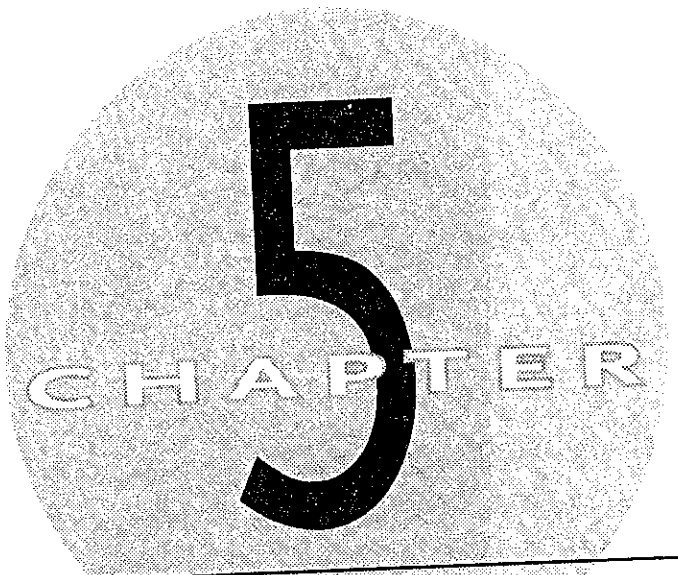


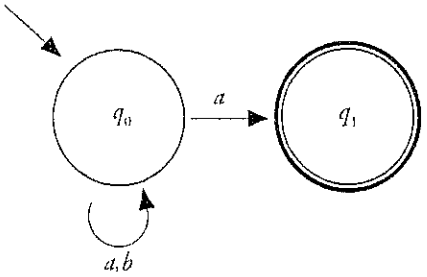
Nondeterministic Finite Automata

A DFA has exactly one transition from every state on every symbol in the alphabet. By relaxing this requirement we get a related but more flexible kind of automaton: the **nondeterministic finite automaton** *(NFA).*

NFAs are a bit harder to think about than DFAs because they do not appear to define simple computational processes. They may seem at first to be unnatural, like puzzles invented by professors for the torment of students. But have patience! NFAs and other kinds of nondeterministic automata arise naturally in many ways, as you will see later in this book, and they too have a variety of practical applications.

5.1 Relaxing a Requirement

Consider the following state machine:



This is not a DFA, since it violates the rule that there must be exactly one transition from every state on every symbol in the alphabet. There is no transition from the q_1 state on either a or b, and there is more than one transition from the q_0 state on an a. This is an example of a nondeterministic finite automaton (NFA).

How can you tell whether such a machine accepts a given string? Consider the string aa. Following the arrows, there are three possible sequences of moves:

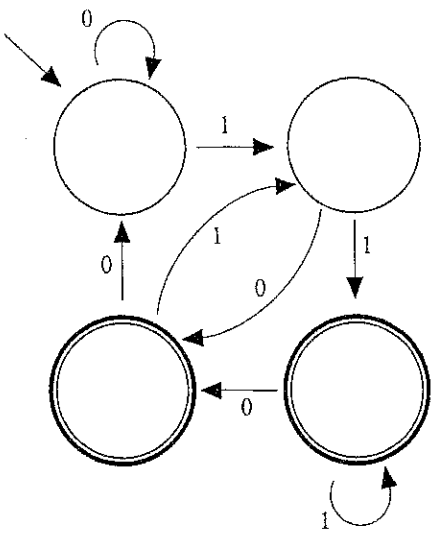
1. from q_0 back to q_0 on the first a, and back to q_0 again on the second;
2. from q_0 back to q_0 on the first a, then to q_1 on the second; and
3. from q_0 to q_1 on the first a, then getting stuck with no legal move on the second.

Only the second of these is an accepting sequence of moves. But the convention is that if there is any way for an NFA to accept a string, that string is taken to be part of the language defined by the NFA. The string aa is in the language of the NFA above because there is a sequence of legal moves that reaches the end of aa in an accepting state. The fact that there are also sequences of legal moves that do not succeed is immaterial.

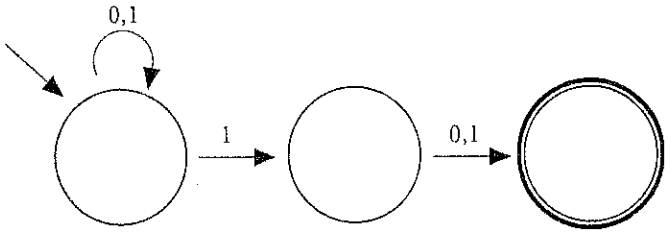
The language defined by the NFA above is $\{xa \mid x \in \{a, b\}^*\}$—that is, the language of strings over the alphabet $\{a, b\}$ that end in a. The machine can accept any such string by staying in the q_0 state until it reaches the a at the end, then making the transition to q_1 as its final move.

A DFA can be thought of as a special kind of NFA—an NFA that happens to specify exactly one transition from every state on every symbol. In this sense every DFA is an NFA, but not every NFA is a DFA. So in some ways the name *nondeterministic finite automaton* is misleading. An NFA might more properly be called a *not-necessarily-deterministic finite automaton*.

The extra flexibility afforded by NFAs can make them much easier to construct. For example, consider the language $\{x \in \{0, 1\}^* \mid$ the next-to-last symbol in x is $1\}$. The smallest DFA for the language is this one:



The connection between the language and this DFA is not obvious. (With some effort you should be able to convince yourself that the four states correspond to the four possibilities for the last two symbols seen—00, 01, 10, or 11—and that the two accepting states are the states corresponding to 10 and 11, where the next-to-last symbol is a 1.) An NFA for the language can be much simpler:

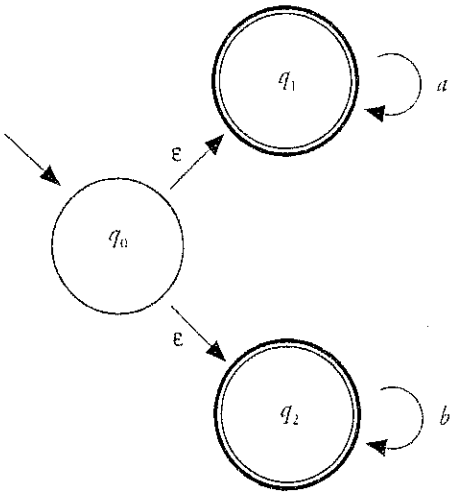


This NFA has fewer states and far fewer transitions. Moreover, it is much easier to understand. It can clearly accept a string if and only if the next-to-last symbol is a 1.

5.2 Spontaneous Transitions

An NFA has one additional trick up its sleeve: it is allowed to make a state transition spontaneously, without consuming an input symbol.

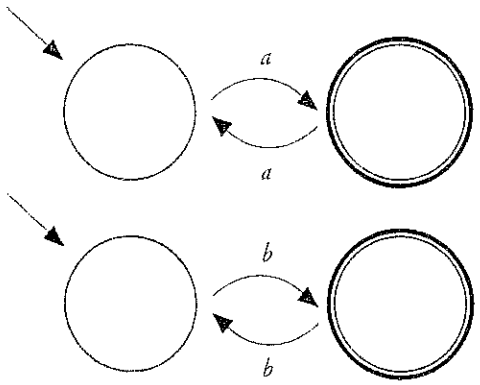
In NFA diagrams the option of making a non-input-consuming transition is shown as an arrow labeled with ε. (Recall that ε is not normally used as a symbol in an alphabet, but stands for the empty string. That makes sense here since no symbols are consumed when an ε-transition is made.) For example, this NFA accepts any string of all *a*s and also accepts any string of all *b*s:



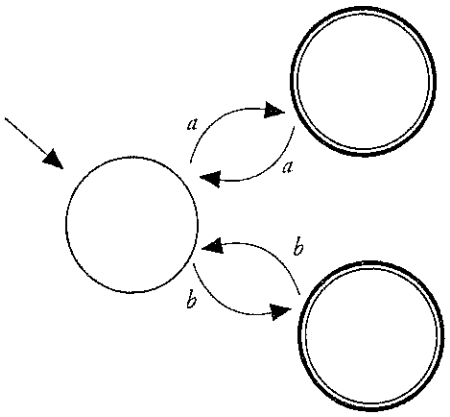
To accept the string *aa*, this NFA would first make an ε-transition to q_1, then make a transition back to q_1 on the first *a*, and finally another transition back to q_1 on the second *a*.

Notice that although q_0 is not an accepting state, this NFA does accept the empty string. Because of the ε-transitions, it has three possible sequences of moves when the input is the empty string: it can make no moves, staying in q_0; it can move to q_1 and end there; or it can move to q_2 and end there. Since two of these are sequences that end in accepting states, the empty string is in the language defined by this NFA. In general, any state that has an ε-transition to an accepting state is, in effect, an accepting state too.

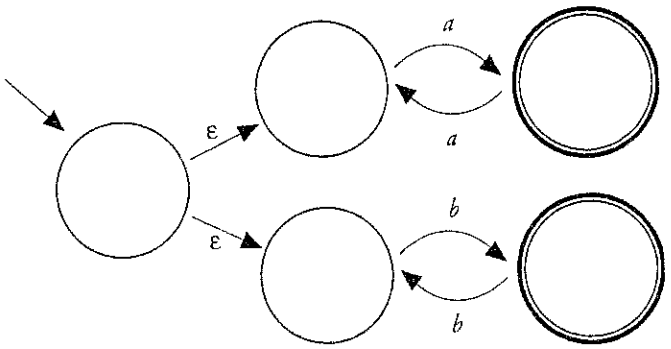
ε-transitions are especially useful for combining smaller automata into larger ones. The example above is an NFA that accepts $\{a^n\} \cup \{b^n\}$, and it can be thought of as the combination of two smaller NFAs, one for the language $\{a^n\}$ and one for the language $\{b^n\}$. For another example, consider these two NFAs:



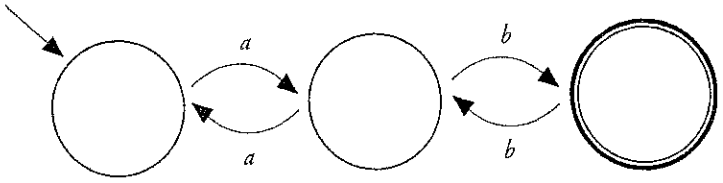
The first accepts $L_1 = \{a^n \mid n \text{ is odd}\}$, and the second accepts $L_2 = \{b^n \mid n \text{ is odd}\}$. Suppose you wanted an NFA for the union of the two languages, $L_1 \cup L_2$. You might think of combining the two smaller NFAs by unifying their start states, like this:



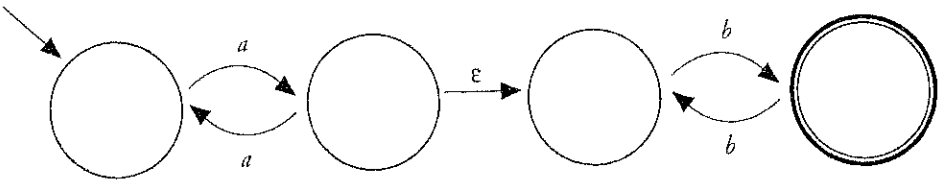
But that does not work—it accepts strings like *aab*, which is not in either of the original languages. Since NFAs may *return* to their start states, you cannot form an NFA for the union of two languages just by combining the start states of their two NFAs. But you can safely combine the two NFAs using ε-transitions, like this:



Similarly, suppose you wanted an NFA for the concatenation of the two languages, $\{xy \mid x \in L_1 \text{ and } y \in L_2\}$. You might think of combining the two smaller NFAs by using the accepting state of the first as the start state of the second, like this:



But again that does not work—it accepts strings like *abbaab*, which is not in the desired language. As before, the problem can be solved using ε-transitions:



Using such techniques, large NFAs can be composed by combining smaller NFAs. This property of NFAs is called *compositionality*, and it is one of the advantages NFAs have over DFAs in some applications. We will see further examples of this in later chapters.

5.3 Nondeterminism

NFAs and DFAs both define languages, but only DFAs really define direct computational procedures for testing language membership. It is easy to check whether a DFA accepts a given input string: you just start in the start state, make one transition for each symbol in the string, and check at the end to see whether the final state is accepting. You can carry this computational procedure out for yourself or write a program to do it. But what about the same test for an NFA—how can you test whether an NFA accepts a given input string? It is no longer so easy, since there may be more than one legal sequence of steps for an input, and you have to determine whether at least one of them ends in an accepting state. This seems to require searching through all legal sequences of steps for the given input string, but how? In what order? The NFA does not say. It does not fully define a computational procedure for testing language membership.

This is the essence of the *nondeterminism* of NFAs:

1. For a given input there can be more than one legal sequence of steps.
2. The input is in the language if at least one of the legal sequences says so.

The first part of nondeterminism is something you may be familiar with from ordinary programming. For example, a program using a random-number generator can make more than one legal sequence of steps for a given input. But randomness is not the same thing as nondeterminism. That second part of nondeterminism is the key, and it is what makes nondeterministic automata seem rather alien to most programmers.

Because of their nondeterminism, NFAs are harder to implement than DFAs. In spite of this higher level of abstraction, NFAs have many practical applications, as we will see in later chapters. We will see algorithmic techniques for deciding whether an NFA accepts a given string, and we will see that, in many applications, the compactness and compositionality of NFAs outweigh the difficulties of nondeterminism.

5.4 The 5-Tuple for an NFA

To prove things about DFAs, we found that diagrams were not very useful. We developed a formalism, the 5-tuple, to represent their essential structure. Now we'll do the same thing for NFAs.

First, we need a preliminary definition. Given any set S, you can form a new set by taking the set of all the subsets of S, including the empty set and S itself. In mathematics this is called the *powerset* of the original set:

If S is a set, the powerset of S is $P(S) = \{R \mid R \subseteq S\}$.

The primary difference between a DFA and an NFA is that a DFA must have exactly one transition from every state on every symbol, while an NFA can have any number, zero or more. To reflect this change in the mathematical description of an NFA, we make a subtle but important change: the output of the transition function, which is a *single state* for the DFA, is a *set of states* for an NFA.

An NFA M is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

Q is the finite set of states

Σ is the alphabet (that is, a finite set of symbols)

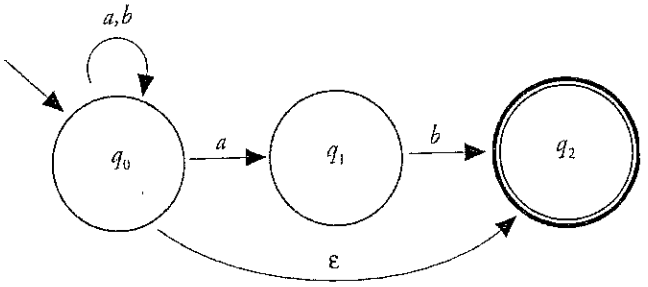
$\delta \in (Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(Q))$ is the transition function

$q_0 \in Q$ is the start state

$F \subseteq Q$ is the set of accepting states

The definition says $\delta \in (Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(Q))$; this is the mathematical way of giving the type of the function, and it says that δ takes two inputs, a state from Q and a symbol from $\Sigma \cup \{\varepsilon\}$, and produces a single output, which is some subset of the set Q. If the NFA is in the q_i state, then $\delta(q_i, a)$ is the set of possible next states after reading the a symbol and $\delta(q_i, \varepsilon)$ is the set of possible next states that can be reached on an ε-transition, without consuming an input symbol. Thus the transition function encodes all the information about the arrows in an NFA diagram. All the other parts of the NFA—the set of states, the alphabet, the start state, and the set of accepting states—are the same as for a DFA.

Consider for example this NFA:



Formally, the NFA shown is $M = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, $F = \{q_2\}$, and the transition function δ is

$$\delta(q_0, a) = \{q_0, q_1\}$$
$$\delta(q_0, b) = \{q_0\}$$
$$\delta(q_0, \varepsilon) = \{q_2\}$$
$$\delta(q_1, a) = \{\}$$
$$\delta(q_1, b) = \{q_2\}$$
$$\delta(q_1, \varepsilon) = \{\}$$
$$\delta(q_2, a) = \{\}$$

$\delta(q_2, b) = \{\}$
$\delta(q_2, \varepsilon) = \{\}$

When the diagram has no arrow from a given state on a given symbol, as for $\delta(q_1, a)$, the transition function reflects this by producing $\{\}$, the empty set, as the set of possible next states. When the diagram has a single arrow from a given state on a given symbol, as for $\delta(q_0, b)$, the transition function produces a singleton set such as $\{q_0\}$, showing that there is exactly one possible next state. When the diagram allows more than one transition from a given state on a given symbol, as for $\delta(q_0, a)$, the transition function produces a set containing more than one possible next states.

The example above demonstrates an odd technicality about NFA alphabets. We assumed for the example that $\Sigma = \{a, b\}$, but all we really know from looking at the diagram is that Σ contains a and b. It could contain other symbols as well. A DFA diagram tells you unequivocally what the alphabet must be, because it gives a transition from every state on every symbol in the alphabet. An NFA need not give transitions on every symbol, so the alphabet is not fully determined by the diagram. Usually this is an inconsequential technicality; the language defined by the machine above is $\{a, b\}^*$, even if we take it that $\Sigma = \{a, b, c\}$. But sometimes you do need to know the exact alphabet (as when taking the complement of the language), and at such times an NFA diagram by itself is inadequate.

5.5 The Language Accepted by an NFA

As an NFA operates, it reads its input string and changes its state. At any point in the NFA's operation, its future depends on two things: the current state, and that part of the input string that is still to be read. Taken together, these two pieces of information are called an *instantaneous description* (*ID*) for the NFA. The following definitions are all made with respect to some fixed NFA $M = (Q, \Sigma, \delta, q_0, F)$:

An instantaneous description for an NFA is a pair (q, x) where
$q \in Q$ is the current state
$x \in \Sigma^*$ is the unread part of the input

The δ function for the NFA determines a relation $\mapsto$ on IDs; we write $I \mapsto J$ if I is an ID and J is an ID that could follow from I after one move of the NFA.

$\mapsto$ is a relation on IDs. For any string $x \in \Sigma^*$ and any $\omega \in \Sigma$ or $\omega = \varepsilon$,
$(q, \omega x) \mapsto (r, x)$ if and only if $r \in \delta(q, \omega)$.

The definition permits $\omega \in \Sigma$ or $\omega = \varepsilon$. Thus the move can be either a move on one symbol of input or an ε-transition. Next we define an extended relation $\mapsto^*$ for sequences of zero or more steps.

$\mapsto^*$ is a relation on IDs, with $I \mapsto^* J$ if and only if there is a sequence of zero or more $\mapsto$ relations that starts with I and ends with J.

Notice here that $\mapsto^*$ is reflexive; for any ID I, $I \mapsto^* I$ by a sequence of zero moves. Using the $\mapsto^*$ relation, we can define the δ^* function for M:

$$\delta^*(q, x) = \{r \mid (q, x) \mapsto^* (r, \varepsilon)\}$$

Thus, for any string $x \in \Sigma^*$ and any state $q \in Q$, $\delta^*(q, x)$ is the set of possible states for the NFA to end up in, starting from q and making some sequence of legal δ-transitions on the symbols in x. Using this extended transition function, it is possible to express the idea that an NFA accepts a string if it has at least one path to an accepting state on that string:

A string $x \in \Sigma^*$ is accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ if and only if $\delta^*(q_0, x)$ contains at least one element of F.

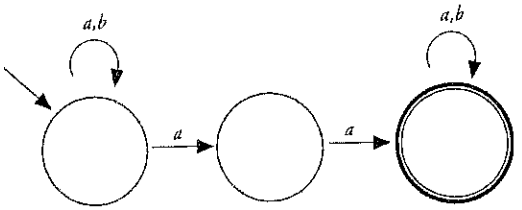
A string x is accepted if and only if the NFA, started in its start state and taken through some sequence of transitions on the symbols in x, has at least one accepting state as a possible final state. We can also define $L(M)$, the language accepted by an NFA M:

For any NFA $M = (Q, \Sigma, \delta, q_0, F)$, $L(M)$ denotes the language accepted by M, which is $L(M) = \{x \in \Sigma^* \mid \delta^*(q_0, x)$ contains at least one element of $F\}$.

Exercises

EXERCISE 1

For each of the following strings, say whether it is in the language accepted by this NFA:



a. *aa*
b. *baabb*
c. *aabaab*
d. *ababa*
e. *ababaab*