St. Francis Xavier University
Department of Computer Science

CSCI 356: Theory of Computing
Lecture 1: Regular Languages
Fall 2022

1 Regular Operations and Regular Languages

In this lecture, we will begin our exploration into the theory of computation by investigating a rather simple model of computation and determining the kinds of languages this model can recognize. Before we get to defining our model, though, we will take a look at the languages themselves.

1.1 Regular Operations

Recall that, if we're given two sets A and B, we can apply certain operations to produce new sets. The set operations we're most familiar with are those of union, intersection, complement, and difference. Since languages are essentially sets, we can similarly apply certain operations to languages in order to produce new languages. Three operations in particular are so important that we give them a special name: the *regular operations*.

Definition 1 (Regular operations)**.** Let L, L_1, and L_2 be languages. Then the regular operations of union, concatenation, and Kleene star are defined as follows:

- Union: $L_1 \cup L_2 = \{w \mid w \in L_1 \text{ or } w \in L_2\}$;
- Concatenation: $L_1 L_2 = \{wv \mid w \in L_1 \text{ and } v \in L_2\}$; and
- Kleene star: $L^* = \bigcup_{i \geq 0} L^i$, where $L^0 = \{\epsilon\}$, $L^1 = L$, and $L^i = \{wv \mid w \in L^{i-1} \text{ and } v \in L\}$.

The union operation, naturally, works in exactly the same way for languages as it does for sets. The other two operations, on the other hand, don't have an exact match to a set operation, but we can reason about them by drawing analogies to other operations we've seen.

The concatenation operation is most similar to our notion of tuples, if we stripped away all of the sequence-y notation; concatenation takes two words and "connects" the end of the first word to the beginning of the second word.

Lastly, the Kleene star operation is somewhat similar to taking a repeated Cartesian product, if we "connect" our elements (words) via concatenation rather than in a tuple. Note that, since the Kleene star allows us to take zero copies of a word, the empty word ϵ is always included in the resulting language.

Example 2. Let $L_1 = \{\mathtt{a}, \mathtt{b}\}$ and $L_2 = \{\mathtt{d}, \mathtt{e}\}$. Then $L_1 \cup L_2 = \{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{e}\}$, $L_1 L_2 = \{\mathtt{ad}, \mathtt{ae}, \mathtt{bd}, \mathtt{be}\}$, $L_1^* = \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \mathtt{aaa}, \mathtt{aab}, \dots\}$, and $L_2^* = \{\epsilon, \mathtt{d}, \mathtt{e}, \mathtt{dd}, \mathtt{de}, \mathtt{ed}, \mathtt{ee}, \mathtt{ddd}, \mathtt{dde}, \dots\}$

1.2 Regular Languages

So, what makes these particular operations so special, and why do we refer to them as the regular operations? As it turns out, taking just these three operations is sufficient to allow us to define the smallest class of languages that is "interesting enough" to study[1]: the *regular languages*.

[1]There is a smaller class called the class of *finite languages*. However, it's not too interesting: it consists only of languages with a finite number of words. Introducing the Kleene star allows us to produce infinite-size languages.

Definition 3 (Regular languages—language-theoretic def'n)**.** Let Σ be an alphabet. The class of regular languages is defined inductively as follows:

1. The empty language, $\emptyset$, is regular.
2. For each $a \in \Sigma$, the language $\{a\}$ is regular.
3. If L_1 and L_2 are regular, then $L_1 \cup L_2$ is regular.
4. If L_1 and L_2 are regular, then $L_1 L_2$ is regular.
5. If L_1 is regular, then L_1^* is regular.

At this point, you might be asking yourself: why do we call these operations and languages "regular"? Stephen Kleene introduced the notion of a regular language in the 1950s, but his justification for the terminology was basically that he couldn't come up with any better name:

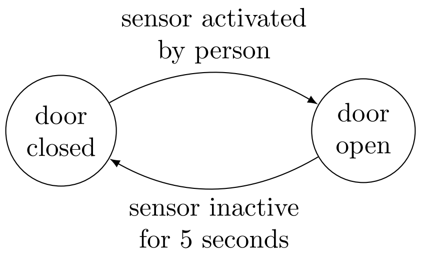
> "We would welcome any suggestions as to a more descriptive term."
> — Stephen Kleene, *Representation of Events in Nerve Nets and Finite Automata*
> RAND Corporation Research Memorandum RM-704, 1951.

Keep the definition of the class of regular languages in mind as we go forward. It will reappear once we introduce our chosen model of computation in this lecture.

2 Finite Automata

The entire point of studying computer science, some might argue, is to determine exactly what computers are capable of. Indeed, humans created computers so that we could pass off boring or repetitive work onto a machine and give our brains a break! However, considering a full computer in the very beginning of our studies is kind of like learning to swim by jumping into the deep end of a pool. In order to learn without getting overwhelmed, we will begin by considering a very simple model of computation that gives us just enough power to actually perform an elementary computation.

If you've ever used a vending machine, or waited in a car at a traffic light, or walked through an automatic door, then you're already familiar with the notion of a *finite automaton*. Consider, for example, how an automatic door works:



The door transitions between two states—closed and open—depending on what the sensor is reporting. The states (circles) represent the door's current status, and the transitions (arrows) correspond to an input given to the door. Note that the door has no way of knowing or remembering that it's closed or open apart from being in a state; it responds solely based on the input it receives from the sensor. This is a finite automaton: an automaton in the sense that it's a machine that performs an action based on predetermined conditions or instructions, and finite in the sense that there's a finite number of possible states the machine can be in at a given time.

2.1 Definition

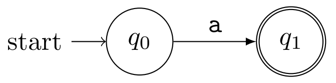
We can use finite automata to model simple computations that take some input word and don't require memory or storage. In a computation, the states of the finite automaton correspond to our current step of the computation. For example, did we just begin the computation, or are we midway through reading some input, or something else? The transitions of the finite automaton take us between states, depending on the label of the transition. If we have, say, a binary word as the input to our finite automaton, then we can transition to a different state depending on whether the next symbol in the word is a `0` or a `1`.

Formally speaking, a finite automaton is just a 5-tuple.

Definition 4 (Finite automaton)**.** A finite automaton is a tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of *states*;
- Σ is an *alphabet*;
- $\delta \colon Q \times \Sigma \to Q$ 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.*

We're already familiar with states and alphabets, and we know a little bit about transitions from our example. The transition function δ is the mathematical formalization of the arrows in our diagram. Given an ordered pair of state and symbol being read, the transition function tells us which state to go to next. For example, if we had a very simple finite automaton like



then the single transition would be represented by the function $\delta(q_0, \mathtt{a}) = q_1$. If a given finite automaton has a large number of transitions, then we can represent each transition concisely in a table format rather than writing each transition out individually.

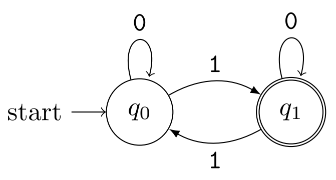
Note that, since we're dealing with a transition *function*, any pair of state and symbol can map to *at most* one state. This condition ensures that we always make the same transition on the same state/symbol pair.

You may have also noticed that the states in our very simple finite automaton had some special flair added to them. The state q_0 has an arrow labelled "start" pointing to it, and the state q_1 has two circles instead of one. This is how we denote initial and final states in our diagram. Initial states have an incoming transition arrow pointing at the state, while final states are double-circled. We typically have just one initial state in a finite automaton, but it's possible to have more than one. On the other hand, we can have as many or as few final states as we want.

Example 5. Consider the finite automaton $\mathcal{M}_1 = (Q, \Sigma, \delta, q_0, F)$ where $Q = \{q_0, q_1\}$, $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, q_0 is the initial state, $F = \{q_1\}$, and δ is defined as follows:

	`0`	`1`
q_0	q_0	q_1
q_1	q_1	q_0

We can draw this finite automaton diagrammatically:



This finite automaton checks whether a binary word has odd parity; that is, whether it contains an odd number of `1`s.

Example 6. Consider the following diagram of a finite automaton:

$$\begin{tikzcd}
\text{start} \arrow[r] & \boxed{q_0} \arrow[loop above, "\texttt{c}"] \arrow[r, bend left, "\texttt{b}"] & q_1 \arrow[l, bend left, "\texttt{c}"] \arrow[r, "\texttt{b}"] & q_2 \arrow[loop above, "\texttt{b,c}"]
\end{tikzcd}$$

This finite automaton checks whether every occurrence of `b` in an input word is immediately followed by an occurrence of `c`.

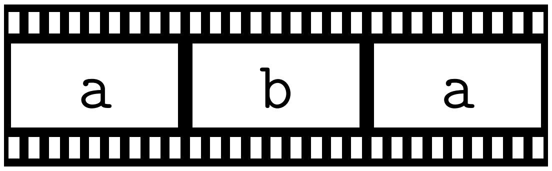
Based on this diagram, we can establish that $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{\texttt{b}, \texttt{c}\}$, q_0 is the initial state, $F = \{q_0\}$, and δ is defined as follows:

	`b`	`c`
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_2

2.2 Computations: Inputs, Accepting, and Rejecting

Now that we know how to define a finite automaton, what can we do with it? Observe that, in our definition, we took care to specify the alphabet Σ. This alphabet gives us information about the kinds of *input words* we can give to a finite automaton. Giving an input word to a finite automaton is much like typing `input()` in a Python program or `scanf()` in a C program; it gives the computer something to read and work with.

When a finite automaton is given an input word, we can imagine the word is written on a reel of film where each symbol in the word has its own frame.



Now, imagine the finite automaton is a film projector, but the rewind button is broken. When we play the film reel starting at the first frame, the projector can only show one frame at a time, and once it moves to the next frame it can never return to the previous one. This is essentially how a finite automaton processes its input: starting with the first symbol of the input word, the finite automaton reads the symbol, transitions to a state, and then moves to the next symbol.

Once the finite automaton reaches the end of its input word, it must make a decision to either *accept* or *reject* the word. Whether or not the finite automaton accepts the input word depends entirely on the state the finite automaton is in at the moment it reaches the end of the word. If the finite automaton is in a final state and it has no more symbols left to read, then it accepts the word. Otherwise, the finite automaton must be in a non-final state, and it therefore rejects the word.

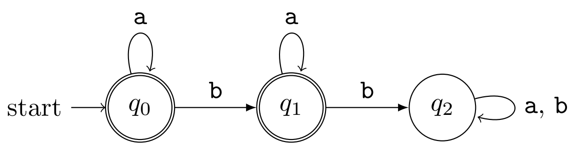
The set of all input words that a finite automaton $\mathcal{M}$ accepts is called the *language* of the finite automaton, denoted $L(\mathcal{M})$, and it's just like any other language: it consists of words over the alphabet Σ. If a finite automaton $\mathcal{M}$ accepts (or *recognizes*[2]) a language A, then $L(\mathcal{M}) = A$. Note that, even though a finite automaton can accept many input words, it can only recognize *one* language.

[2]For clarity's sake, I will try to use the word "accept" only when referring to input words given to a finite automaton, and I will use "recognize" when referring to the language of a finite automaton.

Example 7. Let $\Sigma = \{\texttt{a}, \texttt{b}\}$, and consider the language

$$L_{|w|_{\texttt{b}} \leq 1} = \{w \mid w \text{ contains at most one occurrence of the symbol } \texttt{b}\}.$$

This language can be recognized by the following automaton:



If the input word w contains zero bs, then the finite automaton will remain in the final state q_0. Likewise, if w contains one b, then the finite automaton will enter and remain in the final state q_1. Only if w contains two or more bs does the finite automaton enter the state q_2, where it becomes "stuck" and can no longer accept the input word.

Example 8. A finite automaton with no final states is still able to recognize one language: the empty language, $\emptyset$. This is because the language of input words accepted by the finite automaton is empty.

As a matter of notation, we will refer to the class of languages recognized by *some* finite automaton by the abbreviation DFA. (What does the D mean? We'll find out in the next section...)

We wrap up this section by precisely defining what it means for a finite automaton to accept an input word; that is, by formalizing the notion of an *accepting computation.* We don't need anything new to do this; we already have all the machinery we need.

Definition 9 (Accepting computation of a finite automaton)**.** Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be a 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. $\delta(r_i, w_i) = r_{i+1}$ for all $0 \leq i \leq (n-1)$; and
3. $r_n \in F$.

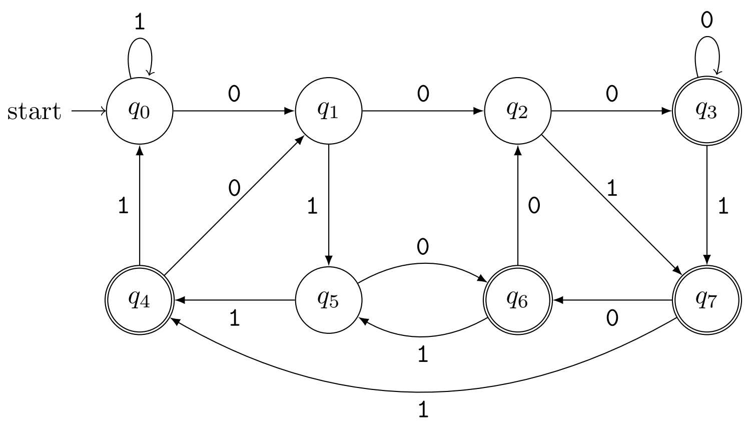
Now that we have the formal notions of a finite automaton and an accepting computation, we can provide an alternative definition of what it means for a language to be regular.

Definition 10 (Regular languages—automata-theoretic def'n)**.** If some finite automaton $\mathcal{M}$ recognizes a language L, then L is regular.

2.3 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, but rather to an element of the power set of states. We'll clarify this in the definition.)

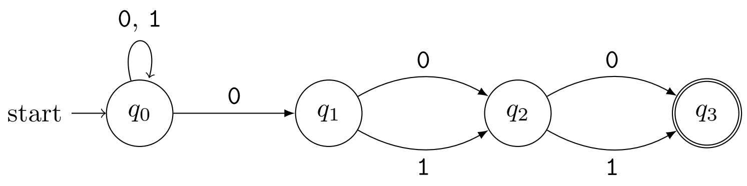
If we get rid of the determinism condition, then the finite automaton can, in a sense, "guess" which step to take at certain points in the computation. If, in a given state, there is more than one transition out of that state on the same symbol, then the finite automaton has multiple options for which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

Definition 11 (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 you can see, the only change we had to make to the definition is in the transition function, where 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.

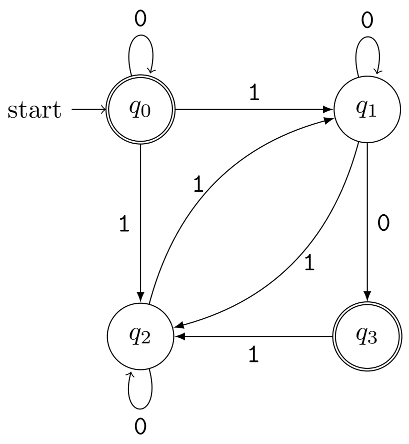
As an illustration of how nondeterminism can simplify the finite automata we construct, think back to our example of the language of words whose third-from-last symbol is `0`. Here is the nondeterministic version of the finite automaton recognizing this language:



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 $\delta(q_0, \mathtt{0}) = \{q_0, q_1\}$, and this abides by our definition since $\{q_0, q_1\} \in \mathcal{P}(Q)$.

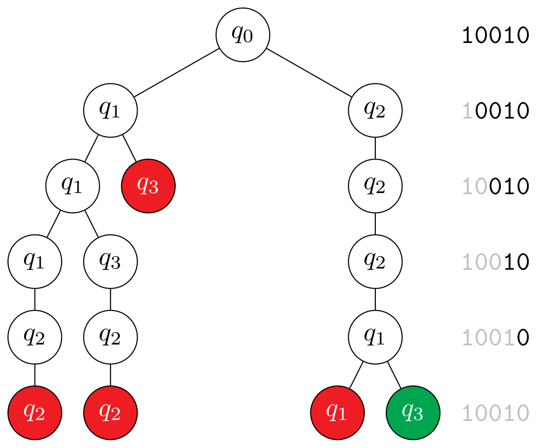
Example 12. The following finite automaton is nondeterministic, because some states have multiple outgoing transitions on the same symbol:



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, 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). In the latter case, that branch dies while the remaining branches continue with their computations. Similarly, if there are no more symbols 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 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.

Example 13. Recall the nondeterministic finite automaton from the previous example. 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 that point in the computation, and the symbols remaining in the input word are listed on the right.



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.

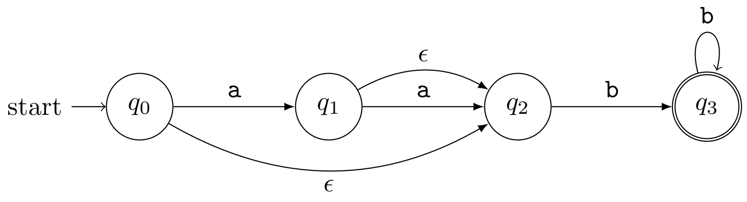
We can formalize the notion of an accepting computation once again for nondeterministic finite automata; the only change we need to make is in the second condition.

Definition 14 (Accepting computation of a nondeterministic finite automaton)**.** Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite automaton, and let $w = w_0 w_1 \ldots w_{n-1}$ be an input word of length n where $w_0, w_1, \ldots w_{n-1} \in \Sigma$. The finite automaton $\mathcal{M}$ accepts the input word w if there exists a sequence of states $r_0, r_1, \ldots, r_n \in Q$ 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. $r_n \in F$.

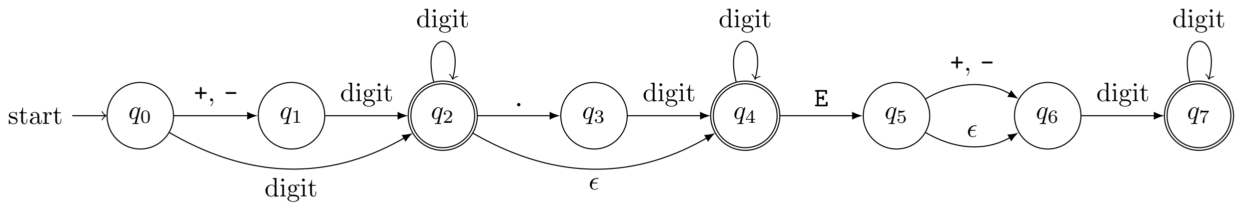
Going one step further, we can take a nondeterministic finite automaton and modify it so that it can transition not just after reading a symbol, but *whenever it wants*. If a certain special transition called an *epsilon transition* exists between two states q_i and q_j, a finite automaton in state q_i 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 15. The following nondeterministic finite automaton uses epsilon transitions:



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

Example 16. 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 the details of nondeterministic finite automata with epsilon transitions, since the model is so similar to the usual nondeterministic finite automaton model. However, we mention it now because it will make some future constructions and proofs much easier for us.

2.4 ϵ-NFA = NFA = DFA

By now, we've learned about three different models of computation: deterministic finite automata, nondeterministic finite automata, and nondeterministic finite automata with epsilon transitions. Going from deterministic to nondeterministic models, we saw that we can construct finite automata that 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 same class of languages.

We will prove this equivalence in two steps, by giving two procedures to convert from a nondeterministic finite automaton with epsilon transitions to one without and to convert from a nondeterministic finite automaton to a deterministic finite automaton.

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.

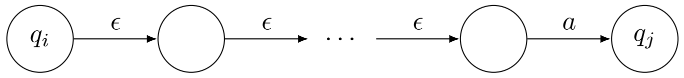
Theorem 17. *Given a nondeterministic finite automaton with epsilon transitions $\mathcal{M}$, we can convert it to a nondeterministic finite automaton $\mathcal{M}'$ without epsilon transitions.*

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:

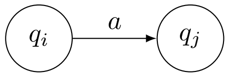
1. Take Q' to be the original state set Q, and remove all states having only epsilon transitions *to* that state. The starting state is not removed, so take $q_0' = q_0$. All final states in $\mathcal{M}$ remain final states in $\mathcal{M}'$ unless they were removed.
2. Take δ' to be the original transition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

3. Add new transitions to the transition function δ' as follows:

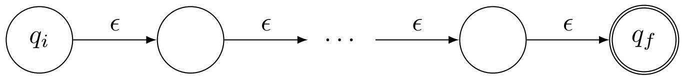
   - If there exists a "chain" of transitions in $\mathcal{M}$ beginning at a state q_i and ending at a state q_j, where all but the last transition is an epsilon transition and the last transition is on some symbol $a \in \Sigma$,

     then replace this "chain" in $\mathcal{M}'$ with a single transition on a between q_i and q_j.

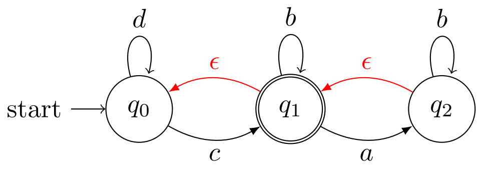
   - If there exists a "chain" of epsilon transitions in $\mathcal{M}$ beginning at a state q_i and ending at a final state $q_f \in F$,

     then remove this "chain" from $\mathcal{M}'$ and make q_i a final state.

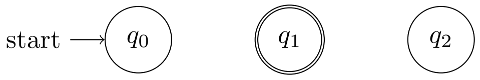
In this way, we have constructed a nondeterministic finite automaton without epsilon transitions recognizing the same language as the original finite automaton. □

Example 18. Consider the following nondeterministic finite automaton with epsilon transitions (highlighted in red):

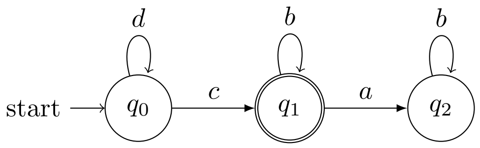


We will use our construction to convert this to a nondeterministic finite automaton without epsilon transitions.

1. First, we take our state set Q' and our initial state q_0'. Since there are no states in this finite automaton having *only* incoming epsilon transitions, we don't need to remove any states.

2. Next, we take our transition function δ' with all epsilon transitions removed. We don't need to remove any other transitions from removed states, since we had no such states in the previous step.

3. Now, we add new transitions to δ' by considering any "chains" in the original finite automaton:

   - For epsilon transition chains ending in a transition on a symbol, we have the following:
     - $q_1 \xrightarrow{\epsilon} q_0 \xrightarrow{c} q_1$ is replaced by $q_1 \xrightarrow{c} q_1$;