Introduction to Formal Languages and Automata Theory

Dr. Mousumi Dutt

Module 2

Regular Expression: Introduction

- A regular expression is a pattern which matches some regular (predictable) text.
- Regular expressions are used in many Unix utilities.
  - like grep, sed, vi, emacs, awk, ...
- The form of a regular expression:
  - It can be plain text ...

  > grep unix file (matches all the appearances of unix)
  - It can also be special text ...

  > grep '[uU]nix' file (matches unix and Unix)

Regular Expression: Introduction

- Regular expressions are different from file name wildcards.
  - Regular expressions are interpreted and matched by special utilities (such as grep).
  - File name wildcards are interpreted and matched by shells.
  - They have different wildcarding systems.
  - File wildcarding takes place first!

```
obelix[1] > grep '[uU]nix' file
obelix[2] > grep [uU]nix file
```

Regular Expression: Introduction

- Offers a declarative way to express the pattern of any string we want to accept
  - E.g., 01*+ 10*
- Automata => more machine-like

  < input: string , output: [accept/reject] >
- Regular expressions => more program syntax-like
- Unix environments heavily use regular expressions
  - E.g., bash shell, grep, vi & other editors, sed
- Perl scripting – good for string processing
- Lexical analyzers such as Lex or Flex

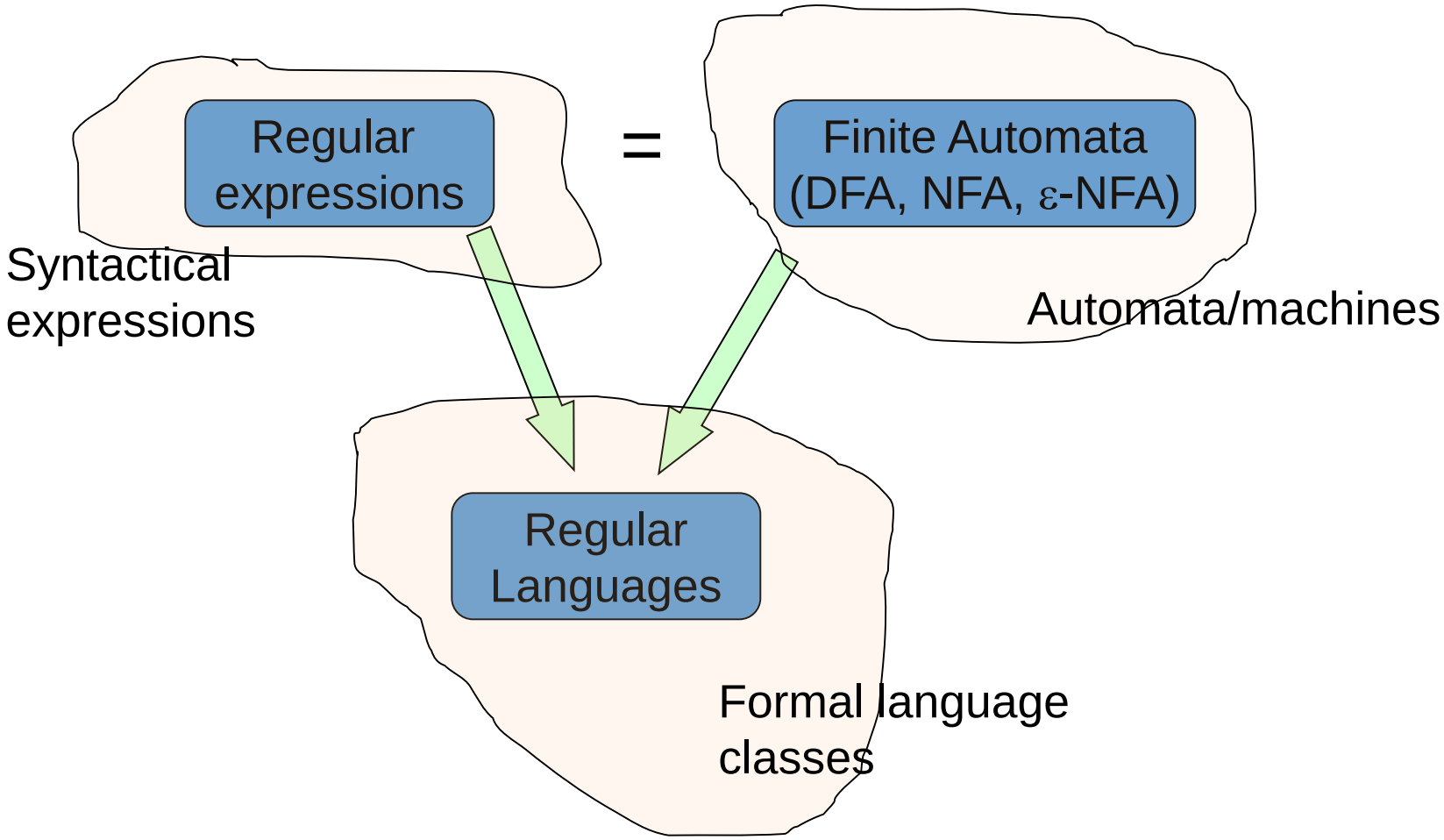
Understanding Regular Expression

- Very powerful and quite cryptic
- Fun once you understand them
- Regular expressions are a language unto themselves
- A language of "marker characters" - programming with characters
- It is kind of an "old school" language - compact

Regular Expression

- A language is regular if there exists a finite acceptor for it
- Every regular language can be desired by some DFA or some NFA
- E.g. whether if a given string is in a certain language
- But in many instances, we need more concise ways of describing regular languages

Regular Expression



Operations on Languages

- Union of two languages:
  - **L U M** = all strings that are either in L or M
  - Note: A union of two languages produces a third language

- Concatenation of two languages:
  - **L . M** = all strings that are of the form *xy*
    s.t., x $\in$ L and y $\in$ M
  - The *dot* operator is usually omitted
    - i.e., **LM** is same as L.M

Operations on Languages

"i" here refers to how many strings to concatenate from the parent language L to produce strings in the language L^i

- Kleene Closure of a given language L:
  - $L^0 = \{\varepsilon\}$
  - $L^1 = \{w \mid \text{for some } w \in L\}$
  - $L^2 = \{ w_1w_2 \mid w_1 \in L, w_2 \in L \text{ (duplicates allowed)}\}$
  - $L^i = \{ w_1w_2\ldots w_i \mid \text{all w's chosen are} \in L \text{ (duplicates allowed)}\}$
  - (Note: the choice of each w_i is independent)
  - **$L^* = \bigcup_{i\geq 0} L^i$** (arbitrary number of concatenations)

Example:

- Let L = { 1, 00}
  - $L^0 = \{\varepsilon\}$
  - $L^1 = \{1,00\}$
  - $L^2 = \{11,100,001,0000\}$
  - $L^3 = \{111,1100,1001,10000,000000,00001,00100,0011\}$
  - **$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$**

Operations on Languages

- L^* is an infinite set iff $|L| \geq 1$ and $L \neq \{\varepsilon\}$ **Why?**
- If $L=\{\varepsilon\}$, then $L^* = \{\varepsilon\}$ **Why?**
- If $L = \Phi$, then $L^* = \{\varepsilon\}$ **Why?**

Σ^* denotes the set of all words over an alphabet Σ

- Therefore, an abbreviated way of saying there is an arbitrary language L over an alphabet Σ is:
  - $L \subseteq \Sigma^*$

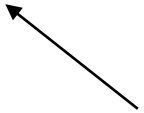
Regular Expression: Definition

- Basis 1: If a is any symbol, then **a** is a RE, and L(**a**) = {a}.
  - Note: {a} is the language containing one string, and that string is of length 1.
- Basis 2: ϵ is a RE, and L(ϵ) = {ϵ}.
- Basis 3: $\emptyset$ is a RE, and L($\emptyset$) = $\emptyset$.

(1) $\emptyset$, ϵ, and a ϵ $\sum$ are all regular expressions

Regular Expression: Definition

- Induction 1: If r_1 and r_2 are regular expressions, then r_1+r_2 is a regular expression, and $L(r_1+r_2) = L(r_1) \cup L(r_2)$.
- Induction 2: If r_1 and r_2 are regular expressions, then r_1r_2 is a regular expression, and $L(r_1r_2) = L(r_1)L(r_2)$.



Concatenation : the set of strings wx such that w Is in $L(r_1)$ and x is in $L(r_2)$.

(2) If r_1 and r_2 are regular expressions, so are r_1+r_2, r_1r_2, r_1^*, and (r_1).

Regular Expression: Definition

- Induction 3: If r is a RE, then r* is a RE, and L(r*) = (L(r))*.

Closure, or "Kleene closure" = set of strings $w_1 w_2 \ldots w_n$, for some $n \geq 0$, where each w_i is in L(r).
Note: when n=0, the string is ϵ.

(3) A string is a regular expression if and only if it can be derived from the primitive regular expression by a finite number of applications of rules in (2)

(4) Any set represented by regular expression is called regular set

Precedence of Operators

- Parentheses may be used wherever needed to influence the grouping of operators.

- Order of precedence is * (highest), then concatenation, then + (lowest).

Language Associated with RE

- RE can be used to describe some simple languages
- If r is a regular expression, we will let L(r) denote the language associated with r
- The language L(r) denoted by any regular expression r is defined by the following rules

1. $\emptyset$ is a RE denoting the empty set
2. ϵ is a RE denoting $\{\epsilon\}$
3. For every a ϵ $\sum$, a is a RE denoting {a}.

Language Associated with RE

- If r_1 and r_2 are regular expressions, then

4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
5. $L(r_1 . r_2) = L(r_1) . L(r_2)$
6. $L((r_1)) = L(r_1)$
7. $L(r_1^*) = (L(r_1))^*$

The last four rules of this definition are used to reduce L(r) to simpler components recursively; the first three are the termination conditions for this recursion

Example

- L(**01**) = {01}.
- L(**01**+**0**) = {01, 0}.
- L(**0**(**1**+**0**)) = {01, 00}.
  - Note order of precedence of operators.
- L(**0***) = {ϵ, 0, 00, 000,… }.
- L((**0**+**10**)*(ϵ+**1**)) = all strings of 0's and 1's without two consecutive 1's.

Example

- ***L = { w | w is a binary string which does not contain two consecutive 0s or two consecutive 1s anywhere)***
  - E.g., w = 01010101 is in L, while w = 10010 is not in L
- Goal: Build a regular expression for L
- Four cases for w:
  - Case A: w starts with 0 and |w| is even
  - Case B: w starts with 1 and |w| is even
  - Case C: w starts with 0 and |w| is odd
  - Case D: w starts with 1 and |w| is odd
- Regular expression for the four cases:
  - Case A: (01)*
  - Case B: (10)*
  - Case C: 0(10)*
  - Case D: 1(01)*
- Since L is the union of all 4 cases:
  - Reg Exp for L = (01)* + (10)* + 0(10)* + 1(01)*
- If we introduce ε then the regular expression can be simplified to:
  - Reg Exp for L = (ε +1)(01)*(ε +0)

Example

Exhibit the language L(a*.(a+b)) in set notation

$L(a^*.(a+b))=L(a^*).L(a+b)$

$=(L(a))^*.(L(a)\cup L(b))$

$=\{\epsilon, a, aa, aaa, \ldots\}\{a,b\}$

$=\{a, aa, aaa, \ldots, b, ab, aab, \ldots\}$

Example

For ∑={a,b}, the expression r=(a+b)*(a+bb) is regular. It denotes the language L(r)={a, bb, aa, abb, ba, bbb, …}

The set of all strings on {a,b} terminated by either a or bb

The expression r=(aa)*(bb)*b denotes the set of all strings with an even no of a's followed by an odd no of b's; that is,

$L(r)=\{a^{2n}b^{2m+1}: n>=0, m>=0\}$

Write the regular expression which includes the set of all strings of 0's and 1's beginning with 0 and ending with 1.

r=0(0+1)*1

Identities for RE

P=Q implies P and Q are equivalent if P Q represents the same set of strings

I1: $\emptyset + R = R$

I2: $\emptyset R = R\emptyset = \emptyset$

I3: $\epsilon R = R \epsilon = R$

I4: $\epsilon^* = \epsilon$ and $\emptyset^* = \epsilon$

I5: $R + R = R$

I6: $R^*R^* = R^*$

I7: $RR^* = R^*R$

I8: $(R^*)^* = R^*$

I9: $\epsilon + RR^* = R^* = \epsilon + R^*R$

I10: $(PQ)^*P = P(QP)^*$

I11: $(P+Q)^* = (P^*Q^*)^* = (P^*+Q^*)^*$

I12: $(P+Q)R = PR + QR$ and

$R(P+Q) = RP + RQ$

Example

L = string in which every 0 is immediately followed by at least two 1's.

RE: (1+011)*

RE: ϵ + 1*(011)*(1*(011)*)*

Let P_1 = 1*(011)*

R= ϵ+ P_1P_1*= P_1* using I9

= (1*(011)*)*

=(P_2*P_3*)* letting P_2 =1, P_3=011

=(P_2+P_3)* using I11

=(1+011)*

Example

Prove (1+00*1)+ (1+00*1)(0+10*1)*(0+10*1)=0*1(0+10*1)*

LHS: (1+00*1)+ (1+00*1)(0+10*1)*(0+10*1)

= (1+00*1)(ϵ +(0+10*1)*(0+10*1)) using I12

= (1+00*1) (0+10*1)* using I9

= (ϵ + 00*) 1 (0+10*1)* using I12 for 1+00*1

= 0*1(0+10*1)* using I9

= RHS

Arden's Theorem

Let P and Q be two regular expressions over ∑. If P does not contain ϵ, then the following equation in R, namely R=Q+RP --(1) has a unique solution (i.e., one and only one solution) given by R=QP*

Proof: R=Q+RP = Q+ (QP*)P = Q(ϵ + P*P) = QP* by I9

This means R+QP* is a solution of R=Q+RP

To prove uniqueness:

Here, replacing R by Q+RP on the R.H.S., we get the equation:

Arden's Theorem

$$Q+RP = Q+ (Q +RP)P$$

$$= Q+ QP + RPP$$

$$= Q+ QP + RP^2$$

$$= Q+ QP + QP^2 + \ldots + QP^i + RP^{i+1}$$

$$= Q(\epsilon + P + P^2 + \ldots + P^i)+ RP^{i+1}$$

$$R= Q(\epsilon + P + P^2 + \ldots + P^i)+ RP^{i+1} \text{ for } i>=0 \text{ --- (2)}$$

Arden's Theorem

Here, i is any arbitrary integer.

Let us take a string $\omega \in R$ | length of $\omega = k$. Substituting k for i in equation (3) we get;

$= Q (\varepsilon + P + P2 + P3 + \ldots. + Pk) + RPk+1$

Since P does not contain ε, ω is not contained in RPk+1 as the shortest string generated form RPk+1 will have a length of k+1.

Since ω is contained in R it must be contained in Q (ε + P + P2 + P3 + …. + Pk).

Conversely, if ω is contained in QP* then for some integer k it must be in Q (ε + P + P2 + P3 + …. + Pk), and hance in R = Q + QP.

Hence, proved.

Application of Arden's Theorem

The main application of Arden's Theorem are as following;

1. It helps to determine the regular expression of finite automata.
2. Arden's theorem helps in checking the equivalence of two regular expressions.

Example

Here the initial state and final state is $\mathbf{q_1}$.

The equations for the three states q1, q2, and q3 are as follows −

$q_1 = q_1a + q_3a + \varepsilon$ (ε move is because q1 is the initial state0

$q_2 = q_1b + q_2b + q_3b$

$q_3 = q_2a$

Now, we will solve these three equations −

$q_2 = q_1b + q_2b + q_3b$

$= q_1b + q_2b + (q_2a)b$ (Substituting value of q_3)

$= q_1b + q_2(b + ab)$

$= q_1b\ (b + ab)^*$ (Applying Arden's Theorem)

$q_1 = q_1a + q_3a + \varepsilon$

$= q_1a + q_2aa + \varepsilon$ (Substituting value of q_3)

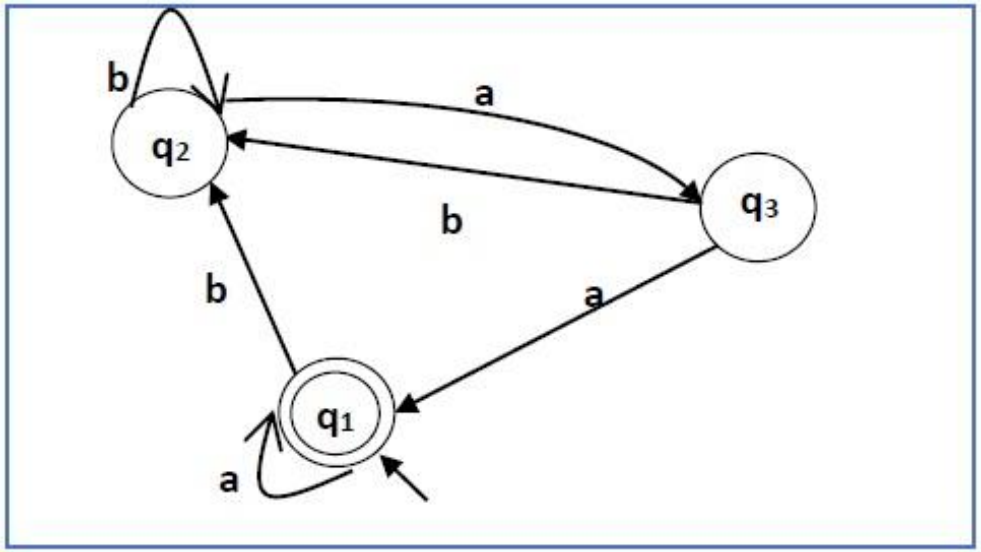
$= q_1a + q_1b(b + ab^*)aa + \varepsilon$ (Substituting value of q_2)

$= q_1(a + b(b + ab)^*aa) + \varepsilon$

$= \varepsilon\ (a + b(b + ab)^*aa)^*$

$= (a + b(b + ab)^*aa)^*$

Hence, the regular expression is $(a + b(b + ab)^*aa)^*$.



Example

Here the initial state is q_1 and the final state is q_2

Now we write down the equations −

$q_1 = q_10 + \varepsilon$

$q_2 = q_11 + q_20$

$q_3 = q_21 + q_30 + q_31$

Now, we will solve these three equations −

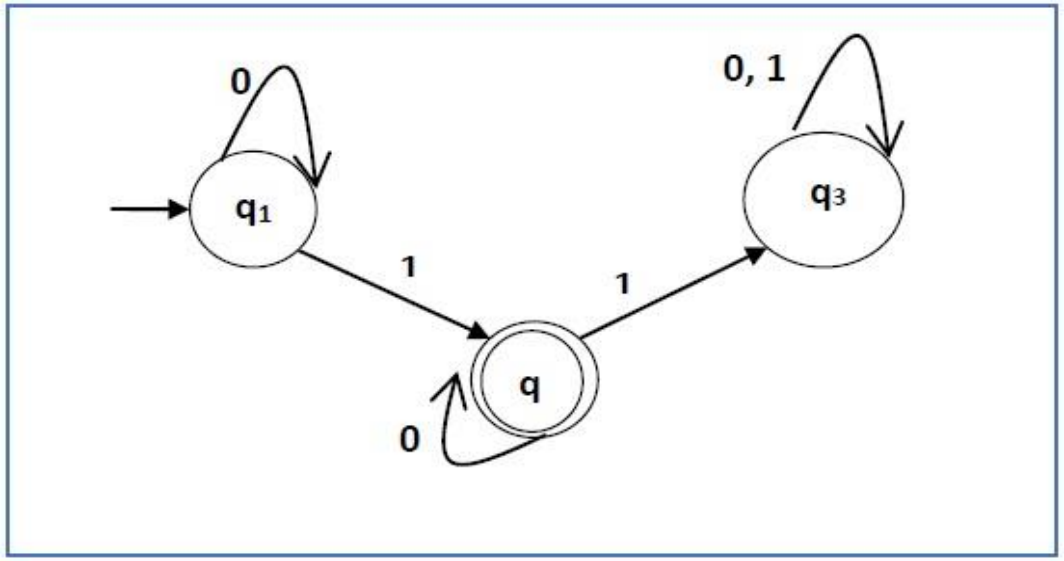
$q_1 = \varepsilon 0^*$ [As, $\varepsilon R = R$]

So, $q_1 = 0^*$

$q_2 = 0^*1 + q_20$

So, $q_2 = 0^*1(0)^*$ [By Arden's theorem]

Hence, the regular expression is 0^*10^*



Regular Expressions and Regular Languages

For every regular language there is a regular expression, and vice versa

To identify whether a string is accepted by regular language finite automaton is used which is of two types

- DFA: deterministic finite automata

On each input there is one and only one state to which the automaton can have transition from its current state

- NFA: non-deterministic finite automata

On each input there can be several states at once

Definition of DFA

A deterministic finite automaton consists of:

1. A finite set of states often denoted Q
2. A finite set of input symbols, often denoted by $\sum$
3. A transition function that takes as arguments a state and an input symbol and returns a state. The transition function will commonly denoted by δ. In our informal graph representation of automata, δ was represented by arcs between states and the labels on the arcs. If q is a state and a is an input symbol, then δ(q,a) is that p such that there is an arc labelled a from q top.
4. A start state, one of the states in Q
5. A set of final accepting states F. The set F is a subset of Q.

Definition of DFA

A deterministic finite automaton will ofte be referred to by its acronym: DFA.

DFA is a five tuple notation: A= (Q, $\sum$, δ, q_0, F)

where A is the name of the DFA, Q is its set of states, $\sum$ its input symbols, δ its transformation function, q_0 its start state, and F its set of accepting states

Transition Diagrams of DFA

A transition diagram for A= (Q, $\sum$, δ, q_0, F) is a graph defined as follows

a) For each state in Q there is a node
b) For each state q in Q and each input symbol a in $\sum$, let δ(q,a)=p. Then the transition diagram has an arc from node q to node p, labelled a.. If there are several input symbols that cause transitions from q to p, then the transition diagram can have one arc, labelled by the list of symbols
c) There is an arrow into the start q_0, labelled start. This arrow does not originate at any node
d) Nodes corresponding to accepting states (those in F) are marked by a double circle. States not in F have a single circle.

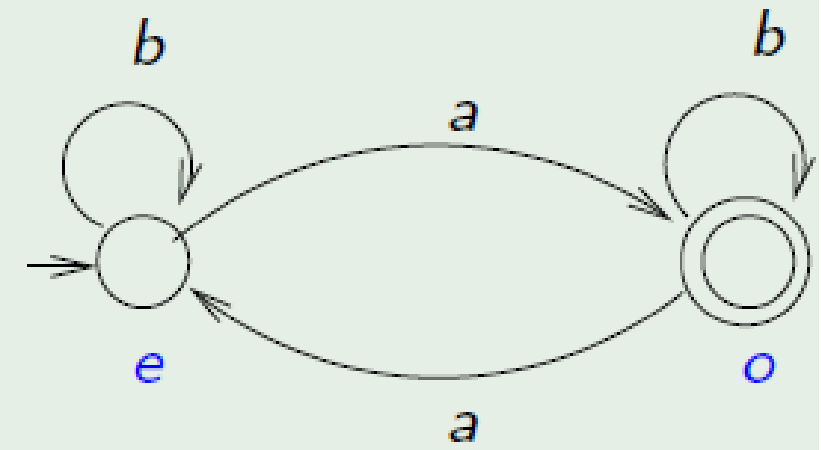
Transition Tables

A transition table is a conventional, tabular representation of a function like δ that takes two arguments and returns a value.

The rows of the table correspond to the states, and the columns correspond to the inputs.

The entry for the row corresponding to state q and the column corresponding to input a is the state δ(q,a)

Example of DFA

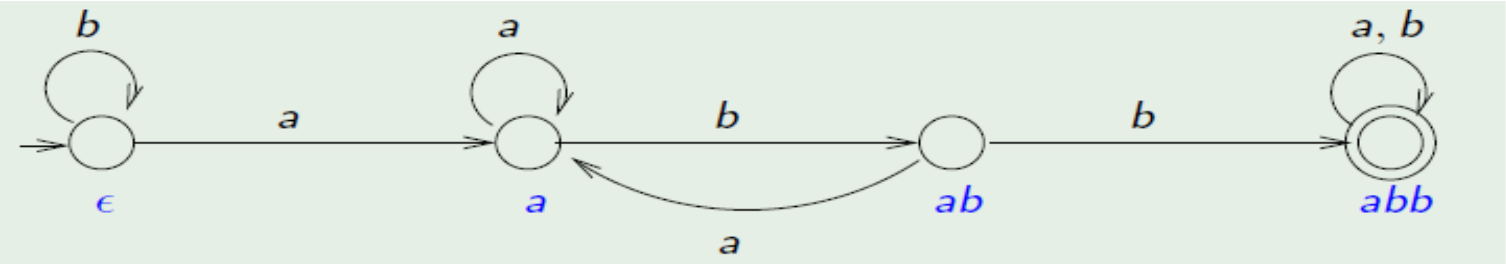


How a DFA works?

Each state represents a property of the input string read so far:
State e: Number of a's seen is even.
State o: Number of a's seen is odd.

States	Input	
	a	b
q0	q1	q0
q1	q0	q1

Example of DFA



Each state represents a property of the input string read so far:

- State ϵ: Not seen *abb* and no suffix in *a* or *ab*.
- State *a*: Not seen *abb* and has suffix *a*.
- State *ab*: Not seen *abb* and has suffix *ab*.
- State *abb*: Seen *abb*.

States	Input	
	a	b
q0	q1	q0
q1	q1	q2
q2	q1	q3
q3	q3	q3

Extended Transition Function

The language of a DFA is the set of labels along all the paths that lead from the start state to any accepting state.

The extended transition function that describes what happened when we start in any state and follow any sequence of inputs

If δ is the transition function, then the extended transition function constructed from δ will be called $\hat{\delta}$

The extended transition function is a function that takes a state q and a string w and returns a state p – the state that the automaton reaches when starting in state q and processing the sequence of inputs w.

We define $\hat{\delta}$ by induction on the length of the input string , as follows:

Extended Transition Function

Basis: $\delta\hat{}(q, \varepsilon) = q$ implies that if we are in state q and read no inputs then we are still in state q

Induction: Suppose w is a string of the form xa; a is the last symbol of w and x is the string consisting of all but the last symbol

w=1101 is broken into x=110 and a=1

Then, $\delta\hat{}(q, w) = \delta(\delta\hat{}(q, x), a)$

To compute $\delta\hat{}(q, w)$, first compute $\delta\hat{}(q, x)$, the state that the automaton is in after processing all but the last symbol of w

Suppose this state is p; that is $\delta\hat{}(q, x) = p$

The $\delta\hat{}(q, w)$ is what we get by making a transition from state p on input a, the last symbol of w., i.e., $\delta\hat{}(q, w) = \delta(p, a)$

The Language of DFA

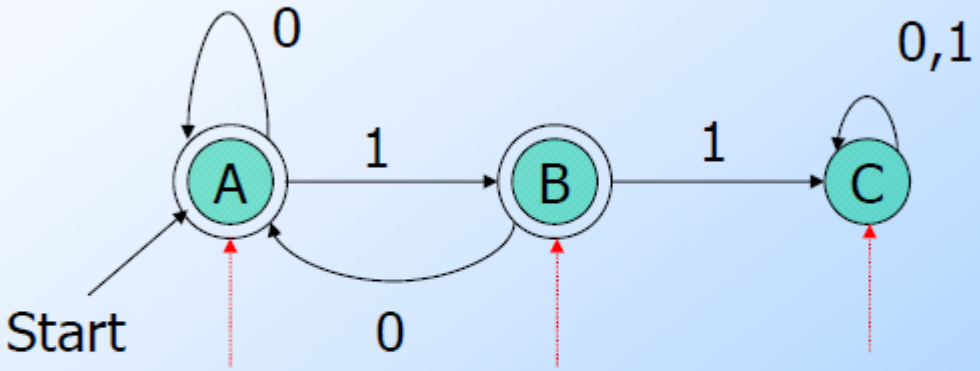
The language of a DFA, $A= (Q, \sum, \delta, q_0, F)$ is denoted by L(A) and is defined by $L(A) = \{w \mid \delta\hat{}(q_0, w) \text{ is in } F\}$

The language of A is the set of strings w that take the start state q_0 to one of the accepting states

If L is L(A) for some DFA, A, then we say L is a regular language

Example

Accepts all strings without two consecutive 1's.



Previous string OK, does not end in 1.

Previous String OK, ends in a single 1.

Consecutive 1's have been seen.

Final states starred

Arrow for start state

Columns = input symbols

	0	1
→ * A	A	B
* B	A	C
C	C	C

Rows = states

	0	1
A	A	B
B	A	C
C	C	C

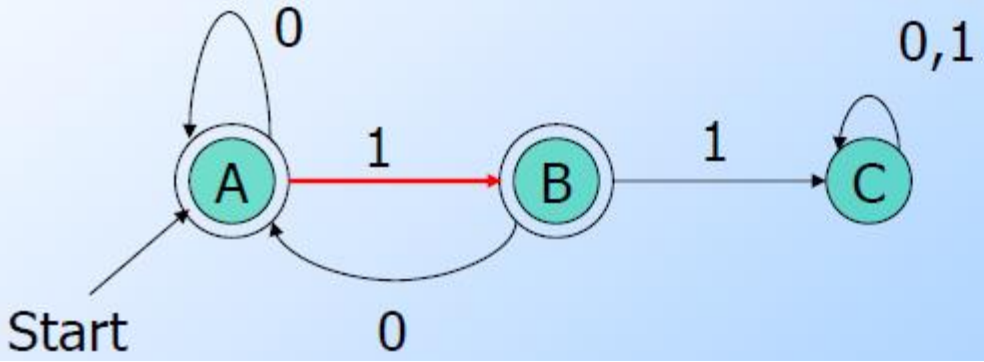
$\delta(B,011) = \delta(\delta(B,01),1) = \delta(\delta(\delta(B,0),1),1) =$

$\delta(\delta(A,1),1) = \delta(B,1) = C$

Example

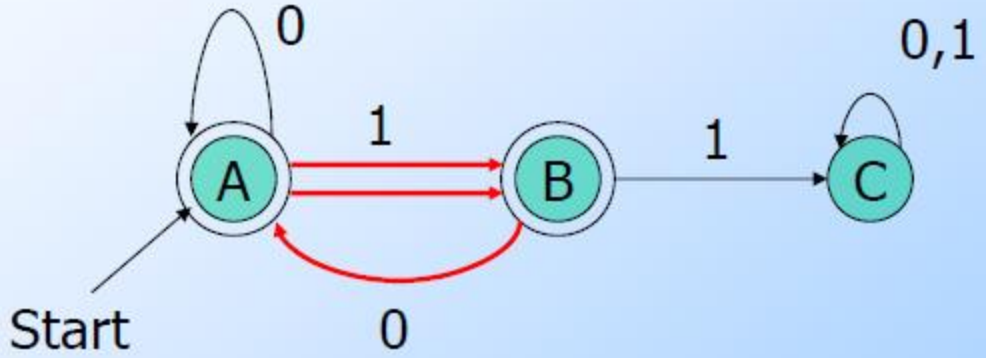
String 101 is in the language of the DFA below.

Follow arc labeled 1.



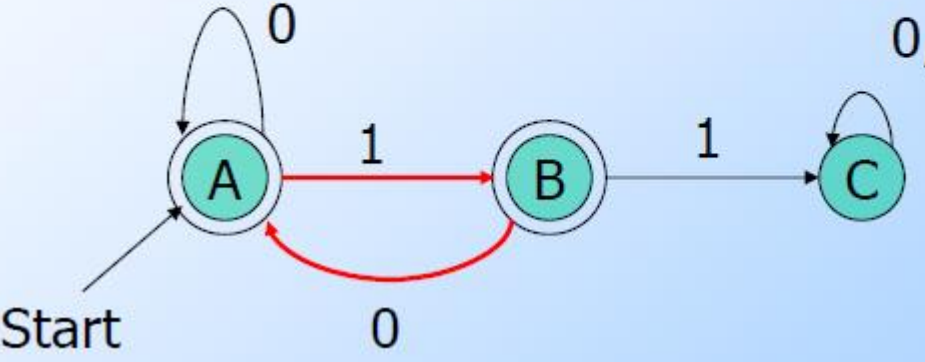
String 101 is in the language of the DFA below.

Finally arc labeled 1 from current state A. Result is an accepting state, so 101 is in the language.



String 101 is in the language of the DFA below

Then arc labeled 0 from current state B.



- The language of our example DFA is:

{w | w is in {0,1}* and w does not have two consecutive 1's}

Such that...

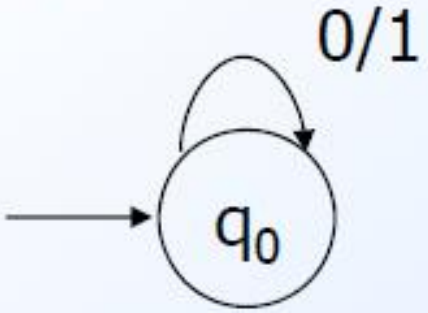
These conditions about w are true.

Read a *set former* as "The set of strings w...

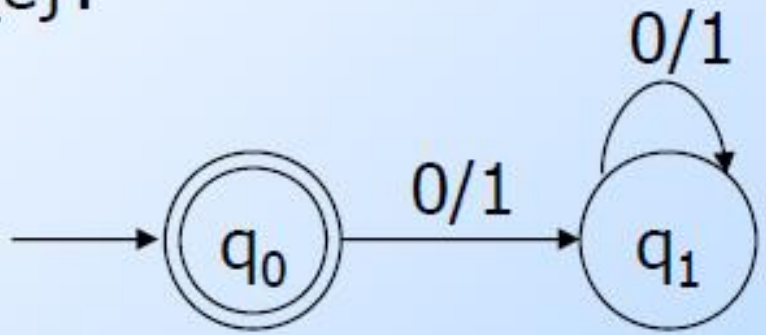
Example

Let $\Sigma = \{0, 1\}$. Give DFAs for $\{\}$, $\{\varepsilon\}$, Σ^*, and Σ^+.

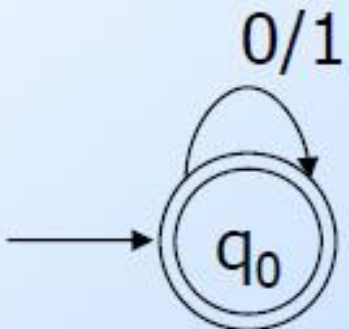
For $\{\}$:



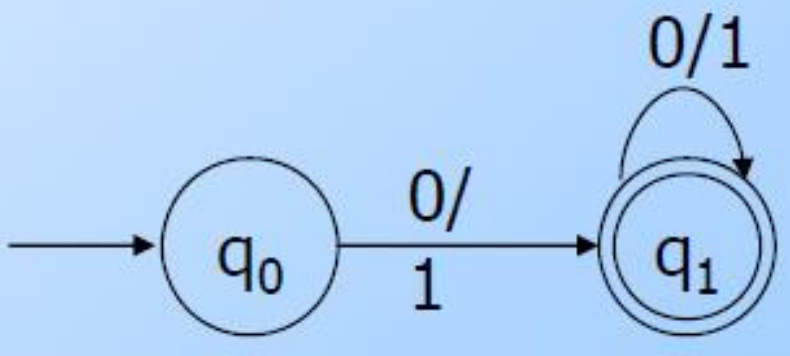
For $\{\varepsilon\}$:



For Σ^*:



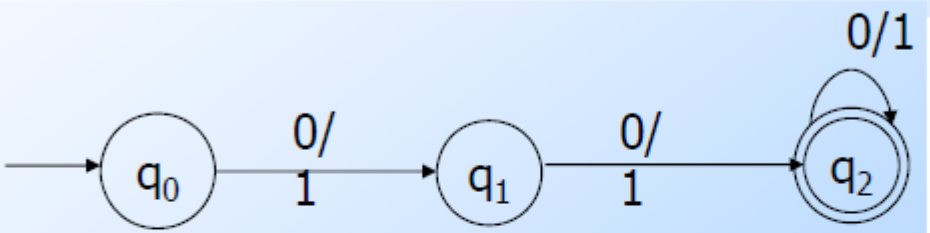
For Σ^+:



Example

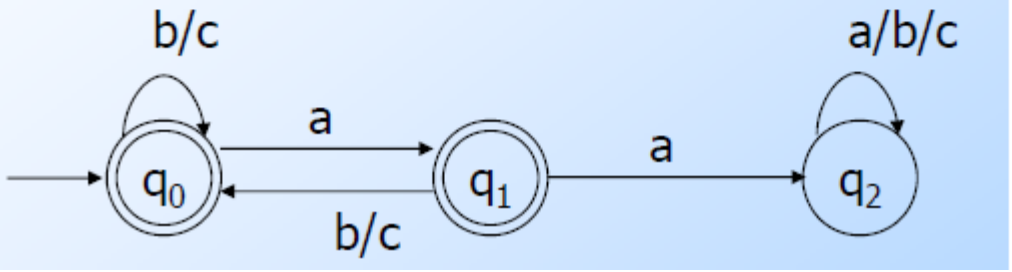
Give a DFA M such that:

L(M) = {x | x is a string of 0's and 1's and |x| >= 2}



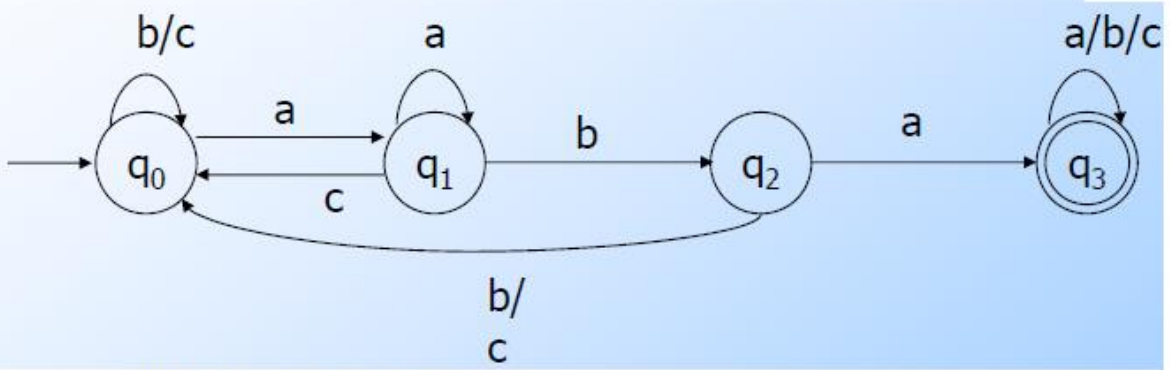
Give a DFA M such that:

L(M) = {x | x is a string of (zero or more) a's, b's and c's such that x does not contain the substring *aa*}



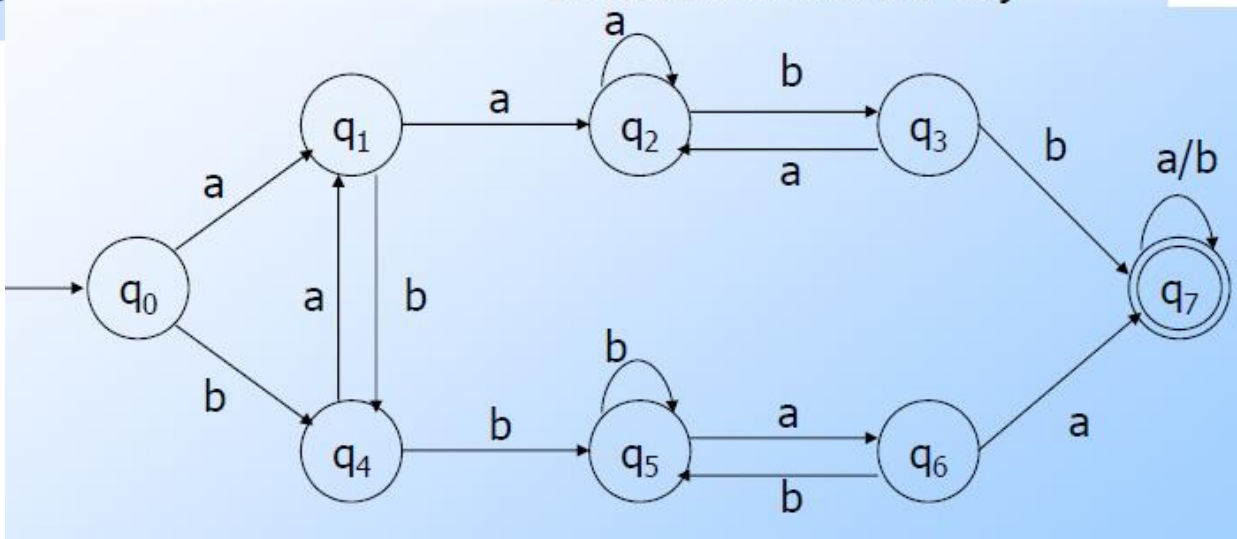
Give a DFA M such that:

L(M) = {x | x is a string of a's, b's and c's such that x contains the substring *aba*}



Give a DFA M such that:

L(M) = {x | x is a string of a's and b's such that x contains both *aa* and *bb*}



Deterministic Finite Automata

Simulating a DFA

Input: An input string x terminated by an EOF. A DFA D with start state s0 and set of accepting states F.

Output: The answer "yes" if D accepts x; "no" otherwise.

```
S: =s0;
C: = nextChar();
While c != EOF do
        S:= move(s, c);
        C:=nextChar();
End;
If s is in F then return "yes"
Else return "no".
```

Non-Deterministic Finite Automata (NFA)

Has the power to be in several states at once
Ability to guess something about its input

Ex: keyword search (long text string)
It is helpful to guess that we are at the beginning of one those strings and use a sequence states to do nothing but check that the string appears, character by character.

Note: All RE/ RL accepted by NFA is also accepted by an equivalent DFA

Non-Deterministic Finite Automata (NFA)

The NFA is represented by, A= (Q, $\sum$, δ, q_0, F)

Where,

1. Q is a finite set of states
2. $\sum$ is a finite set of input symbols
3. q_0 is the start state
4. F a subset of Q, is the set of final (or accepting) states
5. δ, the transition function is a function that takes a state in Q and input symbol in $\sum$ as arguments and returns a subset of Q. Notice that the only difference between an NFA and a DFA is in the type of value that δ returns: a set of states in the case of an NFA and a single state in the case of a DFA

The extended transition function

Basis: $\hat{\delta}(q, \epsilon) = \{q\} \Rightarrow$ without reading any input symbols we are only in the state we began in

Induction: Suppose w is of the form w=xa, where a is the final symbol of w and x is the rest of w.

Also, suppose that $\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_k\}$

Let $\bigcup_{i=1}^{k} \delta(pi, a) = \{r_1, r_2, \ldots, rm\}$

Then $\hat{\delta}(q, w) = \{r_1, r_2, \ldots, rm\}$

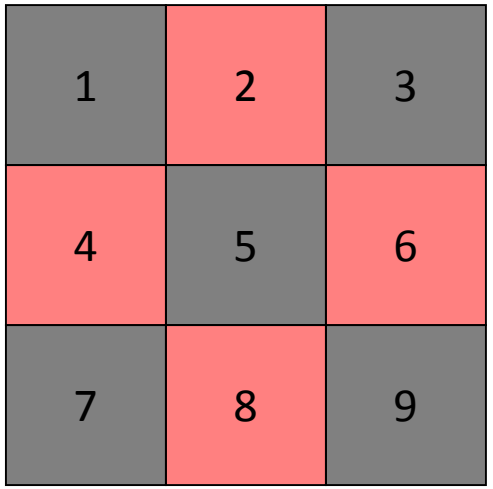
We compute $\hat{\delta}(q, w)$ by first computing $\hat{\delta}(q, x)$, and by then following any transition from any of these states that is labelled a.

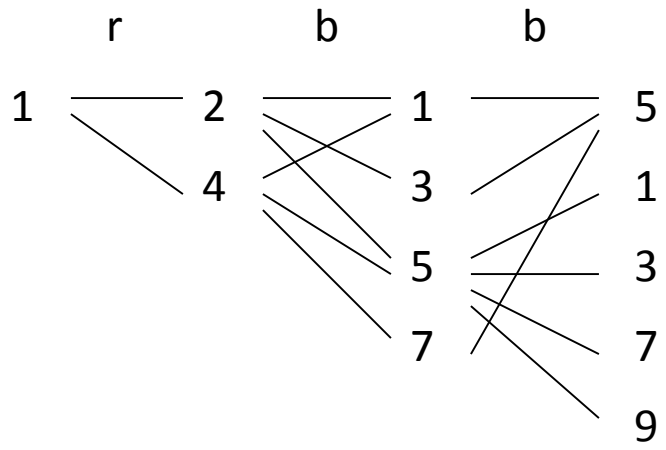
The Language of NFA

The language of a NFA, A= (Q, $\sum$, δ, q_0, F) is denoted by L(A) and is defined by L(A) = {w | $\delta\hat{}(q_0, w) \cap F \neq \emptyset$}

L(A) is the set of strings w in $\sum^*$ such that $\delta\hat{}(q_0, w)$ contains at least one accepting state.

Example: Chessboard



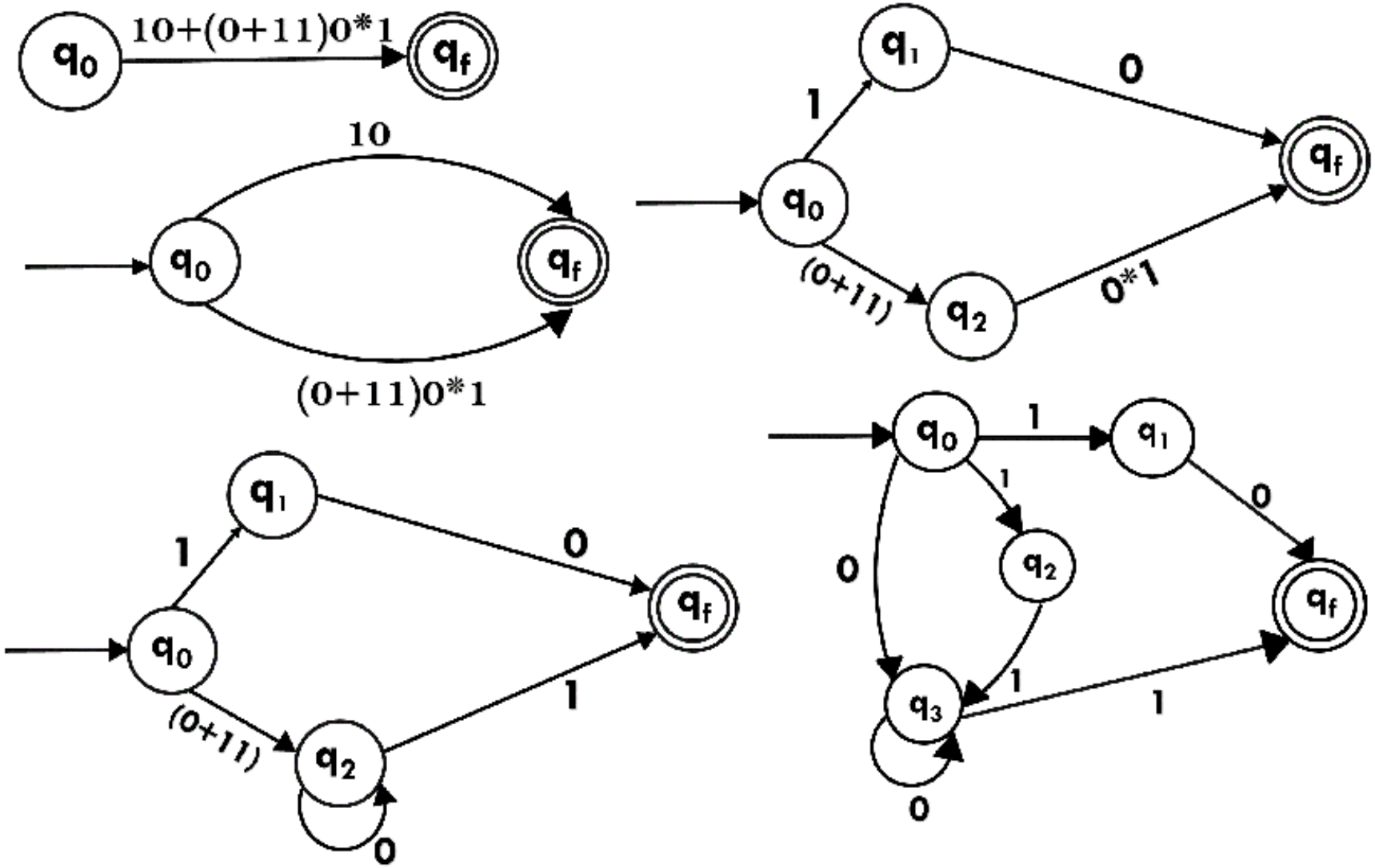


		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

← Accept, since final state reached

Example NFA

regular expression 10 + (0 + 11)0* 1



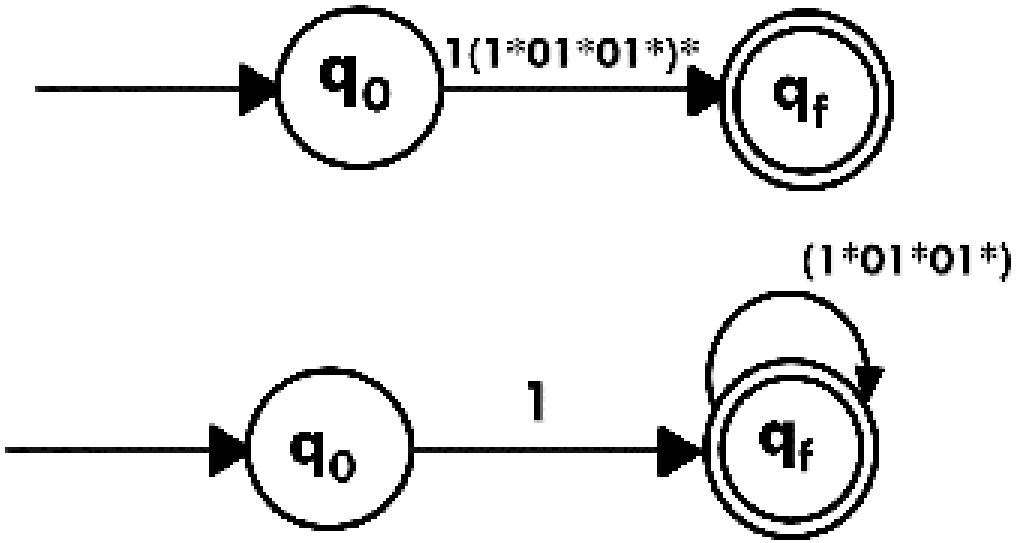
Example NFA

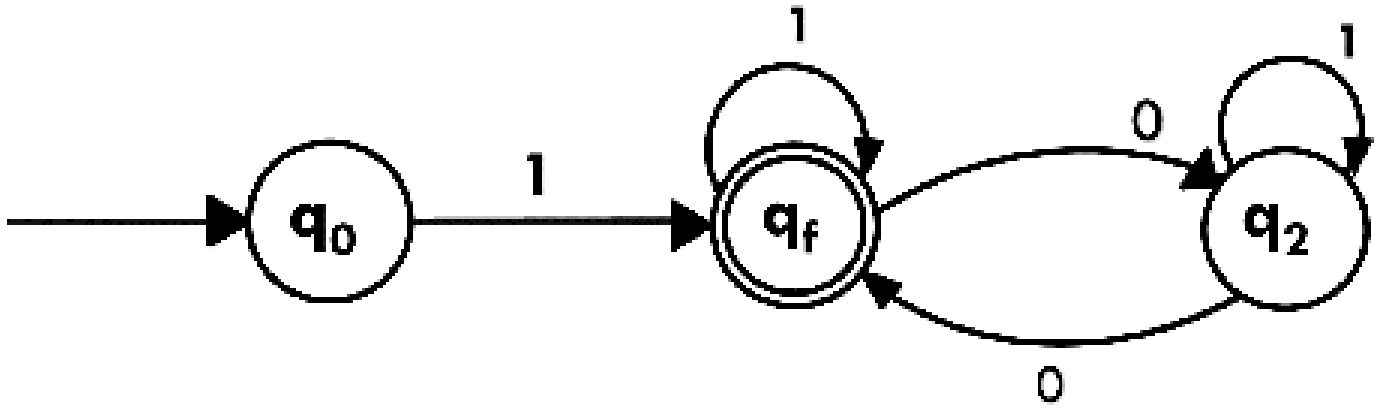
regular expression 10 + (0 + 11)0* 1

State	0	1
→q0	q3	{q1, q2}
q1	qf	φ
q2	φ	q3
q3	q3	qf
*qf	φ	φ

Example NFA

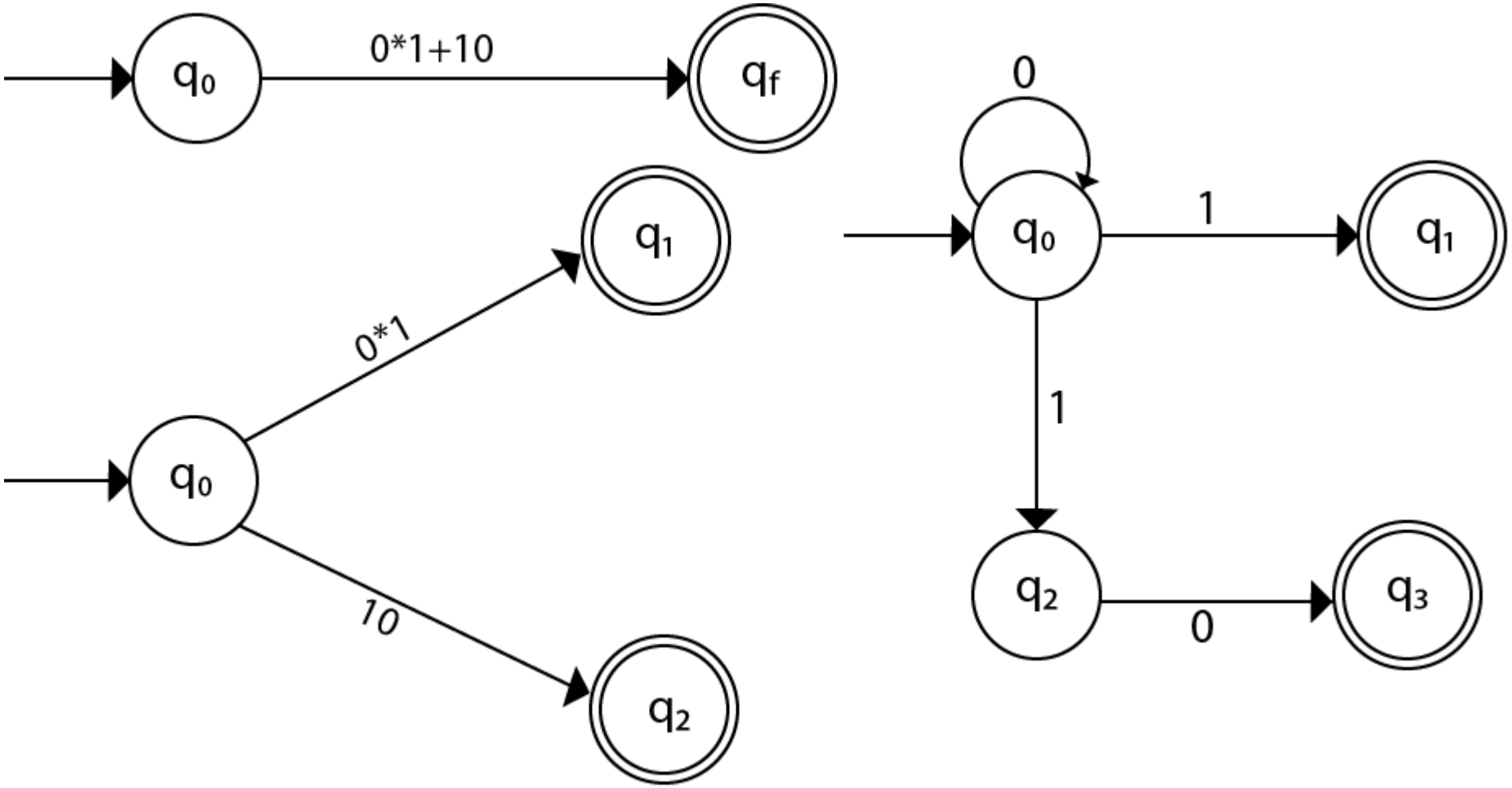
regular expression 1 (1* 01* 01*)*





Example NFA

regular expression 0*1 + 10



Converting NFA to DFA

Step-01:
Let Q' be a new set of states of the DFA. Q' is null in the starting.
Let T' be a new transition table of the DFA.

Step-02:
Add start state of the NFA to Q'.
Add transitions of the start state to the transition table T'.
If start state makes transition to multiple states for some input alphabet, then treat those multiple states as a single state in the DFA

Step-03:
If any new state is present in the transition table T',
Add the new state in Q'.
Add transitions of that state in the transition table T'.

Step-04:
Keep repeating Step-03 until no new state is present in the transition table T'.
Finally, the transition table T' so obtained is the complete transition table of the required DFA.

Converting NFA to DFA: Important Points

After conversion, the number of states in the resulting DFA may or may not be same as NFA.

The maximum number of states that may be present in the DFA are $2^{\text{Number of states in the NFA}}$.

In general, the following relationship exists between the number of states in the NFA and DFA-

$$\mathbf{1 <= n <= 2^m}$$

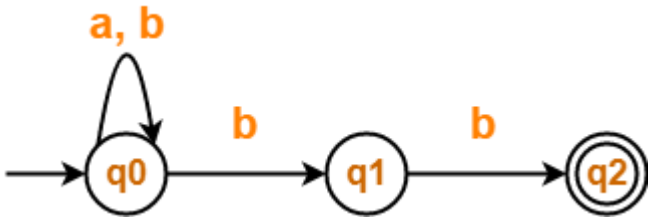
Here,

n = Number of states in the DFA

m = Number of states in the NFA

In the resulting DFA, all those states that contain the final state(s) of NFA are treated as final states.

Converting NFA to DFA: Example

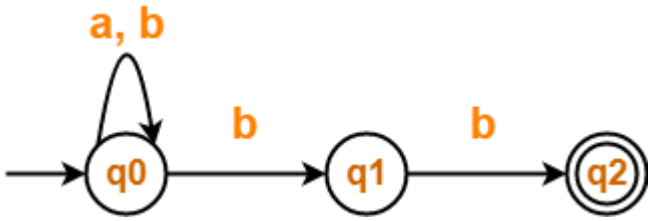


State / Input	a	b
→**q0**	q0	q0, q1
q1	–	*q2
***q2**	–	–

State / Input	a	b
→**q0**	q0	{q0, q1}

State / Input	a	b
→**q0**	q0	{q0, q1}
{q0, q1}	q0	{q0, q1, q2}

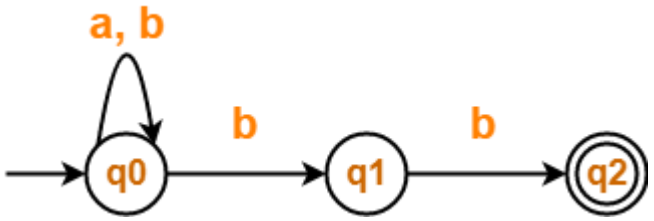
Converting NFA to DFA: Example



State / Input	a	b
→**q0**	q0	q0, q1
q1	–	*q2
***q2**	–	–

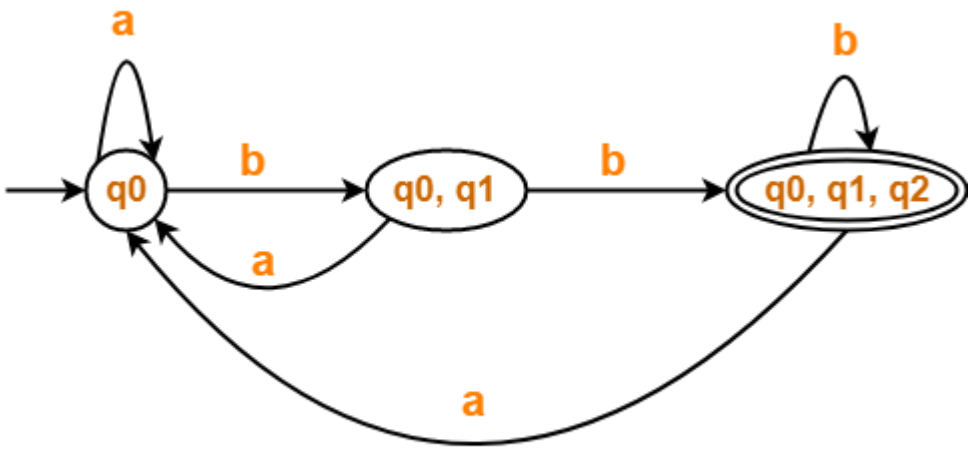
State / Input	a	b
→**q0**	q0	{q0, q1}
{q0, q1}	q0	{q0, q1, q2}
{q0, q1, q2}	q0	{q0, q1, q2}

Converting NFA to DFA: Example



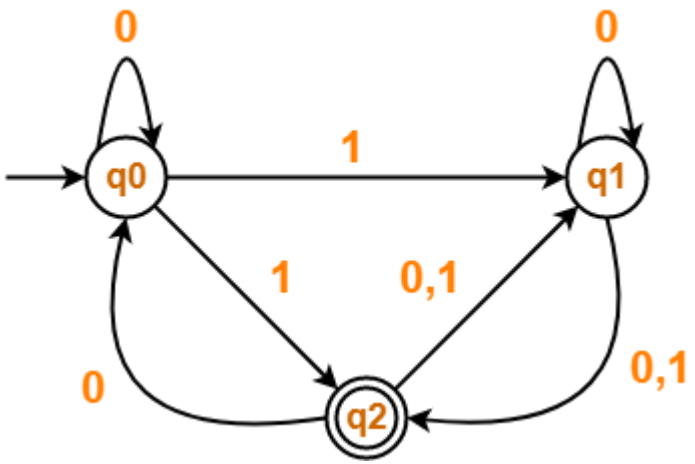
State / Input	a	b
→**q0**	q0	q0, q1
q1	–	*q2
***q2**	–	–

State / Input	a	b
→**q0**	q0	{q0, q1}
{q0, q1}	q0	*{q0, q1, q2}
***{q0, q1, q2}**	q0	*{q0, q1, q2}



Deterministic Finite Automata (DFA)

Converting NFA to DFA: Example



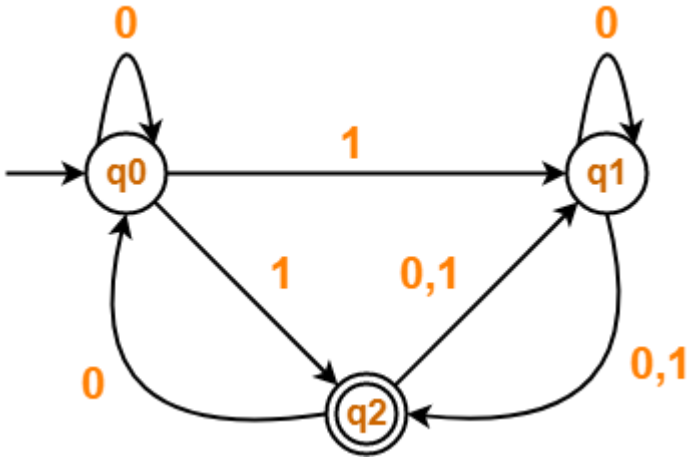
State / Alphabet	0	1
→**q0**	q0	q1, *q2
q1	q1, *q2	*q2
***q2**	q0, q1	q1

State / Alphabet	0	1
→**q0**	q0	{q1, q2}

State / Alphabet	0	1
→**q0**	q0	{q1, q2}
{q1, q2}	{q0, q1, q2}	{q1, q2}

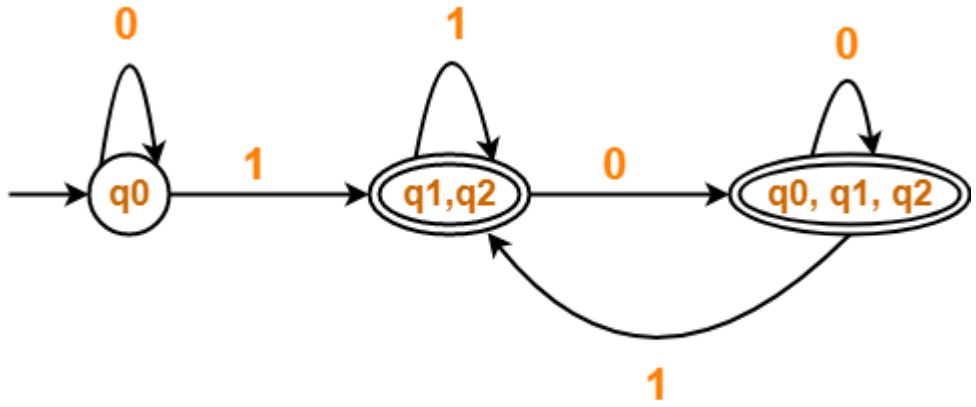
State / Alphabet	0	1
→**q0**	q0	{q1, q2}
{q1, q2}	{q0, q1, q2}	{q1, q2}
{q0, q1, q2}	{q0, q1, q2}	{q1, q2}

Converting NFA to DFA: Example



State / Alphabet	0	1
→**q0**	q0	q1, *q2
q1	q1, *q2	*q2
***q2**	q0, q1	q1

State / Alphabet	0	1
→**q0**	q0	*{q1, q2}
***{q1, q2}**	*{q0, q1, q2}	*{q1, q2}
***{q0, q1, q2}**	*{q0, q1, q2}	*{q1, q2}



Deterministic Finite Automata (DFA)

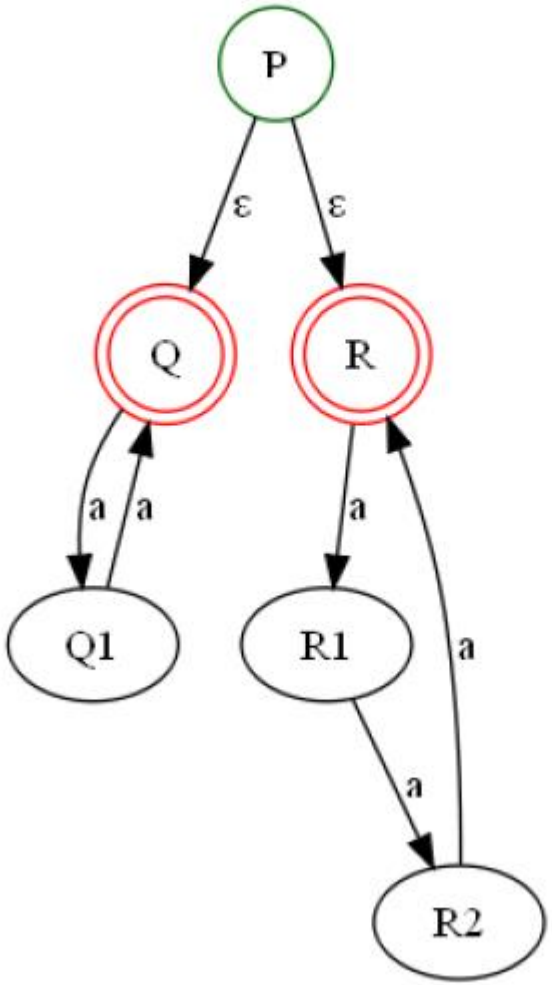
ϵ-NFA

We extend the class of NFAs by allowing instantaneous (ε) transitions:

1. The automaton may be allowed to change its state without reading the input symbol.
2. In diagrams, such transitions are depicted by labeling the appropriate arcs with ε.
3. Note that this does not mean that ε has become an input symbol. On the contrary, we assume that *the symbol* ε *does not belong to any alphabet*.

ϵ-NFA - Example

$\{ a^n \mid n \text{ is even or divisible by 3} \}$



ϵ-NFA - Definition

A ε-NFA is a quintuple $\mathbf{A=(Q,\Sigma,\delta,q_0,F)}$ where

- $\mathbf{Q}$ is a set of *states*
- $\mathbf{\Sigma}$ is the alphabet of *input symbols*
- $\mathbf{q_0 \in Q}$ is the *initial state*
- $\mathbf{F \subseteq Q}$ is the set of *final states*
- $\mathbf{\delta}$: $\mathbf{Q} \times \Sigma_\varepsilon \longrightarrow P(\mathbf{Q})$ is the *transition function*

Note ε is never a member of Σ

Σ_ε is defined to be $(\Sigma \cup \varepsilon)$

ϵ-NFA

ε -NFAs add a convenient feature but (in a sense) they bring us nothing new: they do not extend the class of languages that can be represented. Both NFAs and ε-NFAs recognize exactly the same languages.

ε-transitions are a convenient feature: try to design an NFA for the even or divisible by 3 language that does not use them!

- Hint, you need to use something like the product construction from union-closure of DFAs

ϵ-Closure Recursive Definition

ECLOSE(q) is as follows

BASIS: State q is in ECLOSE

INDUCTION: If state p is in ECLOSE(q) and there is a transition from state p to state r labelled ϵ, then r is in ECLOSE(q).

If δ is the transition function of the ϵ-NFA involves, and p is in ECLOSE(q), then ECLOSE(q) also contains the states in δ(p, ϵ).

ϵ-Closure

ε-closure of a state

The ε-*closure* of the state q, denoted ECLOSE(q), is the set that contains q, together with all states that can be reached starting at q *by following only* ε-*transitions*.

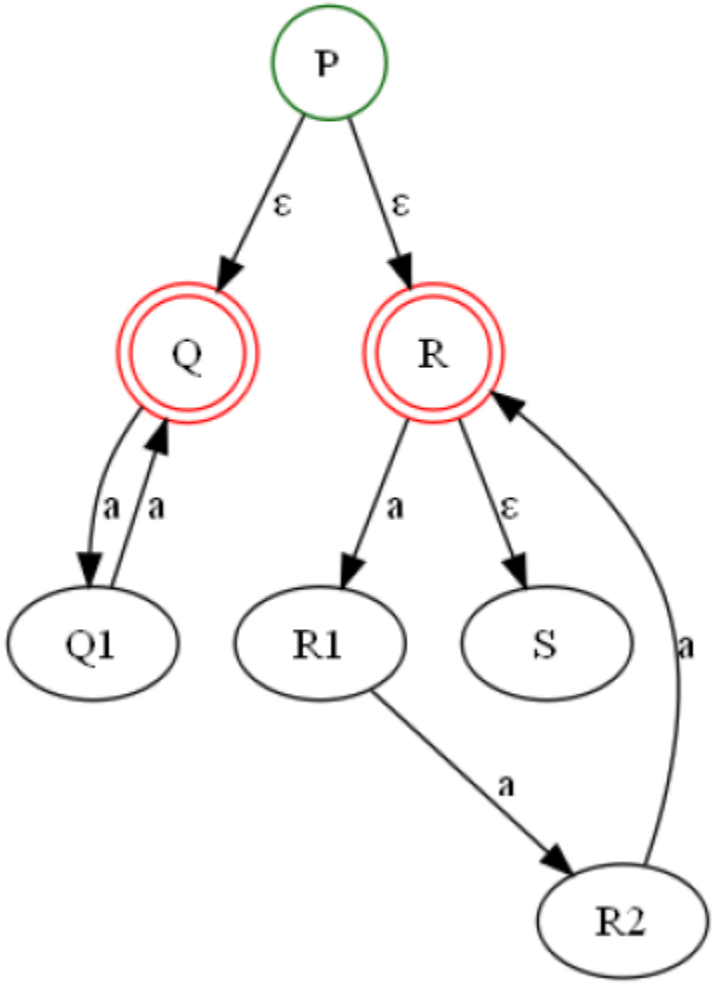
In the above example:

ECLOSE(P) ={P,Q,R,S}

ECLOSE(R)={R,S}

ECLOSE(x)={x} for the remaining 5 states

{Q,Q1,R1,R2,R2}



ϵ-Closure

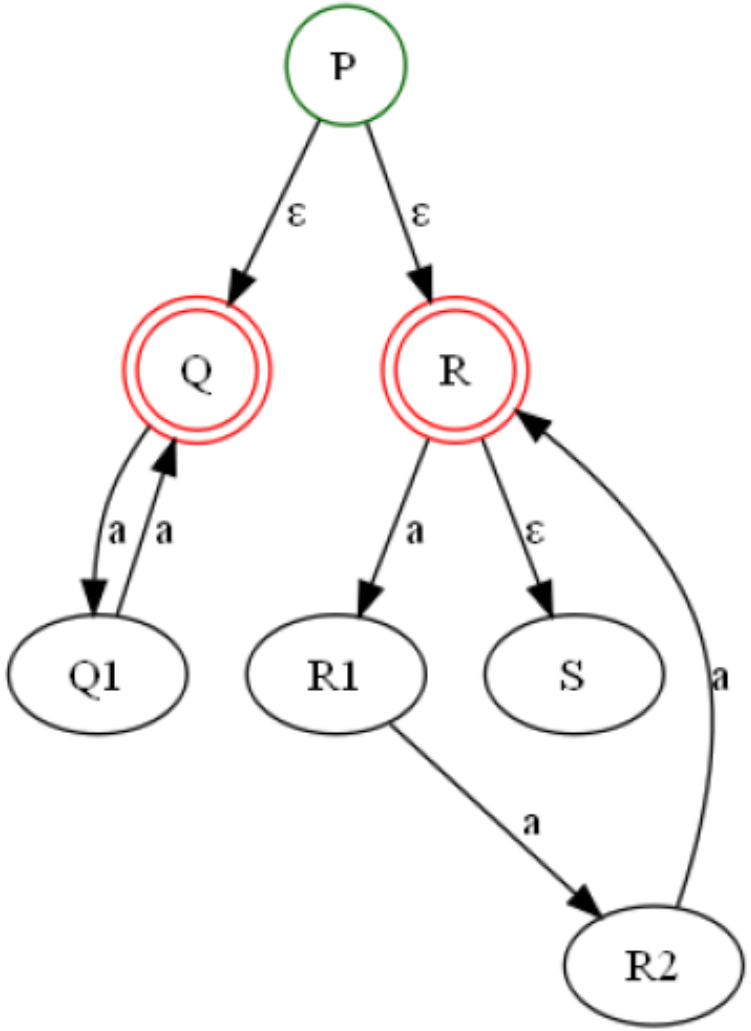
Compute eclose by adding new states until no new states can be added

Start with [P]

Add Q and R to get [P,Q,R]

Add S to get [P,Q,R S]

No new states can be added



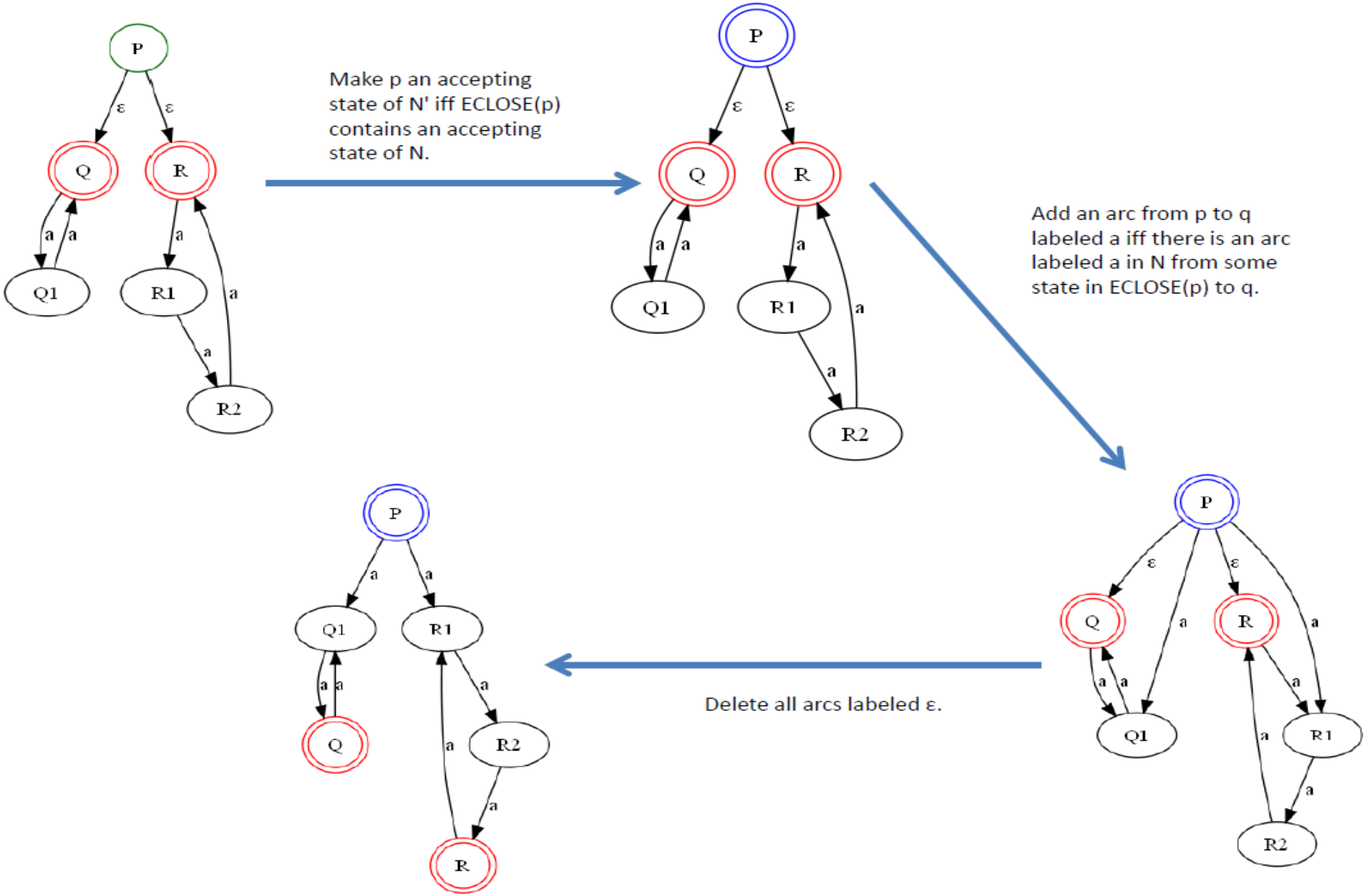
Elimination of ϵ-transition

Given an ε-NFA N, this construction produces an NFA N' such that L(N')=L(N).

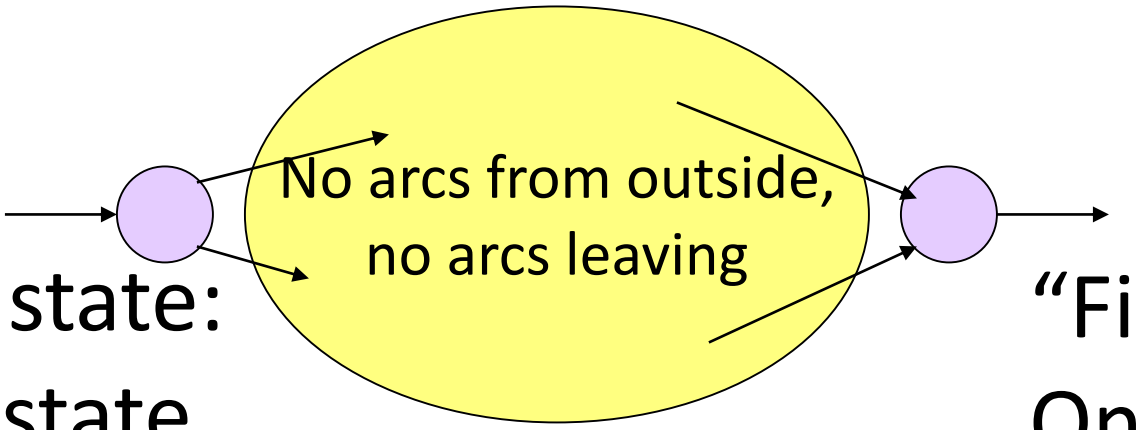
The construction of N' begins with N as input, and takes 3 steps:

1. Make p an accepting state of N' iff ECLOSE(p) contains an accepting state of N.
2. Add an arc from p to q labeled a iff there is an arc labeled a in N from some state in ECLOSE(p) to q.
3. Delete all arcs labeled **ε**.

Elimination of ϵ-transition



Form of ε-NFA's Constructed



Start state:
Only state
with external
predecessors

"Final" state:
Only state
with external
successors

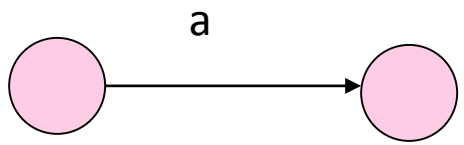
Why does it work?

The language accepted by the automaton is being preserved during the three steps of the construction: $L(N)=L(N_1)=L(N_2)=L(N_3)$

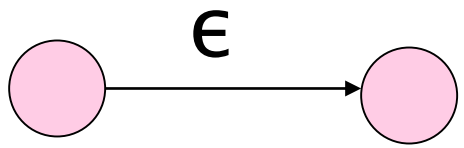
Each step here requires a proof. A Good exercise for you to do!

RE to ε-NFA: Basis

- Symbol **a**:

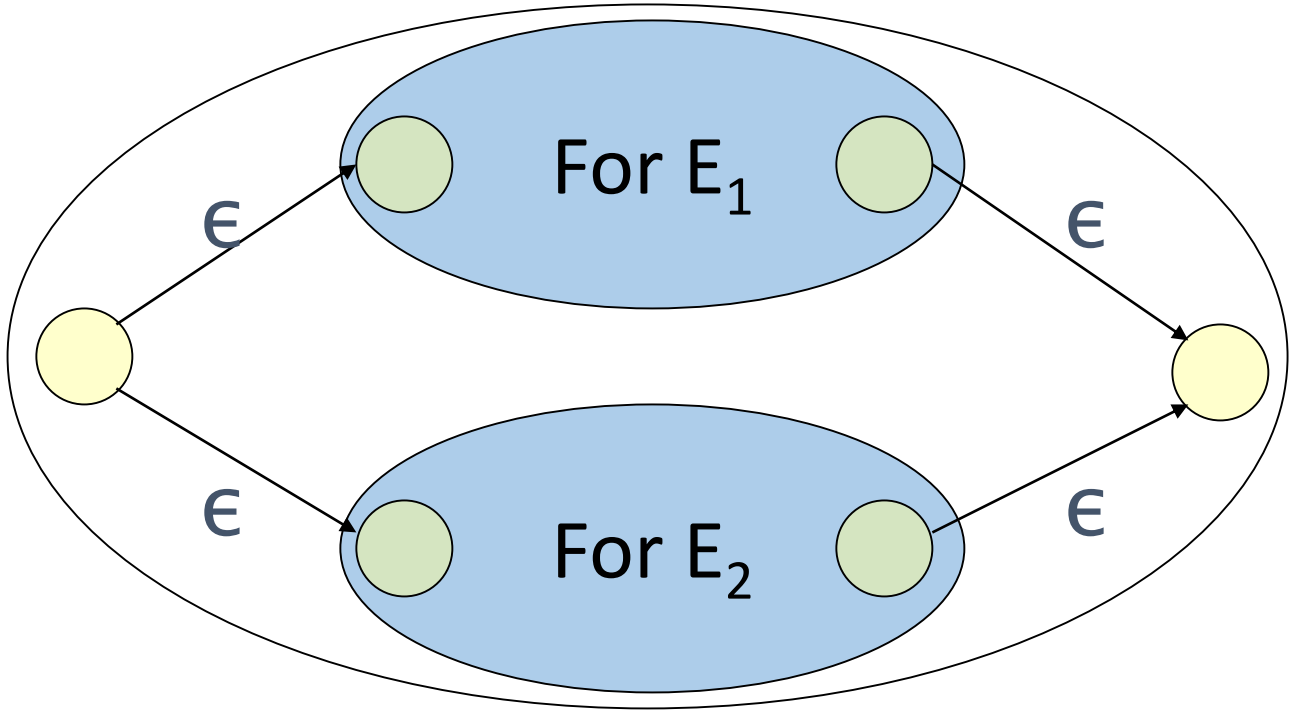


- ϵ:



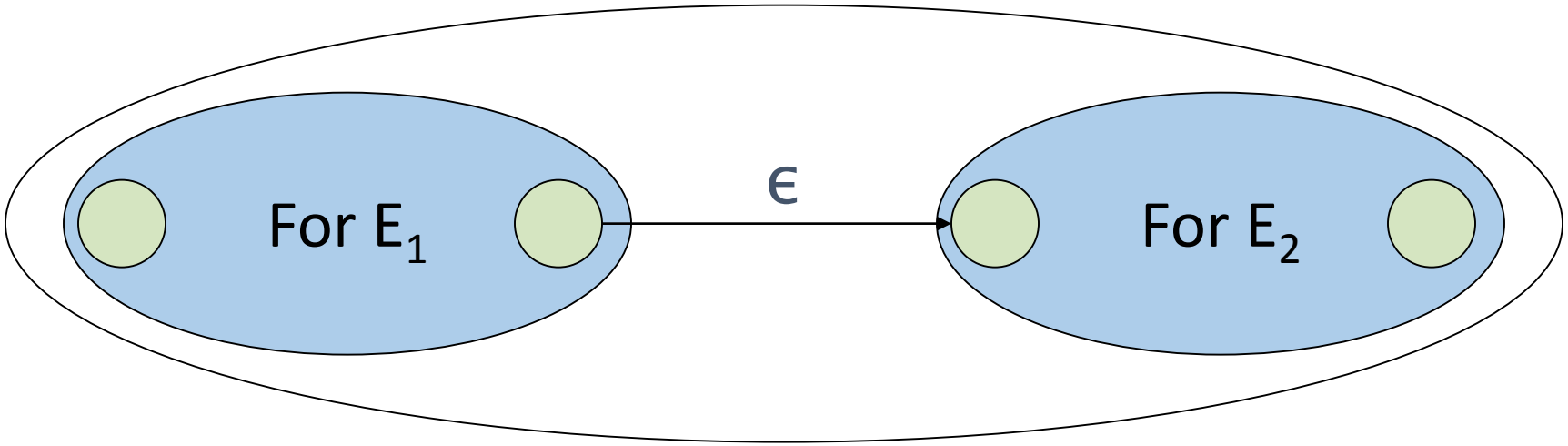
- ∅:

RE to ε-NFA: Union



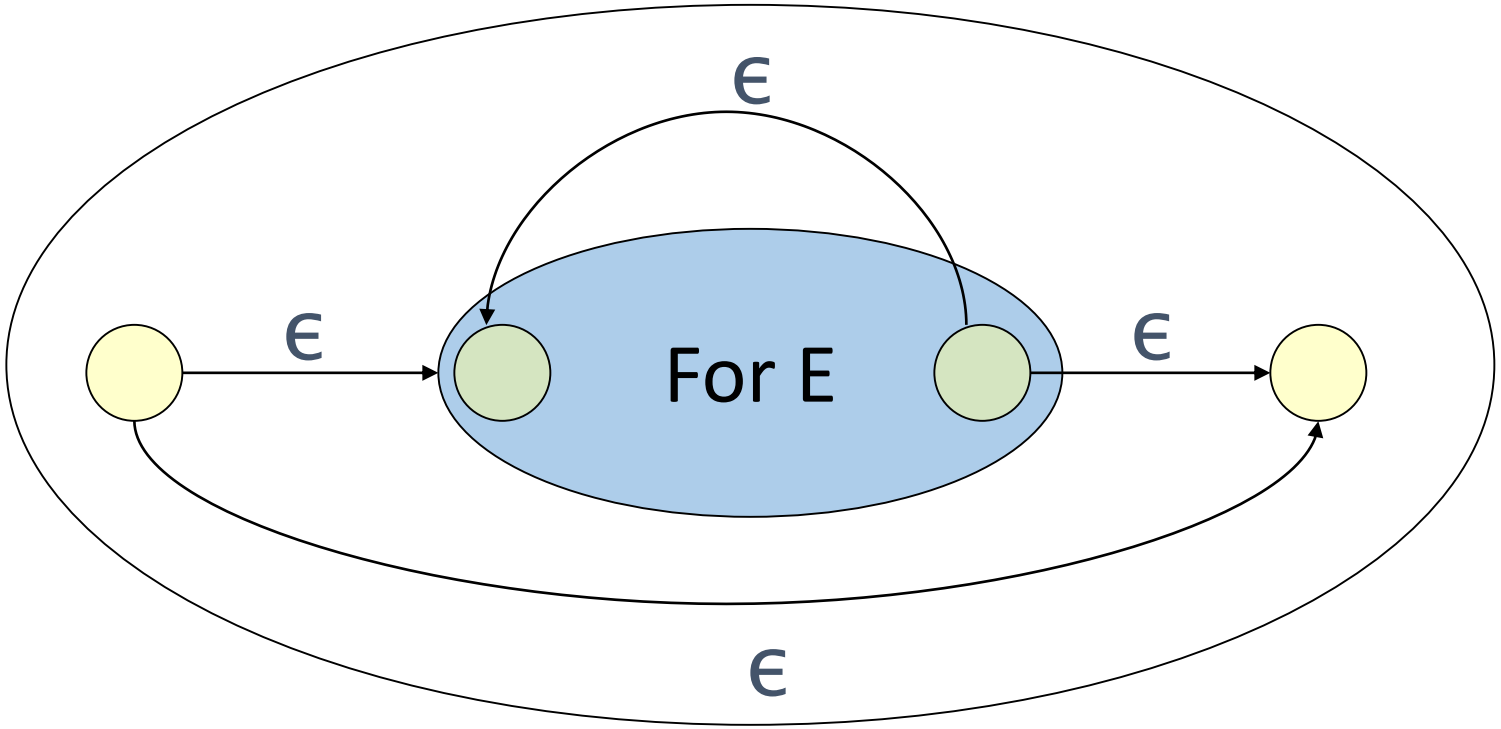
For $E_1 \cup E_2$

RE to ε-NFA: Concatenation



For E_1E_2

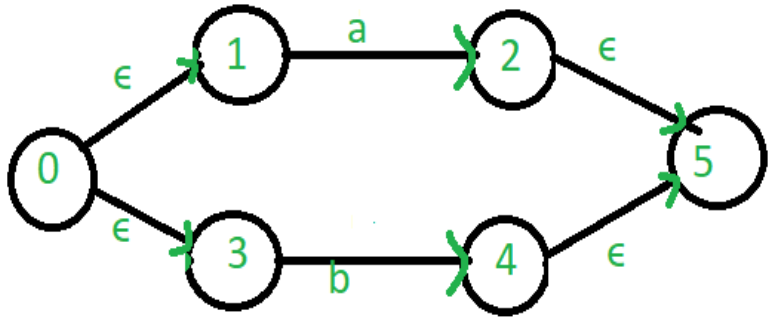
RE to ε-NFA: Closure



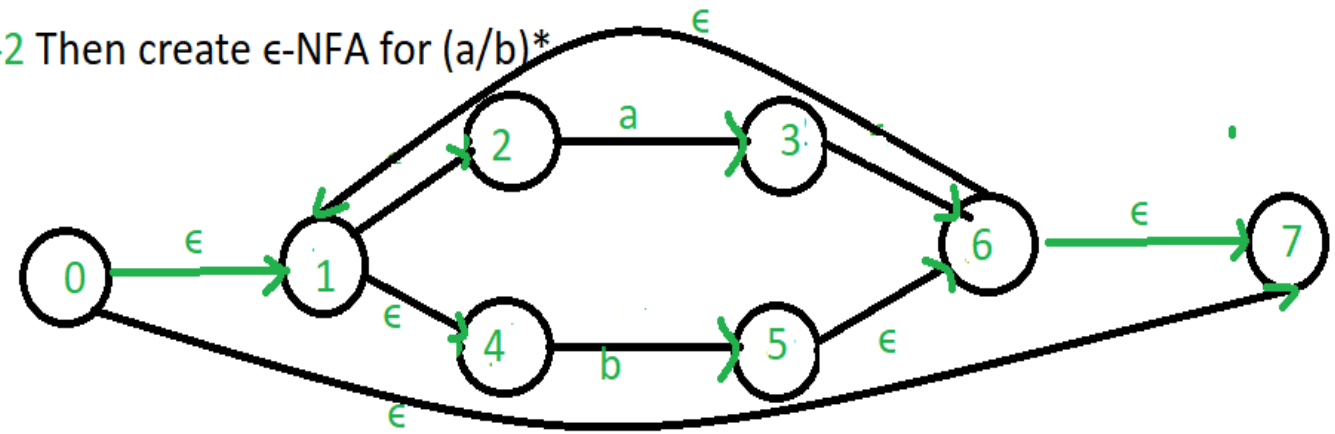
For E*

RE to ε-NFA

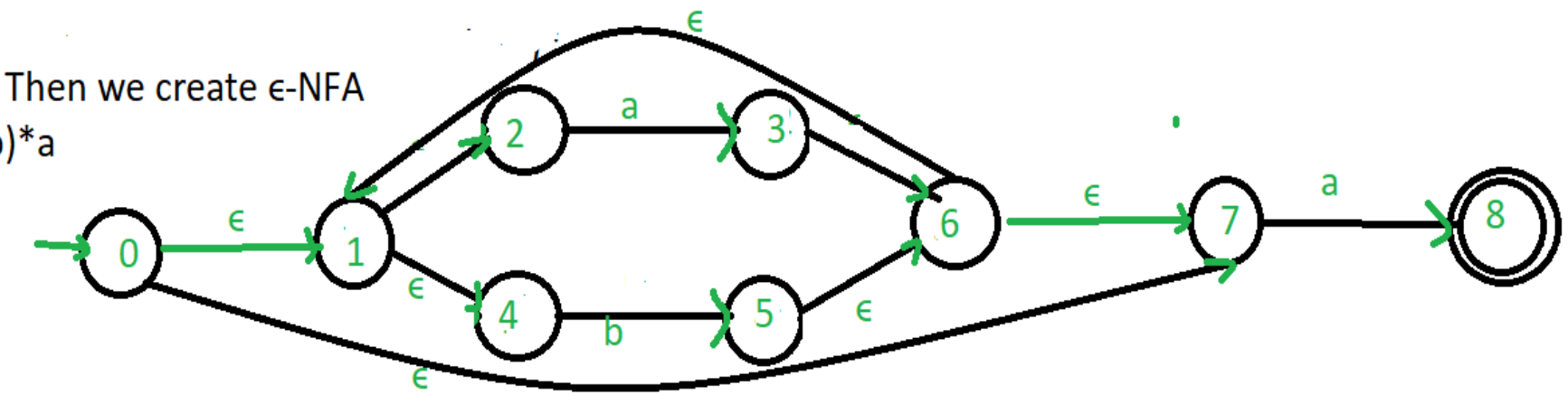
Step-1 First we create ϵ-NFA for (a/b)



Step-2 Then create ϵ-NFA for (a/b)*

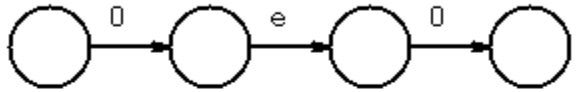


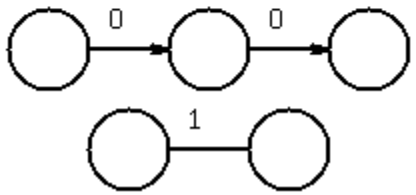
Step-3 Then we create ϵ-NFA for(a/b)*a

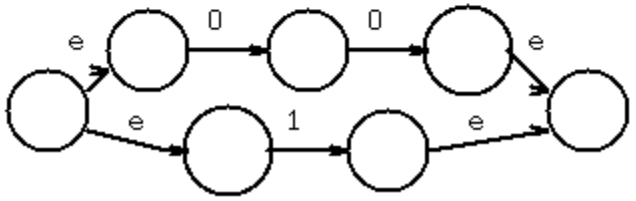


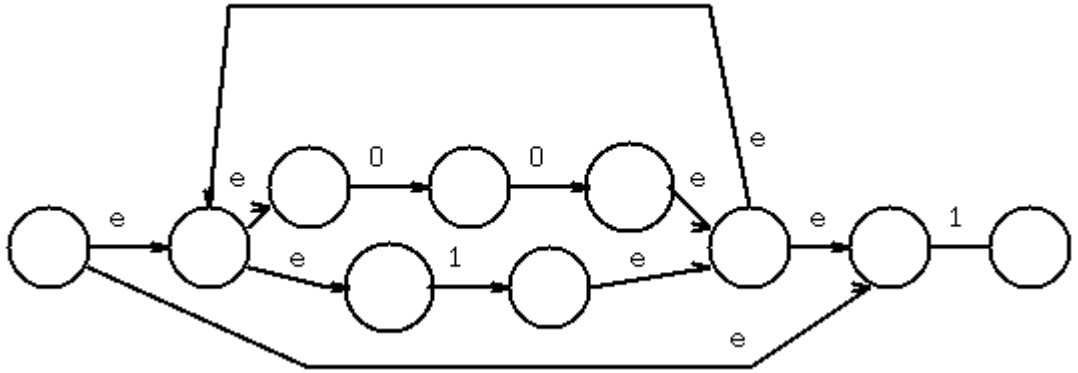
RE to ε-NFA

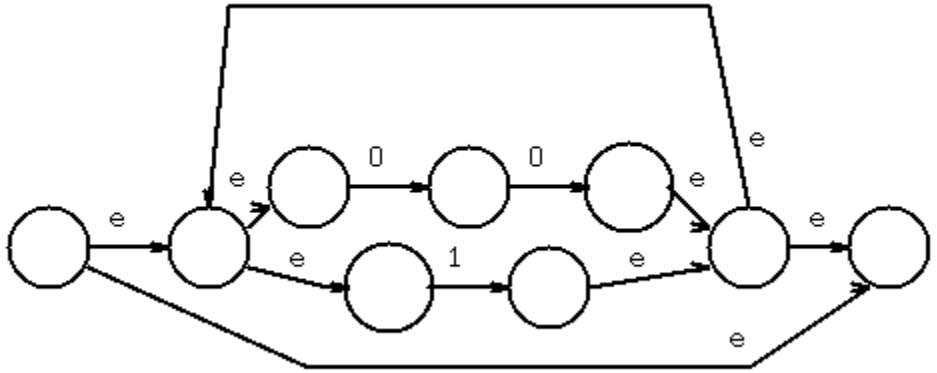
Convert (00 + 1)* 1 (0 +1) to a NFA-epsilon

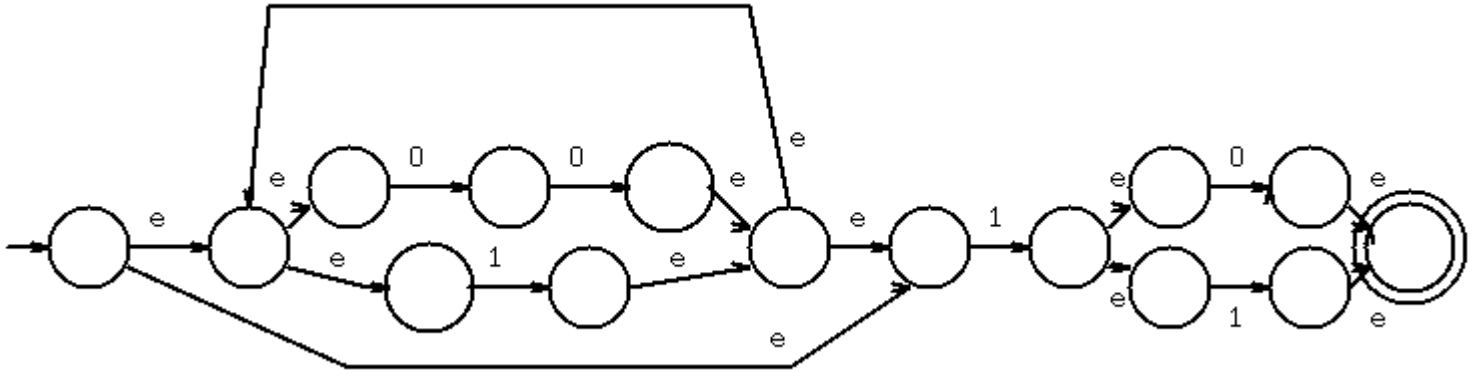












RE to ε-NFA

From regular expression to NFA (Thompson's Construction):

Given a regular expression r over an alphabet Σ, build an NFA N that accepts $L(r)$.
Key: follow the rules that form a regular expression and using the following rules to build an NFA gradually.

1. ε regular expression, construct $N(\varepsilon)$
2. for each symbol a in Σ, construct $N(a)$
3. for regular expression s and t and their corresponding NFA $N(s)$ and $N(t)$
    a. regular expression $s|t$, construct $N(s|t)$
    b. regular expression st, construct $N(st)$
    c. regular expression s^*, construct $N(s^*)$
    d. regular expression (s), construct $N(s)$

The NFA $N(r)$ created following the above procedure has the following properties:

1. the number of states of $N(r)$ is less than or equal to twice of the number of symbols and operators in r.
2. all the NFA's constructed have one start state and one final state.
3. each state in $N(r)$ has either one outgoing transition on a symbol or at most two outgoing ε-transition.

RE to ε-NFA

Simulating an NFA

Input: an NFA *N*, an input string *x*, and a set *F* of final states.

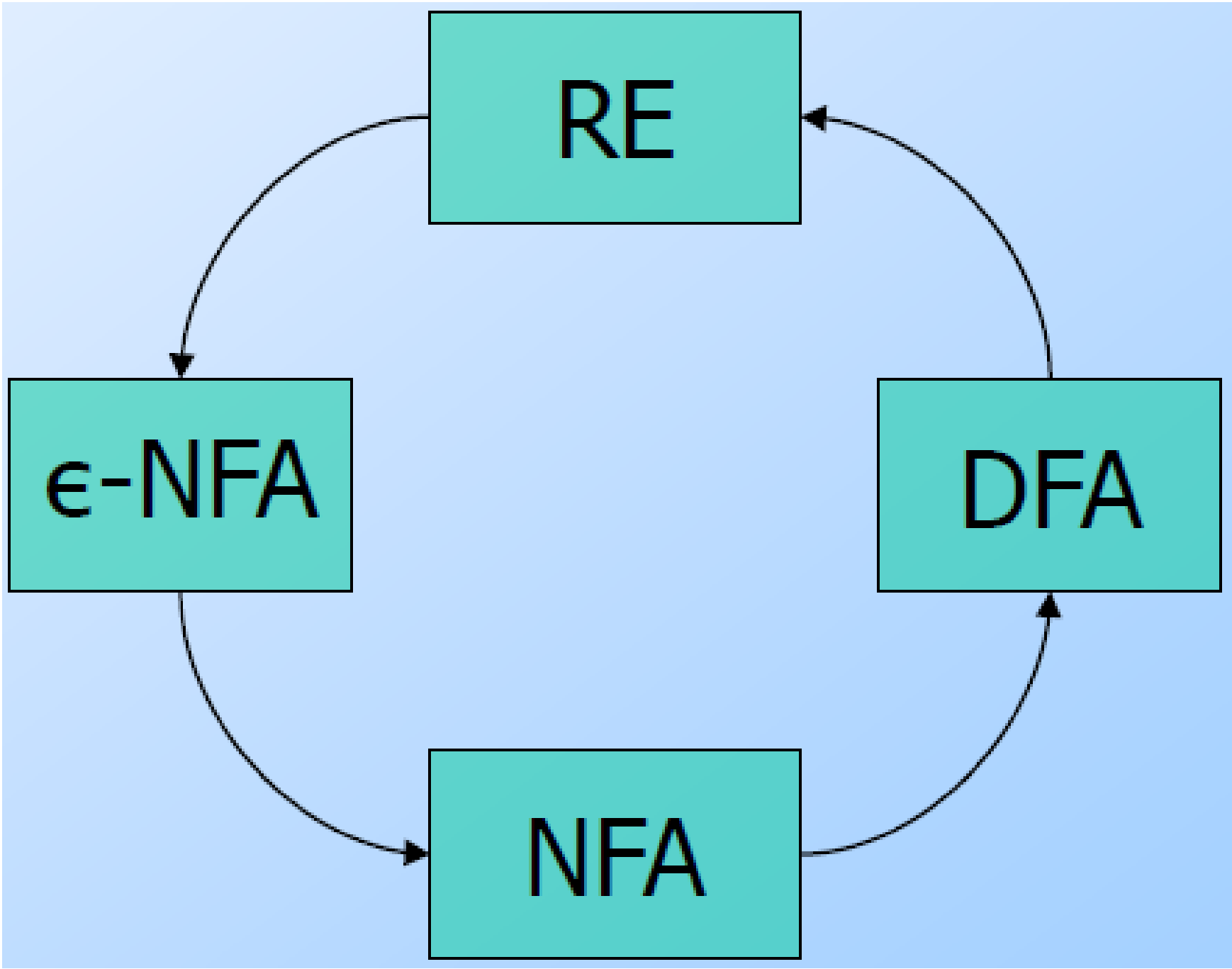
Output: "yes" if *N* accepts *x*; otherwise "no".

```
S := ε-closure({s0})
a := nextChar();
while a != EOF do
        S := ε-closure(move(S, a));
        a := nextChar();
end;
if S ∩ F != ∅ then
        return "yes";
else
        return "no";
```

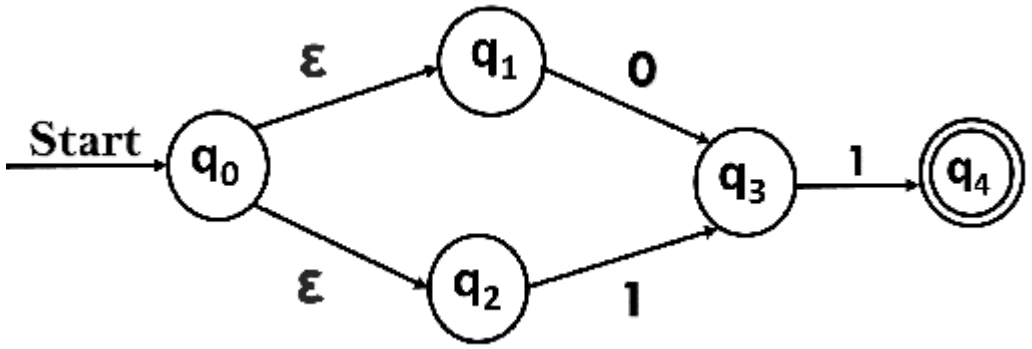
Time complexity is O(|N| × |x|) where |N| is the number of states in N and |x| is the length of x. N has at most twice as many states as |r|. Thus, space complexity is O(|r|).

If we convert NFA to DFA, it generates at most $2^{2|r|}$ states. Therefore, the space complexity for the DFA is $O(2^{2|r|})$. The time complexity is O(|x|).

Relations



ϵ-NFA to DFA



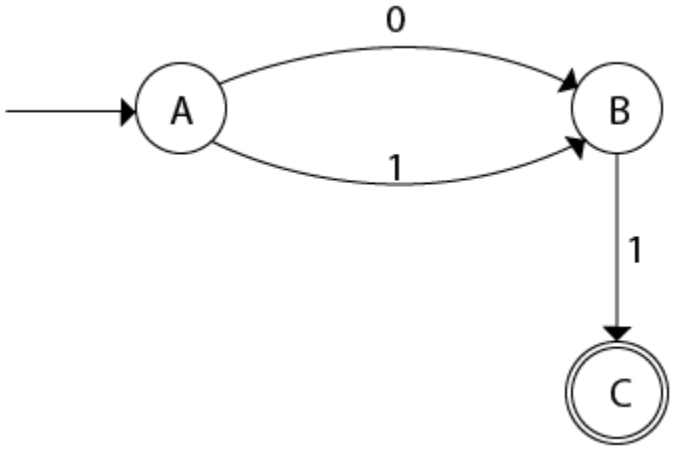
ε-closure {q0} = {q0, q1, q2}

δ'(A, 0) = ε-closure {δ((q0, q1, q2), 0) }
= ε-closure {δ(q0, 0) ∪ δ(q1, 0) ∪ δ(q2, 0) }
= ε-closure {q3}
= {q3} call it as state B.

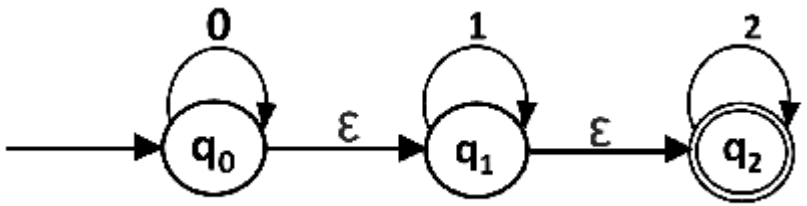
δ'(A, 1) = ε-closure {δ((q0, q1, q2), 1) }
= ε-closure {δ((q0, 1) ∪ δ(q1, 1) ∪ δ(q2, 1) }
= ε-closure {q3}
= {q3} = B.

δ'(B, 0) = ε-closure {δ(q3, 0) }
= φ
δ'(B, 1) = ε-closure {δ(q3, 1) }
= ε-closure {q4}
= {q4} i.e. state C

δ'(C, 0) = ε-closure {δ(q4, 0) }
= ϕ
δ'(C, 1) = ε-closure {δ(q4, 1) }
= ϕ



ϵ-NFA to DFA



ε-closure(q0) = {q0, q1, q2}

ε-closure(q1) = {q1, q2}

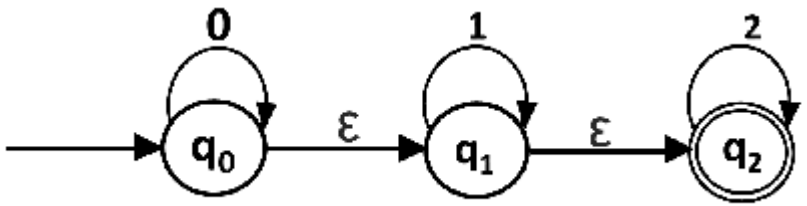
ε-closure(q2) = {q2}

δ'(A, 0) = ε-closure{δ((q0, q1, q2), 0)}
= ε-closure{δ(q0, 0) ∪ δ(q1, 0) ∪ δ(q2, 0)}
= ε-closure{q0}
= {q0, q1, q2}

δ'(A, 1) = ε-closure{δ((q0, q1, q2), 1)}
= ε-closure{δ(q0, 1) ∪ δ(q1, 1) ∪ δ(q2, 1)}
= ε-closure{q1}
= {q1, q2} call it as state B

δ'(A, 2) = ε-closure{δ((q0, q1, q2), 2)}
= ε-closure{δ(q0, 2) ∪ δ(q1, 2) ∪ δ(q2, 2)}
= ε-closure{q2}
= {q2} call it state C

ϵ-NFA to DFA



$$\delta'(B, 0) = \varepsilon\text{-closure}\{\delta((q1, q2), 0)\}$$
$$= \varepsilon\text{-closure}\{\delta(q1, 0) \cup \delta(q2, 0)\}$$
$$= \varepsilon\text{-closure}\{\varphi\}$$
$$= \varphi$$

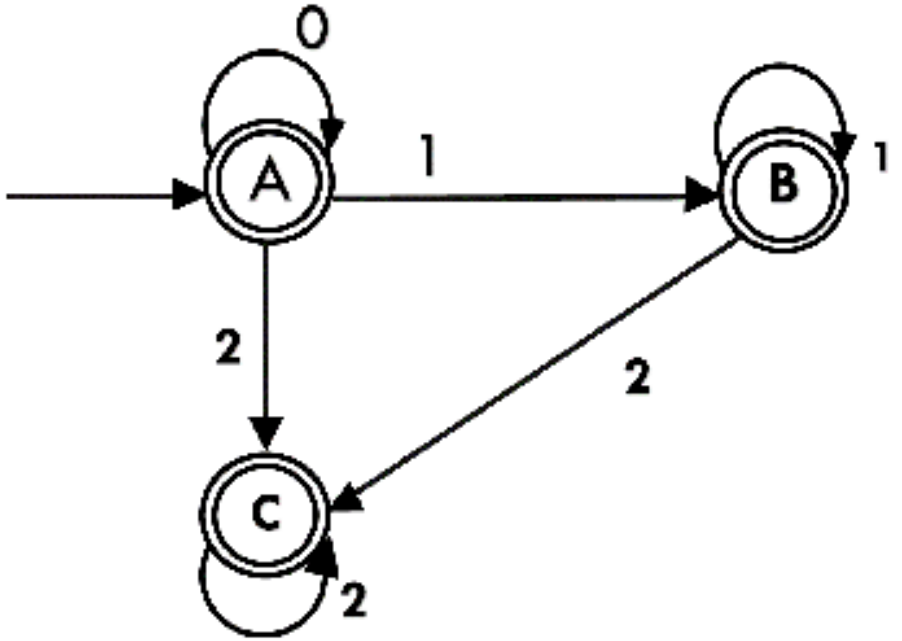
$$\delta'(B, 1) = \varepsilon\text{-closure}\{\delta((q1, q2), 1)\}$$
$$= \varepsilon\text{-closure}\{\delta(q1, 1) \cup \delta(q2, 1)\}$$
$$= \varepsilon\text{-closure}\{q1\}$$
$$= \{q1, q2\} \quad \text{i.e. state B itself}$$

$$\delta'(B, 2) = \varepsilon\text{-closure}\{\delta((q1, q2), 2)\}$$
$$= \varepsilon\text{-closure}\{\delta(q1, 2) \cup \delta(q2, 2)\}$$
$$= \varepsilon\text{-closure}\{q2\}$$
$$= \{q2\} \quad \text{i.e. state C itself}$$

$$\delta'(C, 0) = \varepsilon\text{-closure}\{\delta(q2, 0)\}$$
$$= \varepsilon\text{-closure}\{\varphi\}$$
$$= \varphi$$

$$\delta'(C, 1) = \varepsilon\text{-closure}\{\delta(q2, 1)\}$$
$$= \varepsilon\text{-closure}\{\varphi\}$$
$$= \varphi$$

$$\delta'(C, 2) = \varepsilon\text{-closure}\{\delta(q2, 2)\}$$
$$= \{q2\}$$



NFA to DFA subset construction

Input: An NFA N.
Output: A DFA D accepting the same language.

Operations:
ε-closure(s) is the set of NFA states reachable from s on ε-transitions alone.
ε-closure(T) is the union of *ε-closure(r)* for all *r* in *T*.
move(T, a) is the set of NFA states to which there is a transition on input *a* from some NFA state in T.

```
set the start state to ε-closure(s0) and unmark it.
While there is an unmarked state T in Dstates do
        Mark T
        For each input symbol a do
                If U := ε-closure(move(T, a));
                If U is not in Dstates then
                        Add U as an unmarked state to Dstates;
                Dtran(T, a) := U;
        End;
End;
```

NFA to DFA subset construction

```
Calculate ε-closure:
Push all states in T onto stack;
Initialize ε-closure(T) to T;
While stack <> empty do
        Pop t;
        For each transition (t, u) in ε-transition do
                If u is not in ε-closure(T) then
                        Add u to ε-closure(T);
                        Push u;
                End;
End;
```

Equivalence of RE's and Automata

- We need to show that for every RE, there is an automaton that accepts the same language.
  - Pick the most powerful automaton type: the ϵ-NFA.

- And we need to show that for every automaton, there is a RE defining its language.
  - Pick the most restrictive type: the DFA.

Generalized Transition Graph (GTG)

- Any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state

- A GTG is a transition graph whose edges are labeled with RE; otherwise it is same as the usual transition graph

- The label of any walk from the initial state to a final state is the concatenation of several regular expressions, hence itself a regular expression

Generalized Transition Graph (GTG)

- The strings denoted by such RE are a subset of the language accepted by the GTG with the full language being the union of all such generated subsets

- If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with Ø

- A complete GTG with |V| vertices has exactly $|V|^2$ edges

Generalized Transition Graph (GTG)

procedure: nfa-to-rex

1. Start with an nfa with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.

2. Convert the nfa into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.

3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is

$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*.$$

Generalized Transition Graph (GTG)

4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled

$$r_{pq} + r_{pk} r_{kk}^* r_{kq}$$

for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.

5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states $(q_i, q_j), i \neq k, j \neq k$. At each step apply the simplifying rules

$$r + \varnothing = r,$$

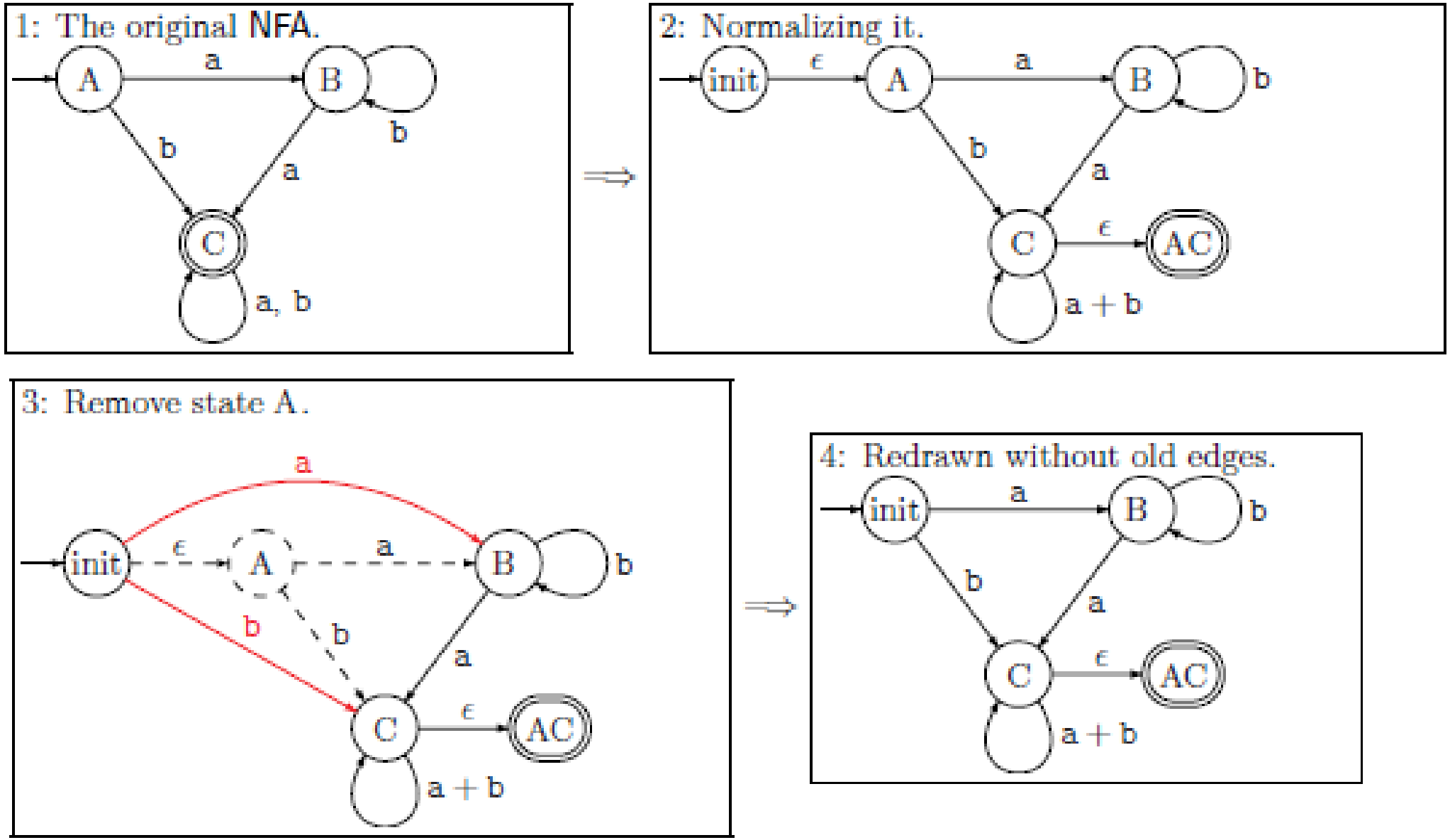
$$r\varnothing = \varnothing,$$

$$\varnothing^* = \lambda,$$

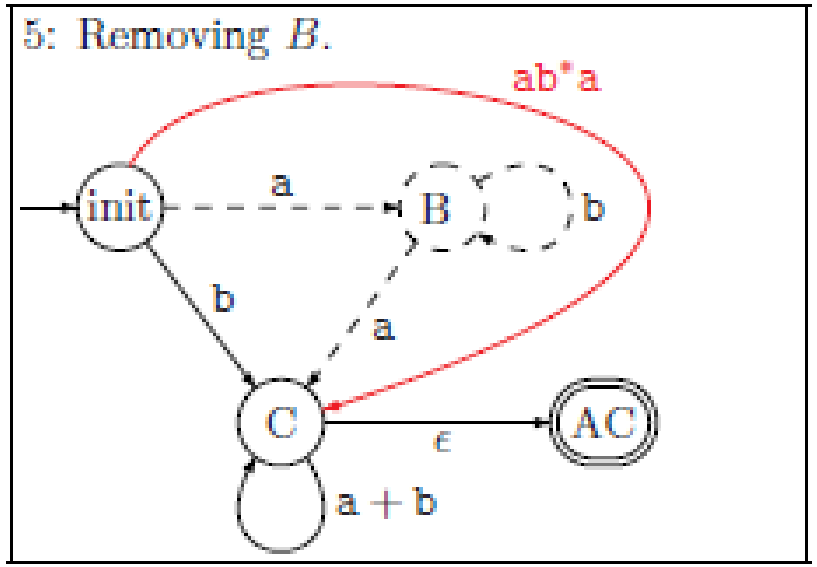
wherever possible. When this is done, remove state q_k.

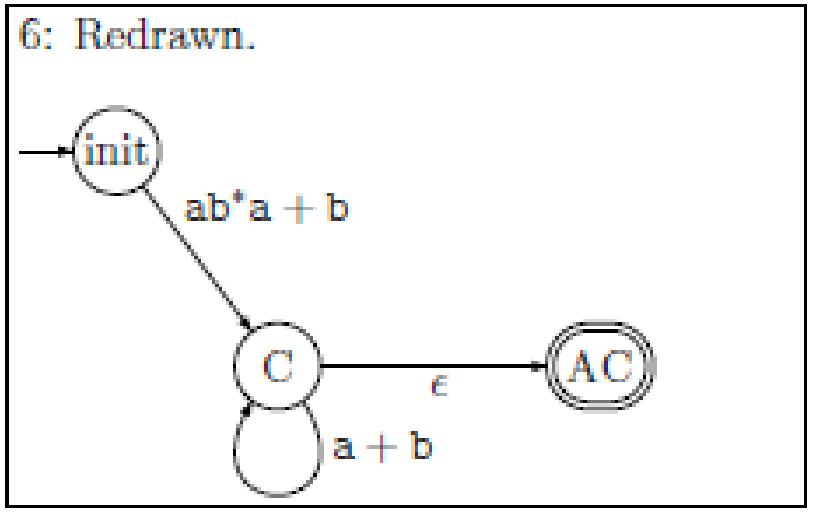
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

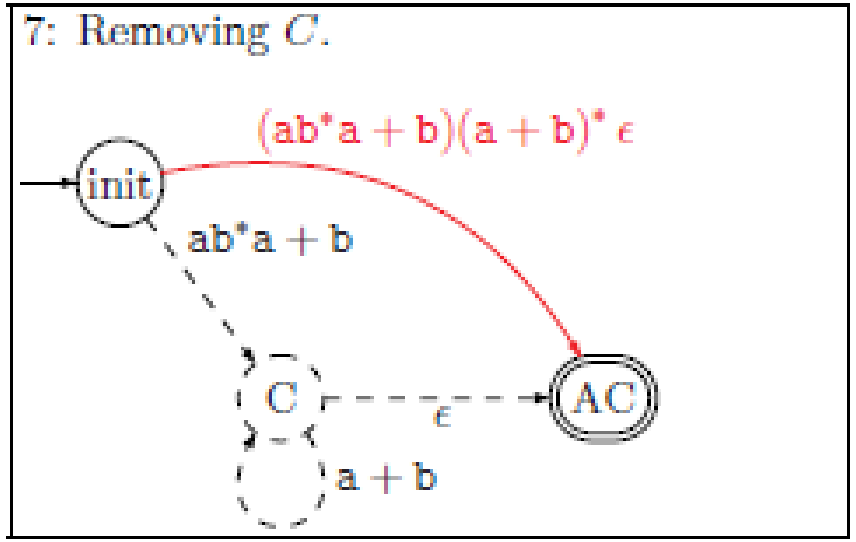
Generalized Transition Graph (GTG)

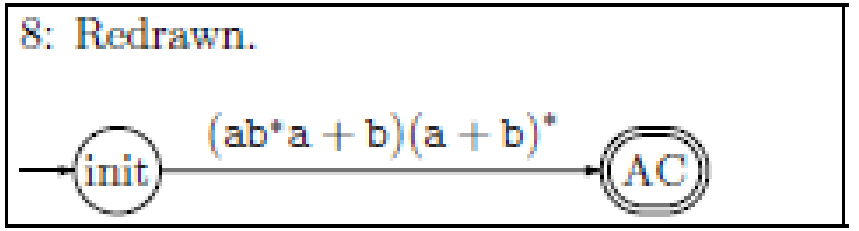


Generalized Transition Graph (GTG)



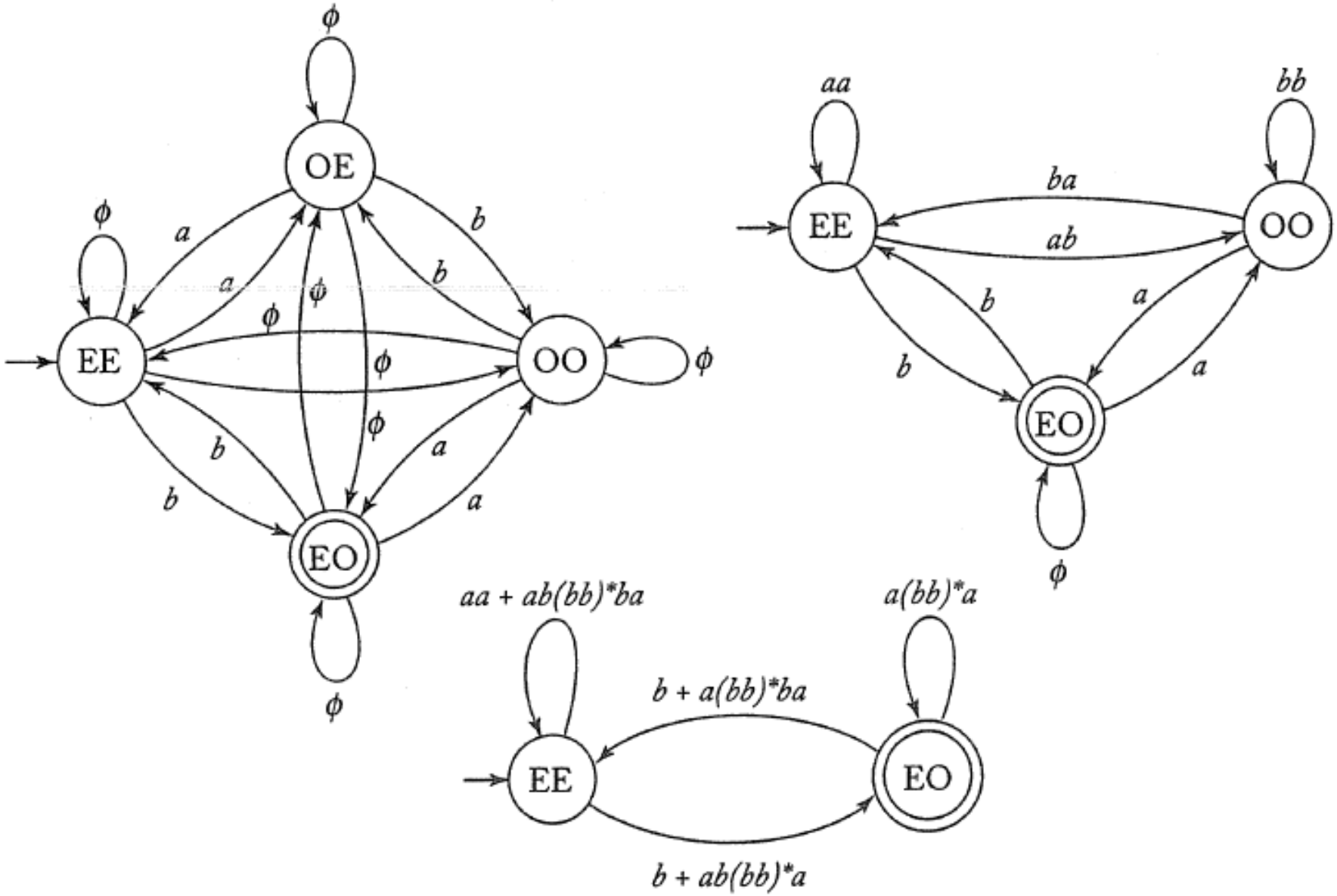






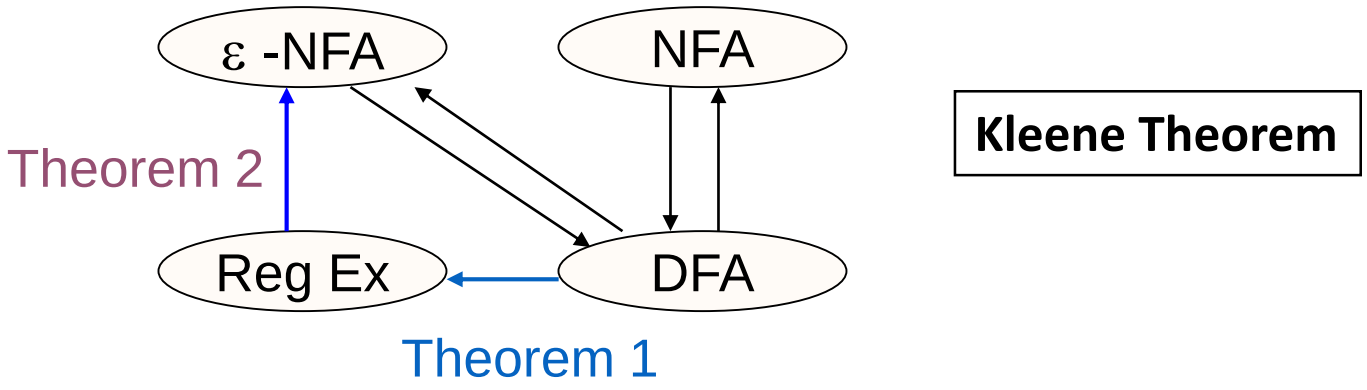
Thus, this automata is equivalent to the regular expression $(ab^*a + b)(a + b)^*$.

Generalized Transition Graph (GTG)



Finite Automata (FA) & Regular Expressions (Reg Ex)

- To show that they are interchangeable, consider the following theorems:
  - *Theorem 1: For every DFA A there exists a regular expression R such that L(R)=L(A)*
  - *Theorem 2: For every regular expression R there exists an ε-NFA E such that L(E)=L(R)*



Important Points

- *Theorem 1: If D is the DFA constructed from NFA N by the subset construction then L(D)=L(N)*
- *Theorem 2: A language L is accepted by some DFA if and only if L is accepted by some NFA*
- *Theorem 3: A language L is accepted by some ϵ-NFA if and only if L is accepted by some DFA*
- *Theorem 4: If L=L(A) for some DFA A, then there is a regular expression R such that LL(R)*

Equivalence of DFA's, NFA's

- A DFA can be turned into an NFA that accepts the same language.
- If $\delta_D(q, a) = p$, let the NFA have $\delta_N(q, a) = \{p\}$.
- Then the NFA is always in a set containing exactly one state – the state the DFA is in after reading the same input.
- Surprisingly, for any NFA there is a DFA that accepts the same language.
- Proof is the *subset construction*.
- The number of states of the DFA can be exponential in the number of states of the NFA.
- Thus, NFA's accept exactly the regular languages.

Subset Construction

- Given an NFA with states Q, inputs Σ, transition function δ_N, state state q_0, and final states F, construct equivalent DFA with:
  - States 2^Q (Set of subsets of Q).
  - Inputs Σ.
  - Start state $\{q_0\}$.
  - Final states = all those with a member of F.

- The transition function δ_D is defined by:

$\delta_D(\{q_1,\ldots,q_k\}, a)$ is the union over all $i = 1,\ldots,k$ of $\delta_N(q_i, a)$.

- Example: We'll construct the DFA equivalent of our "chessboard" NFA.

Subset Construction – (2)

- The transition function δ_D is defined by:

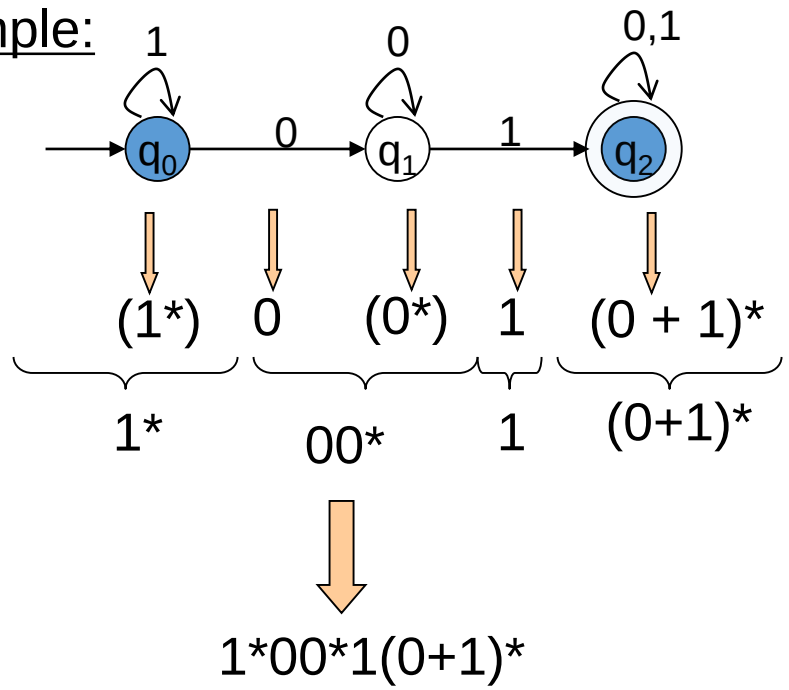
$\delta_D(\{q_1,\ldots,q_k\}, a)$ is the union over all i = 1,…,k of $\delta_N(q_i, a)$.

- Example: We'll construct the DFA equivalent of our "chessboard" NFA.

DFA to RE reconstruction

Informally, trace all distinct paths (traversing cycles only once)
from the start state to *each of the* final states
and enumerate all the expressions along the way

Example:



Theorem: If M is an NFA then L(M) is DFA-recognizable.

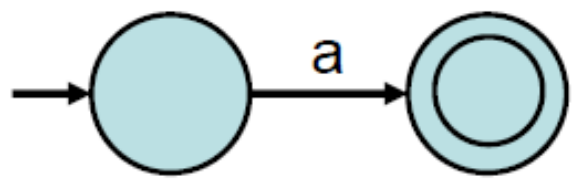
Proof:

- 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: $q_{02} = E(q_{01})$.
- $Q_2 = P(Q_1)$
- $q_{02} = E(q_{01})$
- $F_2 = \{ S \subseteq Q_1 \mid S \cap F_1 \neq \varnothing \}$
  - Accepting states of M_2 are the sets that contain an accepting state of M_1.
- $\delta_2(S, a) = \cup_{r \in S} E(\delta_1(r, a))$
  - 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.

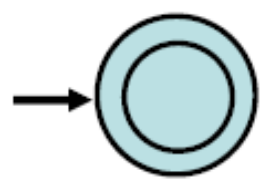
Theorem 1: If R is a regular expression, then L(R) is a regular language (recognized by a FA).

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 }
- Case 2: R = ε
  - L(R) = { ε }



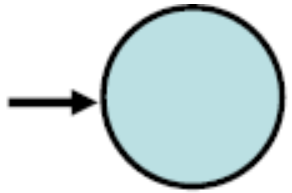
Accepts only a.



Accepts only ε.

– Case 3: $R = \varnothing$

- $L(R) = \varnothing$

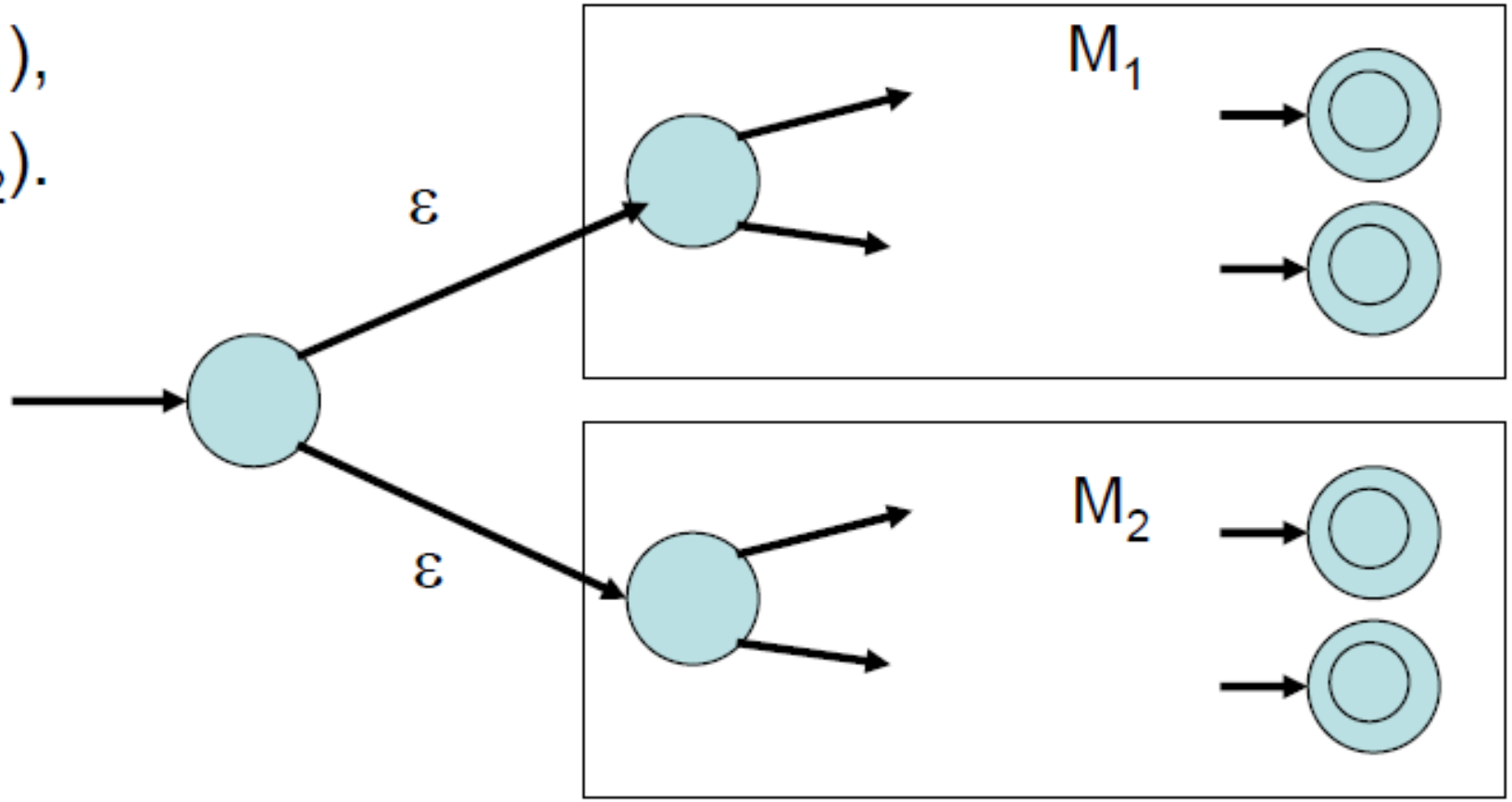


Accepts nothing.

– Case 4: $R = R_1 \cup R_2$

- M_1 recognizes $L(R_1)$,
- M_2 recognizes $L(R_2)$.

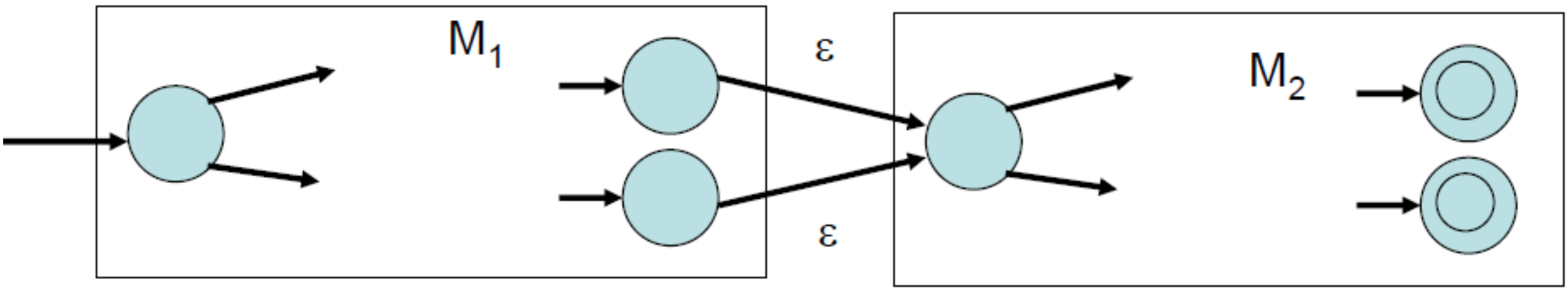
- Same construction we used to show regular languages are closed under union.



– Case 5: $R = R_1 \circ R_2$

- M_1 recognizes $L(R_1)$,
- M_2 recognizes $L(R_2)$.

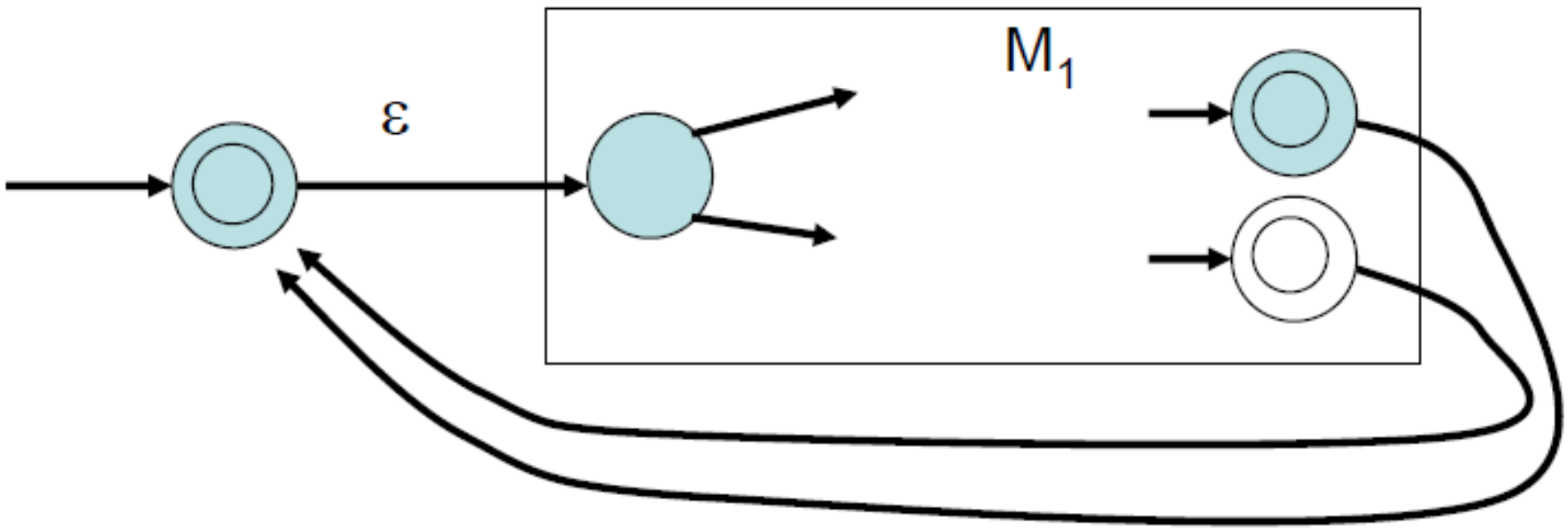
- Same construction we used to show regular languages are closed under concatenation.



– Case 6: $R = (R_1)^*$

- M_1 recognizes $L(R_1)$,

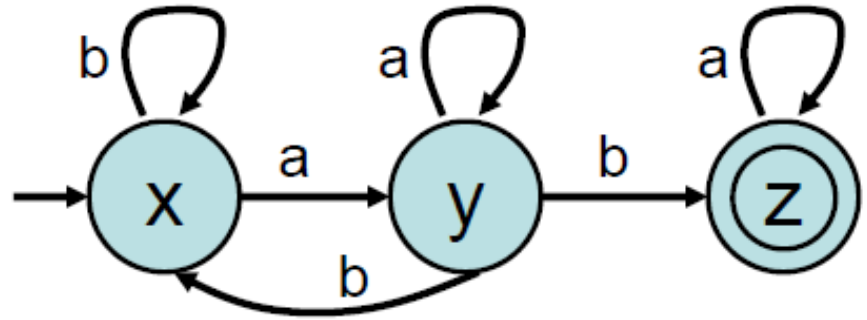
- Same construction we used to show regular languages are closed under star.



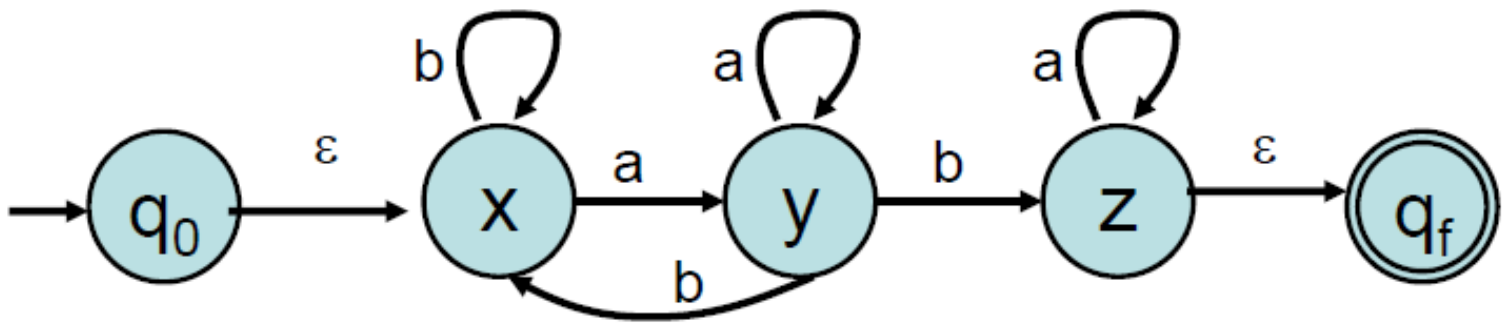
Theorem 2: If L is a regular language, then there is a regular expression R with L = L(R).

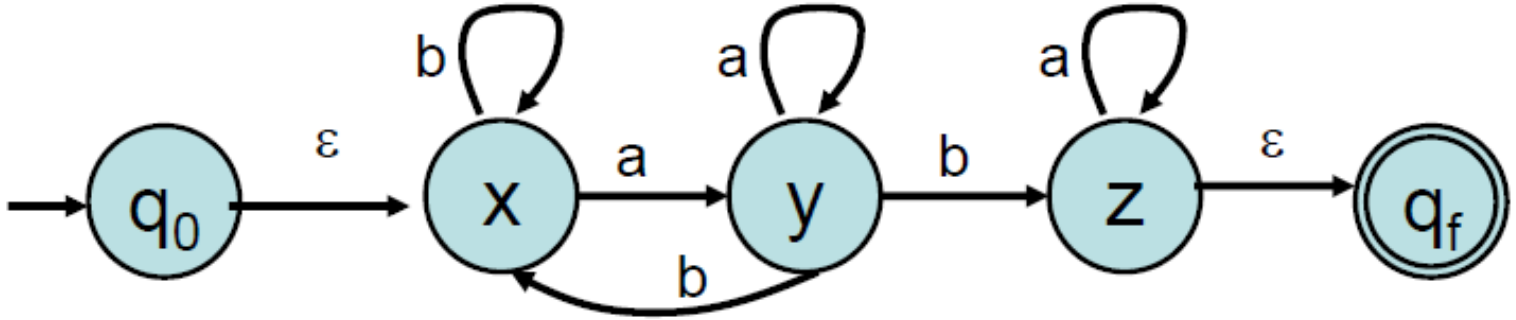
Proof:

- For each NFA M, define a regular expression R with L(R) = L(M).
- Show with an example:



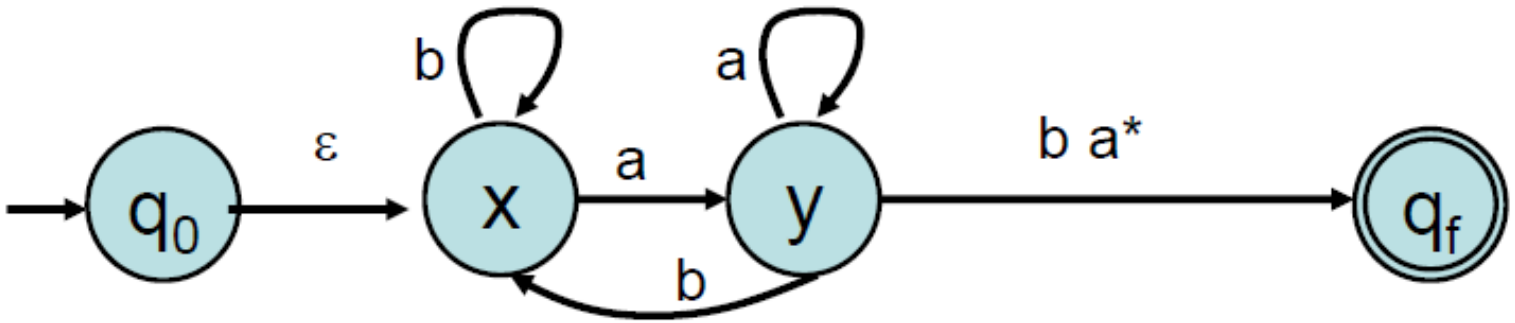
- Convert to a special form with only one final state, no incoming arrows to start state, no outgoing arrows from final state.



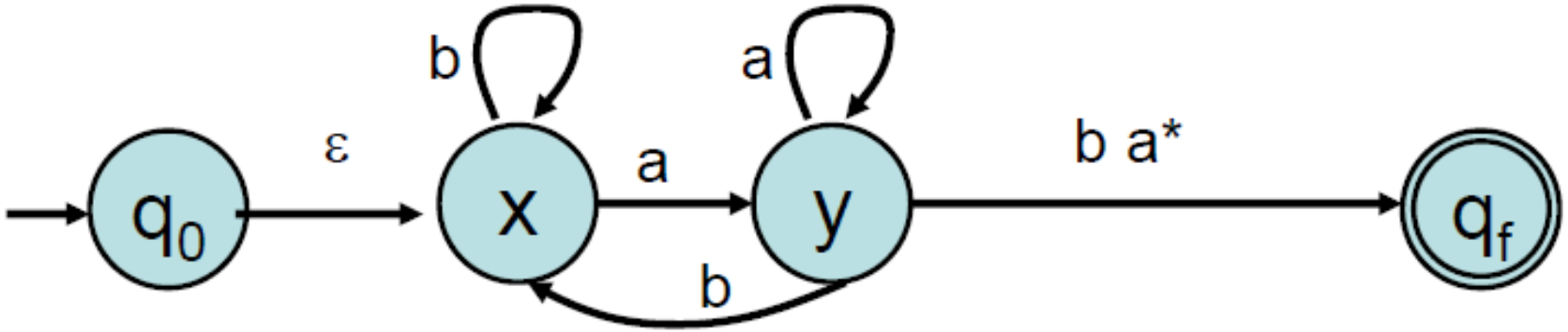


Now remove states one at a time (any order), replacing labels of edges with more complicated regular expressions

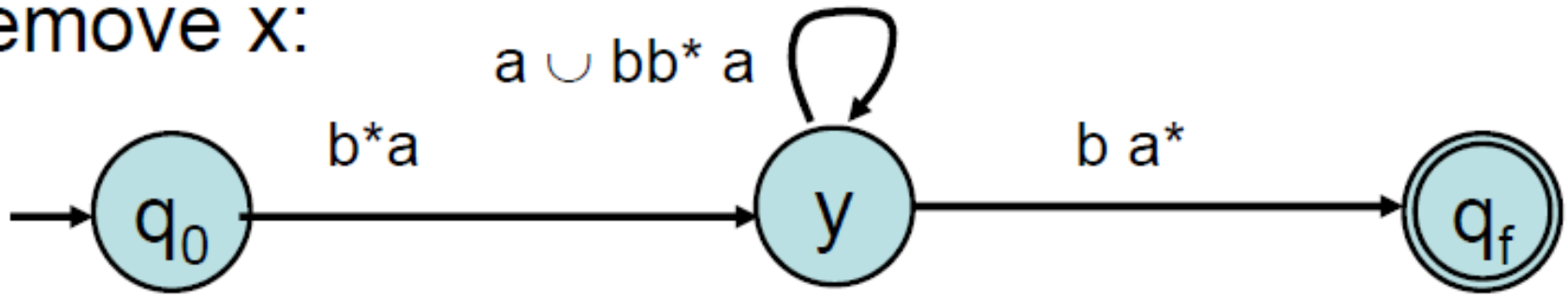
First remove z:



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.

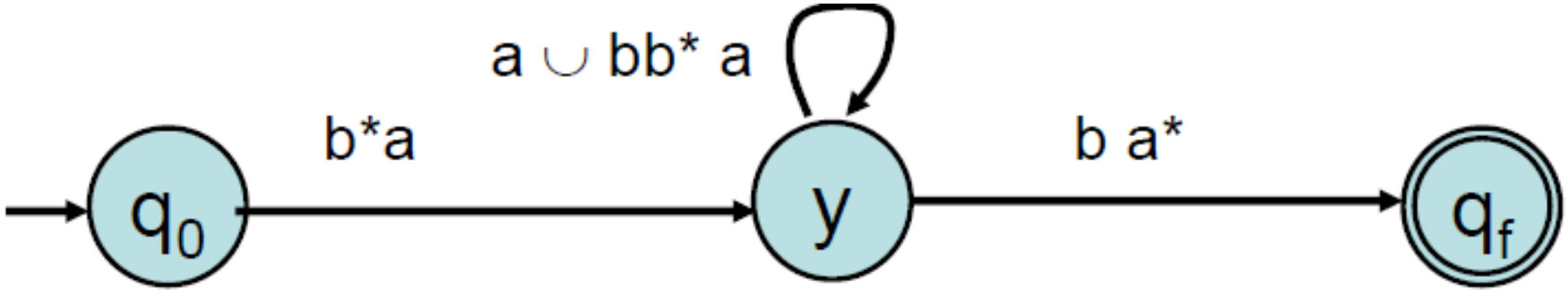


Then remove x:

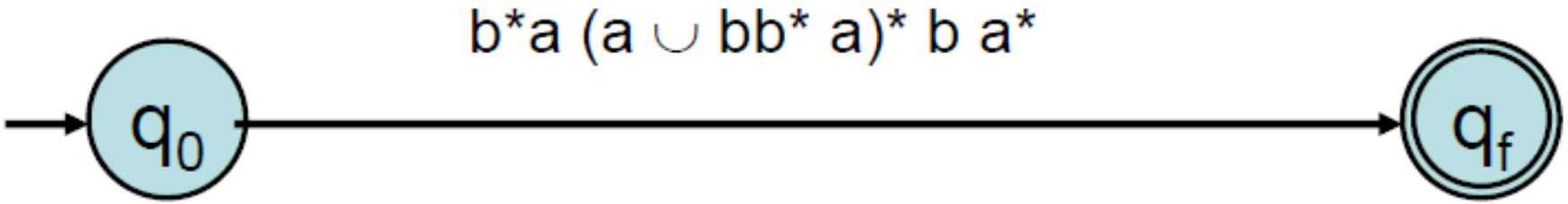


New label b*a describes all strings that can move the machine from q_0 to y, visiting (just) x any number of times.

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



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.

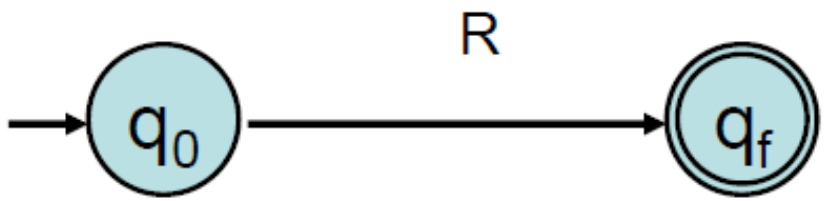
Define a generalized NFA (gNFA).

- 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:

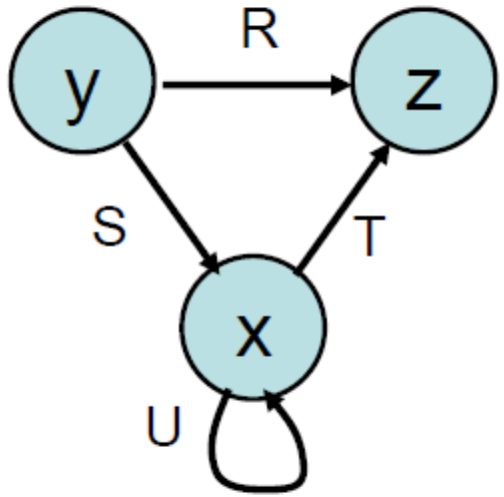


To remove a state x, consider every pair of other states, y and z, including y = z.

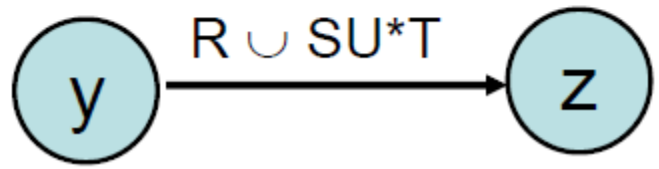
New label for edge (y, z) is the union of two expressions:

- What was there before, and
- One for paths through (just) x.

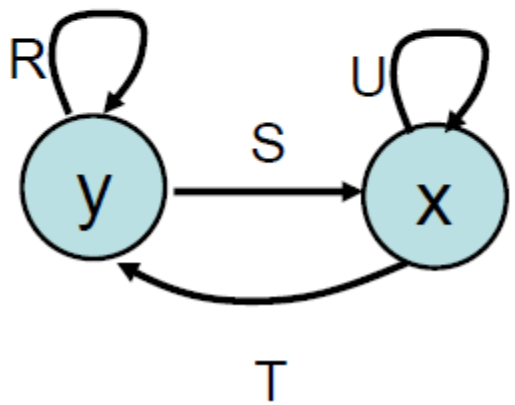
If $y \neq z$:

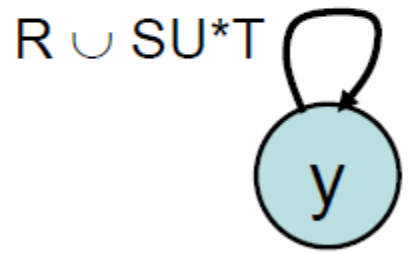


we get:



If y = z:

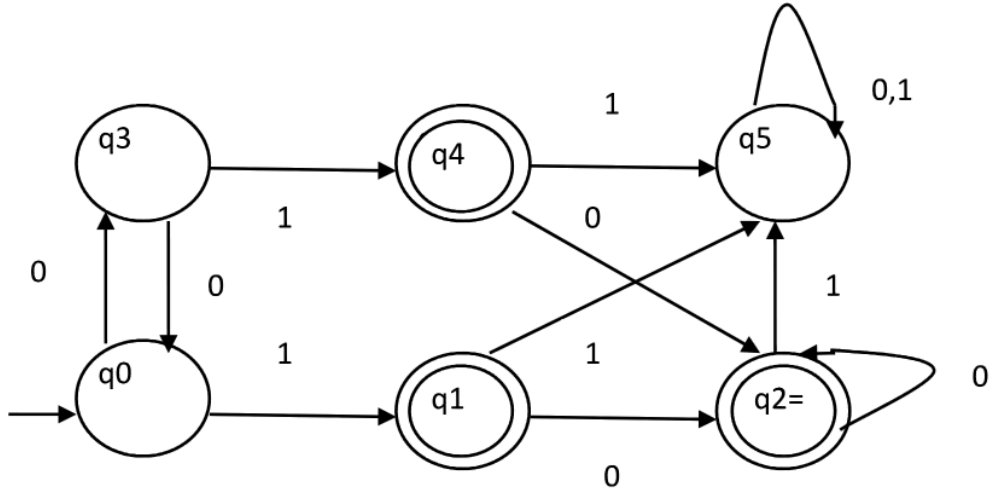




DFA State Minimization

- Suppose there is a DFA D < Q, Σ, q0, δ, F > which recognizes a language L. Then the minimized DFA D < Q', Σ, q0, δ', F' > can be constructed for language L as:
  Step 1: We will divide Q (set of states) into two sets. One set will contain all final states and other set will contain non-final states. This partition is called P_0.
  Step 2: Initialize k = 1
  Step 3: Find P_k by partitioning the different sets of P_{k-1}. In each set of P_{k-1}, we will take all possible pair of states. If two states of a set are distinguishable, we will split the sets into different sets in P_k.
  Step 4: Stop when $P_k = P_{k-1}$ (No change in partition)
  Step 5: All states of one set are merged into one. No. of states in minimized DFA will be equal to no. of sets in P_k.
- **How to find whether two states in partition P_k are distinguishable ?**
  Two states (qi, qj) are distinguishable in partition P_k if for any input symbol a, δ (qi, a) and δ (qj, a) are in different sets in partition P_{k-1}.

DFA State Minimization



Step 1. P0 will have two sets of states. One set will contain q1, q2, q4 which are final states of DFA and another set will contain remaining states. So P0 = { { q1, q2, q4 }, { q0, q3, q5 } }.

Step 2. To calculate P1, we will check whether sets of partition P0 can be partitioned or not:

i) **For set { q1, q2, q4 } :**

δ (q1, 0) = δ (q2, 0) = q2 and δ (q1, 1) = δ (q2, 1) = q5, So q1 and q2 are not distinguishable.

Similarly, δ (q1, 0) = δ (q4, 0) = q2 and δ (q1, 1) = δ (q4, 1) = q5, So q1 and q4 are not distinguishable.

Since, q1 and q2 are not distinguishable and q1 and q4 are also not distinguishable, So q2 and q4 are not distinguishable. So, { q1, q2, q4 } set will not be partitioned in P1

ii) For set { q0, q3, q5 } :

δ (q0, 0) = q3 and δ (q3, 0) = q0

δ (q0, 1) = q1 and δ(q3, 1) = q4

Moves of q0 and q3 on input symbol 0 are q3 and q0 respectively which are in same set in partition P0. Similarly, Moves of q0 and q3 on input symbol 1 are q3 and q0 which are in same set in partition P0. So, q0 and q3 are not distinguishable.

δ (q0, 0) = q3 and δ (q5, 0) = q5 and δ (q0, 1) = q1 and δ (q5, 1) = q5

Moves of q0 and q5 on input symbol 1 are q1 and q5 respectively which are in different set in partition P0. So, q0 and q5 are distinguishable. So, set { q0, q3, q5 } will be partitioned into { q0, q3 } and { q5 }. So,

P1 = { { q1, q2, q4 }, { q0, q3}, { q5 } }

DFA State Minimization

To calculate P2, we will check whether sets of partition P1 can be partitioned or not:

iii)For set { q1, q2, q4 } :

δ (q1, 0) = δ (q2, 0) = q2 and δ (q1, 1) = δ (q2, 1) = q5, So q1 and q2 are not distinguishable.

Similarly, δ (q1, 0) = δ (q4, 0) = q2 and δ (q1, 1) = δ (q4, 1) = q5, So q1 and q4 are not distinguishable.

Since, q1 and q2 are not distinguishable and q1 and q4 are also not distinguishable, So q2 and q4 are not distinguishable. So, { q1, q2, q4 } set will not be partitioned in P2.

iv)For set { q0, q3 } :

δ (q0, 0) = q3 and δ (q3, 0) = q0

δ (q0, 1) = q1 and δ (q3, 1) = q4

Moves of q0 and q3 on input symbol 0 are q3 and q0 respectively which are in same set in partition P1.

Similarly, Moves of q0 and q3 on input symbol 1 are q3 and q0 which are in same set in partition P1. So, q0 and q3 are not distinguishable.

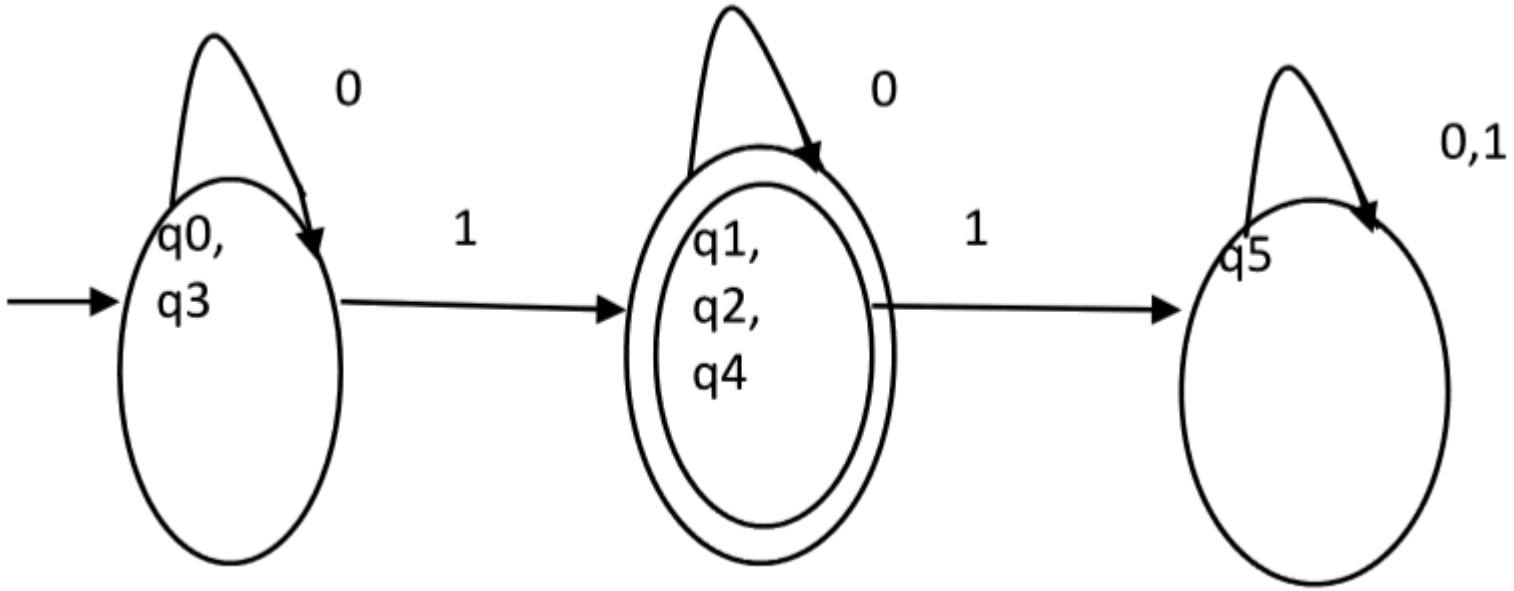
v) For set { q5 }:

Since we have only one state in this set, it can't be further partitioned. So,

P2 = { { q1, q2, q4 }, { q0, q3 }, { q5 } }

Since, P1=P2. So, this is the final partition. Partition P2 means that q1, q2 and q4 states are merged into one. Similarly, q0 and q3 are merged into one.

DFA State Minimization



THANK YOU

More on next class…