

0 Preliminaries

The study of finite automata originally began with theoretical linguists in the mid 20th century trying to understand how language works. They wanted to fully capture the process of language making through a series of formal rules and automatic methods—hence, creating machines that could mimic language “perfectly”. They failed, obviously, because language is constantly evolving and can’t be tied down by a fixed set of rules. However, a lot of their theory became useful in the study of *formal languages* (e.g., programming languages), where the rules are fixed by design in order to avoid unwanted behaviour in machines. This is why a lot of the terminology uses words like ‘alphabets’ and ‘languages’. In this section, we fix a few terms and review some set-theoretic concepts.

An *alphabet* Σ is a finite, non-empty set of symbols. A *string* $w = a_1 a_2 \cdots a_k$ of length k is a sequence of k symbols $a_i \in \Sigma$. We write $|w| = k$ to indicate that the string has length k . We also define for convenience the empty word, ϵ , which has length 0.

Example 0.0.1. For example, given the binary alphabet $\Sigma = \{0, 1\}$, 001 is a word of length 3. •

We define Σ^k to be the set of all k -length words from the alphabet. By convention, $\Sigma^0 = \{\epsilon\}$.

Example 0.0.2. Taking the binary alphabet $\Sigma = \{0, 1\}$ again, we have

$$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\},$$

for example. •

The *Kleene closure* of an alphabet is defined as

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

We also define $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \cdots$. Intuitively Σ^* is the set of all possible finite strings given characters from that alphabet, along with the empty string. Note that Σ^* is always infinite.

Example 0.0.3. For example, the set Σ^* over the binary alphabet is the set of all allowable binary strings, plus the empty word:

$$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \dots\}$$

•
Concatenation of strings is defined in an obvious way: given two strings w_1 and w_2 , concatenating them results in $w_1 w_2$. Concatenating any string with ϵ results in the same string. Similarly, a word w_1 is a *prefix* of w if $w = w_1 w_2$, and a *suffix* of w if $w = w_2 w_1$. We write

$$w^k = \underbrace{w w \cdots w}_{k \text{ times}}$$

and if $w = a_1 a_2 \cdots a_k$ for $a_i \in \Sigma$, then $w^R = a_k a_{k-1} \cdots a_1$ is the reverse of w . We also define *substrings* in an obvious way. A string s is a substring of w if there exist (possibly empty) strings u, v such that $w = usv$.

A *language* L over an alphabet Σ is any set of words over Σ , i.e., a subset $L \subseteq \Sigma^*$. Since languages are sets, we can define the usual set operations of union, intersection, complement, and set difference on them. We

also define some language-specific operations: the concatenation of two languages,

$$L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}, \text{ with } L_1^k = \underbrace{L_1 L_1 \cdots L_1}_{k \text{ times}},$$

and $L_1^0 = \{\epsilon\}$ as usual; and the Kleene closure of a language,

$$L^* = L^0 \cup L^1 \cup L^2 \cup \dots$$

Example 0.0.4. For a simple example, take the languages $L_1 = \{00, 11\}$ and $L_2 = \{01, 10\}$. We then have

$$L_1 L_2 = \{0001, 0010, 1101, 1110\}, L_2 L_1 = \{0100, 0111, 1000, 1011\}.$$

Notice that concatenation is not in general commutative. We also have

$$L_1^* = \{\epsilon, 00, 11, 0000, 0011, 1100, 1111, 000000, 000011, 001100, \dots\}.$$

•

Beyond applications in programming language design, these languages are useful to us because we can represent the solutions of a certain problem with a certain language. For instance, if a given problem is to determine whether or not a binary number is divisible by three, then an equivalent “solution” to that problem would be the set of all binary strings that are divisible by three, along with a process of determining whether or not an arbitrary string belongs to this language. This process of determining whether or not a string is contained inside a language is called a *decision problem*.

1 Deterministic finite automata and regular languages

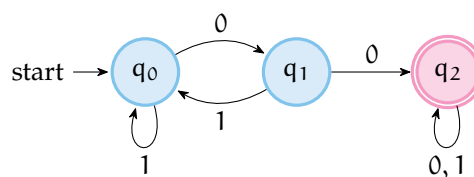
In this chapter, we will cover models of computation that all correspond to (spoilers!) one kind of language: that of the *regular languages*. This will be the weakest form of computation we will consider in this course, but a lot of the same ideas will be present throughout later chapters, so this is a good starting point to gain experience dealing with machines and languages in general. For an overview, we begin by introducing the deterministic finite automata and the kind of languages they compute. Then, we will attempt to prove that these languages are *closed* under certain operations, which will lead us to two new kinds of automata: *nondeterministic* FAs and so-called ϵ -nondeterministic FAs. We will show that these two models of computation, though seemingly more powerful than the DFAs, actually capture the same kinds of languages. Finally, we will create languages that *cannot* be captured by any DFA, using a powerful result known as the pumping lemma.

1.1 DFAs

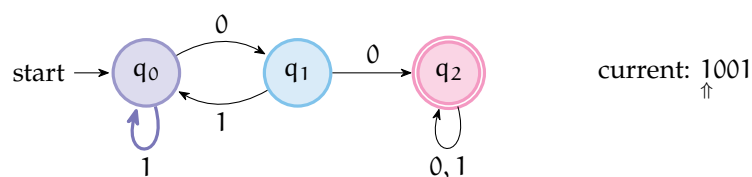
The first model of computation we will tackle is that of a *deterministic finite automaton*, or *DFA*. DFAs consist of finitely many states which are either accepting or not, and we model computation as the DFA taking in *input strings* and moving to a new state at each step of the computation depending on the currently read symbol and the current state the machine is in. Certain states of the DFA will be marked as *accepting* states, so that if the computation given an input string terminates in an accepting state, we say that the string is *accepted* by the DFA. The set of all strings accepted by the DFA is its language.

DFAs being *deterministic* means that there is exactly one way to compute given an input string. It also means that given a state and given a symbol, there will always be a next state to go to; in terms that we will learn later on, this means that the *transition function* is complete over the set of inputs. In general, we will only consider DFAs over the alphabet $\Sigma = \{0, 1\}$.

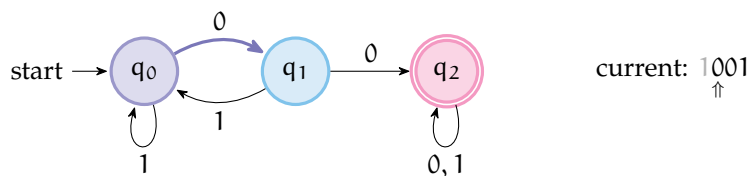
Example 1.1.1. Let $L_1 = \{w \in \Sigma^* \mid 00 \text{ is a substring of } w\}$. Thus, $L_1 = \{00, 001, 100, 000, 1000, 1001, \dots\}$. The DFA that accepts this language is given by:



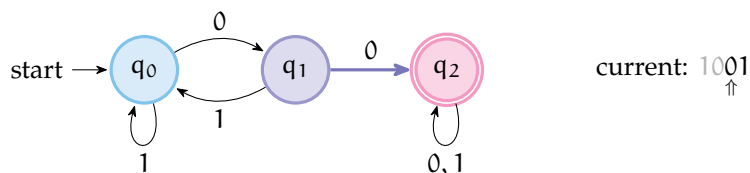
In this diagram, we start the computation at the indicated starting state, and the accepting states are marked with a double outline and a different color. Let's go through the process of accepting a string, 1001.



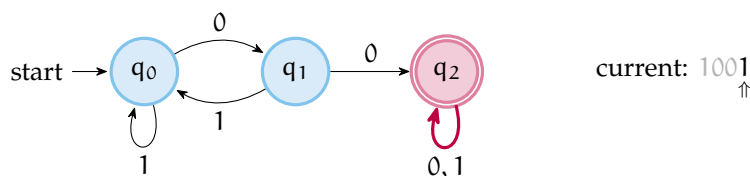
We begin at state q_0 , reading the symbol 1. The arrow coming out of the state labelled with a 1 instructs us to stay in state q_0 when reading that symbol, so we "consume" that symbol and move to the next.



Now we are in state q_0 , reading the symbol 0. The arrow coming out of the state labelled with a 0 instructs us to move to state q_1 , so again we consume the 0 and move on.



Now we are in state q_1 , reading the symbol 0. The arrows instruct us to move to state q_2 and consume the 0.



Now, we have just a 1 left, and the arrows instruct us to stay in place once again. This leaves us with no more symbols to read, which means that the computation has ended. As we can also see, the computation finished in an accepting state, so the string will be accepted. If we repeat the procedure for a string without any 00 contained in it, like 1101, we will see that the computation never reaches beyond state q_1 . Try it out! •

Keeping this example in mind, we can now give the formal definition of a DFA: a deterministic finite automaton M consists of the following data:

- 1 a finite set $Q = \{q_0, q_1, \dots, q_n\}$ of *states*;
- 2 an alphabet of allowed symbols Σ ;
- 3 a *transition function* $\delta : Q \times \Sigma \rightarrow Q$, which takes in as input a state and a symbol from the alphabet, and returns a (possibly different) state;
- 4 an *initial* or *starting state* q_0 ; and
- 5 a subset $F \subseteq Q$ of states known as *final* or *accepting states*.

We may represent this data by writing $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ when we specify a certain DFA.

Example 1.1.2. With the same DFA as the last example, we can see that $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$, the transition function δ is given by the diagram, q_0 is the initial state, and $F = \{q_2\}$. We may also specify a transition function by a complete table of possible inputs and outputs. If we also highlight the starting states and accepting states, then the table conveys the exact same information as the diagram above:

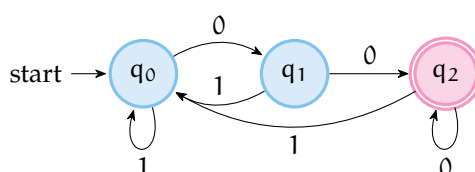
state	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_2	q_0
$q_2 \rightarrow$	q_2	q_2

Here, we signified the starting states by an arrow coming in, and the final states by an arrow coming out. In general, we may use either representation, depending on which is more convenient—though admittedly, I prefer using the diagrams because they’re prettier!

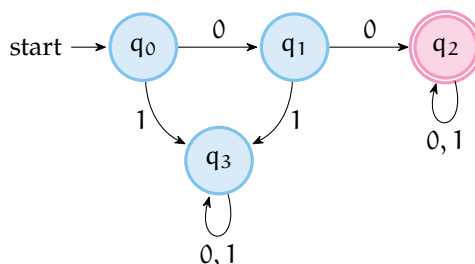


Nerd Interjection! It’s good practice to give meaning to our states, i.e., try to figure out what being in a certain state means. In the previous DFA, we move to q_1 upon reading a 0, but we are only sure that there are two consecutive 0’s if we read another 0 while in q_1 . If we read a 1 instead, we go back to q_0 . Thus, q_1 is a state of “expecting” another 0. Of course, this machine is not really expecting anything, being incapable of thought.

Example 1.1.3. Consider the language $L_1 = \{w \in \Sigma^* \mid 00 \text{ is a suffix of } w\}$. A DFA for this language is:



We can see that this is quite similar to the previous DFA, except we go back to q_0 when we read a 1 in q_2 . Compare that also to the DFA for a seemingly similar language:



This is a DFA for $L_2 = \{w \in \Sigma^* \mid 00 \text{ is a prefix of } w\}$. Obviously, if a string starts with a 1, or a 01, then it is discarded regardless of what comes after. This is represented in the DFA by moving to state q_3 whenever either of these prefixes are read. As we can see, once the DFA goes to q_3 , there is no way out—this is why such states are often called *trap states*.

Example 1.1.4. Let’s transform the second DFA from the previous example into its table representation. We begin at the starting state, q_0 . From here, given a 0, we move to q_1 , and given a 1 we move to q_3 . We represent this by a row of information

state	0	1
$\rightarrow q_0$	q_1	q_3

Then, in q_1 , we move to q_2 given a 0 and to q_3 given a 1. We add this row to our table:

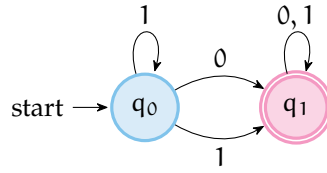
state	0	1
$\rightarrow q_0$	q_1	q_3
q_1	q_2	q_3

We carry on doing this until all states are given rows, leaving us with

state	0	1
$\rightarrow q_0$	q_1	q_3
q_1	q_2	q_3
$q_2 \rightarrow$	q_2	q_2
q_3	q_3	q_3

Make sure you convince yourself that this process is reversible. Try going from this table back to the DFA. •

We can now more formally state the reason DFAs are called “deterministic”. Imagine a DFA like this one:

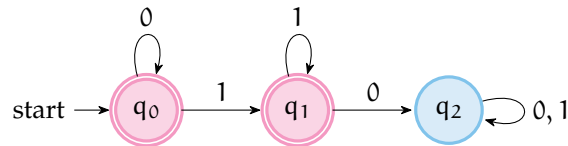


From state q_0 , if we read a 1, we can *either* go to q_1 or loop back and stay in q_0 . Thus, if we take the string 110 for example, we have two possible sequences of states we could go through:

$$q_0 \xrightarrow{1} q_0 \xrightarrow{1} q_1 \xrightarrow{0} q_1 \text{ or } q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_1 \xrightarrow{0} q_1.$$

There is a choice to be made! Compare this to all our previous examples, wherein there was exactly one arrow per symbol coming out of each state. This property ensures that given an arbitrary input string, there is exactly one way of proceeding through the DFA. As it turns out, allowing for multiple transitions per symbol is convenient enough for readability to warrant an entirely new model of computation, that of the *nondeterministic* FAs. We will take a closer look at these in Section 1.3.

Example 1.1.5. All our examples so far have only had one accepting state, but since $F \subseteq Q$ in general, there may be more than one. Take the language $L = \{w \in \Sigma^* \mid w = 0^m 1^n, m, n \geq 0\}$. Recall that in our notation, 0^m means m 0's concatenated together. The DFA for this language is



In this example, we have $F = \{q_0, q_1\}$. •

The next logical step would be to formalize what it means for a DFA to *accept* a string w . We will say that a DFA M accepts $w = a_1 a_2 \cdots a_n$ iff there exists a sequence of states

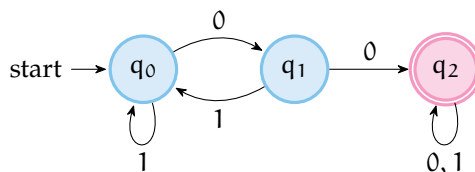
$$q_{(0)} \xrightarrow{a_1} q_{(1)} \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_{(n)}$$

so that the first state in the sequence $q_{(0)} = q_0$ is initial and the last state $q_{(n)} \in F$ is accepting, and for each a_i ,

$$\delta(q_{(i-1)}, a_i) = q_{(i)}.$$

This definition looks intimidating, but all it is saying is that each state in the sequence of states follows the previous one based on the transition function.

Example 1.1.6. Let's look back at our very first example.



Earlier, we went through a step-by-step procedure of accepting the string 1001. We moved in the sequence

$$q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_2 \xrightarrow{1} q_2.$$

In this case, we do have that the first state in the sequence $q_{(0)} = q_0$ is initial, and the last state in the sequence $q_{(5)} = q_2 \in F$ is final. We also know from our diagram that at each state in this sequence, the transition (indicated here by the symbol above each arrow) is valid, i.e., that $\delta(q_{(i-1)}, a_i) = q_{(i)}$. For instance, note that $a_2 = 0$ and $q_{(1)} = q_0$, and we know that $\delta(q_0, 0) = q_1$, which is precisely $q_{(2)}$ in the sequence. •



Nerd Interjection! The distinction between $q_{(i)}$ for the i^{th} state in a *sequence* of states and q_i for the i^{th} state in a *DFA* is quite unfortunate notation for legibility, but it's fairly standard. In any case, there's not really any rule dictating that we need to use q_i for states, so feel free to use a, b, c, \dots instead. Just make sure the symbols you use aren't part of the alphabet, otherwise it can get *really* confusing!

All of that is just a math-y way of saying what us computer scientists understand fairly well as a condition known as “this algorithm terminates with this outcome”. So, if you'd like, you can instead view the outcome of a DFA processing a string as the outcome of this algorithm:

Algorithm 1: The process of accepting or rejecting a string

Input: a string $w = a_1 a_2 \dots a_n$ and a DFA M

Output: either ACCEPT or REJECT

```

1  $q_{\text{curr}} \leftarrow q_0$ ;
2 for  $a_i$  in  $w$  do
3    $q_{\text{curr}} \leftarrow \delta(q_{\text{curr}}, a_i)$ ;
4 if  $q_{\text{curr}} \in F$  then
5   return ACCEPT;
6 return REJECT;
  
```

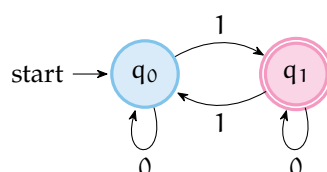
Indeed, this algorithm terminating with ACCEPT is equivalent to the definition we gave earlier of there being an accepting sequence of states, so we can generally adopt either of them as our definition for “DFA M accepts the string w ”. We can even give a *third* formulation using recursion. We construct a new function $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$, which, instead of operating over single symbols, takes in as input entire strings. The function is defined as follows:

- 1 if $w = a$ for some $a \in \Sigma$, then $\hat{\delta}(q_i, a) = \delta(q_i, a)$; and
- 2 if $w = a_1 a_2 \dots a_k$ for some $a_i \in \Sigma$, then $\hat{\delta}(q_i, w) = \delta(\hat{\delta}(q_i, a_1 a_2 \dots a_{k-1}), a_k)$.

Intuitively, all $\hat{\delta}$ does is recursively backtrack until it hits the first symbol, and then iterates through every symbol after that—just like Algorithm 1. Then, we can say that a DFA accepts a string w iff $\hat{\delta}(q_0, w) \in F$ is an accepting state. This definition too is equivalent to our previous ones, and its generally the one we will adopt, if only for notational convenience.

Now that we’ve clarified what it means for a DFA to accept a string, we can define $L(M)$ to be the set of all strings accepted by a certain DFA M , also known as its language. We will say that M *recognizes* L whenever $L = L(M)$. We’ve mostly been providing languages and trying to find DFAs for them, but let’s see how we might go the other way around.

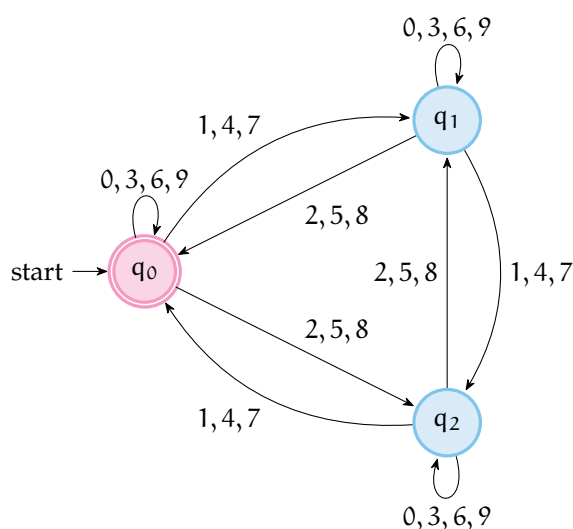
Example 1.1.7. Let M be the following DFA:



Let’s try and figure out what language this DFA recognizes. We can see that reading a 0 makes the computation stay in place, so we can safely assume that the language does not distinguish anything about 0’s, but it switches from accept to non-accept every time it reads a 1. It rejects ϵ , accepts 1, rejects 11, accepts 111, rejects 1111, and so on—it seems to be alternating depending on how many 1’s there are in the string. Thus, $L(M) = \{w \in \Sigma^* \mid w \text{ has an odd number of 1’s}\}$. •

You might be wondering if there’s a way to classify what sorts of languages can be recognized by a DFA without directly referring to a DFA. We’ll look at this question more in depth in Section 1.4, but for now let’s have the following definition: call a language over an alphabet Σ *regular* iff there exists some DFA that recognizes that language.

Example 1.1.8. Here’s another language that a DFA can recognize. Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ be the set of digits in base 10. Thus, strings in Σ^* are just decimal numbers (with some weirdo ones like 000 or 001, but we can just ignore leading 0’s). We’ll show that the language $L = \{w \in \Sigma^* \mid w \text{ is divisible by 3}\}$ is a regular language, by giving this DFA:



Recall that to tell if a number is divisible by 3, we simply add up the digits and check if *that* is divisible by 3. What this machine is doing is keeping track of the remainder mod 3 of the “running total” of the digits! Say

our number is 3147. The DFA reads a 3, so the running total is 3, which is divisible by 3, so we stay in state q_0 . Next, we read a 1, which makes our running total 4. $4 \bmod 3$ is 1, so we go to state q_1 . Next, we read a 4, which makes our running total 8. $8 \bmod 3$ is 2, so we go to state q_2 . Finally, we read a 7, putting our final sum of digits at 15—which is divisible by 3! This is reflected in the fact that the DFA sends us to state q_0 again, the only accepting state, so we accept 3147. •

Via the correspondence we established in Chapter 0 between decision problems and languages, we can say that DFAs can solve the problem of deciding whether or not a number in base 10 is divisible by 3. Quite impressive! With just a few states and a few instructions, we can tell this machine to completely solve this problem. Of course, this is quite an easy problem. How about a problem that a DFA *can't* solve?

1.2 Closure properties

Languages, being sets, can be operated on using established set operations.

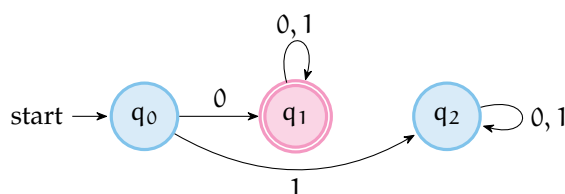
Example 1.2.1. Consider the languages $L_1 = \{w \in \Sigma^* \mid 00 \text{ is a suffix of } w\}$ and $L_2 = \{w \in \Sigma^* \mid w \text{ starts with } 1\}$. These are both regular. Their intersection $L_1 \cap L_2$ would be the set of all strings who start with 1 and end in 00. The first few elements can be listed:

$$L_1 \cap L_2 = \{100, 1000, 1100, 10000, 10100, 11000, 111000, \dots\}.$$

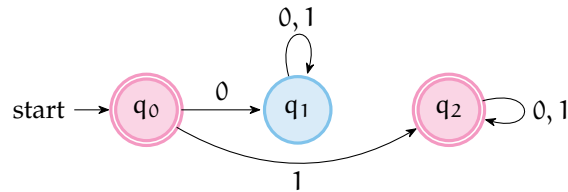
A natural question arises: is the result of applying an operation to one or two regular languages still a regular language? Mathematically, when a type of object satisfies this property, we say that the objects are *closed* under the operation. The prototypical example is the natural numbers, which are closed under addition ($2 + 3 = 5$ and so on) but not closed under subtraction ($2 - 3 = -1$, not a natural number).

The answer, as we will see later, is yes, in the cases of regular set operations. Before we prove these statements in the general case, though, let's think of what we would do given a particular DFA for the case of a set complement. Recall that L^c is the set of all strings over the same alphabet Σ as L but are not themselves members of L . Since we established that a language is regular whenever a DFA recognizes it, if we want to show that L^c is also regular, we need to construct a DFA for it, and we need to do so only given the data of the DFA for L . This turns out to be fairly straightforward.

Example 1.2.2. Consider the DFA M :



Note that $L(M)$ is all strings that start with 0. If we want to accept those strings that M rejects, then we can simply modify M to accept on every non-accepting state, and reject on every accepting state. This process yields the new DFA M' :



It is then clear that $L(M')$ is all strings that start with 1, plus the empty string, which is precisely $L(M)^c$. •

Let's reflect on what we just did. In order to obtain a machine for the complement of a language given a machine for that language, we simply retained all of the original states, the alphabet, the initial state, and the transition function, then flipped all accepting states into non-accepting states and vice-versa. It's not hard to see that this argument works out no matter what the "underlying" DFA is: if in the original DFA M , computation terminates in an accepting state, then in the new M' , computation would terminate in a non-accepting state. We formalize this with the following theorem.

Theorem 1.2.3: Closure under complements

The complement L^c of a regular language L is itself a regular language.

Proof. Let L be a regular language, and $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ be its corresponding DFA. Then, we can construct a new DFA M' for L^c by taking

$$M' = \langle Q, \Sigma, \delta, q_0, Q - F \rangle$$

where $Q - F$ is the set of all non-accepting states in M . Then, $L^c = L(M')$, for if the result of $\hat{\delta}(q_0, w) = q_n \in F$ is an accepting state in M , then this state q_n will not be in $Q - F$, so M' will not accept w . \square

1.3 Nondeterminism

1.4 Regular expressions