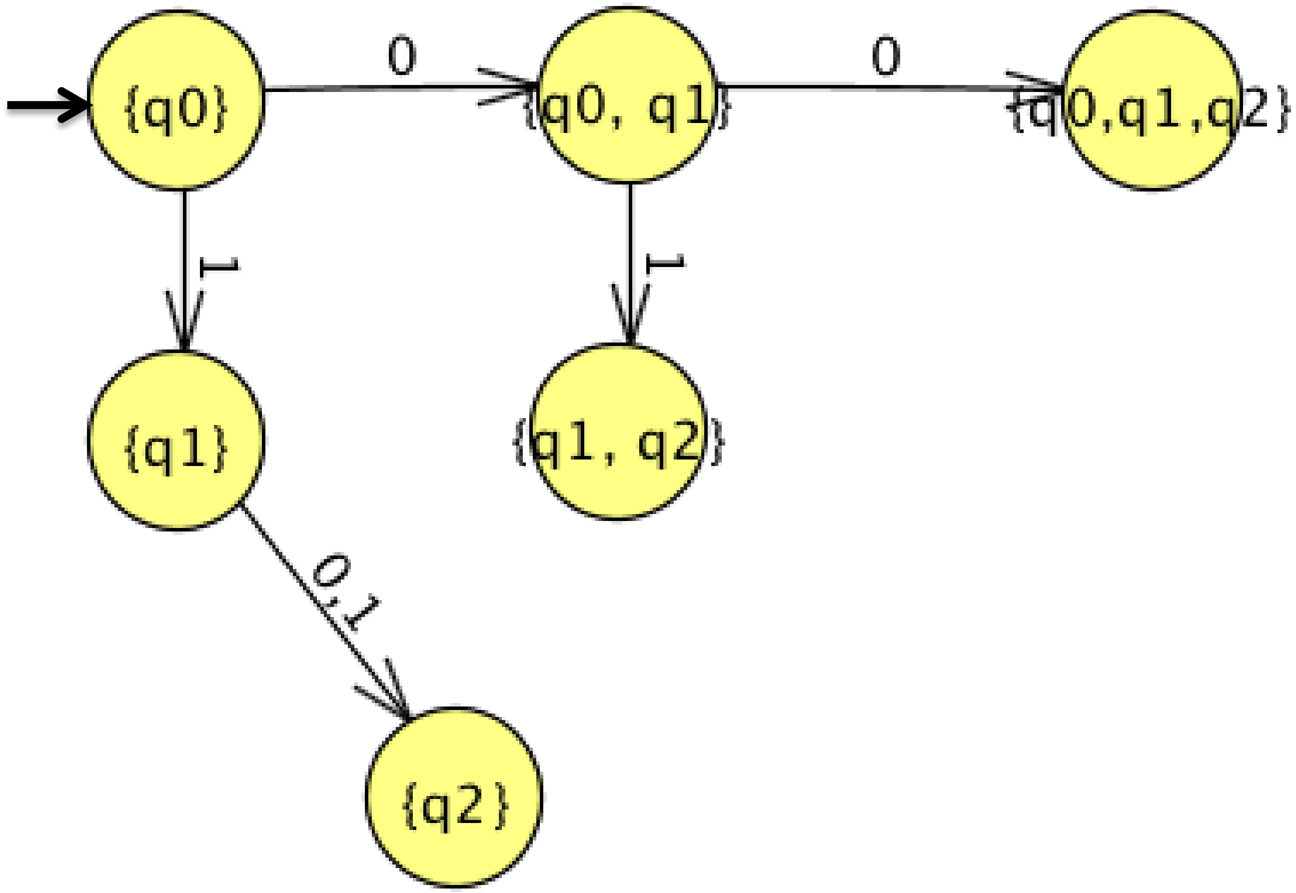


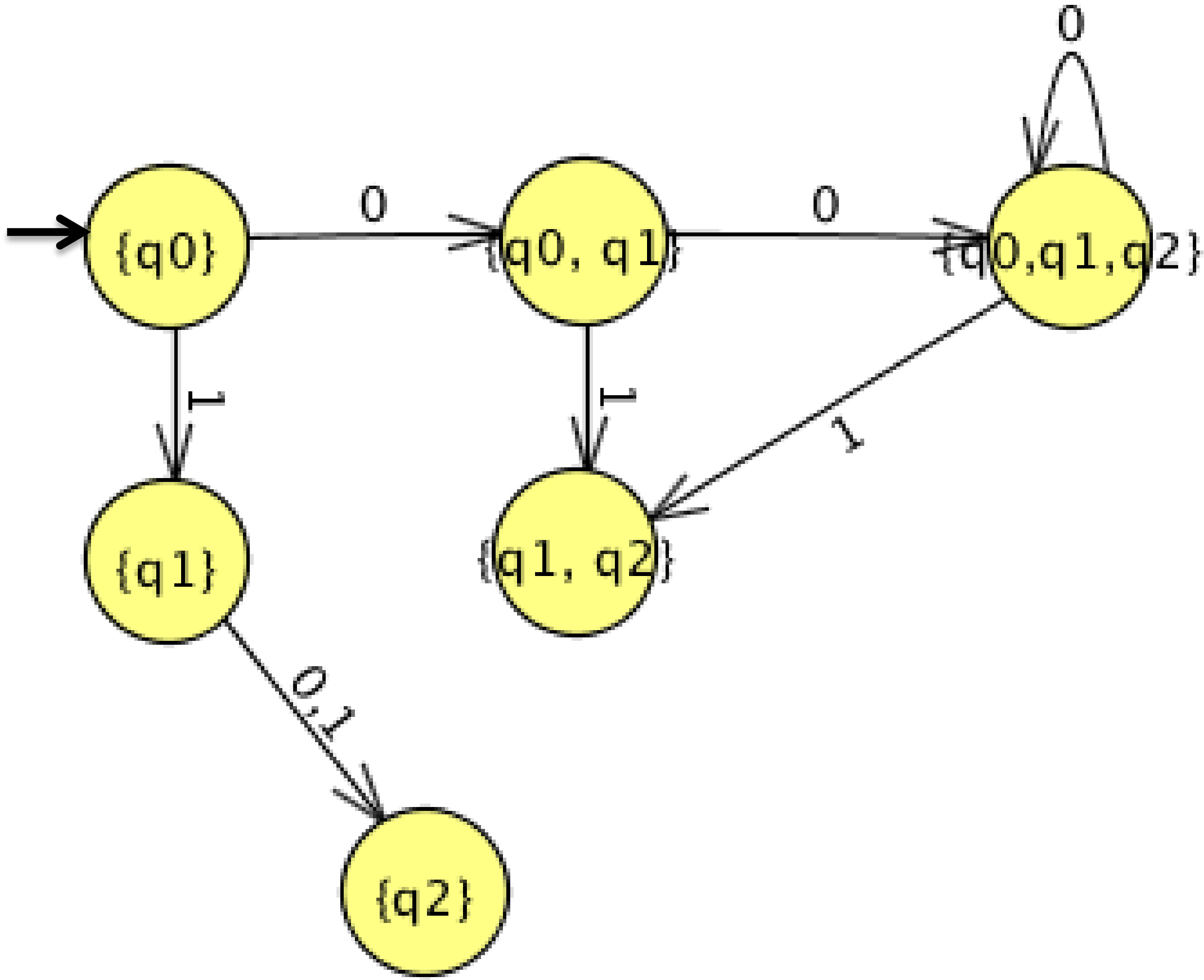


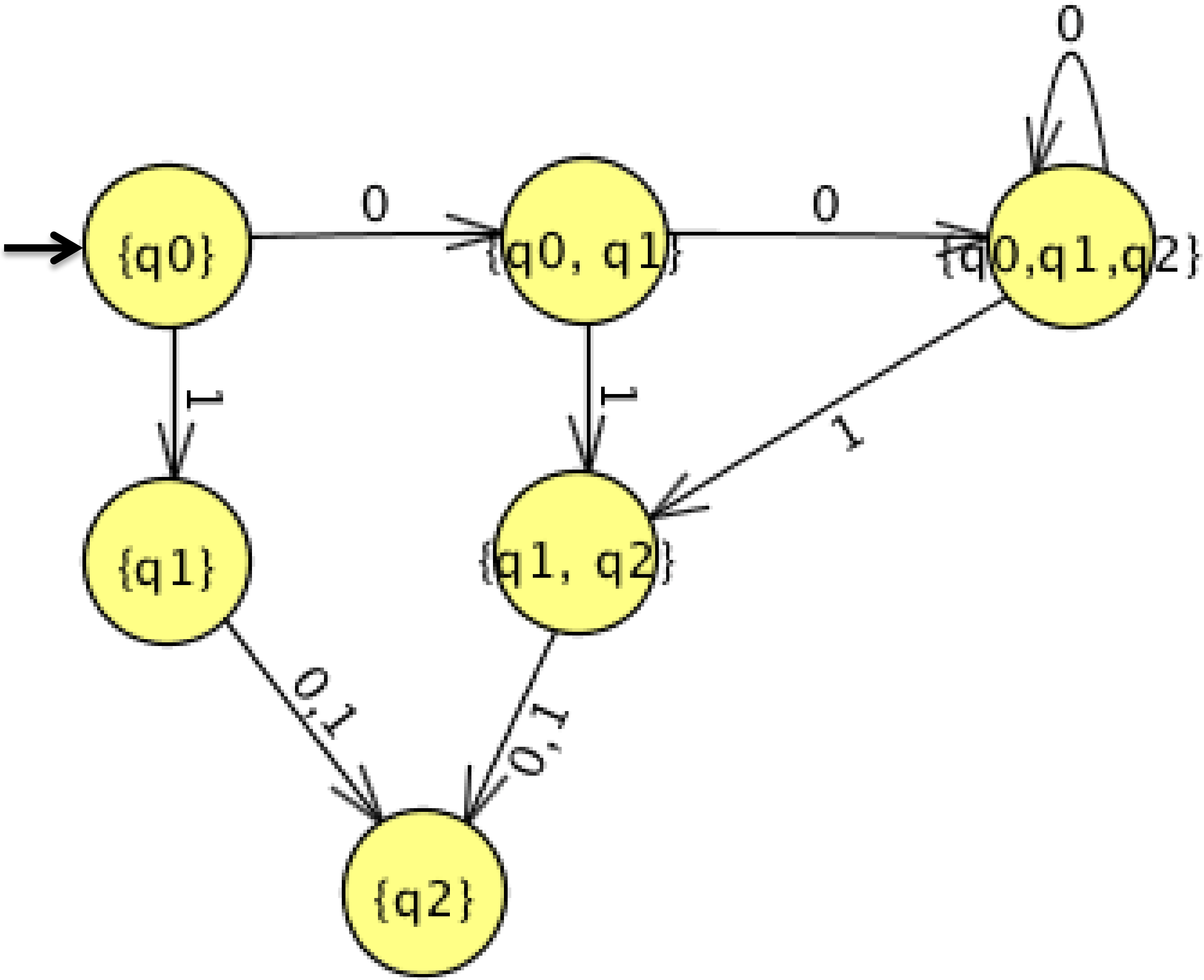


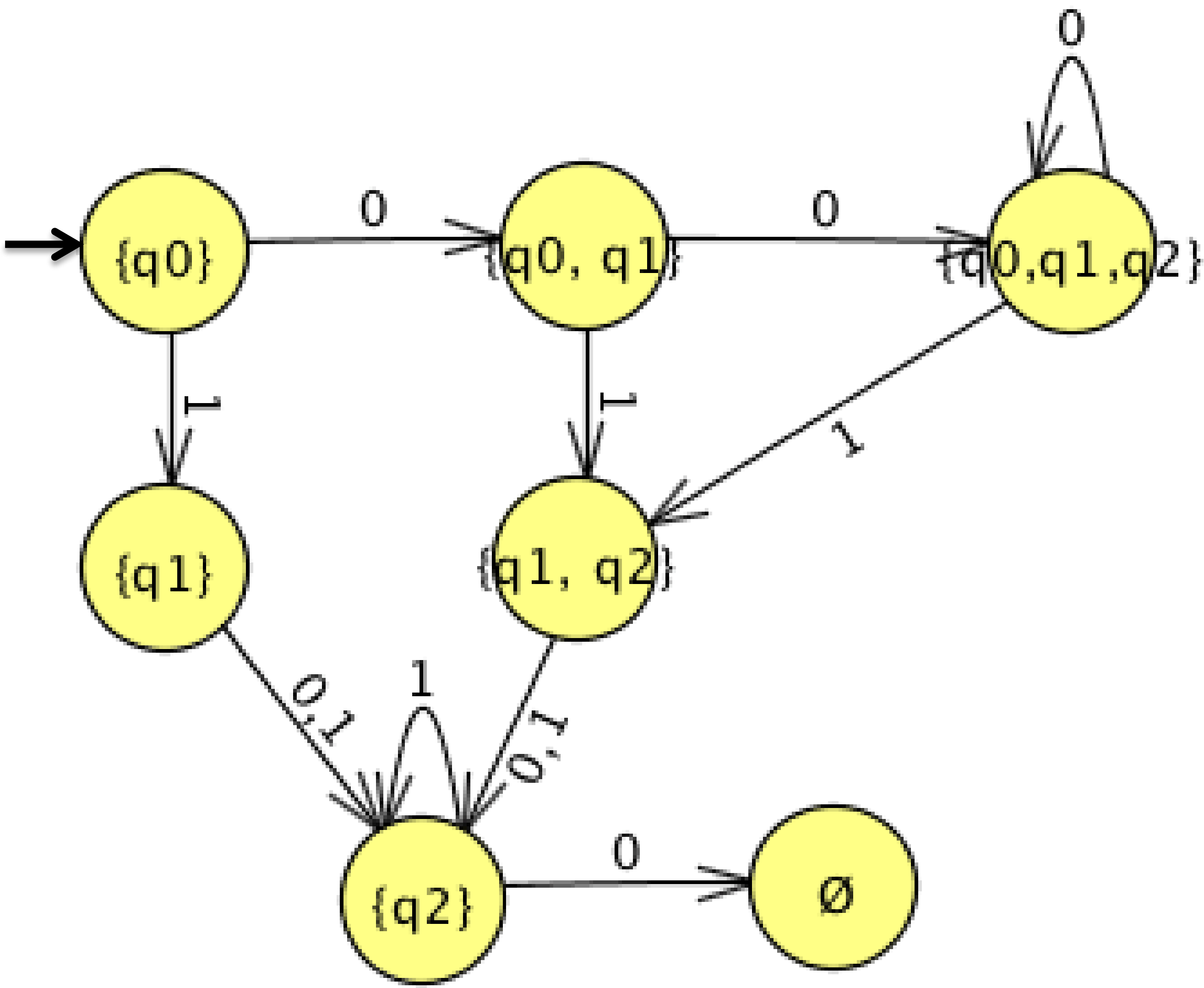
{q0}
0
{q0, q1}
1
{q1}


{q0}
0
{q0, q1}
1
{q1}
0,1
{q2}


{q0}
0
{q0, q1}
0
{q0,q1,q2}
1
1
{q1}
{q1, q2}
0,1
{q2}


0
{q0}
0
{q0, q1}
0
{q0,q1,q2}
1
1
1
{q1}
{q1, q2}
0,1
{q2}


{q0}
{q0, q1}
{q0,q1,q2}
{q1}
{q1, q2}
{q2}
0
0
0
1
1
1
0,1
0,1


{q0}
0
{q0, q1}
0
{q0,q1,q2}
0
1
1
1
{q1}
{q1, q2}
0,1
1
0,1
{q2}
0
Ø

Proof of Equivalence

Theorem: Let M_N be an NFA and M_D be an equivalent DFA obtained by the procedure. Then

$$L(M_N) = L(M_D)$$

We need to show that

if $w \in L(M_N)$ $\quad \Longrightarrow \quad$ $w \in L(M_D)$

Proof of Equivalence by Induction

- Show by induction on |w| that

$$\delta_N(q_0, w) = \delta_D(\{q_0\}, w)$$

Basis: $|w|=0 \rightarrow w = \lambda$

$$\delta_N(q_0, \lambda) = \delta_D(\{q_0\}, \lambda) = \{q_0\}$$

Proof of Equivalence by Induction

- Inductive step: Assume it is true for strings shorter than w. let w = va. So the induction hypothesis is true for v (v is shorter than w).
- Let $\delta_N(q_0, v) = \delta_D(\{q_0\}, v) = S$.
- The extended rule for NFA:

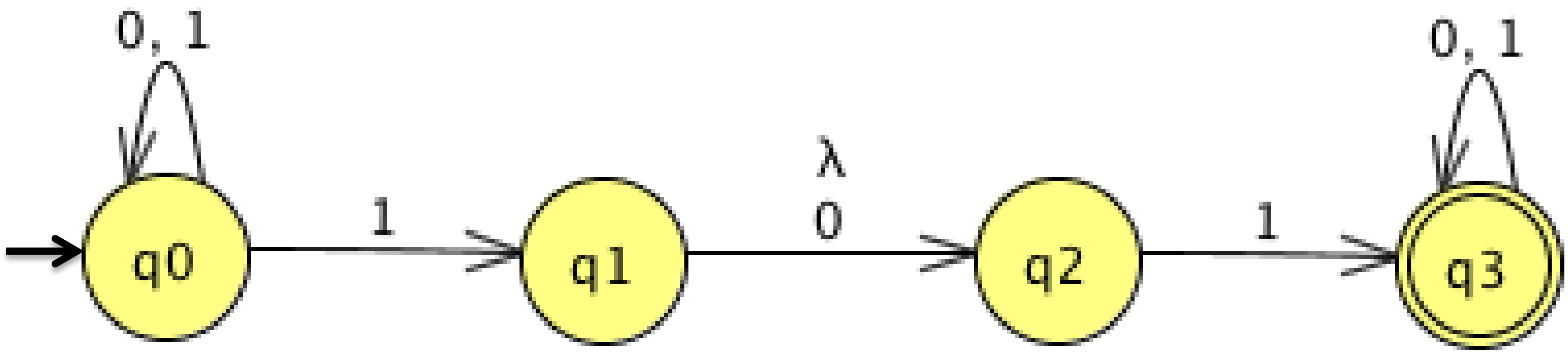
$\delta_N(q_0, w) = \delta_N(q_0, va) =$ T = the union over all states p in S of $\delta_N(p, a)$

- By the procedure we discussed we also know that $\delta_D(\{q_0\}, va)$ is the same set T.
- Therefore $\delta_N(q_0, w) = \delta_D(\{q_0\}, w) = T$.

NFA's with ε transitions

- We can allow state to state transitions on ε input.
- It does not consume the input string.
- Is ε-NFA more powerful than NFA ?

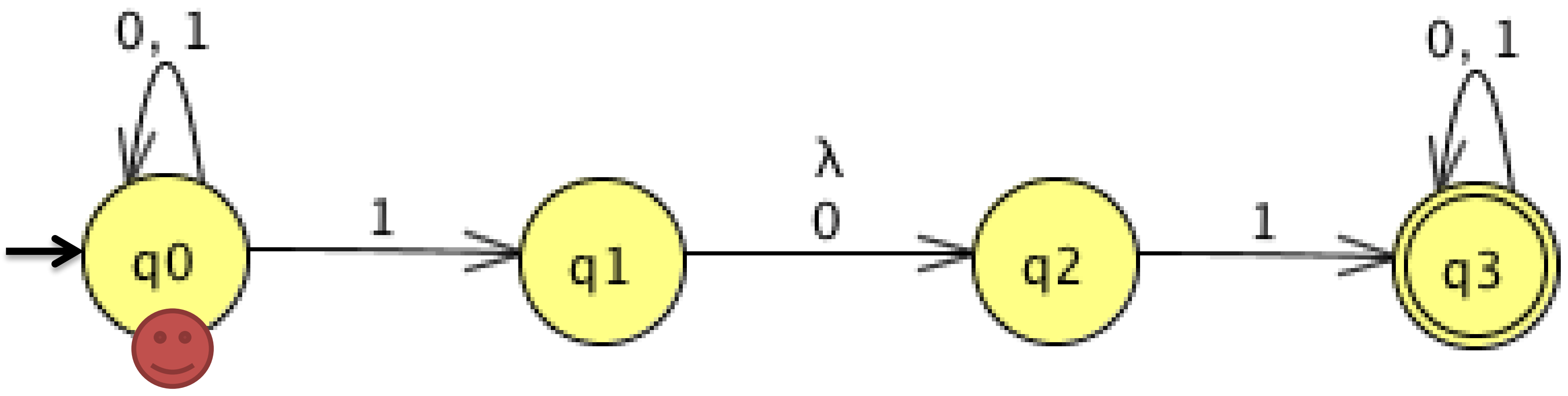
NFA Example



input / state	0	1	λ
q_0	$\{q_0\}$	$\{q_0,q_1\}$	Ø
q_1	$\{q_2\}$	Ø	$\{q_2\}$
q_2	Ø	$\{q_3\}$	Ø
q_3	$\{q_3\}$	$\{q_3\}$	Ø

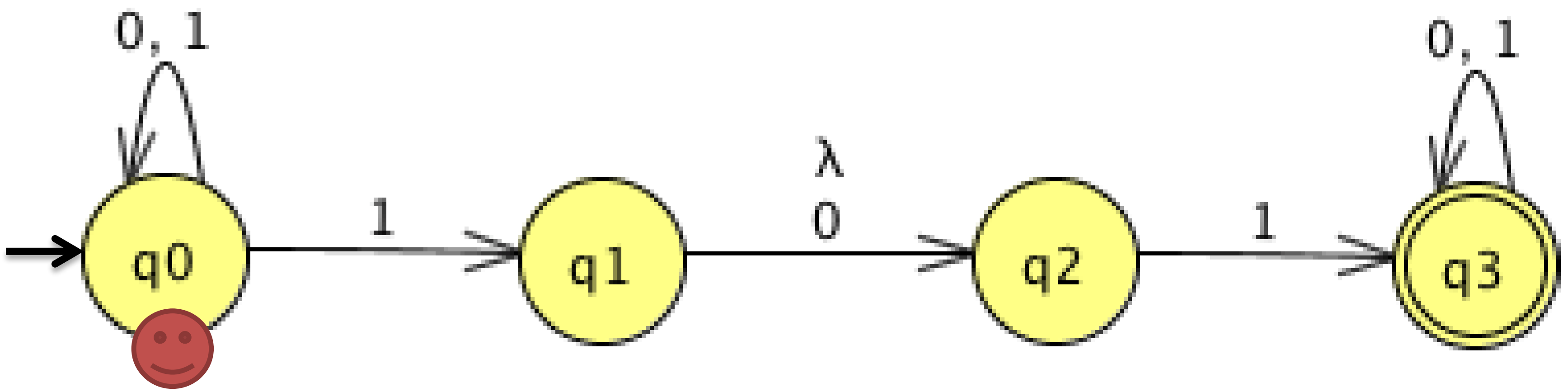
NFA Example

0 1 0 1



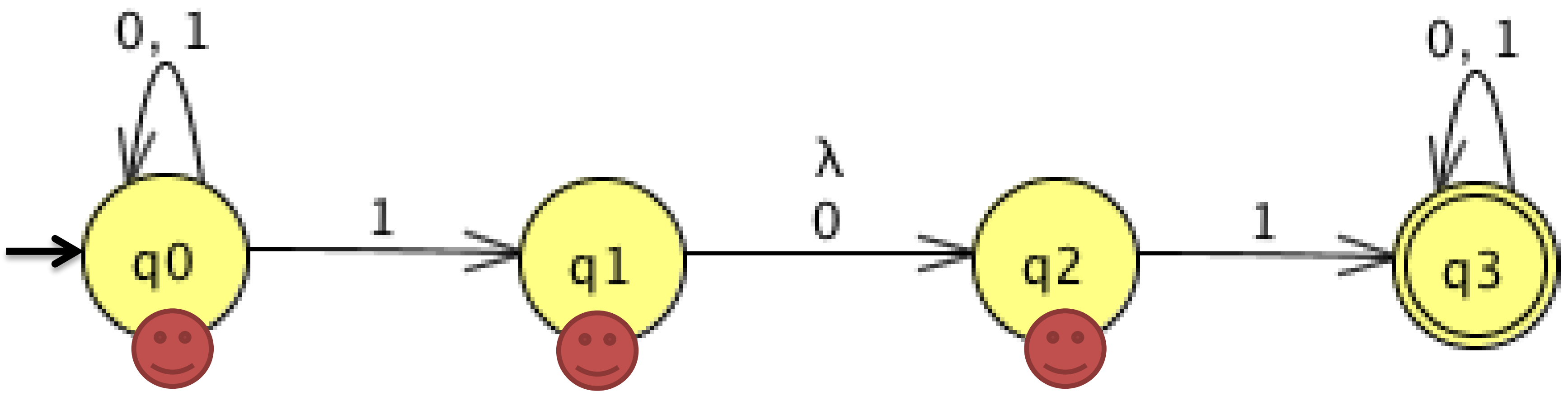
NFA Example

0 1 0 1



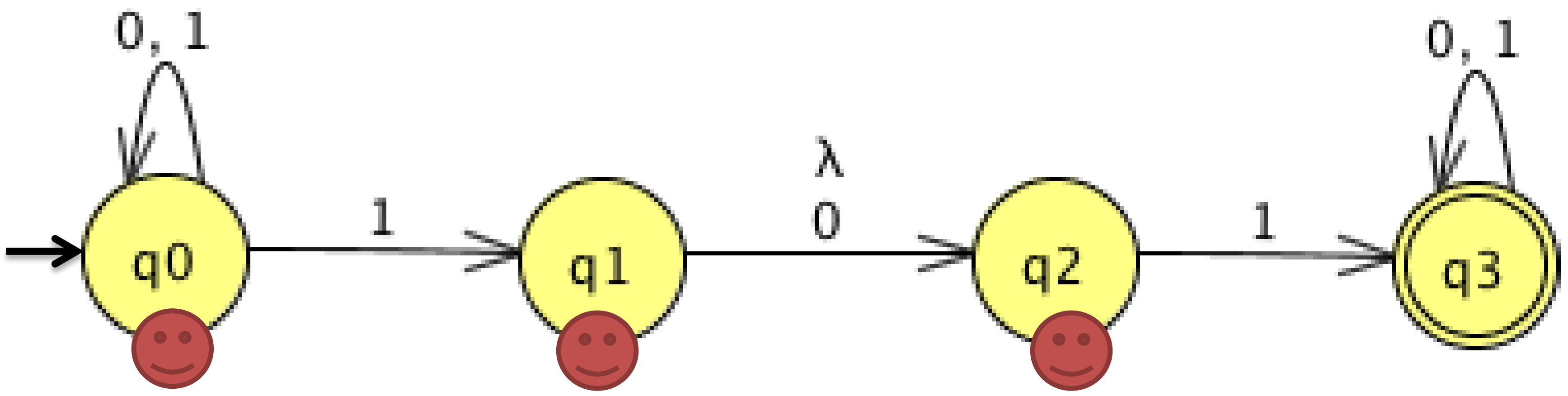
NFA Example

0 1 0 1



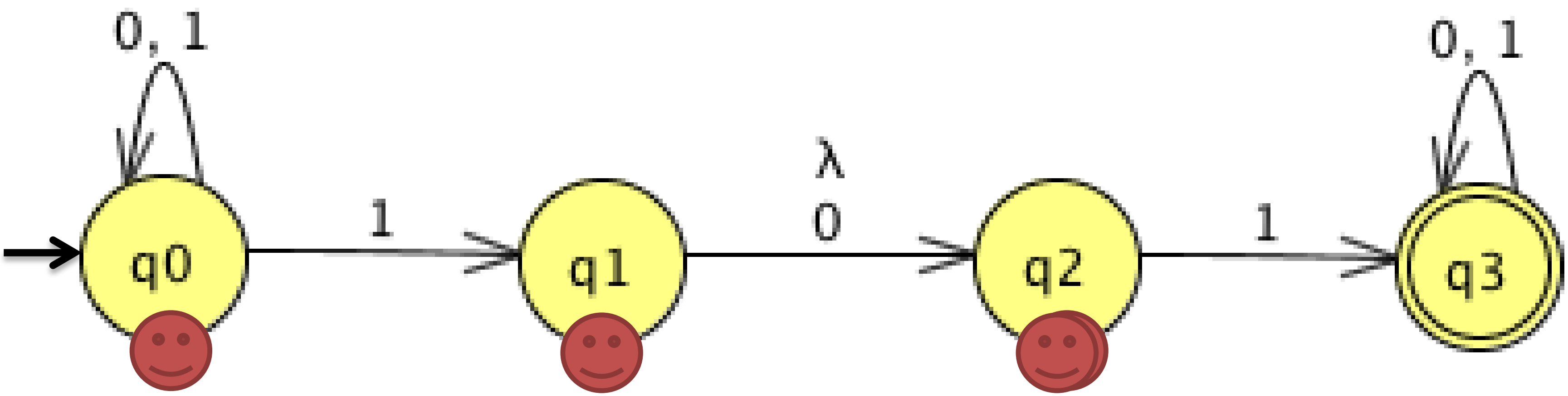
NFA Example

0 1 0 1



NFA Example

0 1 0 1



ε-closure

- The ε-closure of a state q of the NFA will be denoted by E(q).
- E(q) is the set of states that can be reached from q following ε-moves, including q itself.
- The ε-closure of a set of states R = union of the ε-closure of each state.

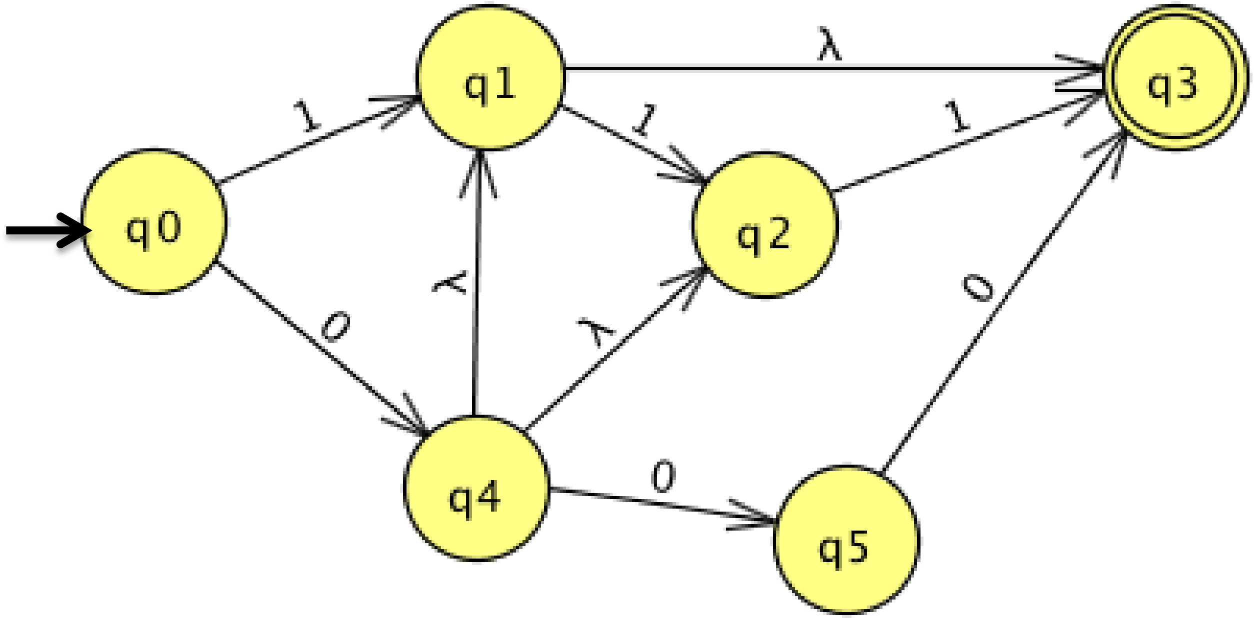
E(R) = { q | q can be reached from R by traveling along zero or more ε transitions}

ε-closure

E(R) = {q | q can be reached from R by traveling along zero or more ε transitions}

$E(q_0) = \{q_0\}$

$E(q_4) = \{q_1, q_2, q_3, q_4\}$



Extended Transition Function

Is intended to tell us where we can get from a given state following a path labeled by a certain string w.

$\hat{\delta}$ is defined by:

$\hat{\delta}(q, \lambda) = E(q)$

Let S be $\hat{\delta}(q, w)$ then:

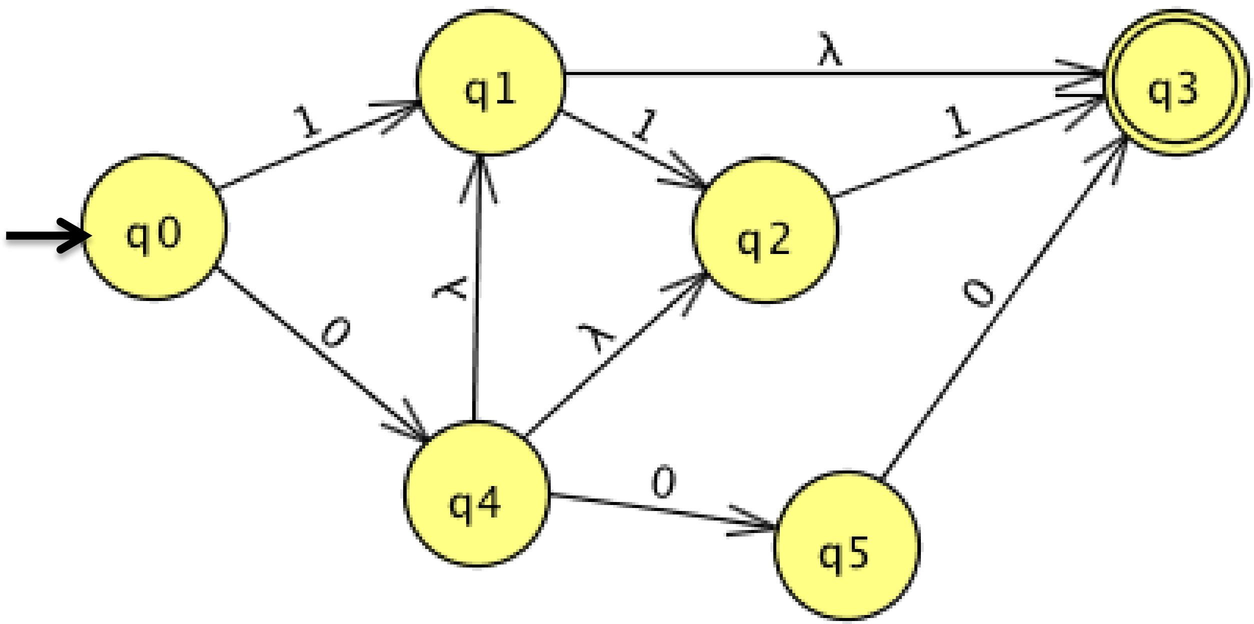
$$\hat{\delta}(q, wa) = \bigcup_{p \in S} E(\delta(p, a))$$

Example

$\hat{\delta}(q_0, \lambda) = E(q_0) = \{q_0\}$

$\hat{\delta}(q_0, 0) = E(\delta(q_0, 0)) = E(\{q_4\}) = \{q_1, q_2, q_3, q_4\}$

$\hat{\delta}(q_0, 01) = E(\{q_2, q_3\}) = \{q_2, q_3\}$



Equivalence of NFA and ε-NFA

- Every NFA is an ε-NFA, it just does not have a ε transition.

- Theorem: If a language L is accepted by an ε-NFA M_E then L is accepted by an NFA M without ε moves.

ε-NFA to NFA

- Given $M_E=(Q, \Sigma, \delta_E, q_0, F)$ construct $M=(Q, \Sigma, \delta', q_0, F')$

  Where F' = the set of states q such that E(q) contains a state of F.

  and compute δ'(q, a) as follows:

  1. Let S = E(q)
  2. $\delta'(q,a) = \bigcup_{p \in S} \hat{\delta}_E(p,a)$

❖ Note that $\hat{\delta}_E(p,a)$ in ε-NFA is actually E(δ(p,a))

ε-NFA to NFA Example

$E(q_0) = \{q_0\}$

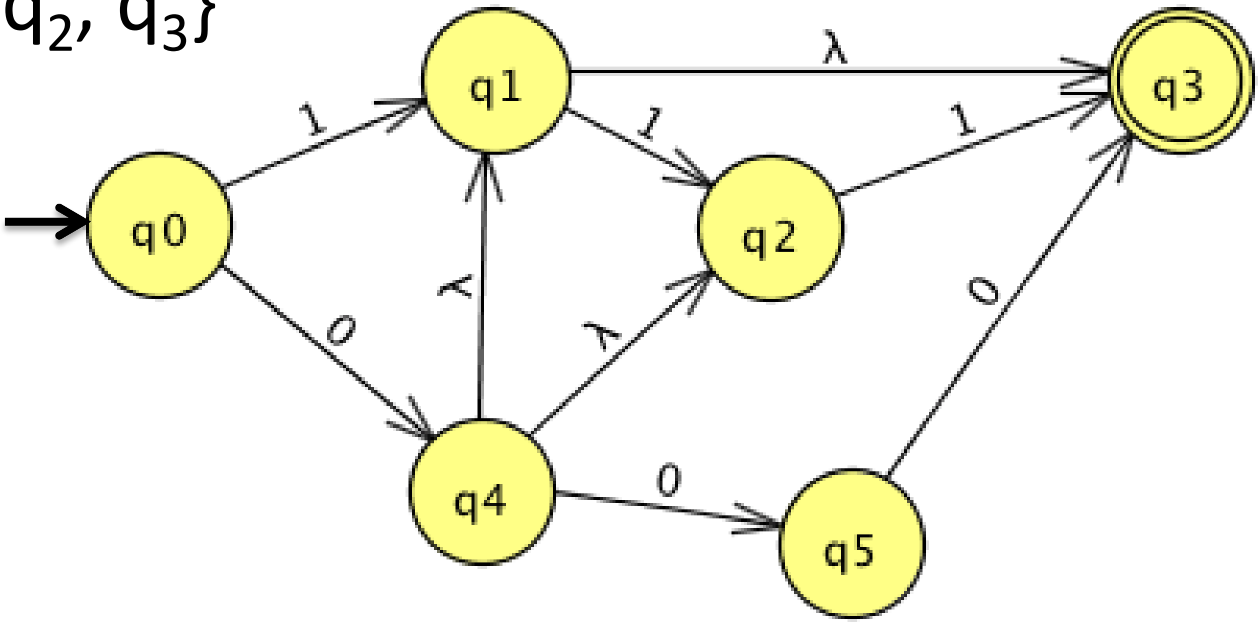
$E(q_1) = \{q_1, q_3\}$

$E(q_2) = \{q_2\}$

$E(q_3) = \{q_3\}$

$E(q_4) = \{q_4, q_1, q_2, q_3\}$

$E(q_5) = \{q_5\}$



ε-NFA to NFA Example

$\delta'(q_0, 0)$ => $S = E(q_0) = \{q_0\}$

$\delta'(q_0, 0) = \hat{\delta}_E(q_0, 0) = E(\delta(q_0,0)) = E(q_4) = \{q_4, q_1, q_2, q_3\}$

$\delta'(q_0, 1) = \hat{\delta}_E(q_0, 1) = E(\delta(q_0,1)) = E(q_1) = \{q_1,q_3\}$

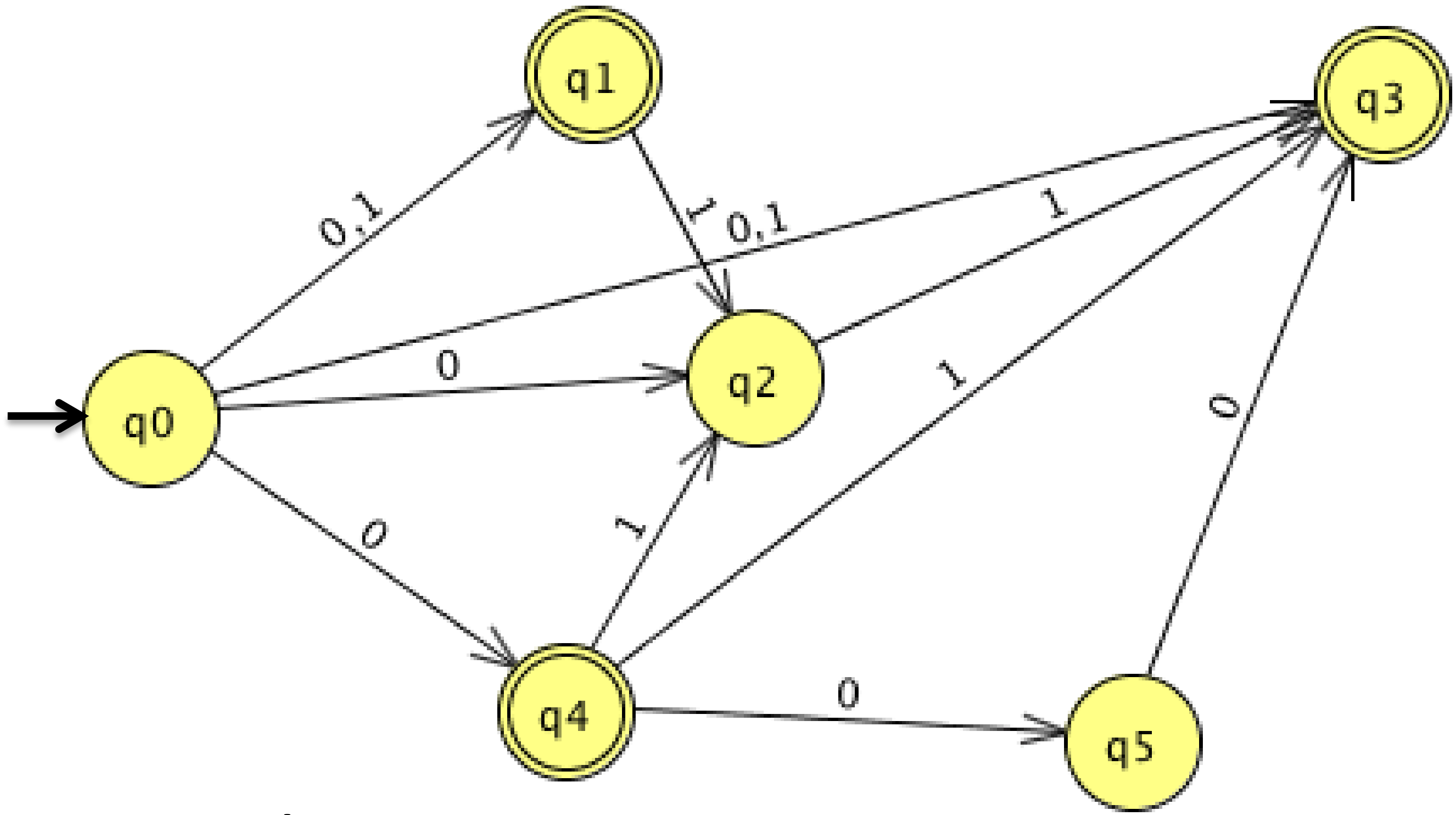
ε-NFA to NFA Example

		E()		Σ				E()
q_0	:	$\{q_0\}$	,	0	➔	$\{q_4\}$	:	$\{q_1,q_2,q_3,q_4\}$
q_0	:	$\{q_0\}$	,	1	➔	$\{q_1\}$	:	$\{q_1,q_3\}$
q_1	:	$\{q_1,q_3\}$	,	0	➔	$\emptyset$	:	$\emptyset$
q_1	:	$\{q_1,q_3\}$	,	1	➔	$\{q_2\}$	:	$\{q_2\}$
q_2	:	$\{q_2\}$	,	0	➔	$\emptyset$	:	$\emptyset$
q_2	:	$\{q_2\}$	,	1	➔	$\{q_3\}$	:	$\{q_3\}$
q_3	:	$\{q_3\}$	,	0	➔	$\emptyset$	:	$\emptyset$
q_3	:	$\{q_3\}$	,	1	➔	$\emptyset$	:	$\emptyset$
q_4	:	$\{q_4, q_1, q_2, q_3\}$	,	0	➔	$\{q_5\}$	:	$\{q_5\}$
q_4	:	$\{q_4, q_1, q_2, q_3\}$	,	1	➔	$\{q_2, q_3\}$	:	$\{q_2,q_3\}$
q_5	:	$\{q_5\}$	,	0	➔	$\{q_3\}$	:	$\{q_3\}$
q_5	:	$\{q_5\}$	,	1	➔	$\emptyset$	:	$\emptyset$

ε-NFA to NFA Example

		E()		Σ				E()
q_0	:	$\{q_0\}$	,	0	→	$\{q_4\}$	:	$\{q_1,q_2,q_3,q_4\}$
q_0	:	$\{q_0\}$	,	1	→	$\{q_1\}$	:	$\{q_1,q_3\}$
q_1	:	$\{q_1,q_3\}$	,	0	→	$\emptyset$	:	$\emptyset$
*q_1	:	$\{q_1,q_3\}$	,	1	→	$\{q_2\}$	:	$\{q_2\}$
q_2	:	$\{q_2\}$	,	0	→	$\emptyset$	:	$\emptyset$
q_2	:	$\{q_2\}$	,	1	→	$\{q_3\}$	:	$\{q_3\}$
*q_3	:	$\{q_3\}$	,	0	→	$\emptyset$	:	$\emptyset$
q_3	:	$\{q_3\}$	,	1	→	$\emptyset$	:	$\emptyset$
*q_4	:	$\{q_4, q_1, q_2, q_3\}$	,	0	→	$\{q_5\}$	:	$\{q_5\}$
q_4	:	$\{q_4, q_1, q_2, q_3\}$	,	1	→	$\{q_2, q_3\}$	:	$\{q_2,q_3\}$
q_5	:	$\{q_5\}$	,	0	→	$\{q_3\}$	:	$\{q_3\}$
q_5	:	$\{q_5\}$	,	1	→	$\emptyset$	:	$\emptyset$

ε-NFA to NFA Example



NFA without ε moves

Summary

- DFA' s, NFA' s, and ϵ–NFA' s all accept exactly the same set of languages: the regular languages.
- The NFA types are easier to design and may have exponentially fewer states than a DFA.
- DFA's are much easier to implement on a computer.