State Machines and Finite Automata

State machines and their state diagrams, runs, deterministic finite automata and their languages, designing automata, and state-transition tables

Last modified: 2026-11-12 01:15:49 (PST)

Source

The topics of this chapter follow the course calendar, and their order follows Kameron Decker Harris's Fall 2026 lecture notes introducing the theory of computation. These come from those notes:

- the three parts of the theory of computation, computability, complexity and automata theory, with the halting problem and the P versus NP problem as examples of the first two;
- automata as the precise form of "computer", and the question of whether a machine accepts a string as the precise form of "problem";
- a state machine drawn as a directed graph, with an arrow into the initial state and a double circle for an accepting state.

These follow the course's handout on graph extensions:

- a finite automaton seen as a directed graph whose vertices are states and whose labeled edges are transitions, with a start state and accepting states;
- reading a string one symbol at a time, following the edge labeled with each symbol, and accepting the string when the machine ends in an accepting state.

The transition function, runs, the extended transition function, languages, dead states, complements and state-transition tables are added here. The notes do not cover nondeterministic automata or regular expressions, so this chapter does not either. The examples, the exercises, the Racket functions and the proofs are new.

State machines

Exercise 1 (A door with a lock)**.** A door is always in one of three conditions: closed, open or locked. Four actions can be done to it:

- `o`: push it open, which opens a closed door and does nothing to a locked or an open one;
- `c`: shut it, which closes an open door and does nothing otherwise;
- `l`: turn the key to lock, which locks a closed door and does nothing otherwise;
- `u`: turn the key to unlock, which unlocks a locked door, leaving it closed, and does nothing otherwise.

The door starts closed, and these actions are done in this order:

$$\texttt{o}, \texttt{l}, \texttt{c}, \texttt{l}, \texttt{o}, \texttt{u}, \texttt{o}.$$

In which condition is the door at the end?

Solution

Solution 1. Follow the door one action at a time:

action		`o`	`l`	`c`	`l`	`o`	`u`	`o`
condition	closed	open	open	closed	locked	locked	closed	open

The first `l` does nothing, because the door is open, and the second `o` does nothing, because the door is locked. At the end the door is open.

Remark 1 (Where state machines come from). The theory of computation (Wikipedia contributors 2026h) asks what computers can do. It has three parts:

- *computability theory* asks which problems a computer can solve at all. For example, no program can decide, for every program P and input I, whether P stops when it runs on I; this is the halting problem (Wikipedia contributors 2026d).
- *complexity theory* asks how much time the problems that can be solved need. Whether every problem whose answers can be checked quickly can also be solved quickly is the P versus NP problem (Wikipedia contributors 2026e), which no one had solved as of 2026.
- *automata theory* makes "computer" and "problem" precise, so that the other two parts can prove things about them. A computer becomes an abstract machine,

called an *automaton* (plural *automata*), and a problem becomes the question of which strings (Definition: String) the machine answers yes for.

The halting problem and the P versus NP problem are about machines that can do anything a general program can, called Turing machines (Wikipedia contributors 2026i). This chapter studies the simplest automata, which remember only one of finitely many situations. They are far less powerful than Turing machines, and cannot model a general program.

Definition 1 (State machine)**.** A **state machine** $M = (S, \Sigma, \delta, s_0)$ consists of

- a nonempty finite set S (Definition: Set), whose elements are called *states*;
- an alphabet Σ (Definition: Alphabet), whose symbols are called *inputs*;
- a function (Definition: Function, map, mapping, transformation) $\delta : S \times \Sigma \to S$ on the Cartesian product (Definition: Cartesian product) $S \times \Sigma$, called the *transition function*;
- an *initial state* $s_0 \in S$.

When M is in state s and reads the input x, it moves to the state $\delta(s, x)$. Names and tuples for these machines vary between sources: for example, the finite-state machines of (Wikipedia contributors 2026b) also carry a set of final states in the tuple.

Example 1 (The door as a state machine)**.** The door of Exercise 1 is a state machine (Definition 1) with

- states $S = \{\text{closed}, \text{open}, \text{locked}\}$;
- inputs $\Sigma = \{\texttt{o}, \texttt{c}, \texttt{l}, \texttt{u}\}$;
- initial state $s_0 = \text{closed}$;
- the transition function δ with

$$\begin{aligned} \delta(\text{closed}, \texttt{o}) &= \text{open}, & \delta(\text{closed}, \texttt{c}) &= \text{closed}, \\ \delta(\text{closed}, \texttt{l}) &= \text{locked}, & \delta(\text{closed}, \texttt{u}) &= \text{closed}, \\ \delta(\text{open}, \texttt{o}) &= \text{open}, & \delta(\text{open}, \texttt{c}) &= \text{closed}, \\ \delta(\text{open}, \texttt{l}) &= \text{open}, & \delta(\text{open}, \texttt{u}) &= \text{open}, \\ \delta(\text{locked}, \texttt{o}) &= \text{locked}, & \delta(\text{locked}, \texttt{c}) &= \text{locked}, \\ \delta(\text{locked}, \texttt{l}) &= \text{locked}, & \delta(\text{locked}, \texttt{u}) &= \text{closed}. \end{aligned}$$

The set $S \times \Sigma$ has $3 \cdot 4 = 12$ pairs, and δ gives one state for each of them, including the pairs where the action does nothing, such as $\delta(\text{locked}, \texttt{o}) = \text{locked}$.

Definition 2 (State diagram)**.** The **state diagram** (Wikipedia contributors 2026f) of a state machine $M = (S, \Sigma, \delta, s_0)$ (Definition 1) is the directed graph (Definition: Directed graph (digraph)) with

- vertex set S;
- for each state s and each input x, one edge from s to $\delta(s, x)$, labeled x.

The labels are extra data attached to the edges: the directed graph itself is only its vertices and edges. It is drawn with an extra arrow, which starts at no state, pointing to the initial state s_0. Two inputs $x \neq y$ with $\delta(s, x) = \delta(s, y)$ give parallel edges (Definition: Parallel edges) of the directed graph, which are drawn as one arrow labeled with all of their inputs.

Example 2 (The state diagram of the door)**.** Figure 1 is the state diagram (Definition 2) of the door (Example 1). The loop at closed is labeled `c,u`: it stands for two parallel loops, one labeled `c` and one labeled `u`, since $\delta(\text{closed}, \texttt{c}) = \delta(\text{closed}, \texttt{u}) = \text{closed}$.

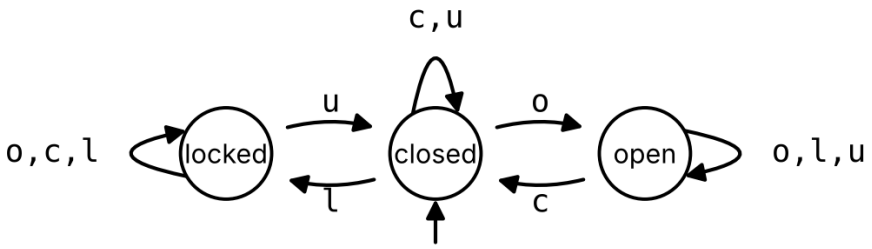


Figure 1: The state diagram of the door.

Remark 2 (Every state has one edge for each input). In the state diagram (Definition 2) of a state machine with inputs Σ, each state s is the start of exactly one edge labeled x for each input x, because $\delta(s, x)$ is one state. So every state has out-degree (Definition: Out-degree) $|\Sigma|$, counting each label of a drawn arrow as its own edge. In Figure 1 every state has out-degree 4. The in-degrees (Definition: In-degree) add up to the same total, $3 \cdot 4 = 12$, as Theorem: Degrees in a directed graph says.

Runs

Exercise 2 (Following the door)**.** The door (Example 1) starts closed, and these actions are done in this order:

$$\texttt{l}, \texttt{o}, \texttt{u}, \texttt{o}, \texttt{l}.$$

Write down the whole sequence of conditions the door passes through, starting with the condition it is in before the first action.

Solution

Solution 2. The sequence starts with closed, and each later condition is δ of the condition before it and the action:

$$\begin{aligned}
\delta(\text{closed}, \texttt{l}) &= \text{locked}, \\
\delta(\text{locked}, \texttt{o}) &= \text{locked}, \\
\delta(\text{locked}, \texttt{u}) &= \text{closed}, \\
\delta(\text{closed}, \texttt{o}) &= \text{open}, \\
\delta(\text{open}, \texttt{l}) &= \text{open}.
\end{aligned}$$

So the sequence is

$$\text{closed}, \text{locked}, \text{locked}, \text{closed}, \text{open}, \text{open}:$$

six conditions for five actions. In Figure 1, they are the states met by following, from closed, the arrows labeled `l`, `o`, `u`, `o` and `l` in turn.

Definition 3 (Run)**.** Let $M = (S, \Sigma, \delta, s_0)$ be a state machine (Definition 1), let $s \in S$, and let $w = x_1 x_2 \cdots x_n$ be a string (Definition: String) of $n \geq 0$ inputs $x_i \in \Sigma$. The **run** of M on w from s (Wikipedia contributors 2026a) is the sequence of states

$$r_0, r_1, \ldots, r_n$$

with $r_0 = s$ and $r_i = \delta(r_{i-1}, x_i)$ for $i = 1, \ldots, n$. The run of M on w, with no state named, is the run from the initial state s_0.

Example 3 (A run of the door)**.** The run (Definition 3) of the door (Example 1) on the string `oclu` is

$$\text{closed}, \text{open}, \text{closed}, \text{locked}, \text{closed},$$

since $\delta(\text{closed}, \texttt{o}) = \text{open}$, $\delta(\text{open}, \texttt{c}) = \text{closed}$, $\delta(\text{closed}, \texttt{l}) = \text{locked}$ and $\delta(\text{locked}, \texttt{u}) = \text{closed}$. The string has 4 inputs, and the run has 5 states. The run on the empty string λ is the one state closed.

Definition 4 (Extended transition function)**.** Let $M = (S, \Sigma, \delta, s_0)$ be a state machine (Definition 1). Its **extended transition function** $\delta^* : S \times \Sigma^* \to S$ (Wikipedia contributors 2026a) gives the state that M reaches from a state s by reading a string $w \in \Sigma^*$ (Definition: String). It is defined recursively (Definition: Recursive definition), following the recursive definition of Σ^*:

- base case: $\delta^*(s, \lambda) = s$;
- recursive rule: if $w \in \Sigma^*$ and $x \in \Sigma$, then $\delta^*(s, wx) = \delta(\delta^*(s, w), x)$.

Example 4 (Unfolding the extended transition function)**.** For the door (Example 1), the recursive rule of Definition 4 peels off the last input three times:

$$\begin{aligned}\delta^*(\text{closed}, \texttt{lou}) &= \delta(\delta^*(\text{closed}, \texttt{lo}), \texttt{u}) \\ &= \delta(\delta(\delta^*(\text{closed}, \texttt{l}), \texttt{o}), \texttt{u}) \\ &= \delta(\delta(\delta(\delta^*(\text{closed}, \lambda), \texttt{l}), \texttt{o}), \texttt{u}).\end{aligned}$$

The base case gives $\delta^*(\text{closed}, \lambda) = \text{closed}$, and then evaluating from the inside out,

$$\begin{aligned}\delta(\delta(\delta(\text{closed}, \texttt{l}), \texttt{o}), \texttt{u}) &= \delta(\delta(\text{locked}, \texttt{o}), \texttt{u}) \\ &= \delta(\text{locked}, \texttt{u}) \\ &= \text{closed}.\end{aligned}$$

So $\delta^*(\text{closed}, \texttt{lou}) = \text{closed}$.

Theorem 1 (A run ends where the extended transition function says)**.** *Let $M = (S, \Sigma, \delta, s_0)$ be a state machine (Definition 1), let $s \in S$, and let $r_0, \ldots, r_n$ be the run (Definition 3) of M on a string $w = x_1 \cdots x_n$ from s. Then*

$$r_n = \delta^*(s, w),$$

where δ^* is the extended transition function (Definition 4).

Proof

Proof. We show, by induction (Definition: Proof by induction) on $k = 0, 1, \ldots, n$, that $r_k = \delta^*(s, x_1 \cdots x_k)$, where $x_1 \cdots x_0$ is the empty string λ.
Base case. For $k = 0$, $r_0 = s = \delta^*(s, \lambda)$, by Definition 3 and the base case of Definition 4.
Inductive step. Suppose $r_{k-1} = \delta^*(s, x_1 \cdots x_{k-1})$ for some k with $1 \leq k \leq n$. Then

$$\begin{aligned} r_k &= \delta(r_{k-1}, x_k) && \text{(definition of the run)} \\ &= \delta(\delta^*(s, x_1 \cdots x_{k-1}), x_k) && \text{(induction hypothesis)} \\ &= \delta^*(s, x_1 \cdots x_{k-1} x_k). && \text{(recursive rule for } \delta^*) \end{aligned}$$

By induction, the equation holds for every $k \leq n$, and $k = n$ gives $r_n = \delta^*(s, w)$. □

Theorem 2 (A run is the only walk that spells its string)**.** *Let $M = (S, \Sigma, \delta, s_0)$ be a state machine (Definition 1), let $s \in S$, and let $w = x_1 \cdots x_n$ be a string over Σ. In the state diagram (Definition 2) of M, there is exactly one walk (Definition: Walk) $v_0, e_1, v_1, \ldots, e_n, v_n$ from s in which each edge e_i is labeled x_i, and its vertices $v_0, v_1, \ldots, v_n$ are the run (Definition 3) of M on w from s.*

Proof

Proof. Let $r_0, \ldots, r_n$ be the run of M on w from s.
There is such a walk. For each $i = 1, \ldots, n$, the state diagram has an edge e_i from r_{i-1} to $\delta(r_{i-1}, x_i) = r_i$, labeled x_i. So $r_0, e_1, r_1, \ldots, e_n, r_n$ is a walk from $r_0 = s$ whose edges are labeled $x_1, \ldots, x_n$.
There is only one. Let $v_0, e_1, v_1, \ldots, e_n, v_n$ be any walk from s with each e_i labeled x_i. We show, by induction (Definition: Proof by induction) on k, that $v_k = r_k$ for $k = 0, \ldots, n$. For $k = 0$, $v_0 = s = r_0$. Suppose $v_{k-1} = r_{k-1}$ for some k with $1 \leq k \leq n$. The edge e_k starts at $v_{k-1} = r_{k-1}$ and is labeled x_k, and the only such edge in the state diagram goes to $\delta(r_{k-1}, x_k) = r_k$. So $v_k = r_k$. The edge e_k is that one edge, so the edges of the two walks are the same too. □

Deterministic finite automata

Exercise 3 (Every a needs a b)**.** A program reads a string (Definition: String) of the symbols `a` and `b`, and must check that every `a` in it is immediately followed by a `b`. Which of these strings pass the check?

$$\texttt{abab}, \quad \texttt{aab}, \quad \texttt{ba}, \quad \texttt{bbab}, \quad \lambda$$

Solution

Solution 3. The strings `abab` and `bbab` pass: each of their `a`s has a `b` right after it. The empty string λ passes too, since it has no `a` that could fail. The string `aab` fails, because its first `a` is followed by an `a`, and `ba` fails, because its `a` is followed by nothing.
To check a string from left to right, the program only has to remember one of three things about what it has read so far: that every `a` so far has its `b`, that the last symbol was an `a` still waiting for its `b`, or that the check has already failed.

Definition 5 (Deterministic finite automaton (DFA))**.** A **deterministic finite automaton**, or **DFA**, (Wikipedia contributors 2026a) $M = (S, \Sigma, \delta, s_0, F)$ is a state machine (S, Σ, δ, s_0) (Definition 1) together with a subset (Definition: Subset) $F \subseteq S$, whose elements are called *accepting states*. Its state diagram is the state diagram (Definition 2) of the state machine, with each accepting state drawn as a double circle.
It is *deterministic* because each state and input give exactly one next state, $\delta(s, x)$.

Example 5 (A DFA for every a needing a b)**.** Figure 2 is the state diagram of the DFA (Definition 5) $A = (S, \Sigma, \delta, q_0, F)$ with $S = \{q_0, q_1, q_2\}$, $\Sigma = \{\texttt{a}, \texttt{b}\}$, $F = \{q_0\}$ and

$$\begin{aligned} \delta(q_0, \texttt{a}) &= q_1, \quad \delta(q_0, \texttt{b}) = q_0, \\ \delta(q_1, \texttt{a}) &= q_2, \quad \delta(q_1, \texttt{b}) = q_0, \\ \delta(q_2, \texttt{a}) &= q_2, \quad \delta(q_2, \texttt{b}) = q_2. \end{aligned}$$

Its states are the three things to remember in Solution 3: q_0 for "every `a` so far has its `b`", q_1 for "the last symbol was an `a` still waiting", and q_2 for "the check has failed".

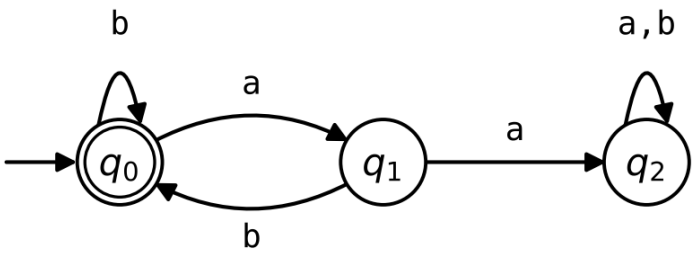


Figure 2: The DFA A, which checks that every `a` is followed by a `b`.

Definition 6 (Accepting a string)**.** A DFA (Definition 5) $M = (S, \Sigma, \delta, s_0, F)$ **accepts** a string $w \in \Sigma^*$ (Definition: String) (Wikipedia contributors 2026a) if $\delta^*(s_0, w) \in F$, where δ^* is the extended transition function (Definition 4). Otherwise M *rejects* w. By Theorem 1, M accepts w exactly when the run (Definition 3) of M on w ends at an accepting state.

Example 6 (Tracing three strings)**.** These are the runs (Definition 3) of the DFA A of Example 5 on three strings of Exercise 3:

string	run	last state	verdict
`abab`	q_0, q_1, q_0, q_1, q_0	q_0	accepts
`aab`	q_0, q_1, q_2, q_2	q_2	rejects
`ba`	q_0, q_0, q_1	q_1	rejects

So A accepts (Definition 6) `abab`, which passes the check of Exercise 3, and rejects `aab` and `ba`, which fail it. Its run on λ is q_0 alone, so it accepts λ.

Example 7 (The door as a DFA)**.** Choosing the accepting states $F = \{\text{open}\}$ makes the door (Example 1) a DFA (Definition 5), D, drawn in Figure 3. It accepts (Definition 6) the strings after which the door is open. For example, it accepts `o`, `luo` and `ol`, whose runs end at open, and rejects λ, `oc` and `lo`, whose runs end at closed, closed and locked.

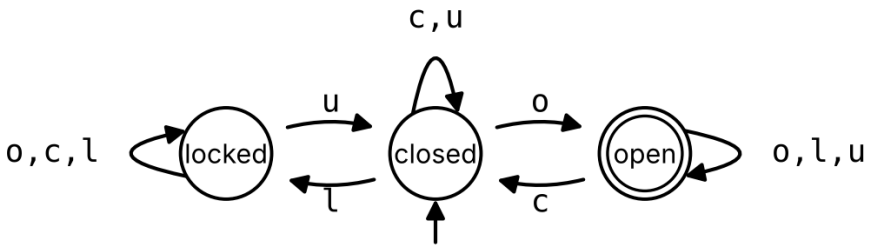


Figure 3: The door as a DFA whose one accepting state is open.

Definition 7 (Language)**.** A **language** over an alphabet Σ (Definition: Alphabet) (Wikipedia contributors 2026c) is a subset (Definition: Subset) of Σ^*, the set of all strings over Σ (Definition: String).

Example 8 (Languages over a and b)**.** Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Each of these sets is a language (Definition 7) over Σ:

- the empty set $\emptyset$ (Definition: Empty set), which has no strings;
- $\{\lambda\}$, which has one string, the empty string;
- $\{\mathtt{b}, \mathtt{ba}, \mathtt{baa}\}$, which has three strings;
- the set of strings with an even number of symbols, which is infinite;
- Σ^* itself.

The first two are different languages: $\emptyset$ has no elements, and $\{\lambda\}$ has one.

Definition 8 (Language of an automaton)**.** The **language** of a DFA $M = (S, \Sigma, \delta, s_0, F)$ (Definition 5) (Wikipedia contributors 2026a) is the language (Definition 7)

$$L(M) = \{w \in \Sigma^* : M \text{ accepts } w\}$$

of the strings that M accepts (Definition 6).

Example 9 (The language of the door)**.** The language (Definition 8) $L(D)$ of the door DFA (Example 7) is the set of strings over $\{\texttt{o}, \texttt{c}, \texttt{l}, \texttt{u}\}$ after which the door is open. So $\texttt{o}, \texttt{luo}, \texttt{ol} \in L(D)$ and $\lambda, \texttt{oc}, \texttt{lo} \notin L(D)$. Every string in $L(D)$ contains an $\texttt{o}$, since a run starts at closed, and the only arrow into open from another state is labeled $\texttt{o}$. The converse fails: $\texttt{oc}$ contains an $\texttt{o}$ but is not in $L(D)$.

Theorem 3 (The language of the a-and-b checker)**.** *The language (Definition 8) of the DFA A of Example 5 is*

$$L(A) = \{w \in \{\texttt{a}, \texttt{b}\}^* : \textit{every } \texttt{a} \textit{ in } w \textit{ is immediately followed by a } \texttt{b}\}.$$

Proof

Proof. Sort the strings over $\{\texttt{a}, \texttt{b}\}$ into three classes:

- class 0: no two $\texttt{a}$s in a row, and not ending in $\texttt{a}$;
- class 1: no two $\texttt{a}$s in a row, and ending in $\texttt{a}$;
- class 2: two $\texttt{a}$s in a row somewhere.

Every string is in exactly one class. A string has every $\texttt{a}$ immediately followed by a $\texttt{b}$ exactly when it is in class 0, since an $\texttt{a}$ that is not followed by a $\texttt{b}$ is followed either by an $\texttt{a}$ (class 2) or by nothing (class 1). Let $P(w)$ be the statement

$$\text{if } w \text{ is in class } k, \text{ then } \delta^*(q_0, w) = q_k.$$

We prove $P(w)$ for every w by structural induction (Definition: Structural induction) on the recursive definition of strings (Definition: String).
Base case. The empty string λ is in class 0, and $\delta^*(q_0, \lambda) = q_0$ (Definition 4), so $P(\lambda)$ holds.
Inductive step. Suppose $P(w)$, and let $x \in \{\texttt{a}, \texttt{b}\}$. By Definition 4, $\delta^*(q_0, wx) = \delta(\delta^*(q_0, w), x)$. We split into cases on the class of w.

- w **in class 0.** By $P(w)$, $\delta^*(q_0, w) = q_0$. Since w does not end in $\texttt{a}$, adding x makes no two $\texttt{a}$s in a row, so wx has none, as w has none. If $x = \texttt{a}$, then wx is in class 1, and $\delta(q_0, \texttt{a}) = q_1$. If $x = \texttt{b}$, then wx is in class 0, and $\delta(q_0, \texttt{b}) = q_0$.

- w **in class 1.** By $P(w)$, $\delta^*(q_0, w) = q_1$. If $x = \texttt{a}$, then wx ends in two `a`s, so it is in class 2, and $\delta(q_1, \texttt{a}) = q_2$. If $x = \texttt{b}$, then wx still has no two `a`s in a row and ends in `b`, so it is in class 0, and $\delta(q_1, \texttt{b}) = q_0$.
- w **in class 2.** By $P(w)$, $\delta^*(q_0, w) = q_2$. Then wx has two `a`s in a row too, so it is in class 2, and $\delta(q_2, x) = q_2$.

In every case $\delta^*(q_0, wx) = q_k$ for the class k of wx, so $P(wx)$ holds. By structural induction, $P(w)$ holds for every string w.
Now let w be any string, in class k. By $P(w)$, $\delta^*(q_0, w) = q_k$, and A accepts w (Definition 6) exactly when $q_k \in F = \{q_0\}$, that is, when $k = 0$. So A accepts exactly the strings of class 0, which are the strings with every `a` immediately followed by a `b`. □

Definition 9 (Dead state)**.** A **dead state** of a DFA $M = (S, \Sigma, \delta, s_0, F)$ (Definition 5) is a state $d \notin F$ with $\delta(d, x) = d$ for every input $x \in \Sigma$.

Example 10 (The dead state of the checker)**.** In the DFA A of Example 5, q_2 is a dead state (Definition 9): it is not accepting, and both of its arrows lead back to it. Once a run (Definition 3) reaches q_2, every later state of the run is q_2, so the run ends at q_2 and A rejects the string. For example, A rejects every string that starts with `aa`, whatever comes after it.

Theorem 4 (Swapping the accepting states)**.** *Let $M = (S, \Sigma, \delta, s_0, F)$ be a DFA (Definition 5), and let*

$$M' = (S, \Sigma, \delta, s_0, S \setminus F)$$

be the DFA with the same states, inputs, transitions and initial state, whose accepting states are the states that are not accepting in M. Then $L(M')$ (Definition 8) is the complement (Definition: Complement) $\Sigma^* \setminus L(M)$ of $L(M)$ in Σ^*.

Proof

Proof. The extended transition function δ^* (Definition 4) is built from S, Σ and δ alone, so M and M' have the same δ^*. Let $w \in \Sigma^*$. By the language of a DFA (Definition 8) and the definition of accepting (Definition 6),

$$\begin{aligned}
w \in L(M') &\iff M' \text{ accepts } w && \text{(language of a DFA)} \\
&\iff \delta^*(s_0, w) \in S \setminus F && \text{(definition of accepting)} \\
&\iff \delta^*(s_0, w) \notin F && \text{(since } \delta^*(s_0, w) \in S\text{)} \\
&\iff M \text{ does not accept } w && \text{(definition of accepting)} \\
&\iff w \notin L(M). && \text{(language of a DFA)}
\end{aligned}$$

So $L(M')$ is the set of strings in Σ^* that are not in $L(M)$.
The step from $\delta^*(s_0, w) \in S \setminus F$ to $\delta^*(s_0, w) \notin F$ uses that δ is a function on all of $S \times \Sigma$, so that $\delta^*(s_0, w)$ is a state of S for every string w. Some drawings of automata leave out a dead state (Definition 9) and the arrows into it, so that some states have no arrow for some input. Swapping the accepting states of such a drawing does not give the complement: a string whose run would enter the missing dead state is rejected before and after the swap. Put the dead state back before swapping. □

Example 11 (The strings that fail the check)**.** Swapping the accepting states of the DFA A of Example 5 gives the DFA A' of Figure 4, whose accepting states are q_1 and q_2. By Theorem 4 and Theorem 3, A' accepts exactly the strings in which some `a` is not immediately followed by a `b`: those with two `a`s in a row, and those that end in `a`. It accepts `aab` and `ba` and rejects `abab` and λ. In A', the state q_2 is no longer a dead state (Definition 9), since it is accepting.

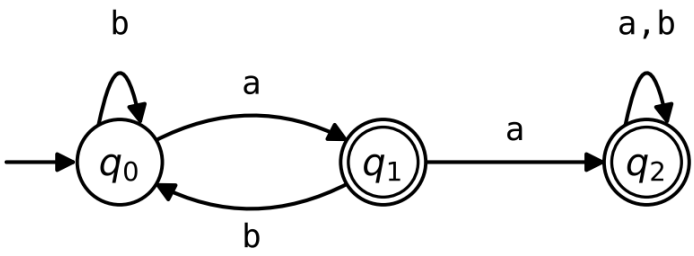


Figure 4: The DFA A', which accepts the strings that A rejects.

Designing a DFA

Exercise 4 (Recognizing decimal numerals)**.** A program reads numbers written with digits and at most one decimal point. To keep the alphabet small, write `d` for any one of the ten digits, so the alphabet (Definition: Alphabet) is $\Sigma = \{\texttt{d}, .\}$. A *numeral* is one or more `d`s, optionally followed by a point . and one or more `d`s. So `d`, `ddd` and `dd.d` are numerals, and λ, `.d`, `d.` and `d.d.d` are not.
Draw the state diagram of a DFA (Definition 5) whose language (Definition 8) is the set of numerals.

Solution

Solution 4. Read from left to right, a numeral goes through four stages, and a string that is not a numeral can go wrong at any point. Give the DFA one state for each thing it has to remember:

- n_0: nothing has been read yet;
- n_1: one or more `d`s and nothing else, which is a numeral, so n_1 is accepting;
- n_2: one or more `d`s and then a point, which is not a numeral yet;
- n_3: one or more `d`s, a point and one or more `d`s, which is a numeral, so n_3 is accepting;
- n_4: the string has already gone wrong, which is a dead state (Definition 9).

A `d` moves n_0 to n_1 and n_2 to n_3, and keeps n_1 at n_1 and n_3 at n_3. A point moves n_1 to n_2. A point anywhere else goes wrong: at the start, right after another point, or after the digits that follow a point. Figure 5 draws this DFA. For example, its run (Definition 3) on `dd.d` is n_0, n_1, n_1, n_2, n_3, which ends at an accepting state, and its run on `d.` is n_0, n_1, n_2, which does not.

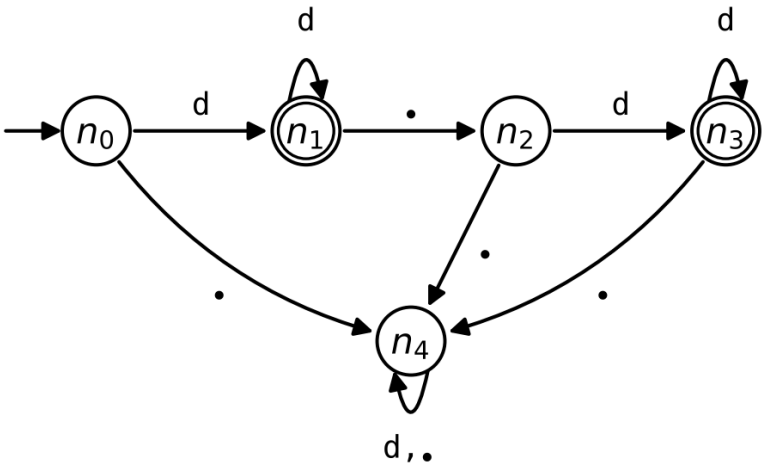


Figure 5: A DFA whose language is the set of numerals.

Remark 3 (Designing a DFA). Solution 4 follows a method that works for many languages (Definition 7):

1. Read a string from left to right, and ask what has to be remembered about the part read so far to decide the rest.
2. Make one state for each different thing to remember, and add a dead state (Definition 9) if a string can go wrong in a way that no later input repairs.
3. Make a state accepting when the part read so far, on its own, is in the language.
4. For each state and each input, decide which state the input leads to.
5. Check the DFA on short strings that should be accepted and on short strings that should be rejected.

A DFA has finitely many states, so this works only when finitely many things need to be remembered.

Example 12 (Counting 0s modulo 3)**.** Consider the strings over $\{0, 1\}$ whose number of 0s leaves remainder 2 when it is divided by 3: 2, 5, 8, and so on. To design a DFA (Definition 5) whose language is this set of strings, it is enough to remember the remainder of the number of 0s read so far when it is divided by 3. So the DFA T of Figure 6 has three states, t_0, t_1 and t_2, with t_k standing for "the remainder is k". A 1 does not change the count, so it keeps each state where it is. A 0 adds one to the count, so it moves t_0 to t_1, t_1 to t_2, and t_2 back to t_0. The initial state is t_0, since no 0s have been read, and t_2 is the one accepting state.
The run (Definition 3) of T on 1001 is t_0, t_0, t_1, t_2, t_2, so T accepts that string, which has two 0s. The run on 0100 is t_0, t_1, t_1, t_2, t_0, so T rejects it, since it has three 0s. T rejects λ, which has no 0s, and accepts 00000, which has five. No state of T is a dead state (Definition 9), since $\delta(t_k, 0)$ is a different state from t_k for each k.

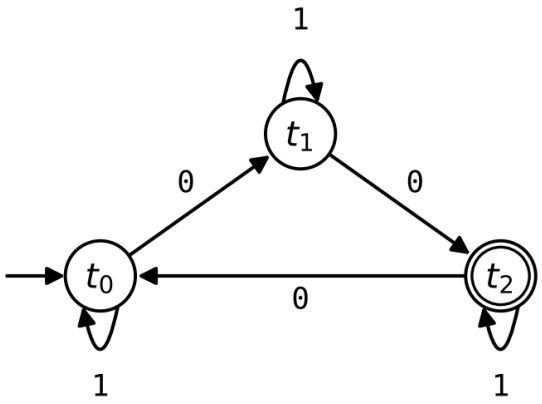


Figure 6: The DFA T, which accepts the strings whose number of 0s leaves remainder 2 when divided by 3.

State-transition tables

Exercise 5 (Reading a DFA from a table)**.** A DFA has states p_0, p_1, p_2 and p_3, inputs 0 and 1, initial state p_0 and one accepting state, p_3. Its transitions are given by this table, in which the entry in the row of a state and the column of an input is the state that the input leads to:

state	0	1
p_0	p_0	p_1
p_1	p_0	p_2

p_2	p_3	p_2
p_3	p_3	p_3

Does this DFA accept the string `0101100`?

Solution

Solution 5. Start at p_0 and look up one entry for each input:

input		`0`	`1`	`0`	`1`	`1`	`0`	`0`
state	p_0	p_0	p_1	p_0	p_1	p_2	p_3	p_3

The run (Definition 3) ends at p_3, which is accepting, so the DFA accepts `0101100` (Definition 6).

Definition 10 (State-transition table)**.** The **state-transition table** (Wikipedia contributors 2026g) of a DFA $M = (S, \Sigma, \delta, s_0, F)$ (Definition 5) has one row for each state and one column for each input, and the entry in the row of s and the column of x is $\delta(s, x)$. The initial state is marked with $\to$ and each accepting state with $*$.

Example 13 (The table of the checker)**.** The state-transition table (Definition 10) of the DFA A of Example 5 is

state	`a`	`b`
$\to * q_0$	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_2

The row of the dead state (Definition 9) q_2 is q_2 in every column.

Example 14 (The table of the door)**.** The state-transition table (Definition 10) of the door DFA D of Example 7 has four columns, one for each input:

state	`o`	`c`	`l`	`u`
→ closed	open	closed	locked	closed
∗ open	open	closed	open	open
locked	locked	locked	locked	closed

It has $3 \cdot 4 = 12$ entries, one for each pair in $S \times \Sigma$, the same 12 values of δ as in Example 1.

Example 15 (What the table's DFA accepts)**.** Figure 7 draws the DFA W of Exercise 5. Its language (Definition 8) is the set of strings over $\{\texttt{0}, \texttt{1}\}$ that contain `110` as three symbols in a row. Each state records how much of `110` the last symbols read have matched:

- p_0: none of it;
- p_1: the last symbol read is a `1` that follows no other `1`, the start of `110`;
- p_2: the last two symbols read are `11`;
- p_3: `110` has appeared, and p_3 is accepting, with both of its arrows leading back to it.

A `0` read at p_1 goes back to p_0, since a string ending in `10` has matched none of `110`, while a `1` read at p_2 stays at p_2, since a string ending in `111` still ends in `11`. So W accepts `0101100`, as in Solution 5, and rejects `101011`, which does not contain `110`.

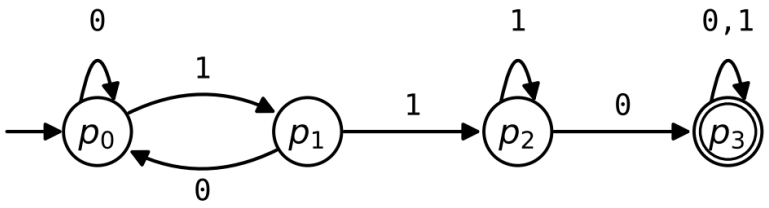


Figure 7: The DFA W, which accepts the strings that contain `110`.

Remark 4 (A table is like a labeled adjacency list). The row of a state s in a state-transition table (Definition 10) lists the ends of the edges that start at s in the state diagram (Definition 2), one for each input, like the list of s in an adjacency list (Definition: Adjacency list) of a directed graph. The table also says which edge has which label, which an adjacency list does not. Every row has exactly $|\Sigma|$ entries, as Remark 2 says.
The likeness stops when two inputs lead from s to the same state. Then the state diagram has parallel edges (Definition: Parallel edges), for which Definition: Adjacency list does not define an adjacency list, and the row of s names that state more than once, as the row of q_2 in Example 13 does.

Automata in Racket

Exercise 6 (One step in Racket)**.** In Racket, the table of the DFA A (Example 13) can be stored as a list with one list for each state, which starts with the state and goes on with a list `(x t)` for each input `x`, where `t` is the state that `x` leads to:

```
(define pairs-table
  '((q0 (#\a q1) (#\b q0))
    (q1 (#\a q2) (#\b q0))
    (q2 (#\a q2) (#\b q2))))

(define (step table s x)
  (second (assoc x (rest (assoc s table)))))
```

Here `#\a` is the character a, and `assoc` is as in Exercise: Looking up a vertex. What does `(step pairs-table 'q1 #\b)` return?

Solution

Solution 6. `(assoc 'q1 pairs-table)` is the row `'(q1 (#\a q2) (#\b q0))`, and its rest is `'((#\a q2) (#\b q0))`. `(assoc #\b ...)` finds `'(#\b q0)` in that list, and its second element is `'q0`. So `(step pairs-table 'q1 #\b)` returns `'q0`, which is $\delta(q_1, \mathtt{b})$ (Example 5).

Example 16 (Running a DFA in Racket)**.** With `step` as in Exercise 6, `run` returns the run (Definition 3) from a state `s` on a list `xs` of inputs, and `accepts?` says whether a DFA accepts (Definition 6) the string whose inputs are the list `xs`:

```
(define (run table s xs)
  (if (empty? xs)
      (list s)
      (cons s (run table (step table s (first xs)) (rest xs)))))

(define (accepts? table start accepting xs)
  (if (member (last (run table start xs)) accepting)
      #t
      #f))
```

- `(last L)` is the last element of the list `L`.
- `(member x L)` is `#f` if `x` is not in the list `L`, and a list that is not `#f` if it is.
- `(string->list w)` is the list of the characters of the string `w`.

`(run pairs-table 'q0 (string->list "abab"))` returns `'(q0 q1 q0 q1 q0)`, and `(run pairs-table 'q0 (string->list "aab"))` returns `'(q0 q1 q2 q2)`, the runs of Example 6. `(accepts? pairs-table 'q0 '(q0) (string->list "abab"))` returns `#t`, `(accepts? pairs-table 'q0 '(q0) (string->list "ba"))` returns `#f`, and `(accepts? pairs-table 'q0 '(q0) (string->list ""))` returns `#t`, since A accepts λ. Let w be the string whose symbols are the elements of `xs`, in order. `run` reads the first symbol of w first, as Definition 3 does, so by Theorem 1 the last state of `(run table s xs)` is $\delta^*(s, w)$.

Example 17 (Checking numerals in Racket)**.** In the form of Exercise 6, the DFA of Solution 4 is the table below, with the symbol `'digit` for the input `d` and the symbol `'point` for the input .:

```
(define numeral-table
  '((n0 (digit n1) (point n4))
    (n1 (digit n1) (point n2))
    (n2 (digit n3) (point n4))
    (n3 (digit n3) (point n4))
    (n4 (digit n4) (point n4))))
```

To check real text, `classify` turns each character into an input, and `numeral?` runs the DFA with `accepts?` from Example 16:

```
(define (classify c)
  (cond [(char<=? #\0 c #\9) 'digit]
        [(char=? c #\.) 'point]
        [else 'other]))

(define (numeral? text)
  (let ([xs (map classify (string->list text))])
    (and (not (member 'other xs))
         (accepts? numeral-table 'n0 '(n1 n3) xs))))
```

- `(char<=? #\0 c #\9)` is true when the character `c` is one of the ten digits 0 to 9.
- `(cond [test1 a] [test2 b] [else c])` returns `a` if `test1` is true, otherwise `b` if `test2` is true, and otherwise `c`.
- `(map f L)` applies `f` to each element of the list `L`.
- `(and p q)` is `#f` if `p` is `#f`, and is `q` otherwise.

A character that is neither a digit nor a point is classified `'other`, which is not an input of the DFA, so `numeral?` returns `#f` for any text that contains one, without running the DFA. Then

```
(map numeral? '("3.14" "2026" "7." ".5" "1.2.3" "d.d" "-3" "\u00BD"))
```

returns `'(#t #t #f #f #f #f #f #f)`: `"3.14"` and `"2026"` are numerals, and the others are not. The letter d in `"d.d"` is classified `'other`, not `'digit`, and so is the one character of `"\u00BD"`, the fraction one half, even though Racket's `char-numeric?` counts it as numeric.

References

Wikipedia contributors. 2026a. *Deterministic Finite Automaton — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Deterministic_finite_automaton.

Wikipedia contributors. 2026b. *Finite-State Machine — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Finite-state_machine.

Wikipedia contributors. 2026c. *Formal Language — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Formal_language.

Wikipedia contributors. 2026d. *Halting Problem — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Halting_problem.

Wikipedia contributors. 2026e. *P Versus NP Problem — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/P_versus_NP_problem.

Wikipedia contributors. 2026f. *State Diagram — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/State_diagram.

Wikipedia contributors. 2026g. *State-Transition Table — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/State-transition_table.

Wikipedia contributors. 2026h. *Theory of Computation — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Theory_of_computation.

Wikipedia contributors. 2026i. *Turing Machine — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Turing_machine.