Formal Languages and the Theory of Computation

Topic 1. Finite automata and regular languages

Han, Nguyen Dinh (han.nguyendinh@hust.edu.vn)

Faculty of Mathematics and Informatics
Hanoi University of Science and Technology

Outline

1

Introduction

What problems can we solve with a computer?

What problems can we solve with a computer?

FACT: *if you pick a totally random problem, the probability that you can solve it is zero*!

We consider **decision problems** - problems with yes/no answers. Examples:

+) Does $123 + 56$ have 3 as a divisor?

+) Is P the shortest path from u to v?

+) Is the program S correct with respect to the initial assertion p and the final assertion q?

NOTE $\rightsquigarrow$ Decision problems do not encompass all possible problems. For instance, "What is $2 + 2$?" is not a decision problem.

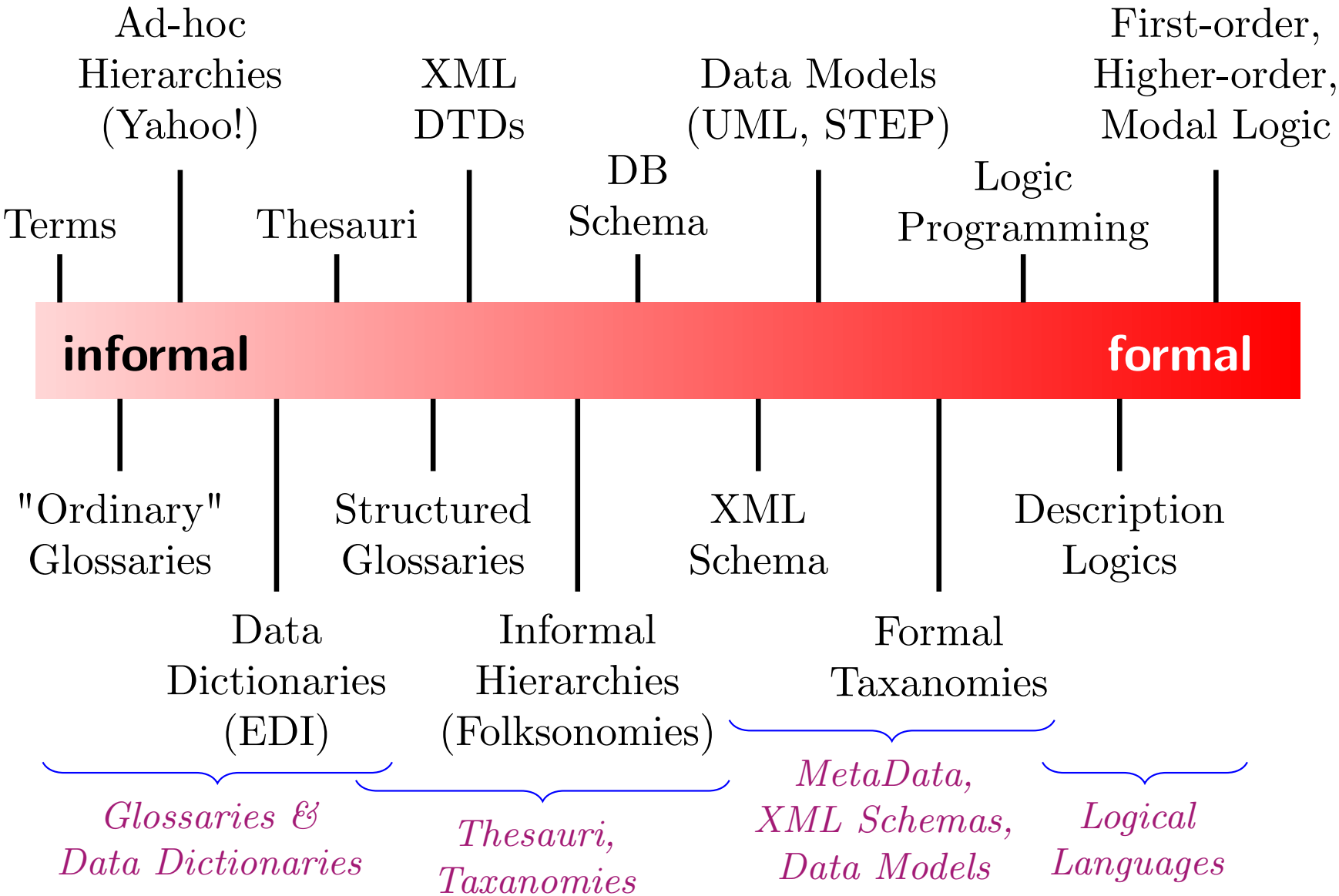
Why restrict ourselves to decision problems?

Many nice mathematical properties:

1) All answers are just one bit, so machines can produce answers more easily
2) No need to worry about what formats the answers will be provided in
3) Easy to use as subroutines

Various problems can be reduced to decision problems. For example, a search problem, which seeks a solution satisfying certain criteria, can be converted into a decision problem by asking whether a solution exists.

How do we encode problems?



How do we encode problems?

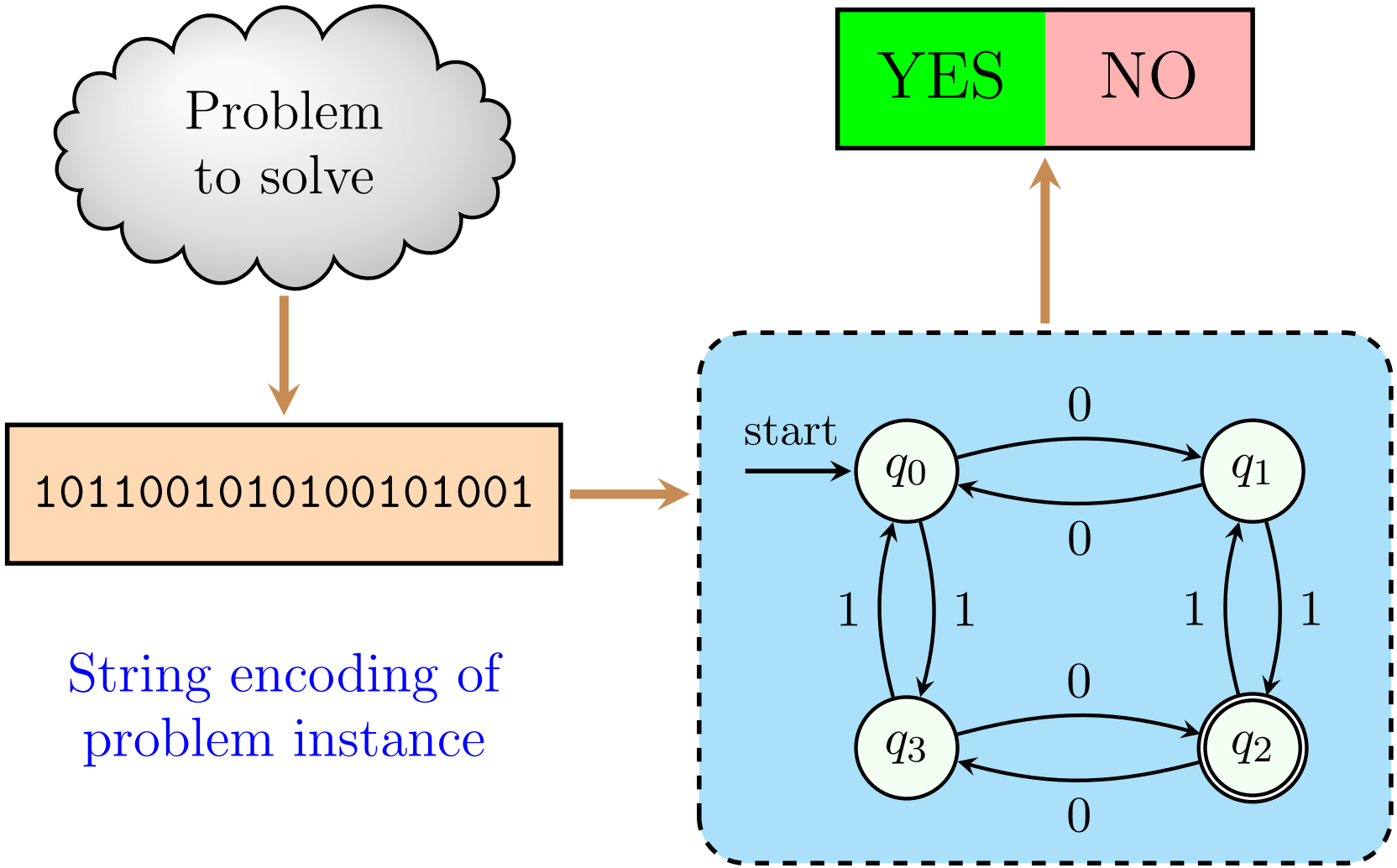
Languages give a compact and flexible way to encode decision problems:

Any decision problem can be
represented by a language of strings
encoding inputs to which the answer is "yes."

Therefore, all the automata we will discuss in this class will be machines for answering the question:

Is string ω in language L?

How do we encode problems?



2

Alphabets and languages

Strings of characters are fundamental building blocks in computer science!

The *alphabet* over which the strings are defined may vary with the application. Here, we define an **alphabet** to be any nonempty finite set of *symbols*. Examples of alphabets:

+) The binary alphabet: $A = \{0, 1\}$

+) The Roman alphabet: $\Sigma = \{a, b, c, \ldots, z\}$

+) The arbitrary alphabet: $\Gamma = \{0, 1, x, y, z\}$

NOTE ⇝ An alphabet can be a finite set of any sort. For simplicity, however, we use as symbols only *letters*, *numerals*, and other characters such as \$, or ♯.

A string over an alphabet is a finite sequence of symbols from that alphabet. Symbols of a string are written next to one another and not separated by commas and parentheses. Examples of strings:

+) 01001 is a string over $A = \{0, 1\}$

+) *abbaaba* is a string over $\Sigma = \{a, b, c, \ldots, z\}$

If ω is a string over Σ, the **length** of ω, denoted by $|\omega|$, is the number of symbols that it contains. For example, the lengths of the strings 01001 and *abc* are 5 and 3, respectively. The string of length zero is called the *empty string* and is written ε.

If ω has length n, we can write $\omega = \omega_1\omega_2\cdots\omega_n$ where each $\omega_i \in \Sigma, i = \overline{1,n}$. The *reverse* of ω, denoted by ω^R, is the string obtained by writing ω in the opposite order (i.e. $\omega^R = \omega_n\cdots\omega_2\omega_1$). For example, $reverse^R = esrever$.

String z is a **substring** of ω if z appears consecutively within ω. For example, substrings of 01001 can be 0, 01 and 100. If we have string x of length m and string y of length n, the **concatenation** of x and y is the string $xy = x_1\cdots x_m y_1\cdots y_n$. To concatenate the string x with itself $k\,(k \geq 1)$ times, we use the superscript notation x^k.

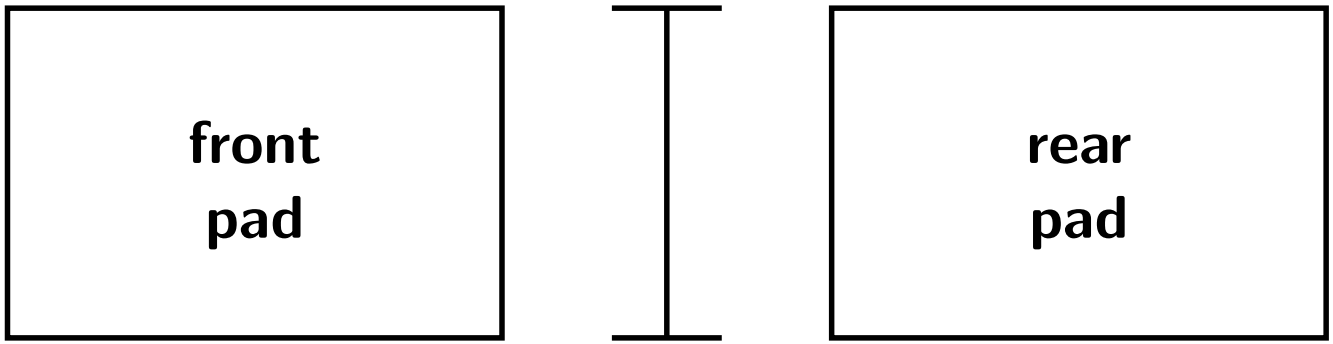
A **language** is a set of strings!

3

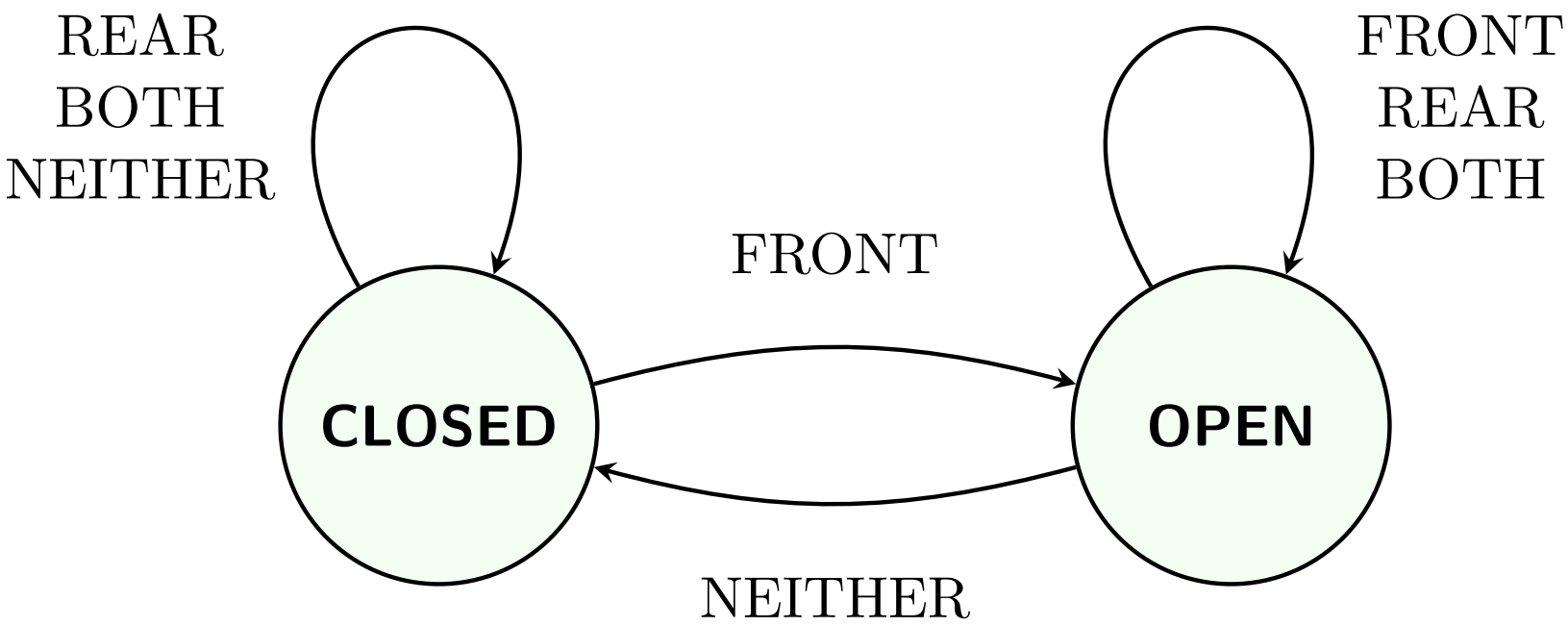
Finite automata

We need a simpler way of discussing computing machines. An **automaton** (plural: automata) is a *mathematical model* of a computing device. Automata make it possible to reason about computability by abstracting away the implementation complexity of real computing systems.

Below, we consider the controller for an automatic door as an example.



The controller is in either of two states: OPEN or CLOSED, representing the corresponding condition of the door. The state diagram for the controller is as follows



The controller moves from state to state, depending on the input it receives. The state transition table for the controller is as follows

state \ input signal	NEITHER	FRONT	REAR	BOTH
CLOSED	CLOSED	OPEN	CLOSED	CLOSED
OPEN	CLOSED	OPEN	OPEN	OPEN

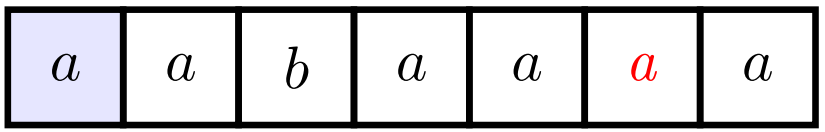
The model of computation we have explored corresponds to different conceptions of what a computer could do. Indeed, the controller is a computer that has a single bit of memory, capable of recording which of the two states the controller is in.

Finite automata are an abstraction of computers with finite resource constraints. Thinking of an automatic door controller as a finite automaton is useful because that suggests standard ways of representation.

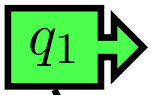
Exercise: Now, try to implement the controller in Python!

3.1

Deterministic finite automata

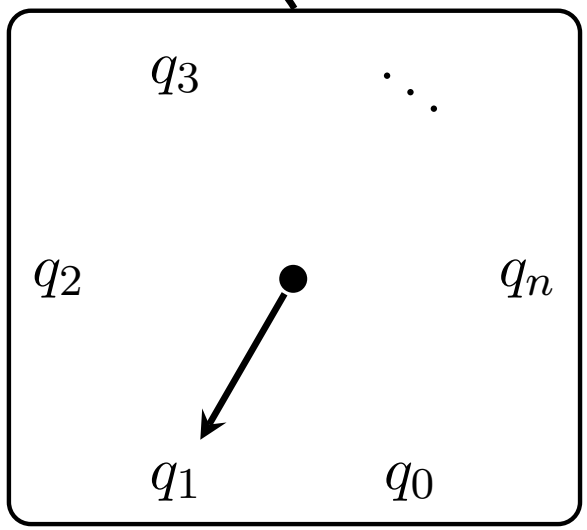


Input tape



Reading head
(moves to the right only)

When the machine is in a given state and reads the next input symbol, we know what is the next state will be - it is so called a *deterministic finite automaton* (DFA for short).



Finite control

Definition 1.1 A *deterministic finite automaton* is a 5−tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the *states*,
2. Σ is a finite set called the *alphabet*,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the *transition function*,
4. q_0 is the *start state* (or the **initial state**),
5. $F \subseteq Q$ is the *set of accept states* (or the **final states**).

Definition 1.2 Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton. Then

1) A *configuration* of M is any element of $Q \times \Sigma^*$, where $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$ is the set of all strings over Σ.

2) The binary relation $\vdash_M$ holds between two configurations of M if and only if the machine can pass from one to the other as a result of a single move (i.e. if (q, ω) and (q', ω') are two configurations of M, then $(q, \omega) \vdash_M (q', \omega')$ if and only if $\omega = a\omega'$ for some symbol $a \in \Sigma$, and $\delta(q, a) = q'$). In this case we say that (q, ω) **yields** (q', ω') **in one step**.

Definition 1.2 (cont.) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton. Then

3) We denote the reflexive, transitive closure of $\vdash_M$ by $\vdash_M^*$. Now, $(q, \omega) \vdash_M^* (q', \omega')$ is read, (q, ω) **yields** (q', ω') (after some number, possibly zero, of steps).

4) A string $\omega \in \Sigma^*$ is said to be **accepted** by M if and only if there is a state $q \in F$ such that $(q_0, \omega) \vdash_M^* (q, \varepsilon)$.

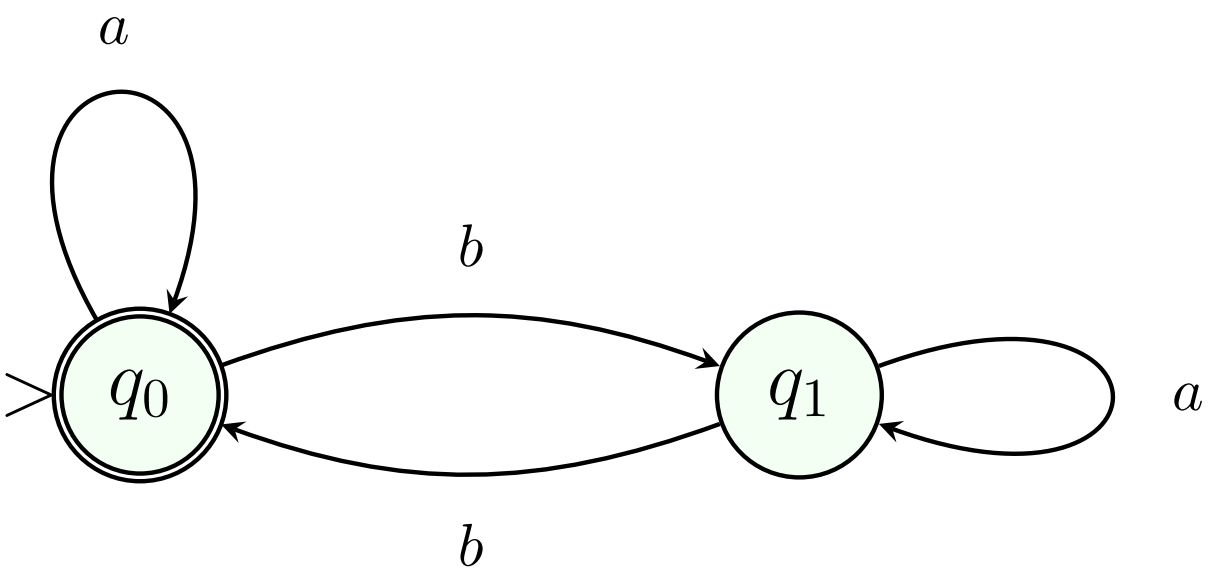
5) **The language accepted by** M, $L(M)$, is the set of all strings accepted by M.

Example 1.1 Let M be the deterministic finite automaton $(Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $F = \{q_0\}$, and δ is the function tabulated below.

$q,$	σ	$\delta(q, \sigma)$
q_0	a	q_0
q_0	b	q_1
q_1	a	q_1
q_1	b	q_0

$L(M)$ is the set of all strings in $\{a, b\}^*$ that have an even number of b's. Now, let check how M accepts some strings in $L(M)$.

The tabular representation of the transition function used in the previous example is not the clearest description of a machine. We generally use a more convenient graphical representation called the **state diagram**. The state diagram of the DFA of Example 1.1 is given below.



Definition 1.3 (Formal definition of computation) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton and let $\omega = \omega_1\omega_2 \cdots \omega_n$ be a string where each ω_i is a member of the alphabet Σ. Then M **accepts** ω if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, \omega_{i+1}) = r_{i+1}$ for $i = 0, 1, \ldots, n-1$, and
3. $r_n \in F$.

Example 1.2 Let $M = (Q, \Sigma, \delta, q_0, F)$ be the DFA from Example 1.1. If M is given the input $\omega = aabbaa$, its initial configuration is $(q_0, \underline{a}abbaa)$. Then

$$
\begin{aligned}
(q_0, \underline{a}aabbaa) &\vdash_M (q_0, \underline{a}abbaa) \vdash_M (q_0, \underline{b}baa) \\
&\vdash_M (q_1, \underline{b}aa) \vdash_M (q_0, \underline{a}a) \\
&\vdash_M (q_0, \underline{a}) \vdash_M (q_0, \underline{\varepsilon})
\end{aligned}
$$

The sequence of states M enters when computing on ω is $r_0 = q_0, q_0, q_0, q_1, q_0, q_0, q_0 = r_6 \in F$. Thus, M accepts ω according to the formal definition of computation.

Problem 1.1 Let $\Sigma = \{a, b\}$ be an alphabet. Construct deterministic finite automata accepting each of the following languages.

(a) $\{\omega \in \Sigma^*$: each a in ω is immediately preceded by a $b\}$.

(b) $\{\omega \in \Sigma^*$: ω has $abab$ as a substring$\}$.

(c) $\{\omega \in \Sigma^*$: ω has neither aa nor bb as a substring$\}$.

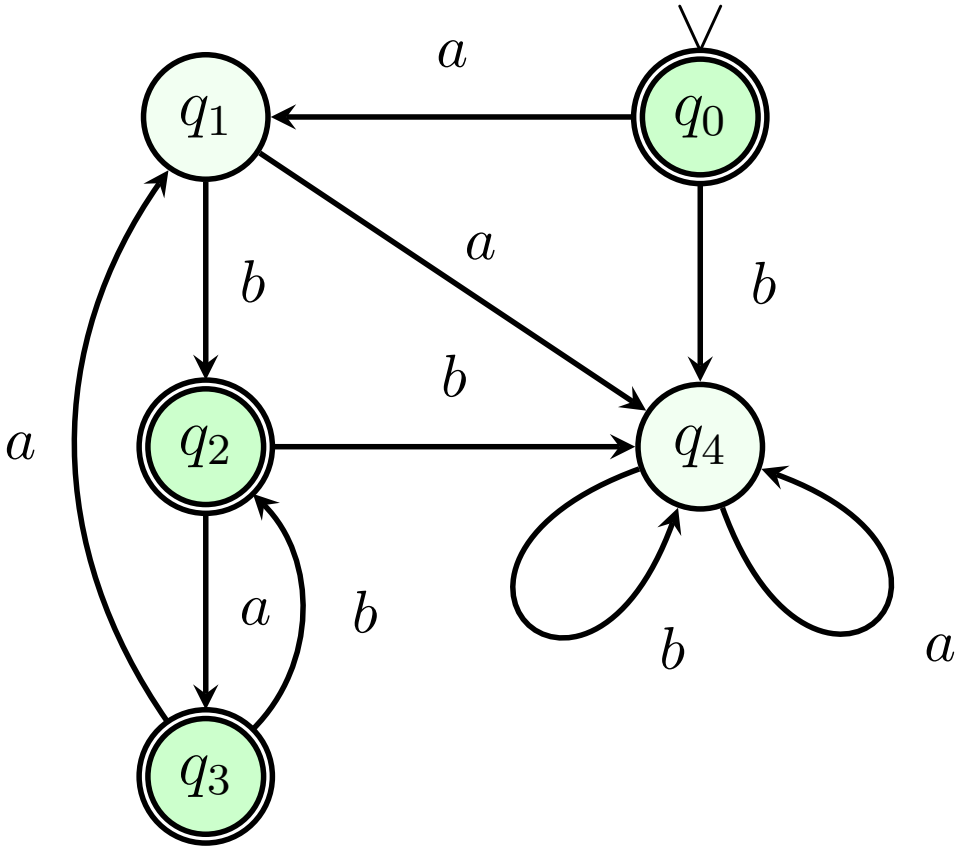
(d) $\{\omega \in \Sigma^*$: ω has an odd number of a's and an even number of b's$\}$.

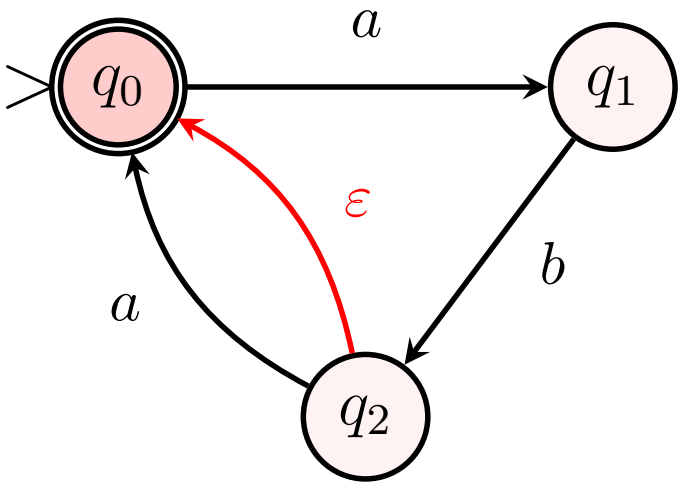
(e) $\{\omega \in \Sigma^*$: ω has both aa and ba as substrings$\}$.

3.2

Nondeterministic finite automata

3.2. Nondeterministic finite automata





These two automata accept the same language:

$\{\varepsilon, ab, aba, abab, \ldots\}$

3.2. Nondeterministic finite automata

The automaton, as it reads the input string, may have several choices for the next state at each step of its computation, is said to be **nondeterministic**.

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

The formal definition of an NFA is similar to that of a DFA. However, they differ in the type of transition function. 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.*

Definition 1.4 A *nondeterministic finite automaton* is a 5−tuple $N = (Q, \Sigma, \delta, q_0, F)$, where

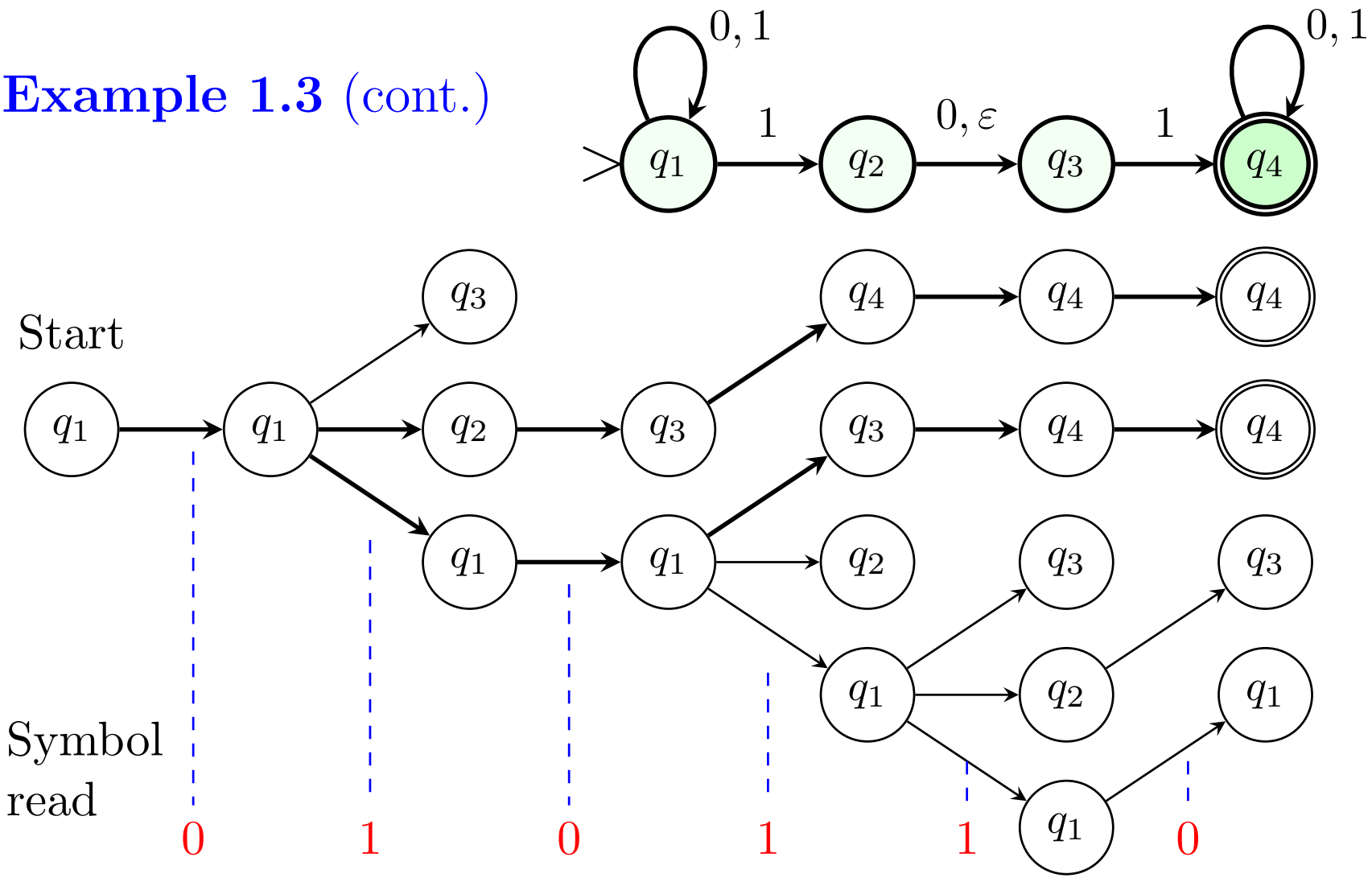
1. Q is a finite set of *states*,
2. Σ is a finite *alphabet*,
3. $\delta : Q \times \left(\Sigma \cup \{\varepsilon\}\right) \longrightarrow \mathcal{P}(Q)$ is the *transition function*. $\mathcal{P}(Q)$, the collection of all subsets of Q, is called the *power set* of Q,
4. $q_0 \in Q$ is the *initial state*, and
5. $F \subseteq Q$ is the *set of accept states*.

Example 1.3 Let N be the nondeterministic finite automaton $(Q, \Sigma, \delta, q_1, F)$, where $Q = \{q_1, q_2, q_3, q_4\}$, $\Sigma = \{0, 1\}$, $F = \{q_4\}$, and δ is the function tabulated below.

q,	σ	$\delta(q, \sigma)$	q,	σ	$\delta(q, \sigma)$	q,	σ	$\delta(q, \sigma)$
q_1	0	$\{q_1\}$	q_1	1	$\{q_1, q_2\}$	q_1	ε	$\emptyset$
q_2	0	$\{q_3\}$	q_2	1	$\emptyset$	q_2	ε	$\{q_3\}$
q_3	0	$\emptyset$	q_3	1	$\{q_4\}$	q_3	ε	$\emptyset$
q_4	0	$\{q_4\}$	q_4	1	$\{q_4\}$	q_4	ε	$\emptyset$

The state diagram for δ and the computation of N on input 010110 will be given in the next slide.

Example 1.3 (cont.)

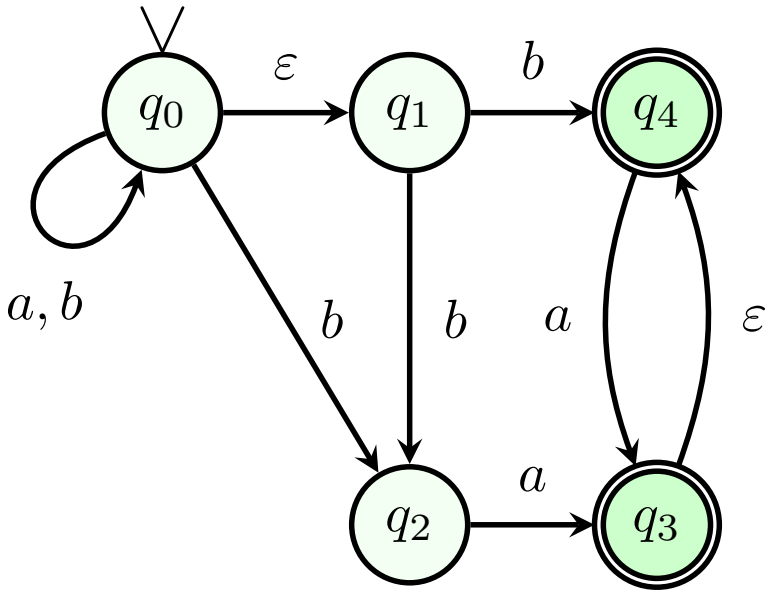


The formal definition of computation for an NFA is similar to that for a DFA. Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and $\omega = \omega_1\omega_2 \cdots \omega_n$ be a string over $\Sigma \cup \{\varepsilon\}$. Then, we say that N **accepts** ω if and only if there is a state $q \in F$ such that $(q_0, \omega) \vdash_M^* (q, \varepsilon)$. Furthermore, $L(N)$, **the language accepted by** N, is the set of all strings accepted by N.

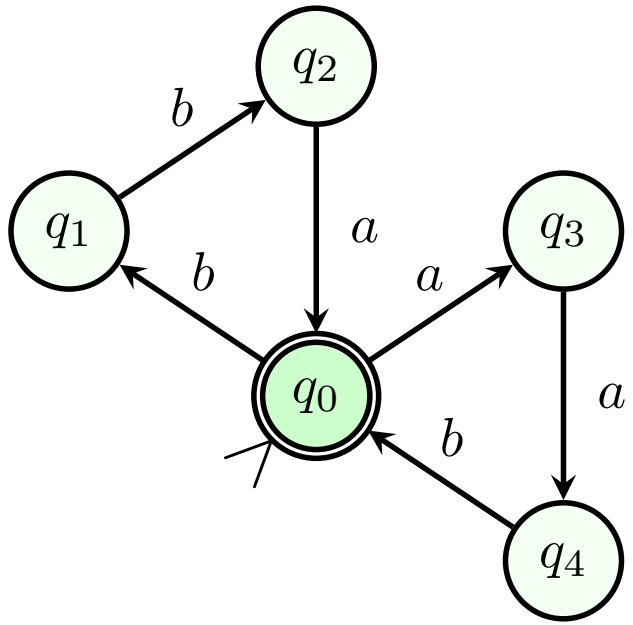
Recall that, a DFA is just *a special type* of an NFA. NFAs are no more powerful than the DFAs in terms of the languages they accept. So, we have

Theorem 1.1 *Every nondeterministic finite automaton has an equivalent deterministic finite automaton.*

Problem 1.2 Construct DFAs equivalent to the following NFAs.



(a)



(b)

Problem 1.3 Draw state diagrams for nondeterministic finite automata that accept the following languages.

(a) $\{ab, ba, aa\}^*$.

(b) $\{ba, b, bb, a\}^*$.

(c) $\{\omega \in \{a_1, a_2, \ldots, a_n\}^*, n \geq 2$: there is a symbol a_i not appearing in $\omega\}$.

(d) $\{\omega \in \{a, b\}^*$: ω contains an occurrence of the pattern bb or of the pattern $bab\}$.

(e) $\{\omega \in \{0, 1\}^*$: ω contains an occurrence of the pattern 01 or of the pattern 010$\}$.

4

Regular languages

4.1

Finite representations of languages

4.1. Finite representations of languages

The representation of languages by finite specifications is *a central issue* in the theory of computation. Recall that, any set of strings over an alphabet Σ will be called a *language*. Thus, we can specify a finite language by listing all its strings (e.g. $\{ab, aba, bb\}$ is a language over $\{a, b\}$). However, we cannot apply this way to specify any infinite language.

In this section, we study finite representations of useful and interesting languages defined below

Definition 1.5 *A language is called a regular language if some finite automaton (deterministic or nondeterministic) accepts/recognizes it.*

As a result in the theory of computation, we are unable to represent all languages finitely. But we want to derive ways of exhibiting particular languages that cannot be represented by various representational methods we study.

By Definition 1.5, we know that the class of languages accepted by finite automata is the same as the class of regular languages. Next, we show how to prove that certain languages are not regular.

Theorem 1.5 *Let L be a regular language. There is an integer $n \geq 1$ such that any string $\omega \in L$ with $|\omega| \geq n$ can be rewritten as $\omega = xyz$ such that $y \neq \varepsilon, |xy| \leq n$, and $xy^i z \in L$ for each $i \geq 0$.*

Theorem 1.5, traditionally called the *pumping theorem*, states that all regular languages have a special property. Hence, to prove that a language is not regular, we show it does not have this property.

Example 1.4 Show that the language $L = \{a^i b^i : i \geq 0\}$ over the alphabet $\Sigma = \{a, b\}$ is not regular.

Proof: Suppose that L is regular. Next, consider the string $\omega = a^n b^n \in L$. By Theorem 1.5, it can be rewritten as $\omega = xyz$ such that $|xy| \leq m = 2n$ and $y \neq \varepsilon$, that is $\omega = a^{n-1}ab^n$, where $x = a^{n-1}, y = a, z = b^n$. However, for $i = 0$, $xy^iz = a^{n-1}b^n \notin L$, contradicting the theorem. □

Can you prove that

the language $\{0^n 1^n 2^n \mid n \geq 0\}$ is not regular?

4.2

Regular expressions

This section we continue our study of finite representations of languages by *regular expressions* - that describe how languages can be built up. But we need basic concepts that allow us to construct regular expressions.

Definition 1.6 Let A and B be languages. We define *the regular operations* **union**, **concatenation**, and **star** as follows

1. **Union**: $A \cup B = \{x \,|\, x \in A \text{ or } x \in B\}$.
2. **Concatenation**: $A \circ B = \{xy \,|\, x \in A \text{ and } y \in B\}$.
3. **Star**: $A^* = \{x_1 x_2 \cdots x_k \,|\, k \geq 0 \text{ and } x_i \in A\}$.

Example 1.5 Let $A = \{good, bad\}$ and $B = \{boy, girl\}$ be languages over $\Sigma = \{a, b, \ldots, z\}$. Then

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

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

$$A^* = \{\varepsilon, good, bad, goodgood, goodbad, \cdots\}.$$

NOTE $\rightsquigarrow$ In arithmetic, the basic objects are numbers and operations for manipulating them can be $+$ or $\times$, such as $3+4\times5 = 23$. In the theory of computation, the objects are languages and regular operations designed for manipulating them.

Now we use the regular operations to build up *regular expressions* describing languages as defined below.

Definition 1.7 We say R is a regular expression if R is

1. $\emptyset$,
2. ε,
3. a for some a in the alphabet Σ,
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.

Example 1.6 Let $\Sigma = \{0, 1\}$ be an alphabet. Then

1. $0^*10^* = \{\omega \,|\, \omega \text{ contains a single } 1\}$.
2. $\Sigma^*1\Sigma^* = \{\omega \,|\, \omega \text{ has at least one } 1\}$.
3. $\Sigma^*001\Sigma^* = \{\omega \,|\, \omega \text{ contains the string } 001 \text{ as a substring}\}$.
4. $1^*\emptyset = \emptyset$ (i.e. concatenate the empty set to any set yields the empty set). $\emptyset^* = \{\varepsilon\}$.
5. $(0^*) \circ (1) \circ (0)^* = 0^*10^*$ (i.e. Parentheses and the notation $\circ$ may be omitted). $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.

4.3

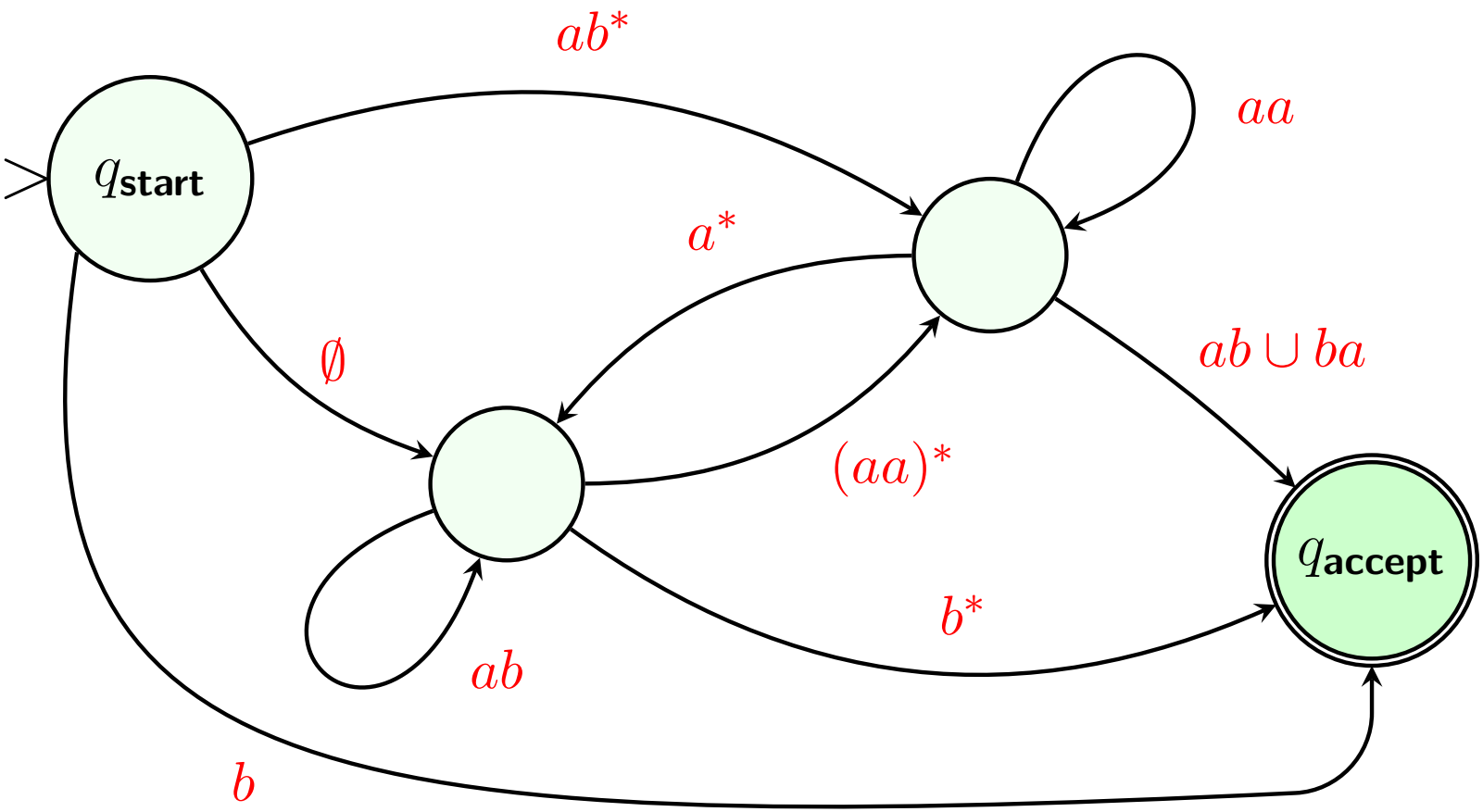
Equivalence of regular expressions and finite automata

We now come to the main result of this section, the identification of two important techniques for finitely specifying languages - in fact, of a language acceptor and a language generator.

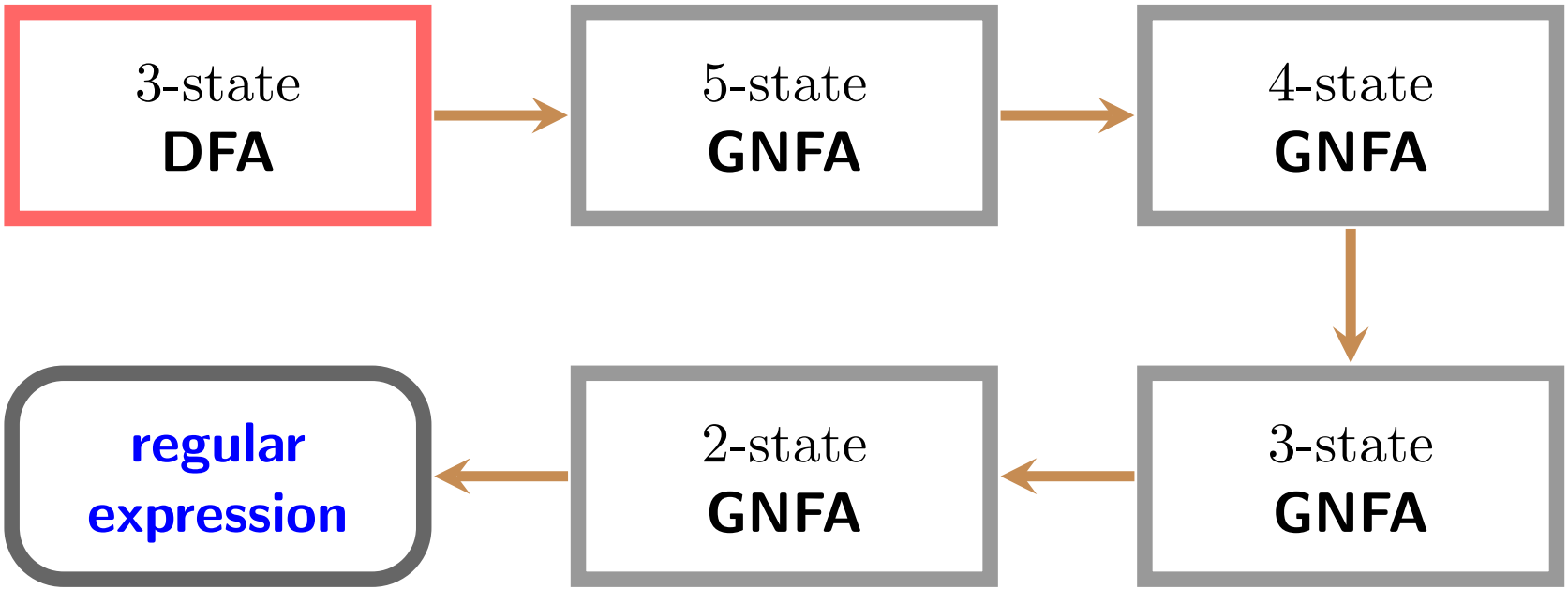
Theorem 1.3 *A language is regular if and only if some regular expression describes it.*

Recall that a regular language is one that is accepted by some finite automaton. Thus, the theorem affirms that regular expressions and finite automata are equivalent in their description power. Indeed any regular expression can be converted into a finite automaton that recognizes the language it describes and vice versa.

4.3. Equivalence of regular expressions and finite automata



A generalized nondeterministic finite automaton



Typical stages in converting a DFA to a regular expression

NOTE ⇝ The procedures for converting a finite automaton to a regular expression and vice versa are given by lemmas of Theorem 1.54 in our textbook.

Can you convert

the regular expression $a(abb)^* \cup b$ to an NFA

using the procedure

given in Theorem 1.54 of our textbook?

Problem 1.4 Give regular expressions generating the following languages.

(a) $\{ab, ba, aa\}^*$.

(b) $\{ba, b, bb, a\}^*$.

(c) $\{\omega \in \{a_1, a_2, \ldots, a_n\}^*, n \geq 2$: there is a symbol a_i not appearing in $\omega\}$.

(d) $\{\omega \in \{a, b\}^*$: ω contains an occurrence of the pattern bb or of the pattern $bab\}$.

(e) $\{\omega \in \{0, 1\}^*$: ω contains an occurrence of the pattern 01 or of the pattern 010$\}$.

Thank you very much for your attention!