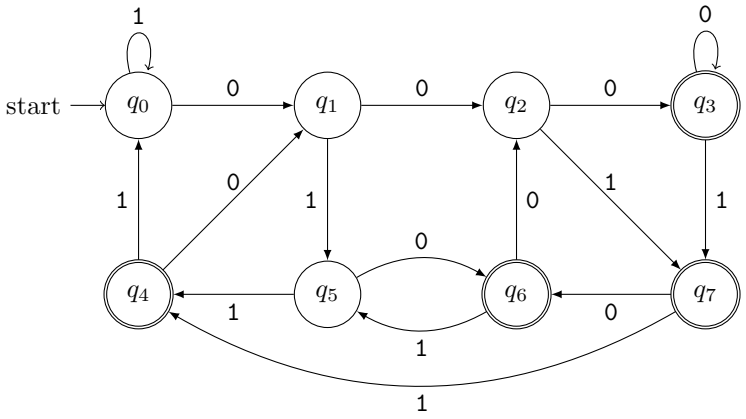
3.4 Nondeterminism

Remember how, when we were discussing the transition function earlier, we mandated a condition that any pair of state and symbol must map to *at most* one state? This condition ensured that if we gave the same input word to the same finite automaton, we would end up with the same result. This is known as *deterministic* computation. (And now you know what the D in DFA stands for!)

While determinism isn't inherently a bad thing, it can unfortunately make our job harder if we're trying to construct a finite automaton that recognizes certain "difficult" languages. For example, suppose we wanted to construct a deterministic finite automaton that recognizes the language of words over the alphabet $\Sigma = \{0, 1\}$ where the third-from-last symbol is 0. This finite automaton should accept input words like 011, 10010, and 1010001010011000, but it should reject input words like 110 or 01. Sounds easy to do, right? After all, we really just need to check one symbol: the symbol in the third-from-last position. As it turns out, however, this is the deterministic finite automaton in question:



Keep in mind also that this deterministic finite automaton *only* works for input words where the third-from-last symbol is 0. If we wanted to, say, check the fourth-from-last symbol, we would need to construct a whole new finite automaton—and this one would have *twice as many* states as our previous one!

So, how do we make our job easier and our finite automata smaller? We get rid of the determinism condition. Specifically, we allow for state/symbol pairs to map to one *or more* states. We're able to preserve the "function" part of our transition function by mapping each state/symbol pair not to multiple different states individually, but rather to a subset of the state set Q.

If we get rid of the determinism condition, then the finite automaton can, in a sense, "guess" which step to take at certain points in the computation. If, in a given state, there is more than one transition out of that state on the same symbol, then the finite automaton has multiple options for which transition it can take. As you might have figured, this property is called *nondeterminism*, and the definition of a nondeterministic finite automaton is nearly identical to our earlier definition of a deterministic finite automaton.

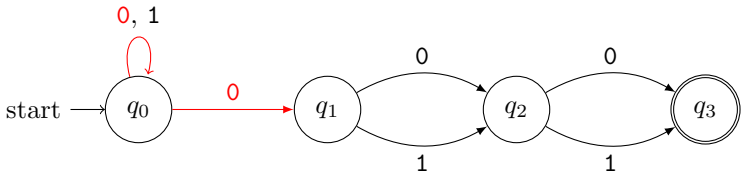
Definition 15 (Nondeterministic finite automaton)**.** A nondeterministic finite automaton is a tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states;
- Σ is an alphabet;
- $\delta\colon Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function;
- $q_0 \in Q$ is the initial or start state; and
- $F \subseteq Q$ is the set of final or accepting states.

As evidenced in the definition, and following our earlier comment, the only change we had to make is in the transition function: we now map to the power set $\mathcal{P}(Q)$ instead of the state set Q. The element of the power

set being mapped to is exactly the subset of states that the nondeterministic finite automaton can transition to from its current state and on its current symbol.

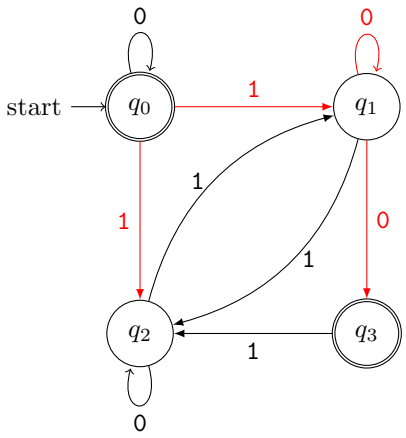
As an illustration of how nondeterminism can simplify the finite automata we construct, let's bring back our example of the language of words whose third-from-last symbol is 0. Here is the nondeterministic version of the finite automaton recognizing this language:



Here, the state q_0 is doing double duty: not only is it reading all of the symbols in the input word up to the third-from-last symbol, but it's also checking that the third-from-last symbol is in fact 0. If it is, then we transition from state q_0 to state q_1, and the remaining states simply read the last two symbols, whatever they may be.

The nondeterminism in this machine is limited to state q_0, where we have two outgoing transitions on the same symbol 0: one transition loops back to the same state q_0, while the other transition takes us to state q_1. We can represent this with the transition function by writing $\delta(q_0, 0) = \{q_0, q_1\}$, and this abides by our definition since $\{q_0, q_1\} \in \mathcal{P}(Q)$.

Example 16. The following finite automaton is nondeterministic, because states q_0 and q_1 each have more than one outgoing transition on the same symbol:



A nondeterministic finite automaton accepts an input word in exactly the same way as a deterministic finite automaton: if the finite automaton is in a final state and there are no more symbols of the input word left to read, then the input word is accepted. If not, then the input word is rejected. We will refer to the class of languages recognized by *some* nondeterministic finite automaton by the abbreviation NFA.

The computation of a nondeterministic finite automaton, however, is slightly different than in the deterministic case. Since the finite automaton can take potentially many transitions from one state/symbol pair, at such a point in the computation, the finite automaton "splits up" and runs multiple copies of itself in parallel. If we were to visualize such a computation, we would obtain a diagram that resembles a tree. In fact, such a visualization is called a *computation tree*!

In each branch of the computation tree, the corresponding copy of the finite automaton continues its computation until it either reaches the end of the input word or finds itself with no more transitions to follow, which could happen if the finite automaton reads a symbol in a state with no outgoing transition on that symbol. If there are no transitions to follow, that branch dies while the remaining branches continue with their computations. Similarly, if there are no symbols left to read in the input word and that copy of the

finite automaton isn't in a final state, that branch dies.

A computation of a nondeterministic finite automaton is therefore accepting only if there exists at least one branch of the computation where the finite automaton is in a final state after reading every symbol of the input word. Just as we did before, we can formalize the notion of an accepting computation for nondeterministic finite automata; the only change we need to make is in the second condition, to account for the change we made to render the transition function nondeterministic.

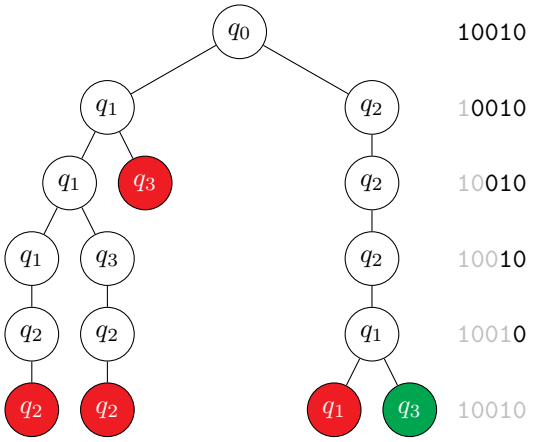
Definition 17 (Accepting computation of a nondeterministic finite automaton)**.** Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite automaton, and let $w = w_0 w_1 \dots w_{n-1}$ be an input word of length n where $w_0, w_1, \dots w_{n-1} \in \Sigma$. The finite automaton $\mathcal{M}$ accepts the input word w if there exists a sequence of states $r_0, r_1, \dots, r_n \in Q$ satisfying the following conditions:

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

Observe that, compared to Definition 11, we substituted inclusion for strict equality in the second condition—this is because the transition function no longer needs to take us *exactly* to state r_{i+1}. Rather, state r_{i+1} needs only to be in the subset mapped to by the transition function.

Example 18. Recall the nondeterministic finite automaton from Example 16. Does this automaton accept the input word `10010`? Let's check by drawing the computation tree.

Each vertex indicates the current state of the finite automaton at a given point in the computation, and the symbols remaining in the input word at that point are listed on the right. A red vertex denotes a rejecting computation, while a green vertex denotes an accepting computation.

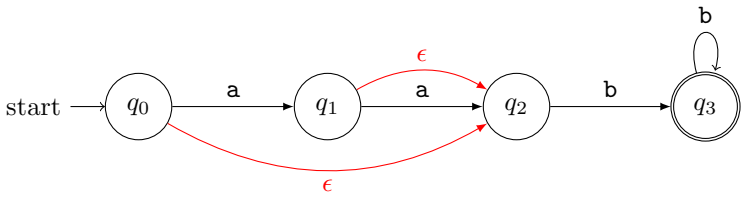


Since there exists at least one branch of the computation tree where the finite automaton is in a final state after reading the entire input word, the finite automaton accepts the word `10010`.

3.5 Epsilon Transitions

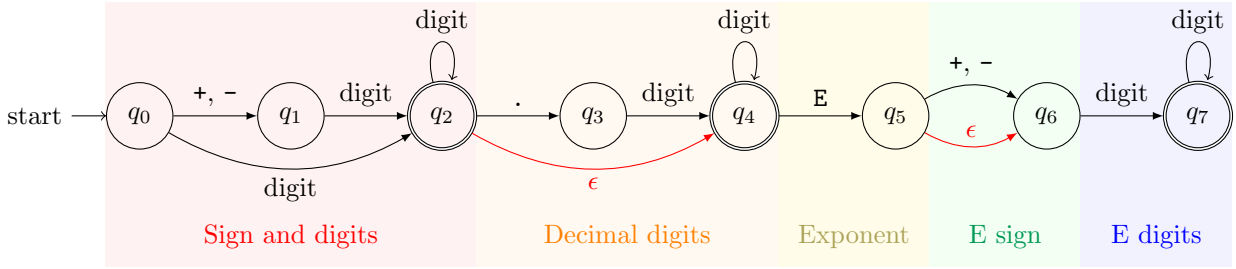
Going one step further, we can take a nondeterministic finite automaton and modify it so that it can transition not just after reading a symbol, but *whenever it wants*. If a certain special transition called an *epsilon transition* exists between two states q_i and q_j, a finite automaton in state q_i can immediately transition to state q_j without reading the next symbol of the input word. We call such a model a nondeterministic finite automaton *with epsilon transitions*, and the class of languages recognized by this model is denoted by ϵ-NFA.

Example 19. The following nondeterministic finite automaton uses epsilon transitions:



This finite automaton accepts all input words starting with zero, one, or two `a`s followed by at least one `b`.

Example 20. The following nondeterministic finite automaton uses epsilon transitions:



This finite automaton recognizes the languages of signed or unsigned floating-point numbers. Some words in this language include `365.25E+2`, `-10E40`, `+2.5`, and `42E-1`. The epsilon transitions allow for words to omit the decimal portion of the number, the sign in the exponent, or both.

Note that adding an epsilon transition to a deterministic finite automaton inherently makes it nondeterministic. This is because we've given the finite automaton the option to transition between two states with or without reading a symbol. There cannot exist a "deterministic finite automaton with epsilon transitions".

We won't spend too much time discussing further details of nondeterministic finite automata with epsilon transitions, since the model is almost identical to the usual nondeterministic finite automaton model. However, we mention it now because, as we're about to see, it makes certain constructions and proofs much easier for us.

3.6 Closure Properties

Closure properties are an important consideration when we discuss any model of computation, since they allow us to determine whether we can apply certain operations to words or languages while still allowing the same model to accept or recognize the result.

We say that a set S is *closed* under an operation $\circ$ if, given any two elements $a, b \in S$, we have that $a \circ b \in S$ as well. You might be familiar with the notion of closure from elsewhere in mathematics: for example, the set of integers is closed under the operations of addition, subtraction, and multiplication, since for all integers a and b, we know that $a + b$, $a - b$, and $a \times b$ are integers. On the other hand, the set of integers is not closed under the operation of division, since (for example) $1, 2 \in \mathbb{Z}$ but $1/2 \notin \mathbb{Z}$.

We can prove all kinds of closure results for languages recognized by finite automata, but here we will focus on three results for each of our three regular operations. In each result, we will follow the same general style of proof to show closure under the specified operation $\circ$: given two finite automata $\mathcal{M}$ and $\mathcal{N}$ recognizing languages $L(\mathcal{M})$ and $L(\mathcal{N})$, we will construct a new finite automaton recognizing the language $L(\mathcal{M}) \circ L(\mathcal{N})$.

Note also that, for each of our three results, we will formulate the statement in terms of nondeterministic finite automata with epsilon transitions. You shouldn't interpret this to mean that deterministic finite automata or nondeterministic finite automata without epsilon transitions are *not* closed under our regular operations—they are! We simply choose to approach each result in this way because the proofs are easiest.

Union

We begin by considering the union operation. To determine whether some input word belongs to the union of two languages $L(\mathcal{A})$ and $L(\mathcal{B})$, we must check that the word is accepted by either $\mathcal{A}$ or $\mathcal{B}$, or by both. Thus, we must essentially perform two parallel "subcomputations" for each of these finite automata. This parallelism means we must also incorporate nondeterminism into our computation, since we don't know in advance which of the two finite automata will accept the word.

Theorem 21. *The class ϵ-NFA is closed under the operation of union.*

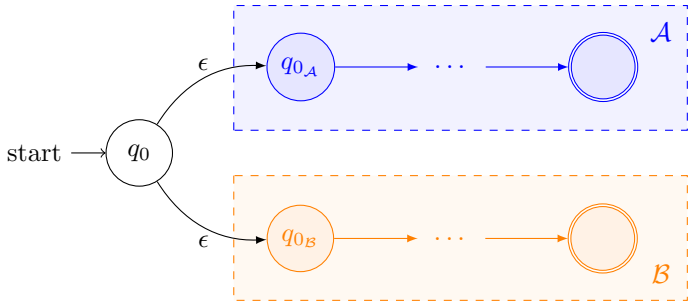
Proof. Suppose we are given two nondeterministic finite automata with epsilon transitions, denoted $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, q_{0_{\mathcal{A}}}, F_{\mathcal{A}})$ and $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, q_{0_{\mathcal{B}}}, F_{\mathcal{B}})$. We construct a finite automaton $\mathcal{C}$ recognizing the language $L(\mathcal{A}) \cup L(\mathcal{B})$ in the following way:

- Take $Q_{\mathcal{C}} = Q_{\mathcal{A}} \cup Q_{\mathcal{B}} \cup \{q_0\}$.
- Take $q_{0_{\mathcal{C}}} = q_0$.
- Take $F_{\mathcal{C}} = F_{\mathcal{A}} \cup F_{\mathcal{B}}$.
- Define $\delta_{\mathcal{C}}$ such that, for all $q \in Q_{\mathcal{C}}$ and for all $a \in \Sigma \cup \{\epsilon\}$,

$$\delta_{\mathcal{C}}(q,a) = \begin{cases} \delta_{\mathcal{A}}(q,a) & \text{if } q \in Q_{\mathcal{A}}; \\ \delta_{\mathcal{B}}(q,a) & \text{if } q \in Q_{\mathcal{B}}; \text{ and} \\ \{q_{0_{\mathcal{A}}}, q_{0_{\mathcal{B}}}\} & \text{if } q = q_0 \text{ and } a = \epsilon. \end{cases}$$

□

Diagrammatically, the "union" finite automaton $\mathcal{C}$ looks like the following:



Concatenation

Next, we consider the concatenation operation. To determine whether some input word belongs to the concatenation language $L(\mathcal{A})L(\mathcal{B})$, we again need to perform two "subcomputations" on both finite automata $\mathcal{A}$ and $\mathcal{B}$, but this time in series. The first part of the word should take us to a final state of $\mathcal{A}$, at which point we will jump to $\mathcal{B}$ to read the remaining second part of the word. However, since we don't know where this "jumping point" is within the word, we again need nondeterminism to guess when we have reached a final state of $\mathcal{A}$.

Theorem 22. *The class ϵ-NFA is closed under the operation of concatenation.*

Proof. Suppose we are given two nondeterministic finite automata with epsilon transitions, denoted $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, q_{0_{\mathcal{A}}}, F_{\mathcal{A}})$ and $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, q_{0_{\mathcal{B}}}, F_{\mathcal{B}})$. We construct a finite automaton $\mathcal{C}$ recognizing the language $L(\mathcal{A})L(\mathcal{B})$ in the following way:

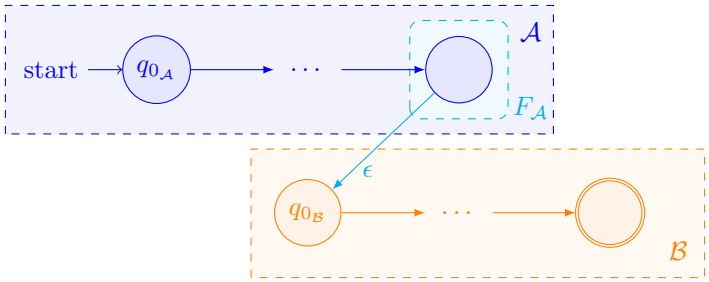
- Take $Q_{\mathcal{C}} = Q_{\mathcal{A}} \cup Q_{\mathcal{B}}$.
- Take $q_{0_{\mathcal{C}}} = q_{0_{\mathcal{A}}}$.

- Take $F_{\mathcal{C}} = F_{\mathcal{B}}$.
- Define $\delta_{\mathcal{C}}$ such that, for all $q \in Q_{\mathcal{C}}$ and for all $a \in \Sigma \cup \{\epsilon\}$,

$$\delta_{\mathcal{C}}(q,a) = \begin{cases} \delta_{\mathcal{A}}(q,a) & \text{if } q \in Q_{\mathcal{A}} \text{ and } q \notin F_{\mathcal{A}}; \\ \delta_{\mathcal{A}}(q,a) & \text{if } q \in F_{\mathcal{A}} \text{ and } a \neq \epsilon; \\ \delta_{\mathcal{A}}(q,a) \cup \{q_{0_{\mathcal{B}}}\} & \text{if } q \in F_{\mathcal{A}} \text{ and } a = \epsilon; \text{ and} \\ \delta_{\mathcal{B}}(q,a) & \text{if } q \in Q_{\mathcal{B}}. \end{cases}$$

□

Diagrammatically, the "concatenation" finite automaton $\mathcal{C}$ looks like the following:



Kleene Star

For our last result, pertaining to the Kleene star, we consider just one finite automaton instead of two. However, the construction process is similar to that which we just saw for concatenation. Since the Kleene star is essentially repeated concatenation, upon reaching a final state of the finite automaton $\mathcal{A}$, we will jump backward to allow us to cycle through the computation again if we desire.

There is one technicality, though: we can't jump backward directly to the original initial state of $\mathcal{A}$, since if that initial state has a looping transition, we might be able to mistakenly accept words not in the original language. Thus, we will jump backward to a new state, and from there we can transition to the original initial state of $\mathcal{A}$.

Theorem 23. *The class ϵ-NFA is closed under the operation of Kleene star.*

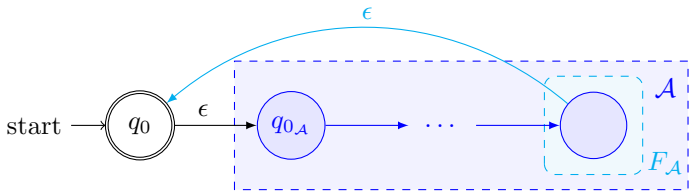
Proof. Suppose we are given a nondeterministic finite automaton with epsilon transitions, denoted $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, q_{0_{\mathcal{A}}}, F_{\mathcal{A}})$. We construct a finite automaton $\mathcal{A}'$ recognizing the language $L(\mathcal{A})^*$ in the following way:

- Take $Q_{\mathcal{A}'} = Q_{\mathcal{A}} \cup \{q_0\}$.
- Take $q_{0_{\mathcal{A}'}} = q_0$.
- Take $F_{\mathcal{A}'} = \{q_0\}$.
- Define $\delta_{\mathcal{A}'}$ such that, for all $q \in Q_{\mathcal{A}'}$ and for all $a \in \Sigma \cup \{\epsilon\}$,

$$\delta_{\mathcal{A}'}(q,a) = \begin{cases} \delta_{\mathcal{A}}(q,a) & \text{if } q \in Q_{\mathcal{A}} \text{ and } q \notin F_{\mathcal{A}}; \\ \delta_{\mathcal{A}}(q,a) & \text{if } q \in F_{\mathcal{A}} \text{ and } a \neq \epsilon; \\ \delta_{\mathcal{A}}(q,a) \cup \{q_0\} & \text{if } q \in F_{\mathcal{A}} \text{ and } a = \epsilon; \text{ and} \\ \{q_{0_{\mathcal{A}}}\} & \text{if } q = q_0 \text{ and } a = \epsilon. \end{cases}$$

□

Diagrammatically, the "Kleene star" finite automaton $\mathcal{A}'$ looks like the following:



4 Equivalence of Models

By now, we've learned about a handful of different models of computation: regular expressions, deterministic finite automata, nondeterministic finite automata, and nondeterministic finite automata with epsilon transitions. Regular expressions and, more generally, regular operations give us a textual, language-oriented way of reasoning about regular languages, while finite automata allow us to think in terms of machines. While these two approaches may seem far apart, there actually isn't as much difference between them as one might think.

Let's focus on finite automata for a moment. Going from deterministic to nondeterministic models, we saw that we can construct finite automata that both recognize the same language and are easier to understand—for instance, by virtue of having fewer states or transitions. By introducing epsilon transitions, we learned that we don't even necessarily need to read symbols in order to transition from one state to another.

It seems that this ongoing weakening of conditions keeps giving us models that can "do more". You may be surprised to learn, however, that all of these models of computation are equivalent in terms of the languages they can recognize! No matter what flavour of finite automaton we have, we can still only recognize the one class of regular languages.

We will prove this automaton equivalence in two steps. First, we will devise a procedure to convert from a nondeterministic finite automaton with epsilon transitions to one without. Afterward, we will see how to convert from a nondeterministic finite automaton to a deterministic finite automaton.

4.1 ϵ-NFA = NFA

In our first procedure, we will use the notion of *epsilon closure* to remove epsilon transitions from a nondeterministic finite automaton. The epsilon closure of a state q is the set of states where there exists some sequence of epsilon transitions from q to that state. Note that the epsilon closure of q always includes q itself.

Theorem 24. *Given a nondeterministic finite automaton with epsilon transitions $\mathcal{M}$, we can convert it to a nondeterministic finite automaton $\mathcal{M}'$ without epsilon transitions.*

Proof. Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite automaton with epsilon transitions. We will construct an equivalent nondeterministic finite automaton $\mathcal{M}' = (Q', \Sigma, \delta', q_0', F')$ without epsilon transitions in the following way:

1. Take Q' to be the original state set Q, and remove all states having only epsilon transitions *to* that state. The starting state is not removed, so take $q_0' = q_0$. All final states in $\mathcal{M}$ remain final states in $\mathcal{M}'$ unless they were removed.
2. Take δ' to be the original transition function δ, but with all epsilon transitions removed. For all states removed in the previous step, also remove all transitions *from* that state.