REGULAR EXPRESSIONS

Dr. Hatem Moharram

THE REGULAR OPERATIONS

We introduced and defined finite automata and regular languages. We now begin to investigate their properties. Doing so will help develop a toolbox of techniques for designing automata to recognize particular languages. The toolbox also will include ways of proving that certain other languages are nonregular (i.e., beyond the capability of finite automata).

In arithmetic, the basic objects are numbers and the tools are operations for manipulating them, such as + and ×. In the theory of computation, the objects are languages and the tools include operations specifically designed for manipulating them. We define three operations on languages, called the *regular operations*, and use them to study properties of the regular languages.

DEFINITION 1.23

Let A and B be languages. We define the regular operations *union*, *concatenation*, and *star* as follows:

- **Union:** $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
- **Concatenation:** $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$.
- **Star:** $A^* = \{x_1 x_2 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$.

The union operation takes all the strings in both A and B and lumps them together into one language.

The concatenation operation attaches a string from A in front of a string from B in all possible ways to get the strings in the new language.

The star operation is a *unary operation* instead of a *binary operation*. It works by attaching any number of strings in A together to get a string in the new language A*.

The empty string ε is always a member of A*, no matter what A is.

EXAMPLE 1.24

Let the alphabet be the standard 26 letters {a, b, . . . , z}. If A = {good, bad} and B = {boy, girl}, then

$A \cup B$ = {good, bad, boy, girl},

$A \circ B$ = {goodboy, goodgirl, badboy, badgirl}, and

A^* = {ε, good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, goodgoodbad, gooadgdbood, goodbadbad, . . . }.

Let $\mathcal{N} = \{1, 2, 3, \ldots\}$ be the set of natural numbers.

When we say that $\mathcal{N}$ is ***closed under multiplication***, we mean that for any x and y in $\mathcal{N}$, the product $x \times y$ also is in $\mathcal{N}$. In contrast, $\mathcal{N}$ is not closed under division, as 1 and 2 are in $\mathcal{N}$ but 1/2 is not.

A collection of objects is ***closed* under some operation** if applying that operation to members of the collection returns an object still in the collection.

We show that

"the collection of regular languages is closed under all three of the regular operations".

THEOREM 1.25

The class of regular languages is closed under the union operation.
In other words, if A1 and A2 are regular languages, so is A1 ∪ A2.

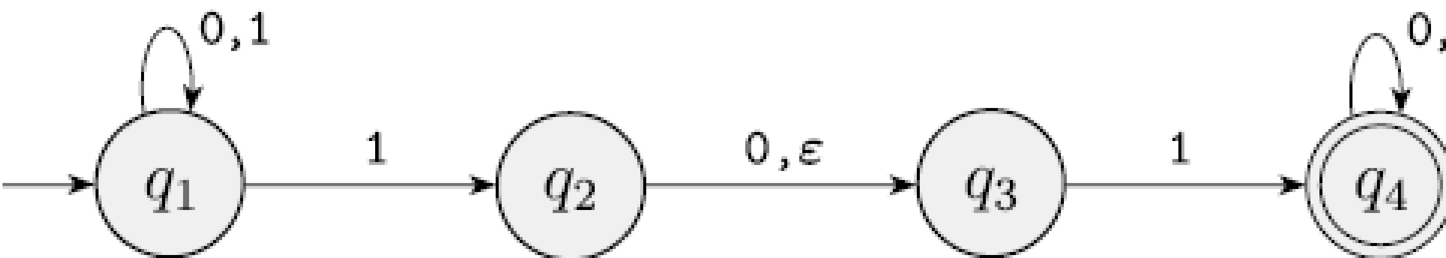
THEOREM 1.26

The class of regular languages is closed under the concatenation operation.
In other words, if A1 and A2 are regular languages then so is A1 ° A2.

1.2 NONDETERMINISM

When the machine is in a given state and reads the next input symbol, we know what the next state will be—it is determined. We call this *deterministic* computation. In a *nondeterministic* machine, several choices may exist for the next state at any point.

Nondeterminism is a generalization of determinism, so every deterministic finite automaton (DFA) is automatically a nondeterministic finite automaton (NFA).



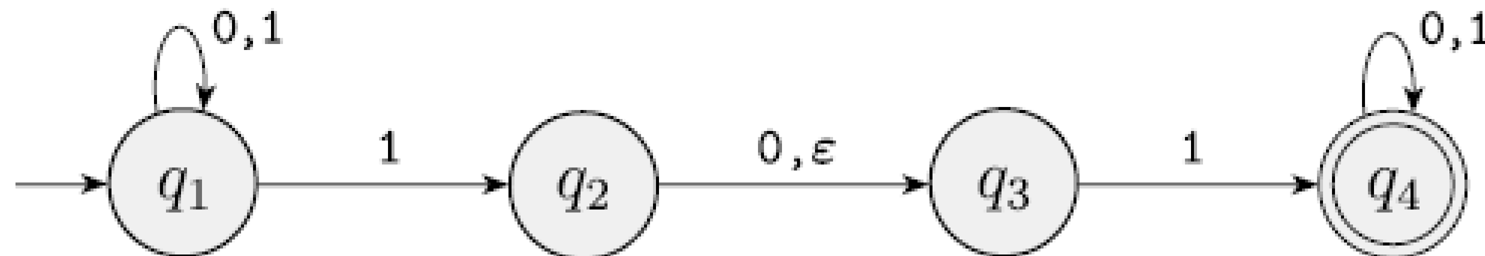
The nondeterministic finite automaton N_1

State q_1 has one exiting arrow for 0, but it has two for 1; q_2 has one arrow for 0, but it has none for 1. In an NFA, a state may have zero, one, or many exiting arrows for each alphabet symbol.

The difference between a deterministic finite automaton (DFA) and a nondeterministic finite automaton (NFA) is:

First, every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. The NFA violates that rule.

Second, in a DFA, labels on the transition arrows are symbols from the alphabet. This NFA has an arrow with the label ε. In general, an NFA may have arrows labeled with members of the alphabet or ε. Zero, one, or many arrows may exit from each state with the label ε.



How does an NFA compute?

Suppose that we are running an NFA on an input string and come to a state with multiple ways to proceed.

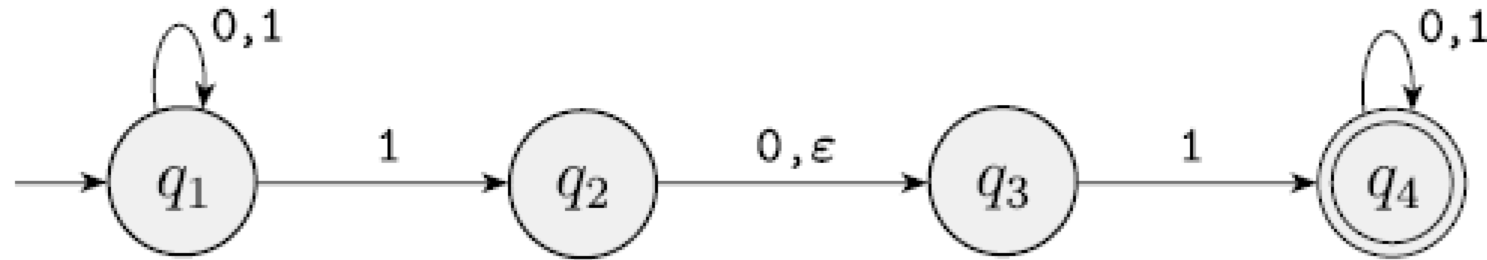
After reading that symbol, the machine splits into multiple copies of itself and follows all the possibilities in parallel, and continues as before.

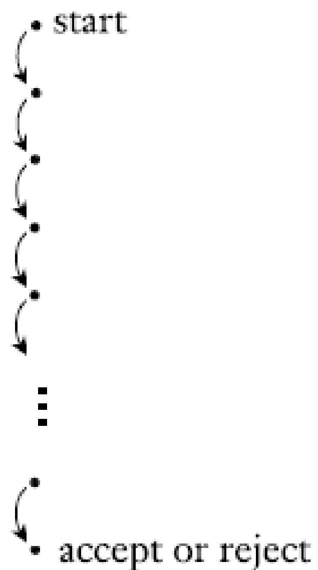
If there are subsequent choices, the machine splits again.

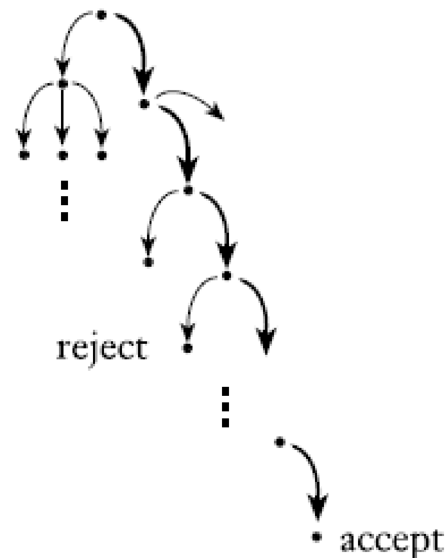
If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it.

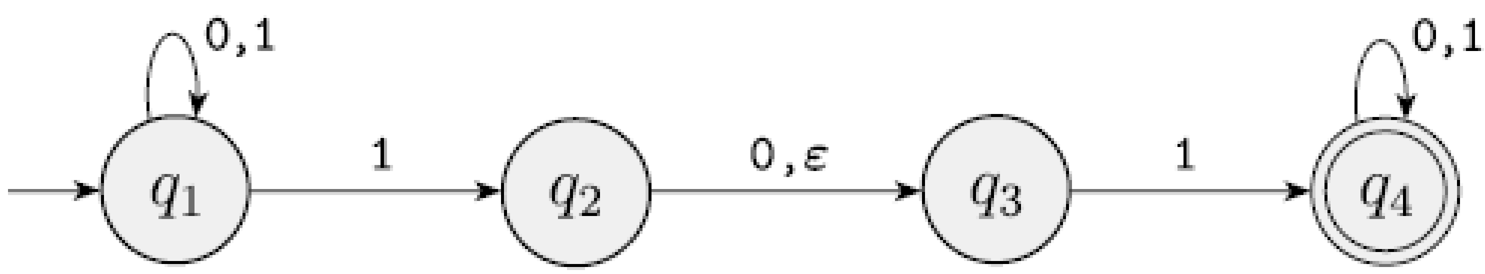
Finally, if any one of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

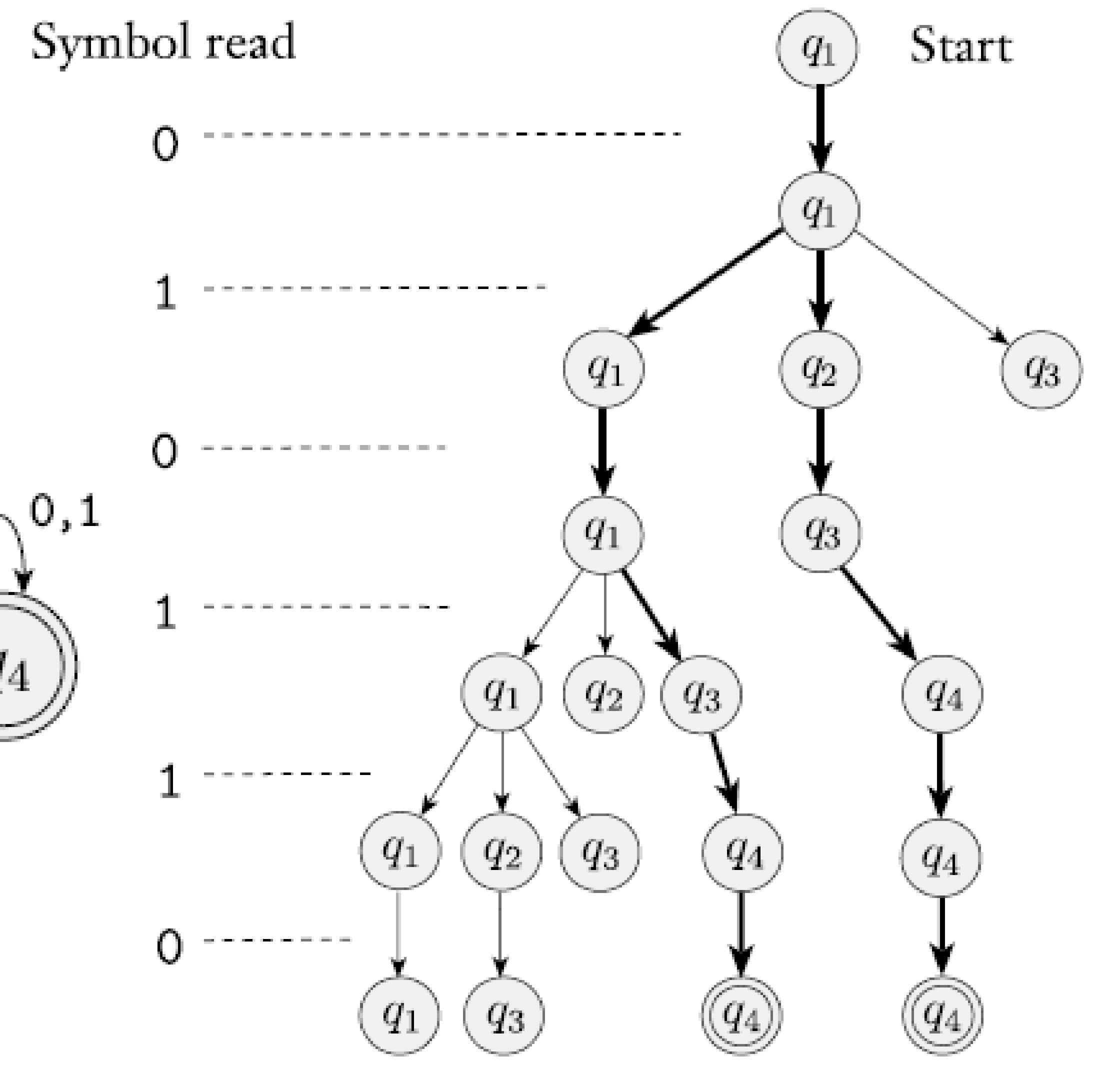
If a state with an ε symbol on an exiting arrow is encountered, something similar happens. Without reading any input, the machine splits into multiple copies, one following each of the exiting ε -labeled arrows and one staying at the current state. Then the machine proceeds nondeterministically as before.











The computation of N_1 on input 010110

as q_4 is an accept state, N_1 accepts this string.

By continuing to experiment in this way, you will see that N_1 accepts all strings that contain either 101 or 11 as a substring.

Nondeterministic finite automata are useful in several respects:

- As we will show, every NFA can be converted into an equivalent DFA,

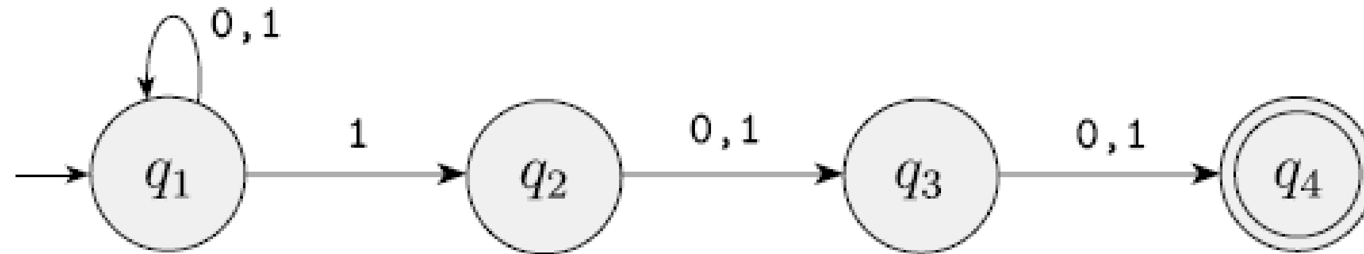
- constructing NFAs is sometimes easier than directly constructing DFAs.

- An NFA may be much smaller than its deterministic counterpart, or its functioning may be easier to understand.

- Nondeterminism in finite automata is also a good introduction to nondeterminism in more powerful computational models because finite automata are especially easy to understand.

EXAMPLE 1.30

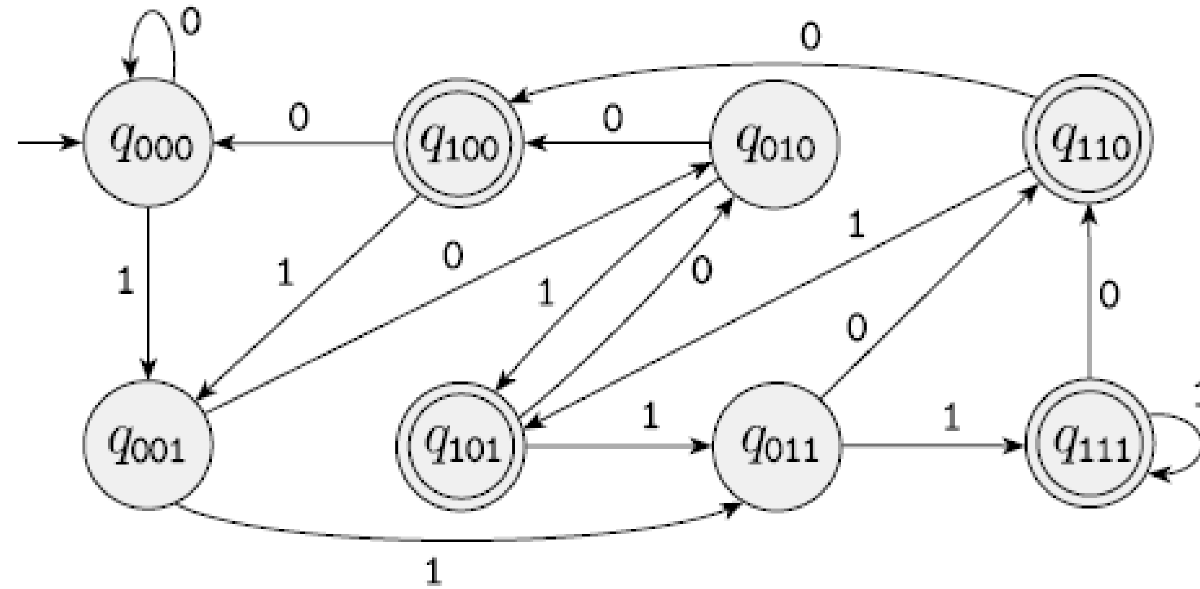
Let A be the language consisting of all strings over {0,1} containing a 1 in the third position from the end (e.g., 000100 is in A but 0011 is not). The following four-state NFA N_2 recognizes A.



The NFA N_2 recognizing A

One good way to view the computation of this NFA is to say that it stays in the start state q_1 until it "guesses" that it is three places from the end. At that point, if the input symbol is a 1, it branches to state q_2 and uses q_3 and q_4 to "check" on whether its guess was correct.

As mentioned, every NFA can be converted into an equivalent DFA; but sometimes that DFA may have many more states. The smallest DFA for A contains eight states. Furthermore, understanding the functioning of the NFA is much easier, as you may see by examining the following figure for the DFA.

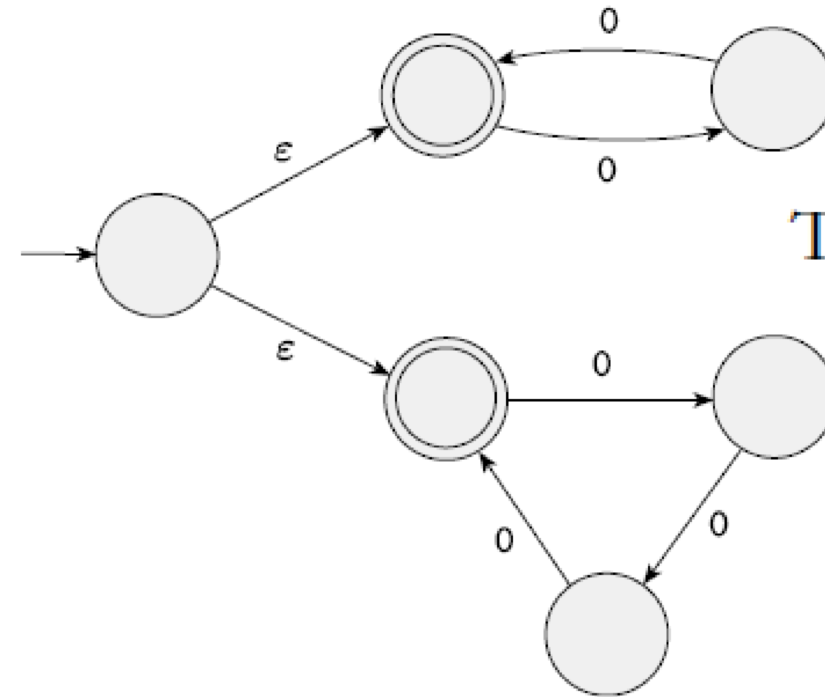


A DFA recognizing A

EXAMPLE 1.33

The following NFA N_3 has an input alphabet {0} consisting of a single symbol. An alphabet containing only one svmbol is called a *unary alphabet*.

This machine demonstrates the convenience of having ε arrows. It accepts all strings of the form 0^k where k is a multiple of 2 or 3. (Remember that the superscript denotes repetition, not numerical exponentiation.) For example, N_3 accepts the strings ε, 00, 000, 0000, and 000000, but not 0 or 00000.

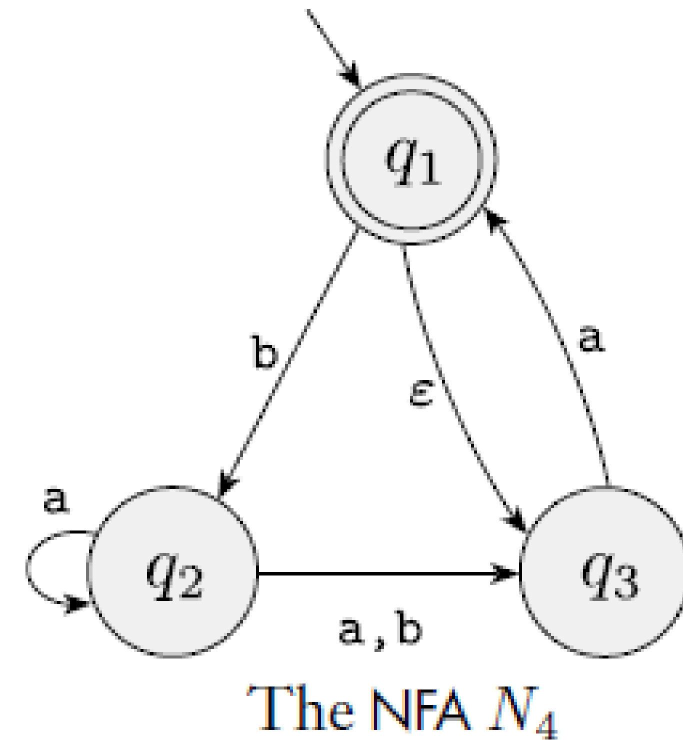


The NFA N_3

Think of the machine operating by initially guessing whether to test for a multiple of 2 or a multiple of 3 by branching into either the top loop or the bottom loop and then checking whether its guess was correct. Of course, we could replace this machine by one that doesn't have ε arrows or even any nondeterminism at all, but the machine shown is the easiest one to understand for this language.

EXAMPLE 1.35

Practice with it to satisfy yourself that it accepts the strings ε, a, baba, and baa, but that it doesn't accept the strings b, bb, and babba. Later we use this machine to illustrate the procedure for converting NFAs to DFAs.



The NFA N_4

FORMAL DEFINITION OF A NONDETERMINISTIC FINITE AUTOMATON

In a DFA, the transition function takes a state and an input symbol and produces the next state.

In an NFA, the transition function takes a state and an input symbol *or the empty string* and produces *the set of possible next states*.

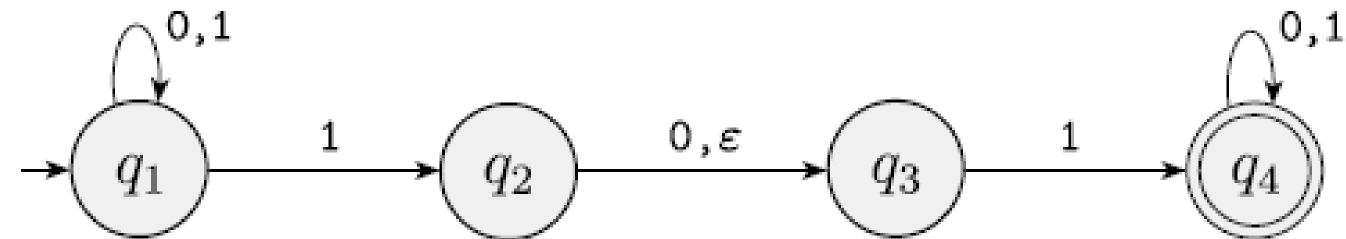
In order to write the formal definition, we need to set up some additional notation.

For any set $\mathcal{Q}$ we write $\mathcal{P}(\mathcal{Q})$ to be **the collection of all subsets of $\mathcal{Q}$**. Here $\mathcal{P}(\mathcal{Q})$ is called the ***power set*** of $\mathcal{Q}$. For any alphabet Σ we write Σ_ε to be $\Sigma \cup \{\varepsilon\}$. Now we can write the formal description of the type of the transition function in an NFA as $\delta: \mathcal{Q} \times \Sigma_\varepsilon \to \mathcal{P}(\mathcal{Q})$.

EXAMPLE 1.38

Recall the NFA N_1:

The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where



1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0,1\}$
3. is given as

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$,

4. q_1 is the start state, and
5. $F = \{q_4\}$.

DEFINITION 1.37

A *nondeterministic finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta : Q \times \Sigma_\varepsilon \rightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

Let N = (Q, Σ, δ , q_0, F) be an NFA and w a string over the alphabet. Then we say that N *accepts* w if we can write w as $w = y_1 y_2 \cdots y_m$, where each y_i is a member of Σ_ε and a sequence of states $r_0, r_1, \ldots,$ r_m exists in Q with three conditions:

1. $r_0 = q_0$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$, for i = 0, . . . ,m − 1, and
3. $r_m \in F$.

Condition 1 says that the machine starts out in the start state.
Condition 2 says that state r_{i+1} is one of the allowable next states when N is in state r_i and reading y_{i+1}.
condition 3 says that the machine accepts its input if the last state is an accept state.

EQUIVALENCE OF NFAS AND DFAS

Deterministic and nondeterministic finite automata recognize the same class of languages. Such equivalence is both surprising and useful.

It is surprising because NFAs appear to have more power than DFAs, so we might expect that NFAs recognize more languages.

It is useful because describing an NFA for a given language sometimes is much easier than describing a DFA for that language.

Say that two machines are *equivalent* if they recognize the same language.

THEOREM 1.39

Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

Theorem 1.39 states that every NFA can be converted into an equivalent DFA. Thus nondeterministic finite automata give an alternative way of characterizing the regular languages.

COROLLARY 1.40

A language is regular if and only if some nondeterministic finite automaton recognizes it.

we define *E(R)* to be the collection of states that can be reached from members of R by going only along ε arrows, including the members of R themselves.

EXAMPLE 1.41

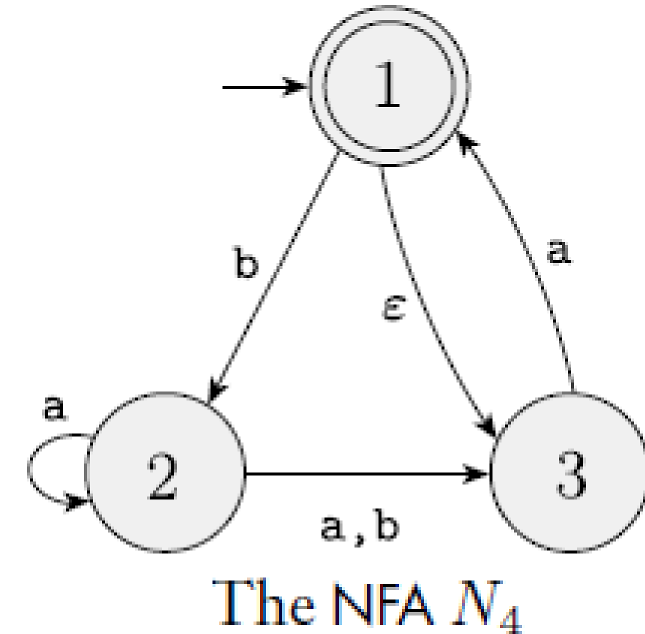
Let N_4 be the machine that appears in Example 1.35. For clarity, we have relabeled the states of N_4 to be {1, 2, 3}. Thus in the formal description of $N_4 = (Q, \{a,b\}, \delta, 1, \{1\})$, the set of states Q is {1, 2, 3}.

To construct a DFA D that is equivalent to N_4,

1- determines D's states:

N_4 has three states, {1, 2, 3}, so we construct D with eight states, one for each subset of N_4's states. We label each of D's states with the corresponding subset. Thus D's state set is

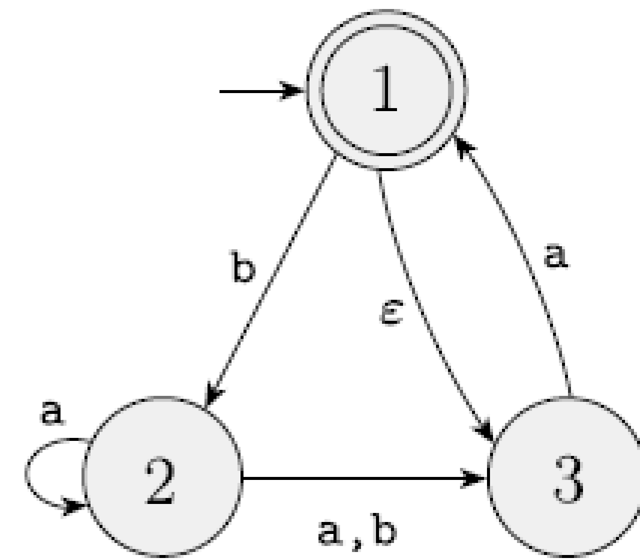
$\{ \varphi, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$



The NFA N_4

.

2- determines the start and accept states of D:



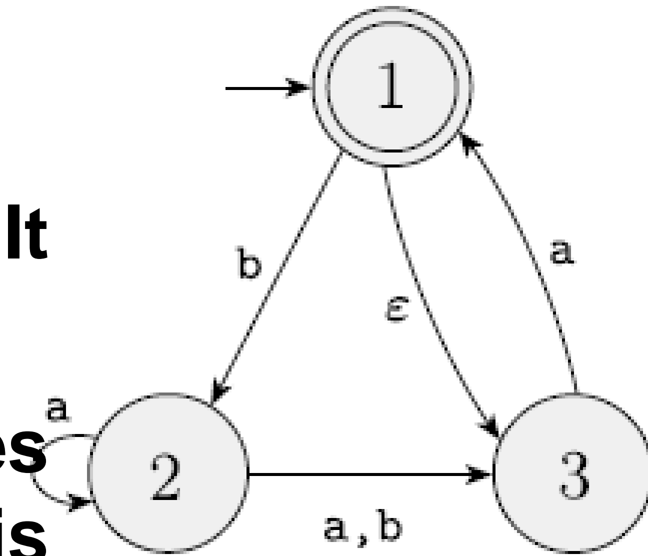
The start state is E({1}), the set of states that are reachable from 1 by traveling along ε arrows, plus 1 itself. An ε arrow goes from 1 to 3, so E({1}) = {1, 3}.

The new accept states are those containing N_4's accept state; thus **{{1},{1,2},{1,3},{1,2,3}}.**

3- determines D's transition function:

Each of D's states goes to one place on input ***a*** and one place on input ***b***. We illustrate the process of determining the placement of D's transition arrows with a few examples.

In D, **state {2}** goes to {2,3} on input *a* because in N_4, state 2 goes to both 2 and 3 on input *a* and we can't go farther from 2 or 3 along ε arrows. State {2} goes to state {3} on input *b* because in N_4, state 2 goes only to state 3 on input *b* and we can't go farther from 3 along ε arrows. $\{2\} \xrightarrow{a} \{2,3\}$, $\{2\} \xrightarrow{b} \{3\}$



State {1} goes to φ on *a* because no *a* arrows exit it. It goes to {2} on *b*. $\{1\} \xrightarrow{a} \varphi$, $\{1\} \xrightarrow{b} \{2\}$
Note that the procedure in Theorem 1.39 specifies that we follow the ε arrows *after* each input symbol is read.

State {3} goes to {1,3} on *a* because in N_4, state 3 goes to 1 on *a* and 1 in turn goes to 3 with an ε arrow. State {3} on *b* goes to φ.
$\{3\} \xrightarrow{a} \{1,3\}$, $\{3\} \xrightarrow{b} \varphi$

State {1,2} on *a* goes to {2,3} because 1 points at no states with *a* arrows, 2 points at both 2 and 3 with *a* arrows, and neither points anywhere with ε arrows. State {1,2} on *b* goes to {2,3}.

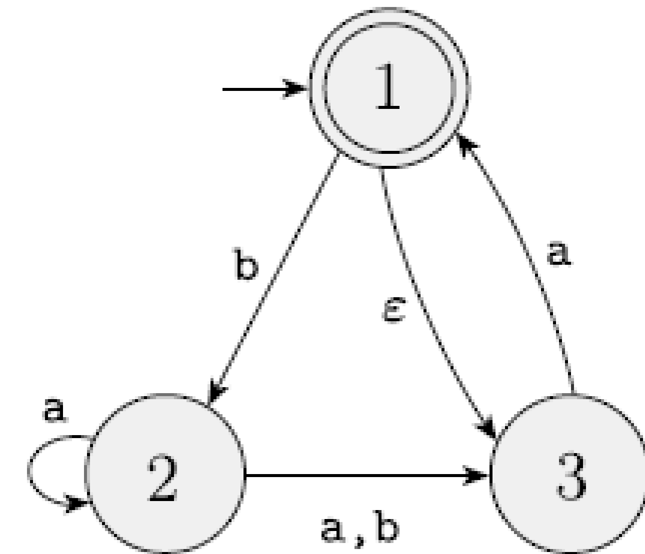
$\{1,2\} \overset{a}{\rightarrow} \{2,3\}$, $\{1,2\} \overset{b}{\rightarrow} \{2,3\}$

Continuing in this way, we obtain the following diagram for D.

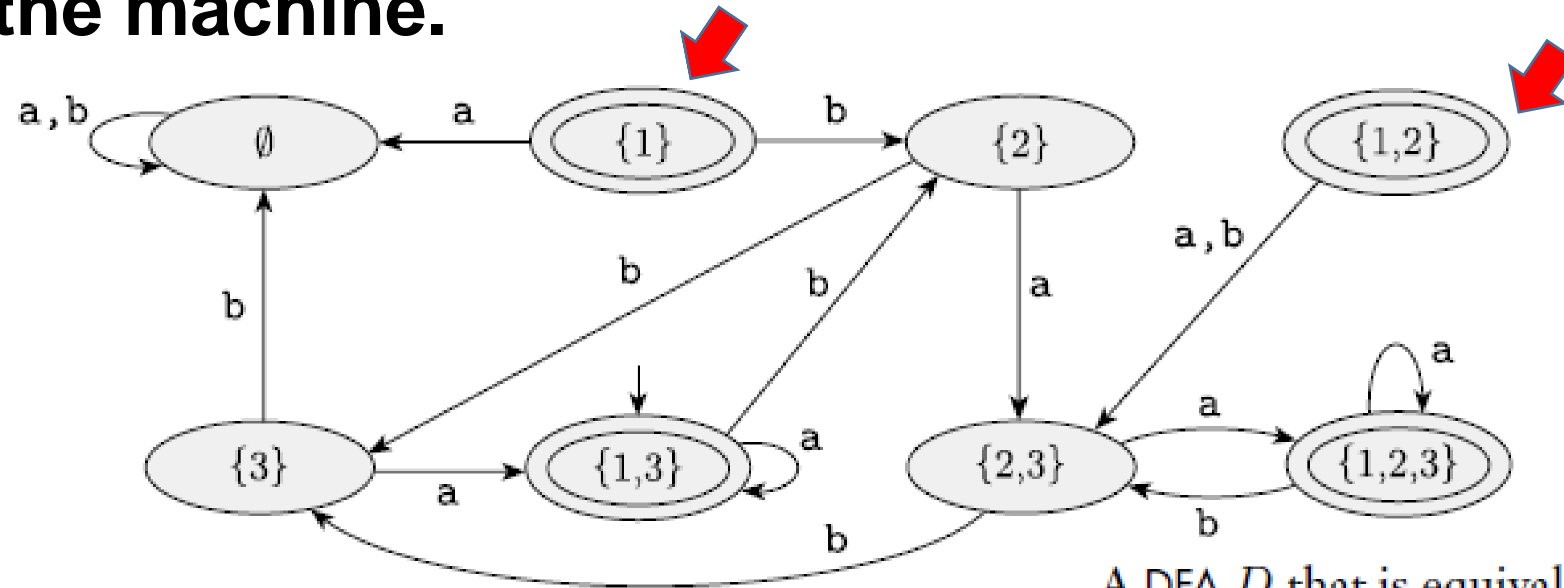
$\{1,3\} \overset{a}{\rightarrow} \{1,3\}$, $\{1,3\} \overset{b}{\rightarrow} \{2\}$

$\{2,3\} \overset{a}{\rightarrow} \{1,2,3\}$, $\{2,3\} \overset{b}{\rightarrow} \{3\}$

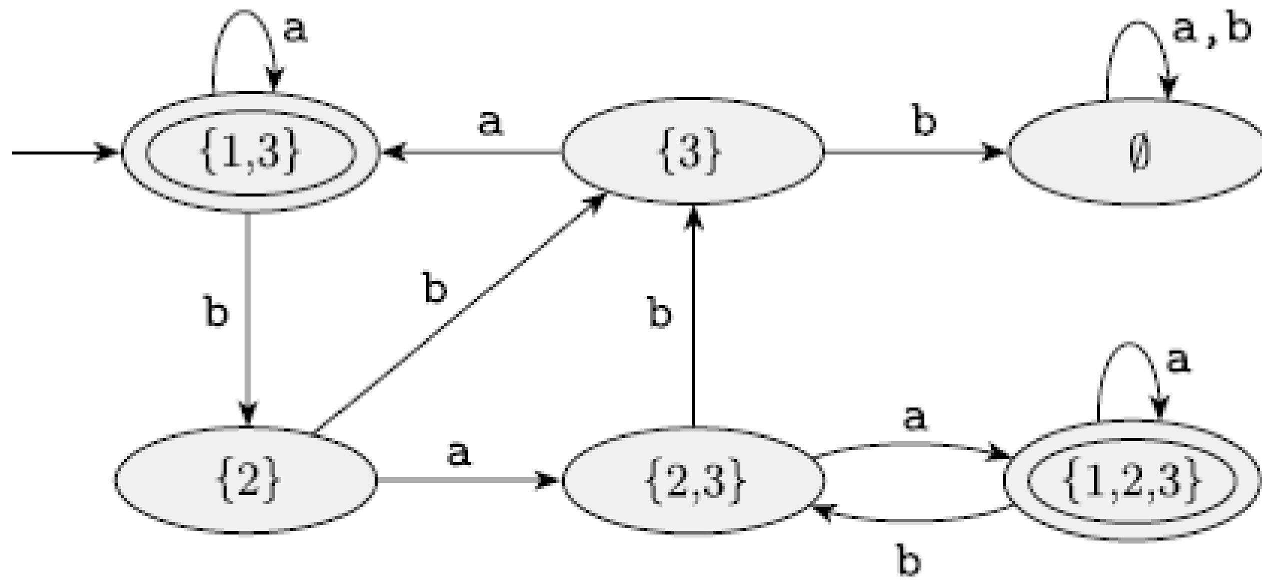
$\{1,2,3\} \overset{a}{\rightarrow} \{1,2,3\}$, $\{1,2,3\} \overset{b}{\rightarrow} \{2,3\}$



We may simplify this machine by observing that no arrows point at states {1} and {1, 2}, so they may be removed without affecting the performance of the machine.



A DFA D that is equivalent to the NFA N_4



DFA D after removing unnecessary states

CLOSURE UNDER THE REGULAR OPERATIONS

we can use the regular operations to build up expressions describing languages, which are called *regular expressions*. An example is:

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

the symbols 0 and 1 are shorthand for the sets {0} and {1}. So $(0 \cup 1)$ means $(\{0\} \cup \{1\})$. The value of this part is the language {0,1}.

The part 0^* means $\{0\}^*$, and its value is the language consisting of all strings containing any number of 0s.

the concatenation symbol often is implicit in regular expressions. Thus $(0 \cup 1)0$ actually is shorthand for $(0 \cup 1) \circ 0$.

the value is the language consisting of all strings starting with a 0 or a 1 followed by any number of 0s.

EXAMPLE 1.51

Another example of a regular expression is $(0 \cup 1)^*$.

It starts with the language $(0 \cup 1)=(\{0\} \cup \{1\})= \{0,1\}$ and applies the operation.

The value of this expression, $(0 \cup 1)^*$, is the language consisting of all possible strings of 0s and 1s.

If $\Sigma = \{0,1\}$, we can write Σ as shorthand for the regular expression $(0 \cup 1)$.

More generally, if Σ is any alphabet, the regular expression Σ describes the language consisting of all strings of length 1 over this alphabet, and Σ^* describes the language consisting of all strings over that alphabet.

Similarly, $\Sigma^* 1$ is the language that contains all strings that end in a 1. The language $(0 \Sigma^*) \cup (\Sigma^* 1)$ consists of all strings that start with a 0 or end with a 1.

In arithmetic, we say that × has precedence over + to mean that when there is a choice, we do the × operation first. Thus in 2+3×4, the 3×4 is done before the addition. To have the addition done first, we must add parentheses to obtain (2 + 3) × 4.

In regular expressions:

1- the star operation * is done first,

2- concatenation ° , and finally

3- union ∪,

unless parentheses change the usual order.

FORMAL DEFINITION OF A REGULAR EXPRESSION

DEFINITION 1.52

Say that R is a *regular expression* if R is

1. *a* for some *a* in the alphabet Σ,
2. ε
3. φ
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. $(R_1)^*$, where R_1 is a regular expression.

In items 1 and 2, the regular expressions *a* and ε represent the languages {*a*} and {ε}, respectively.
In item 3, the regular expression φ represents the empty language.
In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the star of the language R_1, respectively.

Don't confuse the regular expressions ε and φ:

The expression ε represents the language containing a single string—namely, the empty string.

whereas φ represents the language that doesn't contain any strings.

let R^+ be shorthand for RR^*. In other words, whereas R^* has all strings that are 0 or more concatenations of strings from R, the language R^+ has all strings that are 1 or more concatenations of strings from R.
So $R^+ \cup \varepsilon = R^*$.

In addition, we let R^k be shorthand for the concatenation of k R's with each other.
When we want to distinguish between a regular expression R and the language that it describes, we write L(R) to be the language of R.

EXAMPLE 1.53

In the following instances, we assume that the alphabet is {0,1}.

1. $0^*10^* = \{w \mid w \text{ contains a single } 1\}$.

2. $\Sigma^*1\Sigma^* = \{w \mid w \text{ has at least one } 1\}$.

3. $\Sigma^*001\Sigma^* = \{w \mid w \text{ contains the string } 001 \text{ as a substring}\}$.

4. $1^*(01^+)^* = \{w \mid \text{every } 0 \text{ in } w \text{ is followed by at least one } 1\}$.

5. $(\Sigma\Sigma)^* = \{w \mid w \text{ is a string of even length}\}$.

6. $(\Sigma\Sigma\Sigma)^*$ = {w| the length of w is a multiple of 3}.

7. $01 \cup 10$ = {01, 10}

8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1$ = {w| w starts and ends with the same symbol}.

9. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$ = {0, 01, 011,0111,……., 1,11,111,…}

The expression $0 \cup \varepsilon$ describes the language $\{0,\varepsilon\}$, so the concatenation operation adds either 0 or ε before every string in 1.

10. $(0 \cup \varepsilon)(1 \cup \varepsilon)$ = $\{\varepsilon,0,1,01\}$

11. $1^* \varphi = \varphi$. Concatenating the empty set to any set yields the empty set.

12. $\varphi^* = \{\varepsilon\}$. the star operation can put together 0 strings, giving only the empty string.

Regular Expressions	Regular Set
$(0 \cup 10^*)$	L= { 0, 1, 10, 100, 1000, 10000, … }
(0^*10^*)	L={1, 01, 10, 010, 0010, …}
$(0 \cup \varepsilon)(1 \cup \varepsilon)$	L= {ε, 0, 1, 01}
$(a \cup b)^*$	Set of strings of a's or b's of any length including the null string. So L= { ε, a, b,aa,ab,ba,bb,…….}
$(a \cup b)^*abb$	Set of strings of a's or b's ending with the string abb, So L = {abb, aabb, babb, aaabb, ababb, …………..}
$(11)^*$	Set consisting of even number of 1's including empty string, So L= {ε, 11, 1111, 111111, ……….}
$(aa)^*(bb)^*b$	Set of strings consisting of even number of a's followed by odd number of b's , so L= {b, aab, aabbb, aabbbbb, aaaab, aaaabbb, …………..}
$(aa \cup ab \cup ba \cup bb)^*$	String of a's and b's of even length can be obtained by concatenating any combination of the strings aa, ab, ba and bb including null, so L= {aa, ab, ba, bb, aaab, aaba, …………..}

If we let R be any regular expression, we have the following identities:

$$R \cup \varphi = R.$$

Adding the empty language to any other language will not change it.

$$R \circ \varepsilon = R.$$

Joining the empty string to any string will not change it.

However, exchanging φ and ε in the preceding identities may cause the equalities to fail.

$R \cup \varepsilon$ may not equal R.

For example, if $R = 0$, then $L(R) = \{0\}$ but $L(R \cup \varepsilon) = \{0, \varepsilon\}$.

$R \circ \varphi$ may not equal R.

For example, if $R = 0$, then $L(R) = \{0\}$ but $L(R \circ \varphi) = \varphi$.

EQUIVALENCE WITH FINITE AUTOMATA

Any regular expression can be converted into a finite automaton that recognizes the language it describes, and vice versa.

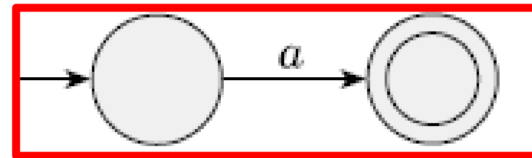
Recall that a regular language is one that is recognized by some finite automaton.

THEOREM 1.54

A language is regular if and only if some regular expression describes it.

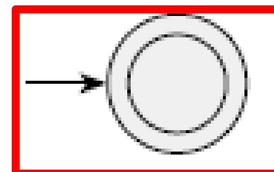
Say that we have a regular expression R describing some language A. We show how to convert R into an NFA recognizing A. If an NFA recognizes A then A is regular.

1. R = a for some $a \in \Sigma$. Then L(R) = {a}, and the following NFA recognizes L(R).



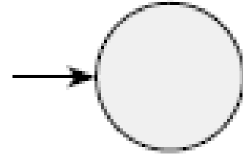
Formally, N = $\{\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\}\}$, where we describe δ by saying that $\delta(q_1, a) = \{q_2\}$ and that $\delta(r, b) = \varphi$; for $r \neq q_1$ or $b \neq a$.

2. R = ε. Then L(R) = {ε}, and the following NFA recognizes L(R).



Formally, N = { {q1}, Σ, δ, q1, {q1}}, where $\delta(r, b) = \varphi$; for any r and b

3. R = φ. Then L(R) = φ, and the following NFA recognizes L(R).



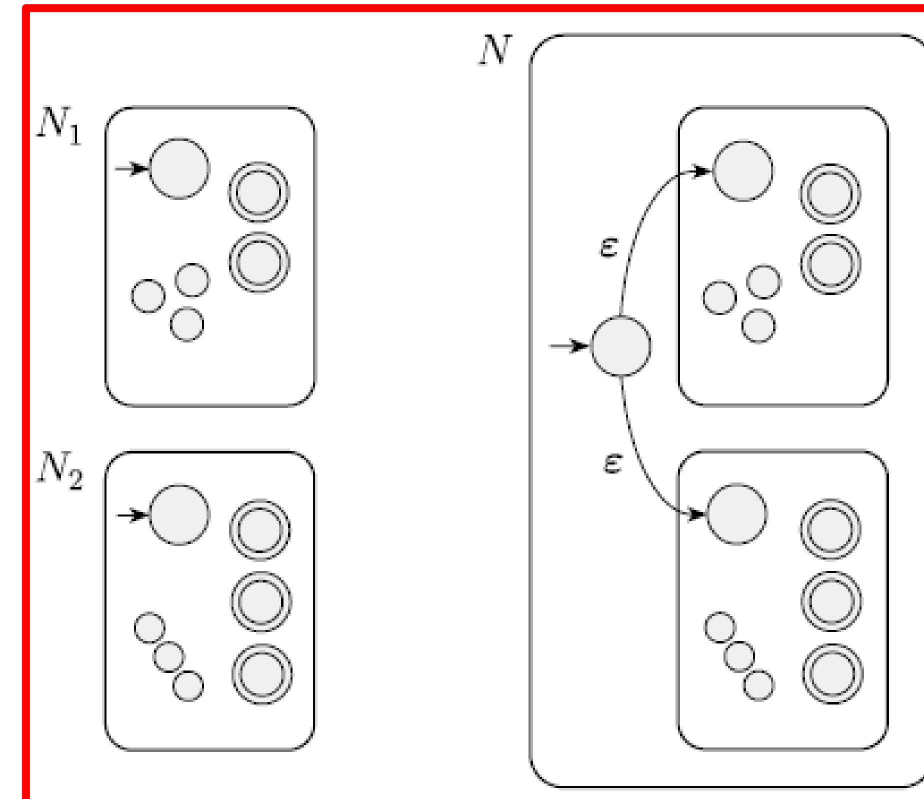
Formally, N ={{q}, Σ, δ, q, φ }, where δ(r, b) = φ for any r and b.

4. $R = R_1 \cup R_2$.

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize R_1, and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize R2. Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize $R_1 \cup R_2$.

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$.
2. The state q_0 is the start state of N.
3. The set of accept states $F = F_1 \cup F_2$.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon. \end{cases}$$

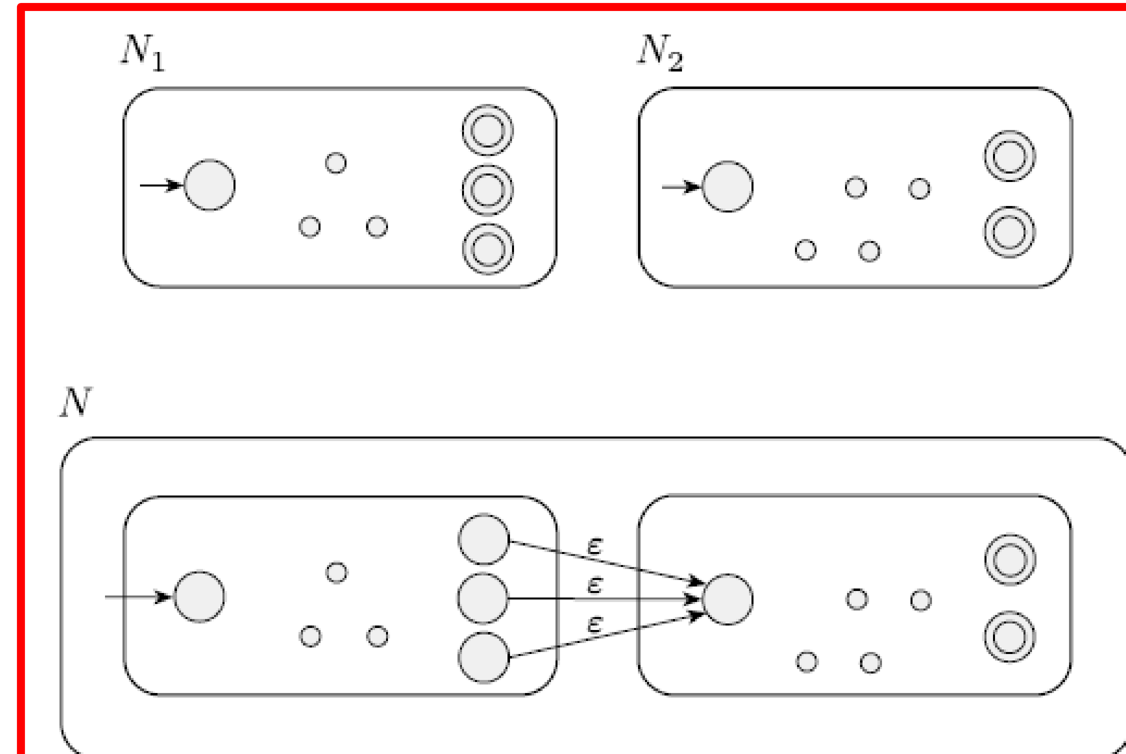


5. $R = R_1 \circ R_2$.

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize R_1, and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize R2. Construct $N = (Q, \Sigma, \delta, q_1, F_2)$ to recognize $R_1 \circ R_2$.

1. $Q = Q_1 \cup Q_2$.
2. The state q_1 is the same as the start state of R1.
3. The accept states F_2 are the same as the accept states of R_2.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1 \text{ and } a = \varepsilon \\ \delta_2(q,a) & q \in Q_2. \end{cases}$$

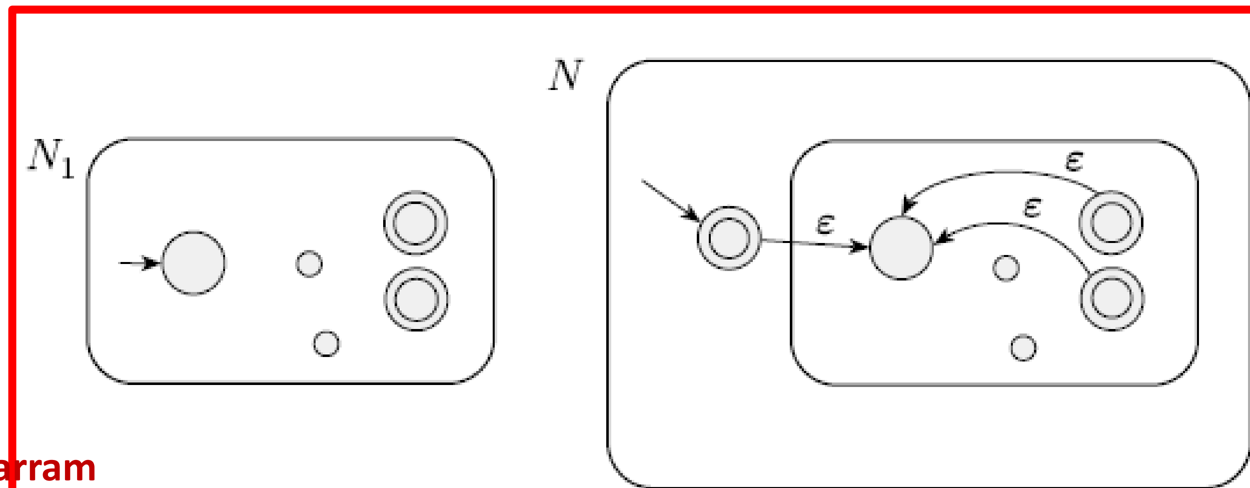


6. R = R*$_1$

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize R_1. Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize R^*_1.

1. $Q = \{q_0\} \cup Q_1$.
2. The state q_0 is the new start state of R.
3. $F=\{q_0\} \cup F$.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

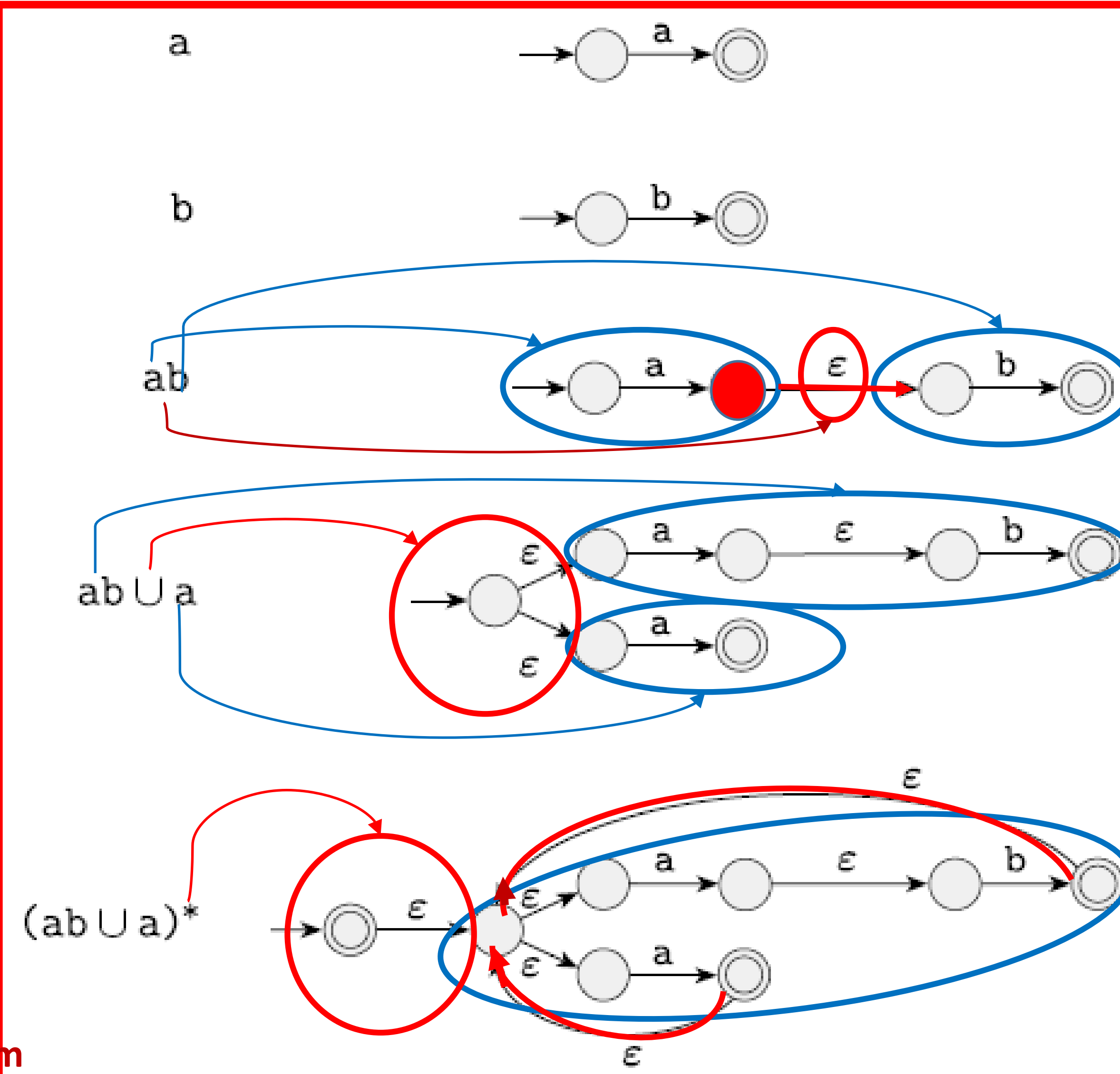
$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ and } a = \varepsilon \\ \{q_1\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon. \end{cases}$$



EXAMPLE 1.56

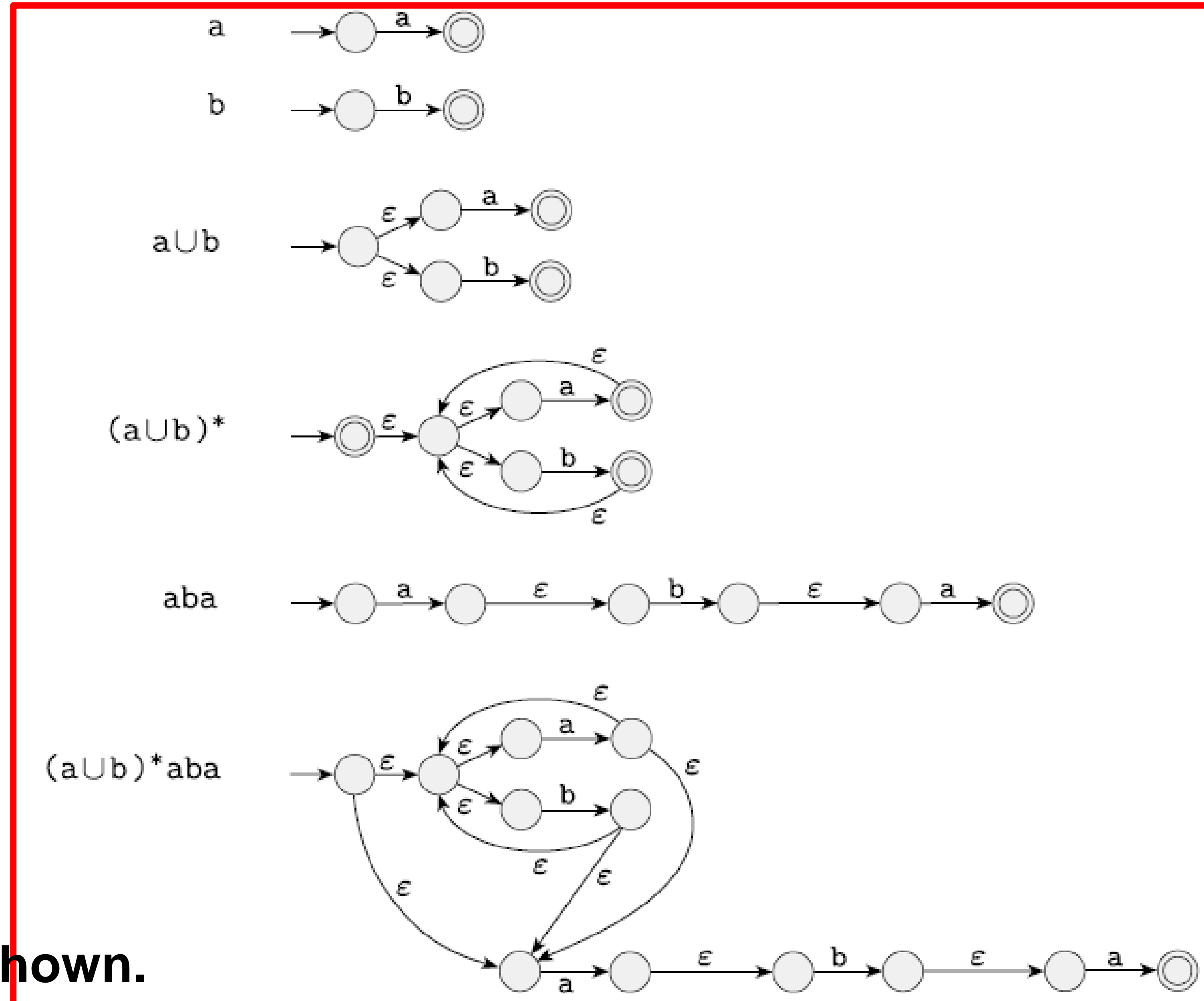
We convert the regular expression $(ab \cup a)^*$ to an NFA in a sequence of stages.

We build up from the smallest subexpressions to larger subexpressions until we have an NFA for the original expression, as shown in the following diagram.
Note that this procedure generally doesn't give the NFA with the fewest states.
In this example, the procedure gives an NFA with eight states, but the smallest equivalent NFA has only two states.



EXAMPLE 1.58

convert the regular expression (a∪b)*aba to an NFA.



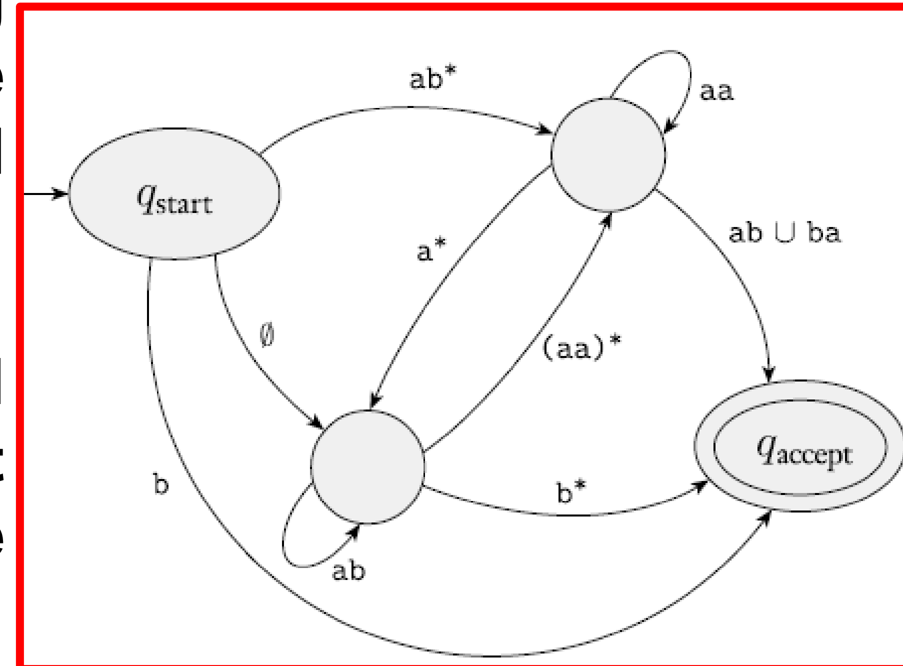
A few of the minor steps are not shown.

Generalized nondeterministic finite automata GNFA **are simply nondeterministic finite automata wherein the transition arrows may have any regular expressions as labels, instead of only members of the alphabet or ε.**

The GNFA reads blocks of symbols from the input, not necessarily just one symbol at a time as in an ordinary NFA.

The GNFA moves along a transition arrow connecting two states by reading a block of symbols from the input, which themselves constitute a string described by the regular expression on that arrow.

A GNFA is nondeterministic and so may have several different ways to process the same input string. It accepts its input if its processing can cause the GNFA to be in an accept state at the end of the input.

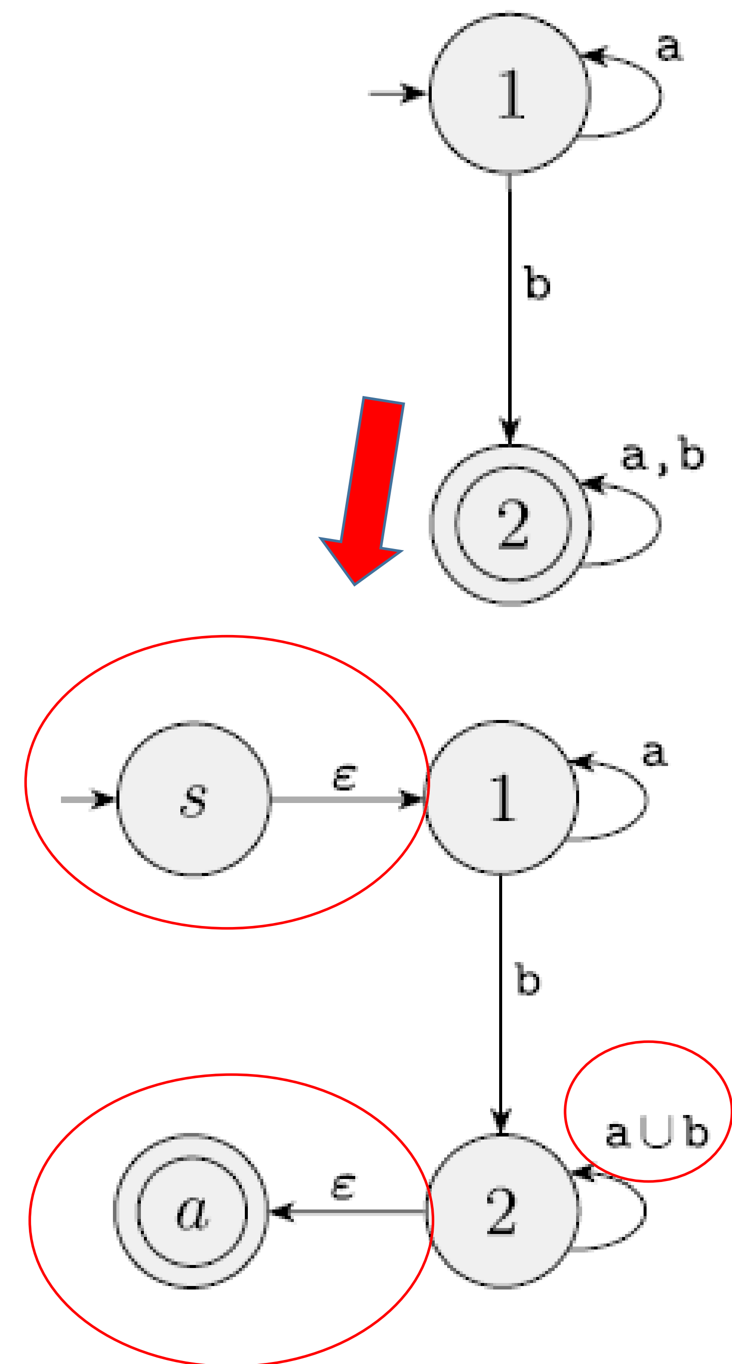


We can easily convert a DFA into a GNFA in the special form.

1- We add a new start state with an ε arrow to the old start state and a new accept state with ε arrows from the old accept states.

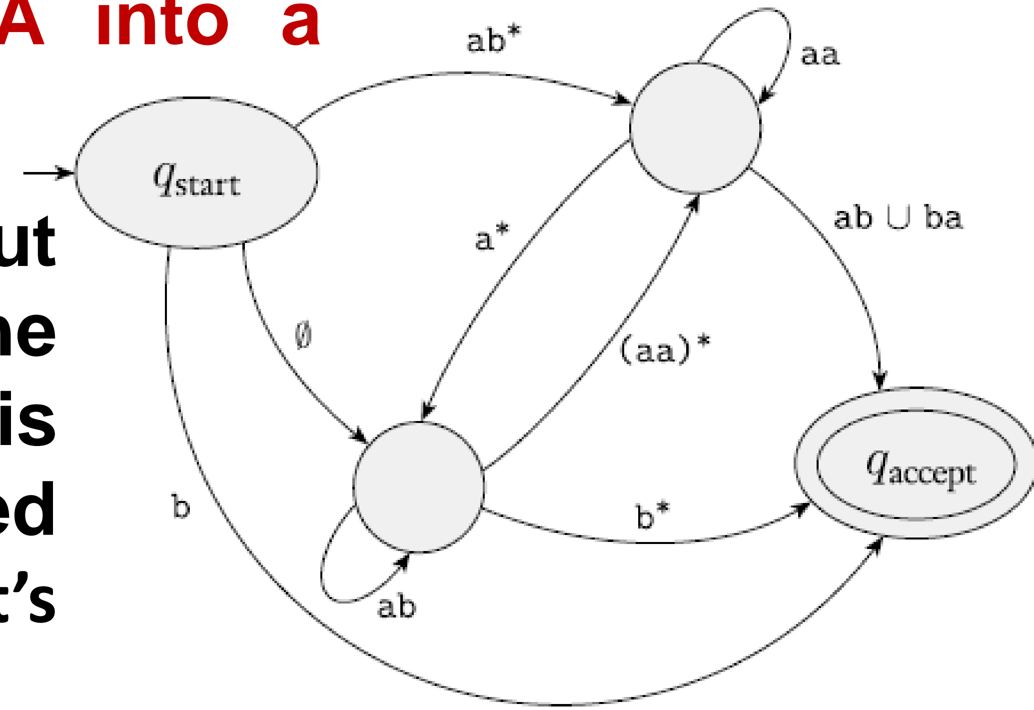
2- If any arrows have multiple labels (or if there are multiple arrows going between the same two states in the same direction), we replace each with a single arrow whose label is the union of the previous labels.

3- We add arrows labeled φ between states that had no arrows.



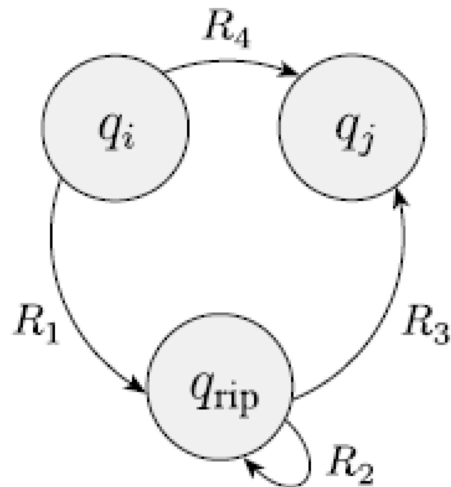
Now we show how to convert a GNFA into a regular expression.

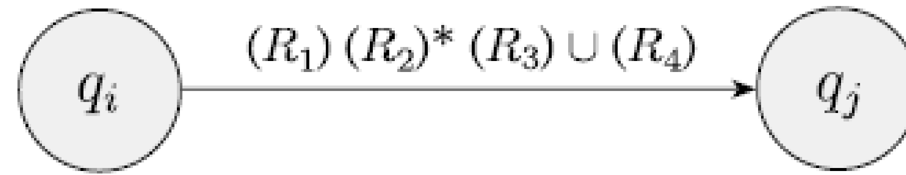
We do so by selecting a state, ripping it out of the machine, and repairing the remainder so that the same language is still recognized. Any state will do, provided that it is not the start or accept state. Let's call the removed state q_{rip}.



A generalized nondeterministic finite automaton

take the machine from q_i to q_j either directly or via q_{rip}.





In the old machine, if

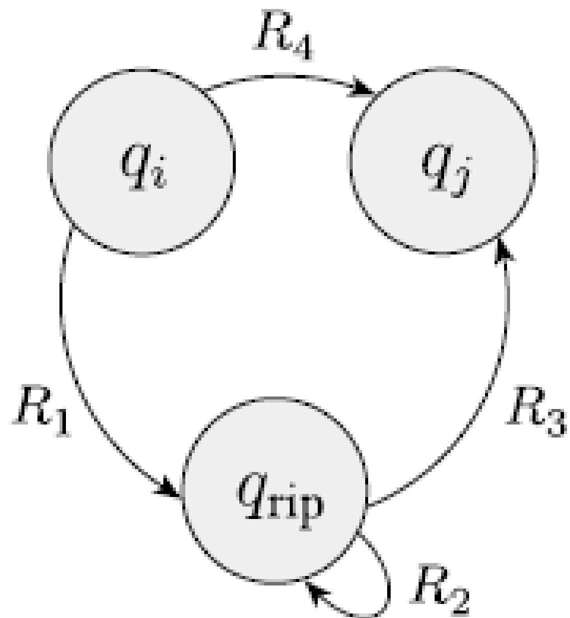
1. q_i goes to q_{rip} with an arrow labeled R_1,

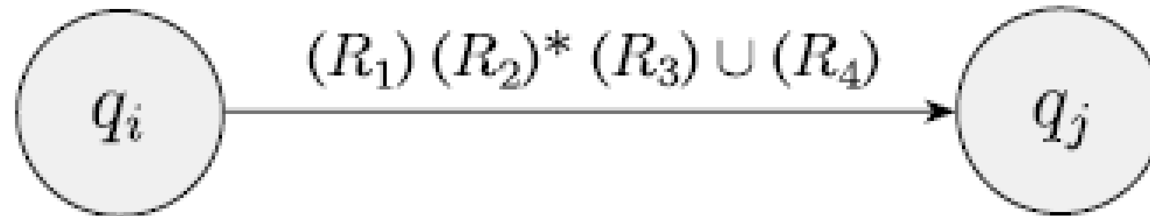
2. q_{rip} goes to itself with an arrow labeled R_2,

3. q_{rip} goes to q_j with an arrow labeled R_3, and

4. q_i goes to q_j with an arrow labeled R_4,

then in the new machine, the arrow from q_i to q_j gets the label $(R_1)(R_2)^*(R_3) \cup (R_4)$.





We use the procedure CONVERT(G), which takes a GNFA and returns an equivalent regular expression.

CONVERT(G):

1. Let k be the number of states of G.

2. If k = 2, then G must consist of a start state, an accept state, and a single arrow connecting them and labeled with a regular expression R. Return the expression R.

3. If k > 2, we select any state $q_{rip} \in Q$ different from q_{start} and q_{accept} and let G' be the GNFA (Q',Σ, δ', q_{start}, $q\mathit{accept}$), where

$$Q' = Q - \{q_{rip}\},$$

and for any $q_i \in Q' - \{q_{accept}\}$ and any $q_j \in Q' - \{q_{start}\}$, let

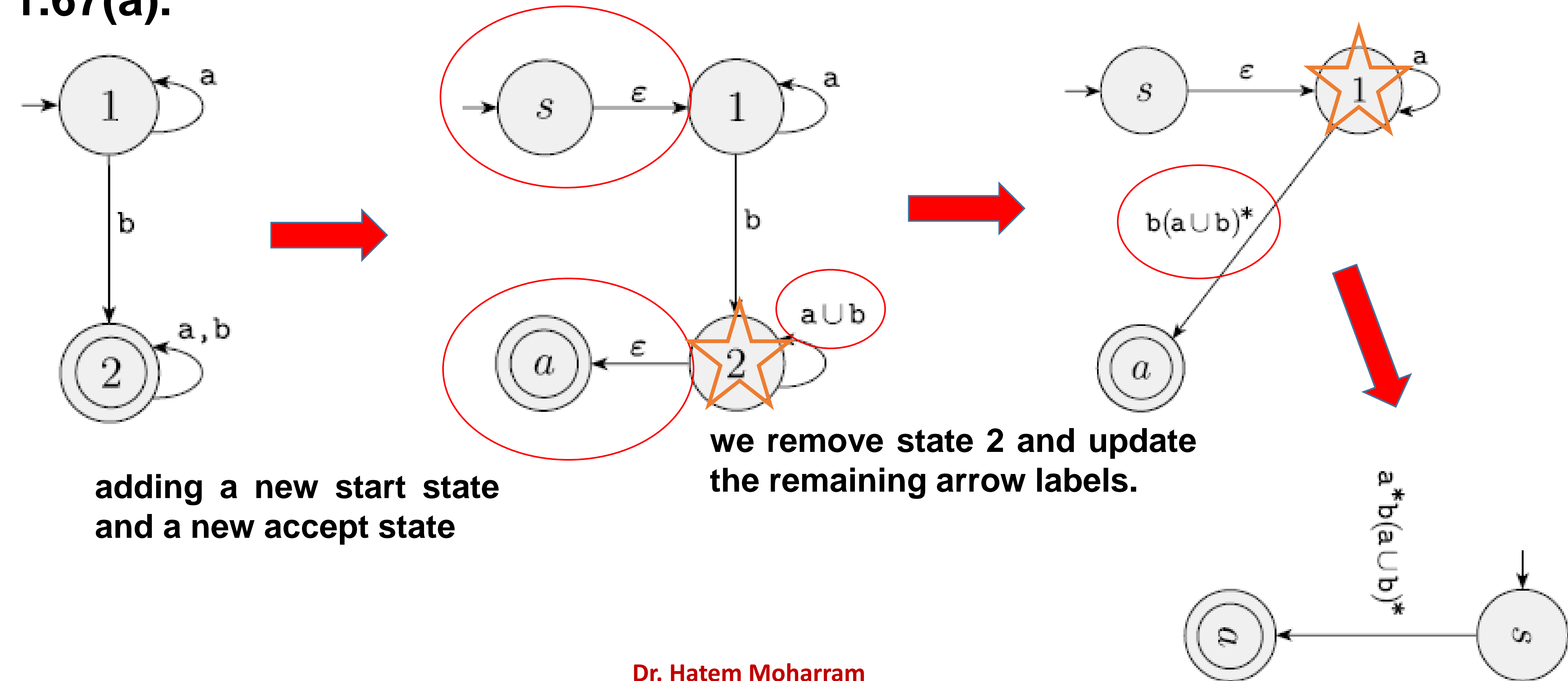
$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R3) \cup (R4),$$

for $R_1 = \delta(q_i, q_{rip})$, $R_2 = \delta(q_{rip}, q_{rip})$, $R_3 = \delta(q_{rip}, q_j)$, and $R_4 = \delta(q_i, q_j)$.

4. Compute CONVERT(G') and return this value.

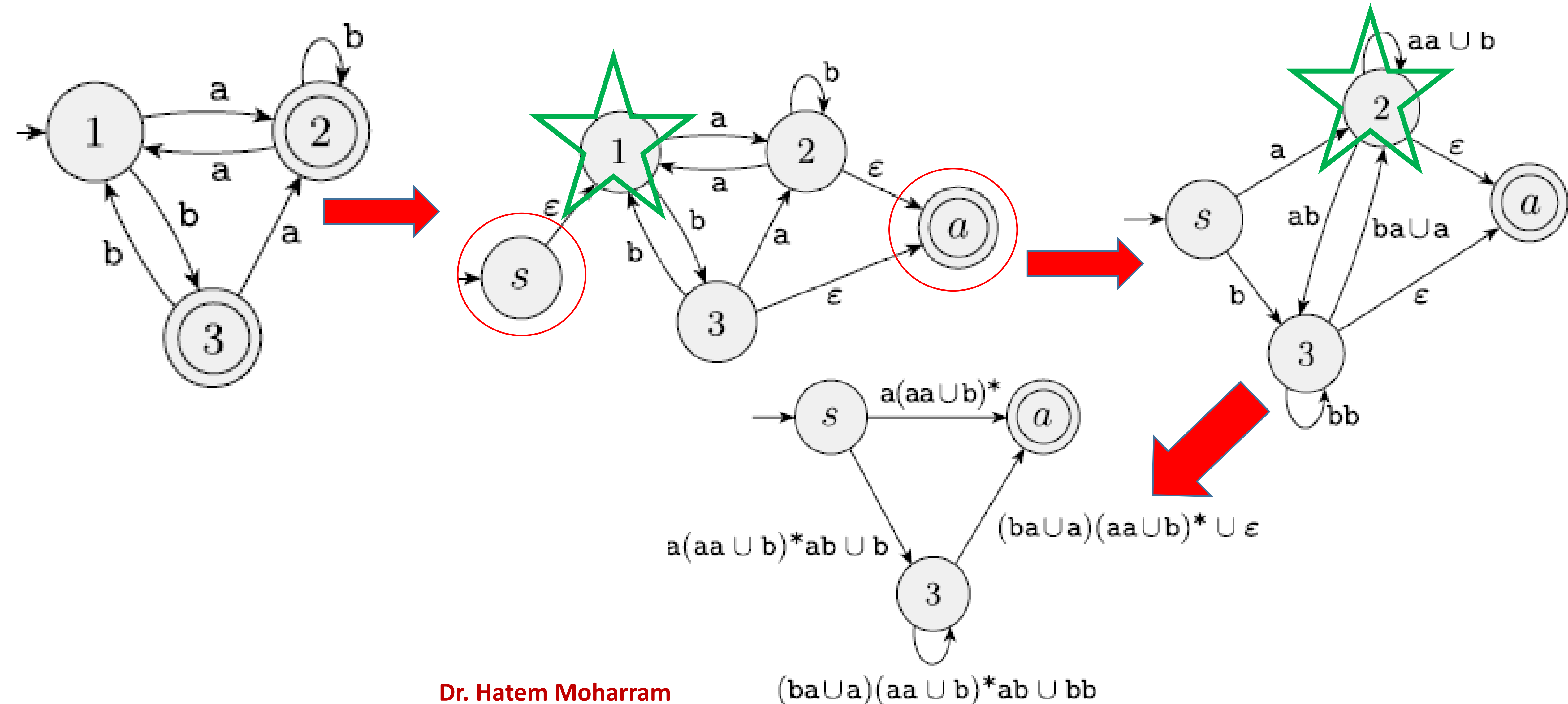
EXAMPLE 1.66

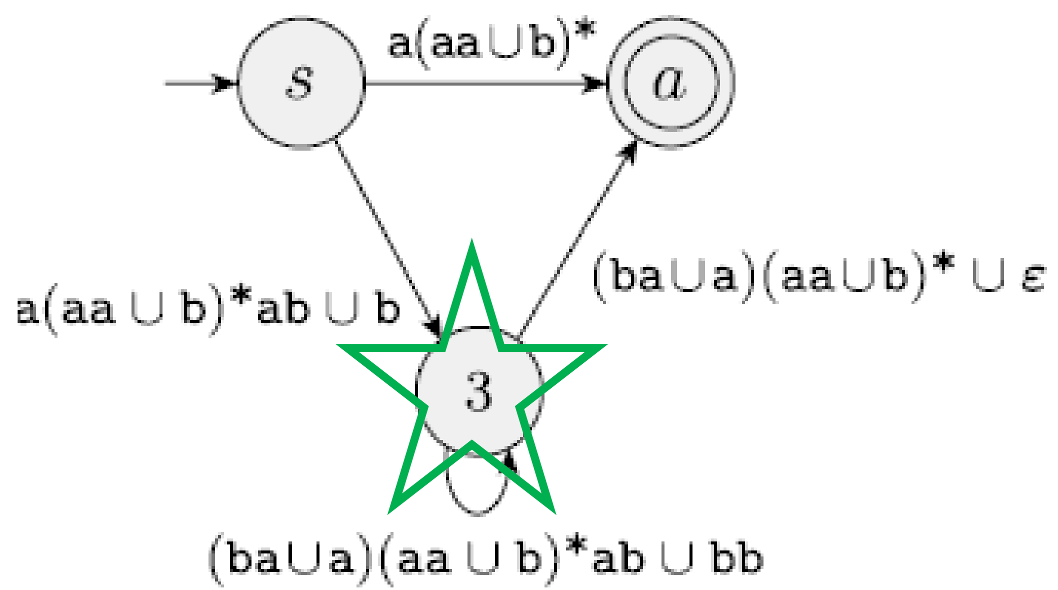
In this example, we use the preceding algorithm to convert a DFA into a regular expression. We begin with the two-state DFA in Figure 1.67(a).



EXAMPLE 1.68

In this example, we begin with a three-state DFA. The steps in the conversion are shown in the following figure.





$(a(aa\cup b)^*ab\cup b)((ba\cup a)(aa\cup b)^*ab\cup bb)^*((ba\cup a)(aa\cup b)^*\cup \varepsilon)\cup a(aa\cup b)^*$

NONREGULAR LANGUAGES

In this section, we show how to prove that certain languages cannot be recognized by any finite automaton.

Let's take the language B = $\{0^n1^n | n \geq 0\}$. If we attempt to find a DFA that recognizes B, we discover that the machine seems to need to remember how many 0s have been seen so far as it reads the input. Because the number of 0s isn't limited, the machine will have to keep track of an unlimited number of possibilities. But it cannot do so with any finite number of states.

because the language appears to require unbounded memory doesn't mean that it is nonregular. It does happen to be true for the language B; but other languages seem to require an unlimited number of possibilities, yet actually they are regular.

For example, consider two languages over the alphabet $\Sigma = \{0,1\}$:

C = {w| w has an equal number of 0s and 1s}, and
D = {w| w has an equal number of occurrences of 01 and 10 as substrings}.

At first glance, a recognizing machine appears to need to count in each case, and therefore neither language appears to be regular. As expected, C is not regular, but surprisingly D is regular!

we show how to prove that certain languages are not regular.

THE PUMPING LEMMA FOR REGULAR LANGUAGES

The *pumping lemma* states that all regular languages have a special property.

If we can show that a language does not have this property, we are guaranteed that it is not regular.

The property states that all strings in the language can be “pumped” if they are at least as long as a certain special value, called the *pumping length*.

That means each such string contains a section that can be repeated any number of times with the resulting string remaining in the language.

THEOREM 1.70

Pumping lemma: If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. **for each $i \geq 0$, $xy^iz \in A$,**
2. **$|y| > 0$, and**
3. **$|xy| \leq p$.**

Recall the notation where $|s|$ represents the length of string s, y^i means that i copies of y are concatenated together, and y^0 equals ε.
When s is divided into xyz, either x or z may be ε, but condition 2 says that $y \neq \varepsilon$. Observe that without condition 2 the theorem would be trivially true. Condition 3 states that the pieces x and y together have length at most p.

EXAMPLE 1.73

Let B be the language $\{0^n1^n | n \geq 0\}$. We use the pumping lemma to prove that B is not regular. The proof is by contradiction.

Solution:

Assume to the contrary that B is regular. Let *p* be the pumping length given by the pumping lemma. Choose $s = 0^p1^p$. Because $s \in B$ and $|s| > p$, the pumping lemma guarantees that *s* can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in B. We consider three cases to show that this result is impossible.

1. The string *y* consists only of 0s. In this case, the string *xyyz* has more 0s than 1s and so is not a member of B, violating condition 1 of the pumping lemma. This case is a contradiction.

2. The string y consists only of 1s. This case also gives a contradiction.

3. The string y consists of both 0s and 1s. In this case, the string *xyyz* may have the same number of 0s and 1s, but they will be out of order with some 1s before 0s. Hence it is not a member of B, which is a contradiction.

Thus a contradiction is unavoidable if we make the assumption that B is regular, so B is not regular.

EXAMPLE 1.74

Let C = {w| w has an equal number of 0s and 1s}. We use the pumping lemma to prove that C is not regular.

Solution

Assume to the contrary that C is regular. Let p be the pumping length given by the pumping lemma. Let s be the string 0^p1^p. With s being a member of C and having length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in C. We would like to show that this outcome is impossible.

But wait, it *is* possible! If we let x and z be the empty string and y be the string 0^p1^p, then xy^iz always has an equal number of 0s and 1s and hence is in C. So it *seems* that s can be pumped.

Here condition 3 in the pumping lemma is useful. It stipulates that when pumping s, it must be divided so that $|xy| \leq p$. That restriction on the way that s may be divided makes it easier to show that the string $s = 0^p1^p$ we selected cannot be pumped. If $|xy| \leq p$, then y must consist only of 0s, so $xyyz \notin C$. Therefore, s cannot be pumped. That gives us the desired contradiction.

EXAMPLE 1.75

Let F = {ww| w ∈ {0,1}*}. We show that F is nonregular, using the pumping lemma.

Solution

Assume to the contrary that F is regular. Let *p* be the pumping length given by the pumping lemma. Let *s* be the string 0^p10^p1. Because *s* is a member of F and *s* has length more than *p*, the pumping lemma guarantees that *s* can be split into three pieces, *s* = *xyz*, satisfying the three conditions of the lemma. We show that this outcome is impossible.

Condition 3 is once again crucial because without it we could pump *s* if we let *x* and *z* be the empty string. With condition 3 the proof follows because *y* must consist only of 0s, so *xyyz* ∉ F.