The art of art, the glory of expression
and the sunshine of the light of letters is simplicity.
Nothing is better than simplicity . . . .
nothing can make up for excess or for the lack of definiteness.
— Walt Whitman, Preface to *Leaves of Grass* (1855)

Freedom of choice
Is what you got.
Freedom from choice
Is what you want.
— Devo, "Freedom of Choice", *Freedom of Choice* (1980)

4   Nondeterminism

4.1   Nondeterministic State Machines

The following diagram shows something that looks like a finite-state machine over the alphabet $\{0,1\}$, but on closer inspection, it is not consistent with our earlier definitions. On one hand, there are two transitions out of s for each input symbol. On the other hand, states a and b are each missing an outgoing transition.
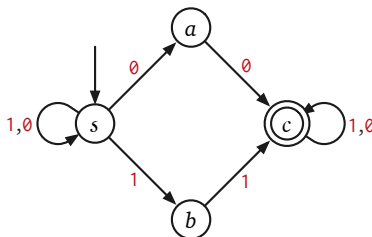
A nondeterministic finite-state automaton

Nevertheless, there is a sense in which this machine "accepts" the set of all strings that contain either 00 or 11 as a substring. Imagine that when the machine reads a symbol in state s, it makes a **choice** about which transition to follow. If the input string contains the substring 00, then it is *possible* for the machine to end in the accepting state c, by *choosing* to move into state a when it reads a 0 immediately before another 0. Similarly, if the input string contains the substring 11, it is *possible* for the machine to end in the accepting state c. On the other hand, if the input string does not contain either 00 or 11—or in other words, if the input alternates between 0s and 1s—there are no choices that lead the machine to the accepting state. If the machine incorrectly chooses to transition to state a and then reads a 1, or transitions to b and then reads 0, it explodes; the only way to avoid an explosion is to stay in state s.

This object is an example of a ***nondeterministic finite-state automaton***, or ***NFA***, so named because its behavior is not uniquely *determined* by the input string. Formally, every NFA has five components:

- An arbitrary finite set Σ, called the ***input alphabet***.

- Another arbitrary finite set Q, whose elements are called ***states***.

- An arbitrary ***transition*** function $\delta\colon Q \times \Sigma \to 2^Q$.

- A ***start state*** $s \in Q$.

- A subset $A \subseteq Q$ of **accepting states**.

The only difference from the formal definition of *deterministic* finite-state automata is the domain of the transition function. In a DFA, the transition function always returns a single state; in an NFA, the transition function returns always a *set* of states, which could be empty, or all of Q, or anything in between.

Just like DFAs, the behavior of an NFA is governed by an **input string** $w \in \Sigma^*$, which the machine reads one symbol at a time, from left to right. Unlike DFAs, however, an NFA does not maintain a single current state, but rather a *set* of current states. Whenever the NFA reads a symbol a, its set of current states changes from C to $\delta(C, a) := \bigcup_{q \in C} \delta(q, a)$. After all symbols have been read, the NFA **accepts** w if its current state set contains *at least one* accepting state and **rejects** w otherwise. In particular, if the set of current states ever becomes empty, it will stay empty forever, and the NFA will reject.

For example, given the input string `01101001`, our example NFA's set of current states evolves as follows.

$$\{s\} \xrightarrow{0} \{s, a\} \xrightarrow{1} \{s, b\} \xrightarrow{1} \{s, b, c\} \xrightarrow{0} \{s, a, c\} \xrightarrow{1} \{s, b, c\} \xrightarrow{0} \{s, a, c\} \xrightarrow{0} \{s, a, c\} \xrightarrow{1} \{s, b, c\}$$

This sequence of transitions is drawn more explicitly in the figure below. (Notice the explosions whenever the NFA reads a `1` in state a, or reads a `0` in state b.) Because the final set of states $\{a, b, c\}$ contains the accept state c, our NFA accepts this input strung.
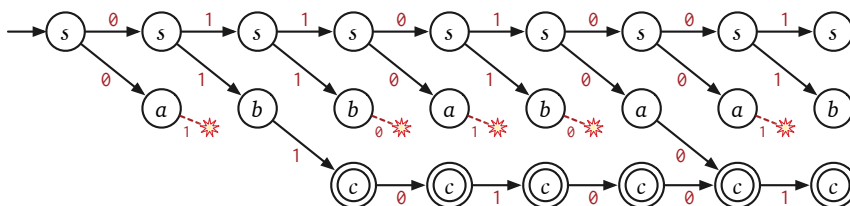
Running our example NFA on the input string `01101001`.

More formally, for any NFA $(\Sigma, Q, \delta, s, A)$, we recursively define an *extended transition function* $\delta^*: Q \times \Sigma^* \to 2^Q$ that takes a state and a *string* as input, and returns a set of states as output:

$$\delta^*(q, w) := \begin{cases} \{q\} & \text{if } w = \varepsilon, \\ \displaystyle\bigcup_{r \in \delta(q,a)} \delta^*(r, x) & \text{if } w = ax. \end{cases}$$

For example, in our example NFA, we have $\delta^*(s, \texttt{01101001}) = \{s, b, c\}$ and $\delta^*(s, \texttt{01010}) = \{s, a\}$. Finally, the NFA **accepts** a string $w \in \Sigma^*$ if and only if $\delta^*(s, w) \cap A \neq \varnothing$.

We can equivalently define an NFA as *any* directed graph whose vertices are the states Q, and whose edges are labeled with symbols from Σ. Unlike DFAs, we do not require that every vertex has exactly one outgoing edge with each label; it may have several such edges or none. An NFA accepts a string w if the graph contains *at least one* walk from the start state to an accepting state whose label is the string w.

★★★

> It's arguably more natural to an arbitrary *set* of start states $S \subseteq Q$ instead of just one. Then an NFA accepts a string w if and only if there is a sequence of transitions consistent with w from *some* start state to *some* accepting state, or more formally if $\delta^*(S, q) \cap A \neq \varnothing$. Change the definition and chase through all the theorems? Or prove equivalence and bounce back and forth, like we already do for ε-transitions?

4.2 Intuition

There are at least three useful ways to think about non-determinism.

Clairvoyance. Whenever an NFA reads symbol a in state q, it *chooses* the next state from the set $\delta(q,a)$, always *magically* choosing a state that leads to the NFA accepting the input string, unless no such choice is possible. Even if there is only one sequence of choices out of 14,000,605 that leads the machine to an accepting state, the NFA magically makes that choice.[1] As the BSD fortune file put it, "Nondeterminism means never having to say you are wrong."[2] Of course real machines can't actually look into the future; that's why I used the word "magic".

Parallel threads. An arguably more "realistic" view is that when an NFA reads symbol a in state q, it spawns an independent execution thread for each state in $\delta(q,a)$. In particular, if $\delta(q,a)$ is empty, the current thread simply dies. The NFA accepts if *at least one* thread is in an accepting state after it reads the last input symbol. This is the intuition illustrated in the figure on the previous page.

Equivalently, we can imagine that when an NFA reads symbol a in state q, it branches into several parallel universes, one for each state in $\delta(q,a)$. If $\delta(q,a)$ is empty, the NFA destroys the universe (including itself) to protect the Sacred Timeline. Similarly, if the NFA finds itself in a non-accepting state when the input ends, the NFA destroys the universe to protect the Sacred Timeline. Thus, when the input is gone, only universes in which the NFA somehow chose a path to an accept state still exist. One slight disadvantage of this metaphor is that if an NFA reads a string that is not in its language, it destroys *all* universes.

Proofs/oracles. Finally, we can treat NFAs not as a mechanism for *computing* something, but as a mechanism for *verifying proofs*. If we want to *prove* that a string w contains one of the suffixes 00 or 11, it suffices to demonstrate a single walk in our example NFA that starts at s and ends at c, and whose edges are labeled with the symbols in w. Equivalently, whenever the NFA faces a nontrivial choice, the prover can simply tell the NFA which state to move to next.

This intuition can be formalized as follows. Consider a *deterministic* finite state machine whose input alphabet is the product $\Sigma \times \Omega$ of an **input** alphabet Σ and an **oracle** alphabet Ω. Equivalently, we can imagine that this DFA reads simultaneously from two strings of the same length: the *input* string w and the *oracle* string Ω. In either formulation, the transition function has the form $\delta\colon Q \times (\Sigma \times \Omega) \to Q$. As usual, this DFA accepts the pair $(w,\Omega) \in (\Sigma \times \Omega)^*$ if and only if $\delta^*(s,(w,\Omega)) \in A$. Finally, M **nondeterministically accepts** the string $w \in \Sigma^*$ if there is an oracle string $\Omega \in \Omega^*$ with $|\Omega| = |w|$ such that $(w,\Omega) \in L(M)$.
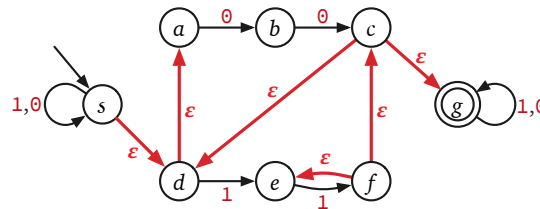
4.3 ε-Transitions

It is fairly common for NFAs to include so-called **ε-transitions**, which allow the machine to change state without reading an input symbol. An NFA with ε-transitions accepts a string w if and only if there is a sequence of transitions $s \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_\ell} q_\ell$ where the final state q_ℓ is accepting, each a_i is either ε or a symbol in Σ, and $a_1 a_2 \cdots a_\ell = w$.

---

[1]Yes, even if it means Black Widow and Iron Man have to die.

[2]This sentence is a riff on a horrible aphorism that was (sadly) popular in the US in the 70s and 80s. Fortunately, everyone seems to have forgotten the original saying, except maybe for that one time it was mocked on *The Simpsons*. Ah, who am I kidding? Nobody remembers *The Simpsons* either.

For example, consider the following NFA with ε-transitions. (For this example, we indicate the ε-transitions using large red arrows; we won't normally do that.) This NFA deliberately has more ε-transitions than necessary.

A (rather silly) NFA with ε-transitions

The NFA starts as usual in state s. If the input string is 100111, the the machine might non-deterministically choose the following transitions and then accept.

$$s \xrightarrow{\varepsilon} s \xrightarrow{1} s \xrightarrow{\varepsilon} d \xrightarrow{\varepsilon} a \xrightarrow{0} b \xrightarrow{0} c \xrightarrow{\varepsilon} d \xrightarrow{1} e \xrightarrow{1} f \xrightarrow{\varepsilon} e \xrightarrow{1} f \xrightarrow{\varepsilon} c \xrightarrow{\varepsilon} g$$

More formally, the transition function in an NFA with ε-transitions has a slightly larger domain $\delta \colon Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$. The **$\varepsilon$-reach** of a state $q \in Q$ consists of all states r that satisfy one of the following recursive conditions:

- either $r = q$,

- or $r \in \delta(q', \varepsilon)$ for some state q' in the ε-reach of q.

In other words, r is in the ε-reach of q if there is a (possibly empty) sequence of ε-transitions leading from q to r. For instance, in the example NFA above, the ε-reach of state f is the set $\{a, c, d, e, f, g\}$, but the ε-reach of state a is just $\{a\}$.

Now we redefine the extended transition function $\delta^* \colon Q \times \Sigma^* \to 2^Q$, which transitions on arbitrary strings, as follows:

$$\delta^*(p, w) := \begin{cases} \varepsilon\text{-}reach(p) & \text{if } w = \varepsilon, \\ \displaystyle\bigcup_{r \in \varepsilon\text{-}reach(p)} \bigcup_{q \in \delta(r,a)} \delta^*(q, x) & \text{if } w = ax. \end{cases}$$

If we abuse notation by writing $\delta(S, a) = \bigcup_{q \in S} \delta(q, a)$ and $\delta^*(S, w) = \bigcup_{q \in S} \delta^*(q, w)$ and $\varepsilon\text{-}reach(S) = \bigcup_{q \in S} \varepsilon\text{-}reach(q)$ for any subset of states $S \subseteq Q$, this definition simplifies as follows:

$$\delta^*(p, w) := \begin{cases} \varepsilon\text{-}reach(p) & \text{if } w = \varepsilon, \\ \delta^*(\delta(\varepsilon\text{-}reach(p), a), x) & \text{if } w = ax. \end{cases}$$

Finally, as usual, an NFA with ε-transitions accepts a string w if and only if $\delta^*(s, w)$ contains at least one accepting state.

Although it may appear at first that ε-transitions give us a more powerful set of machines, NFAs with and without ε-transitions are actually equivalent. Given an NFA $M = (\Sigma, Q, s, A, \delta)$ with ε-transitions, we can construct an equivalent NFA $M' = (\Sigma, Q', s', A', \delta')$ without ε-transitions as follows:

$$Q' := Q$$
$$s' = s$$
$$A' = \left\{ q \in Q \mid \varepsilon\text{-}reach(q) \cap A \neq \varnothing \right\}$$
$$\delta'(q, a) = \delta(\varepsilon\text{-}reach(q), a)$$

Straightforward definition-chasing now implies that M and M' accept exactly the same language. Thus, whenever we reason about or design NFAs, we are free to either allow or forbid ε-transitions, whichever is more convenient for the task at hand.

For example, our previous NFA with ε-transitions can be transformed into an equivalent NFA without ε-transitions, as shown in the figure below. The NFA on the right has two unreachable states a and d, but whatever.
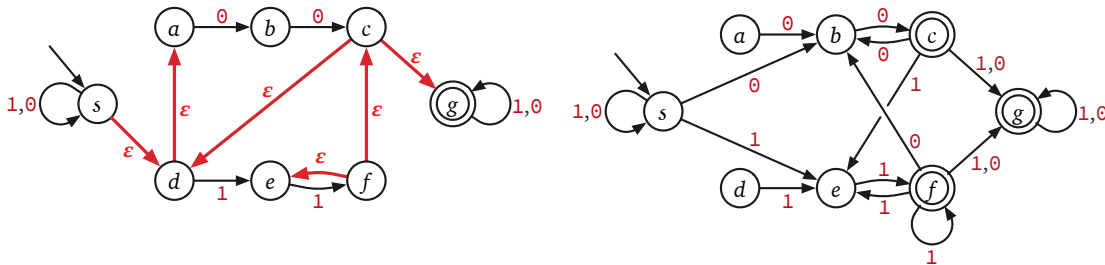
A (rather silly) NFA with ε-transitions, and an equivalent NFA without ε-transitions

★★★

This reduction might be easier to understand incrementally.

- For every transition pair $p \xrightarrow{\varepsilon} q \xrightarrow{a} r$, add a direct transition $p \xrightarrow{a} r$. This addition does not change the accepted language.

- For each transition $p \xrightarrow{\varepsilon} q$ where q is an accepting state, make p an accepting state. This modification does not change the accepted language.

- When no more of the previous modifications are possible, delete all ε-transitions. This modification does not change the accepted language.

4.4 Kleene's Theorem

We are now finally in a position to prove the following fundamental fact, first observed by Stephen Kleene in 1951:

Theorem 4.1. *A language L can be described by a regular expression if and only if L is the language accepted by a DFA.*

We will prove Kleene's fundamental theorem in four stages:

- Every DFA can be transformed into an equivalent NFA.
- Every NFA can be transformed into an equivalent DFA.
- Every regular expression can be transformed into an equivalent NFA.
- Every NFA can be transformed into an equivalent regular expression.

The first of these four transformations is completely trivial; a DFA is just a special type of NFA where the transition function always returns a single state. Unfortunately, the other three transformations require a bit more work.

4.5 NFA to DFA: The Subset Construction

In the parallel-thread model of NFA execution, an NFA does not have a single current state, but rather a *set* of current states. The evolution of this set of states is *determined* by a modified

transition function $\delta': 2^Q \times \Sigma \to 2^Q$, defined by setting $\delta'(P,a) := \bigcup_{p \in P} \delta(p,a)$ for any set of states $P \subseteq Q$ and any symbol $a \in \Sigma$. When the NFA finishes reading its input string, it accepts if and only if the current set of states intersects the set A of accepting states.

This formulation makes the NFA completely deterministic! We have just shown that any NFA $M = (\Sigma, Q, s, A, \delta)$ is equivalent to a DFA $M' = (\Sigma, Q', s', A', \delta')$ defined as follows:

$$Q' := 2^Q$$
$$s' := \{s\}$$
$$A' := \{S \subseteq Q \mid S \cap A \neq \varnothing\}$$
$$\delta'(q',a) := \bigcup_{p \in q'} \delta(p,a) \qquad \text{for all } q' \subseteq Q \text{ and } a \in \Sigma.$$

Similarly, any NFA with ε-transitions is equivalent to a DFA defined as follows:

$$Q' := 2^Q$$
$$s' := \{s\}$$
$$A' := \{S \subseteq Q \mid \varepsilon\text{-}reach(S) \cap A \neq \varnothing\}$$
$$\delta'(q',a) := \bigcup_{p \in q'} \bigcup_{r \in \varepsilon\text{-}reach(p)} \delta(r,a) \qquad \text{for all } q' \subseteq Q \text{ and } a \in \Sigma.$$

This conversion from NFA to DFA is often called the **subset construction**, but that name is somewhat misleading; it's not a "construction" so much as a change in perspective.

For example, the subset construction converts the 4-state NFA on the first page of this note into the following 16-state DFA. To simplify notation, I've named each DFA state using a simple string, omitting the braces and commas from the corresponding subset of NFA states; for example, DFA state sbc corresponds to the subset $\{s, b, c\}$ of NFA states.
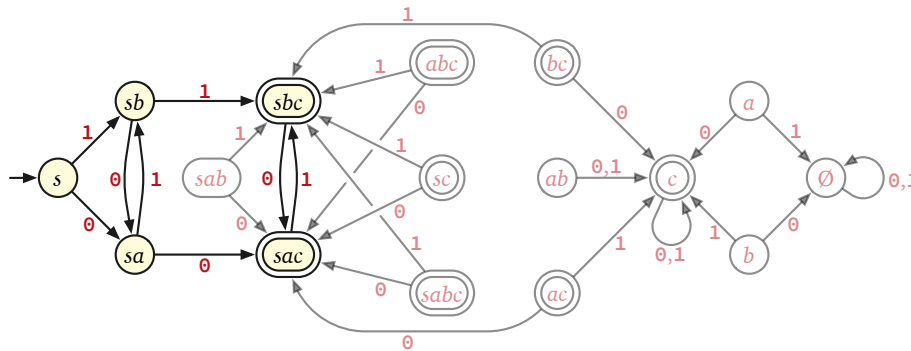
The 16-state DFA obtained from our first 4-state NFA by the subset construction.
Only the five yellow states are reachable from the start state.

An obvious disadvantage of this "construction" is that it (usually) leads to DFAs with far more states than necessary, in part because many states cannot even be reached from the start state. In the example above, there are eleven unreachable states; only five states are reachable from s.

Incremental Subset Construction

Instead of building the entire subset DFA and then discarding the unreachable states, we can avoid the unreachable states from the beginning by constructing the DFA incrementally, essentially by performing a breadth-first search of the DFA graph.

To execute this algorithm by hand, we prepare a table with $|\Sigma| + 3$ columns, with one row for each DFA state we discover. In order, these columns record the following information:

- The DFA state (as a subset of NFA states)

- The ε-reach of the corresponding subset of NFA states

- Whether the DFA state is accepting (that is, whether the ε-reach intersects A)

- The output of the transition function for each symbol in Σ.

We start with DFA-state $\{s\}$ in the first row and first column. Whenever we discover an unexplored state in one of the last $|\Sigma|$ columns, we copy it to the left column in a new row. To reduce notational clutter, we write all subsets of NFA states without braces or commas.

For example, given the NFA with ε-transitions from Section 4.3, the standard subset construction would produce a DFA with 256 states, but the incremental subset construction produces an nine-state DFA, described by the following table and illustrated on the next page. We would fill in the first row, for the starting DFA state s, as follows:

- The ε-reach of NFA state s is $\{s, a, d\}$, so we write sad in the first column.

- None of the NFA states $\{s, a, d\}$ is an accepting state, so $\{s\}$ is not an accepting state of the DFA, so we do *not* check the second column.

- Next, $\delta'(\{s, a, d\}, 0) = \delta(s, 0) \cup \delta(a, 0) \cup \delta(d, 0) = \{s\} \cup \{b\} \cup \varnothing = \{s, b\}$, so we write sb in the third column. Because sb does not already appear in the first column in any existing row, we have discovered a new DFA state! We start a new row for DFA state sb.

- Finally, $\delta'(\{s, a, d\}, 1) = \delta(s, 1) \cup \delta(a, 1) \cup \delta(d, 1) = \{s\} \cup \varnothing \cup \{e\} = \{s, e\}$, so we write se in the fourth column, and we start a new row for the new DFA state se.

We now have two new rows to fill in, corresponding to states sb and se. The algorithm continues filling in rows (and discovering new rows) until all rows are filled, ending with the following table:

q'	ε-reach(q')	$q' \in A'?$	$\delta'(q', 0)$	$\delta'(q', 1)$
s	sad		sb	se
sb	$sabd$		sbc	se
se	$sade$		sb	sef
sbc	$sabcdg$	✓	$sbcg$	seg
sef	$sacdefg$	✓	sbg	$sefg$
$sbcg$	$sabcdg$	✓	$sbcg$	seg
seg	$sadeg$	✓	sbg	$sefg$
sbg	$sabdg$	✓	$sbcg$	seg
$sefg$	$sacdefg$	✓	sbg	$sefg$

Although it avoids unreachable states, the incremental subset algorithm still gives us a DFA with far more states than necessary, intuitively because it keeps looking for 00 and 11 substrings even after it's already found one. After all, after the NFA finds both 00 and 11 as substrings, it doesn't kill all the other parallel execution threads, because it *can't*. NFAs often have significantly fewer states than equivalent DFAs, but that efficiency also makes them kind of stupid.
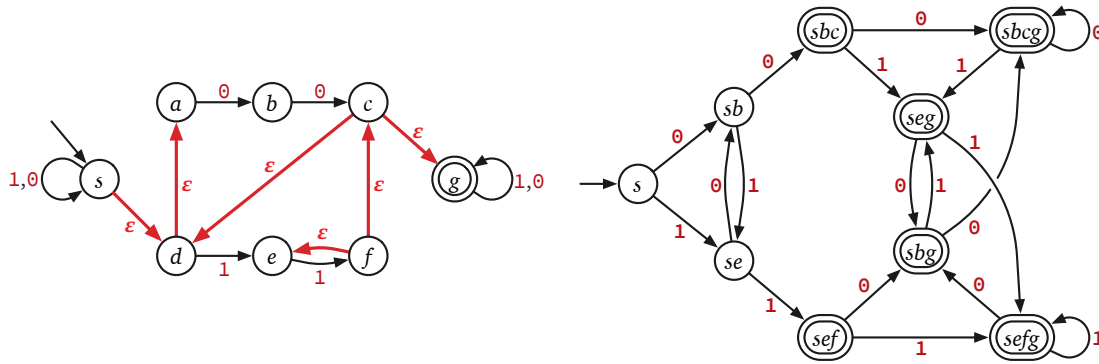
An eight-state NFA with ε-transitions, and the output of the incremental subset construction for that NFA.

4.6 Regular Expression to NFA: Thompson's Algorithm

We now turn to the core of Kleene's theorem, which claims that regular languages (described by regular expressions) and automatic languages (accepted by finite-state automata) are the same.

Lemma 4.2. *Every regular language is accepted by a nondeterministic finite-state automaton.*

Proof: In fact, we will prove the following stronger claim: Every regular language is accepted by an NFA *with exactly one accepting state, which is different from its start state*. The following construction was first described by Ken Thompson in 1968. Thompson's algorithm actually proves a stronger statement: For any regular language L, there is an NFA that accepts L that has exactly one accepting state t, which is distinct from the starting state s.

Let R be an arbitrary regular expression over an arbitrary finite alphabet Σ. Assume that for any sub-expression S of R, the language described by S is accepted by an NFA with one accepting state distinct from its start state, which we denote pictorially by ⭢◯ s ◎. There are six cases to consider—three base cases and three recursive cases—mirroring the recursive definition of a regular expression.

- If $R = \varnothing$, then $L(R) = \varnothing$ is accepted by the trivial NFA ⭢◯   ◎.

- If $R = \varepsilon$, then $L(R) = \{\varepsilon\}$ is accepted by a different trivial NFA ⭢◯⭢$^\varepsilon$◎.

- If $R = w$ for some non-empty string $w \in \Sigma^+$, then $L(R) = \{w\}$ is accepted by an NFA consisting of a single chain of transitions from s to t, labeled by the symbols in w in order. For example, the language $\{101\}$ is accepted by the NFA ⭢◯⭢^{1}◯⭢^{0}◯⭢^{1}◎.

- Suppose $R = ST$ for some regular expressions S and T. The inductive hypothesis implies that the languages $L(S)$ and $L(T)$ are accepted by NFAs ⭢◯ s ◎ and ⭢◯ T ◎, respectively. Then $L(R) = L(ST) = L(S) \bullet L(T)$ is accepted by the NFA ⭢◯ s ◯⭢$^\varepsilon$◯ T ◎, built by connecting the two component NFAs in series.

- Suppose $R = S + T$ for some regular expressions S and T. The inductive hypothesis implies that the languages $L(S)$ and $L(T)$ are accepted by NFAs ⭢◯ s ◎ and ⭢◯ T ◎, respectively. Then $L(R) = L(ST) = L(S) \bullet L(T)$ is accepted by the NFA ⭢◯ $^\varepsilon$ ◯ S ◯ $^\varepsilon$ ◎, built by connecting the two component NFAs in parallel with new start and accept states.

8

- Finally, suppose $R = S^*$ for some regular expression S. The inductive hypothesis implies that the languages $L(S)$ is accepted by an NFA $\rightarrow\!\circ\,\boxed{s}\,\circledcirc$. Then the language $L(R) = L(S^*) = L(S)^*$

  is accepted by the NFA $\rightarrow\!\circ\rightarrow\circ\,\boxed{s}\,\circ\rightarrow\circledcirc$.

In all cases, the language $L(R)$ is accepted by an NFA with one accepting state, which is different from its start state, as claimed.   □

As an example, given the regular expression $(0+10^*1)^*$ of strings containing an even number of 1s, Thompson's algorithm produces a 14-state NFA shown on the next page. As this example shows, Thompson's algorithm tends to produce NFAs with many redundant states. Fortunately, just as there are for DFAs, there are algorithms that can reduce any NFA to an equivalent NFA with the smallest possible number of states.
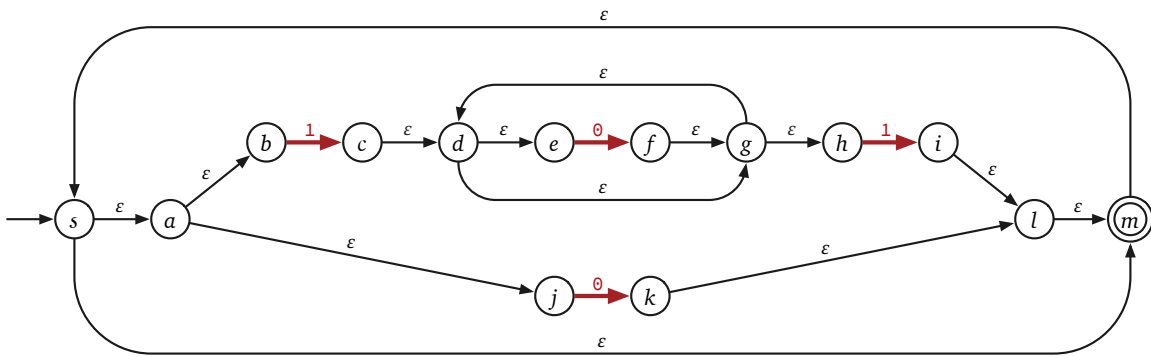
The NFA constructed by Thompson's algorithm for the regular expression $(0+10^*1)^*$.
The four non-ε-transitions are drawn with with bold red arrows for emphasis.

Interestingly, applying the incremental subset algorithm to Thompson's NFA tends to yield a DFA with relatively *few* states, in part because the states in Thompson's NFA tend to have large ε-reach, and in part because relatively few of those states are the targets of non-ε-transitions. Starting with the Thompson's NFA for $(0+10^*1)^*$, for example, the incremental subset construction yields a DFA with just five states.

| q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	$sabjm$	✓	k	c
k	$sabjklm$	✓	k	c
c	$cdegh$		f	i
f	$defgh$		f	i
i	$sabjilm$	✓	k	c
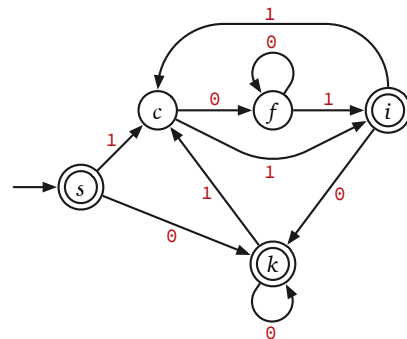
The DFA computed by the incremental subset algorithm from Thompson's NFA for $(0+10^*1)^*$.

This DFA can be further simplified to just two states, by observing that all three accepting states are equivalent, and that both non-accepting states are equivalent. But still, five states is pretty good, especially compared with the $2^{14} = 16384$ states that the naïve subset construction would yield!

4.7 Another Example

Here is another example of all the algorithms we've seen so far, starting with the regular expression $(0 + 1)^*(00 + 11)(0 + 1)^*$, which describes the language accepted by our very first example NFA. Thompson's algorithm constructs the following 26-state monster:
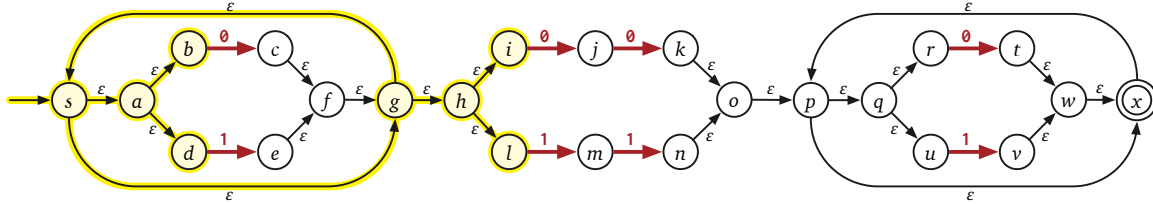
Thompson's NFA for the regular expression $(0 + 1)^*(00 + 11)(0 + 1)^*$,
with the ε-reach of the start state s highlighted.

Given this NFA as input, the incremental subset construction computes the following table, leading to a DFA with just nine states. Yeah, the ε-reaches get a bit ridiculous; unfortunately, this is typical for NFAs produced by Thompson's algorithm, which have lots of ε-transitions. As usual, the resulting DFA still has far more states than necessary, but *significantly* less than the $2^{24} = 16\,777\,216$ states that the naïve subset construction would give us.

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	$sabdghil$		cj	em
cj	$sabcdfghijl$		cjk	em
em	$sabdefghilm$		emn	cj
cjk	$sabcdfghijklopqrux$	✓	$cjkt$	emv
emn	$sabdefghilmnopqrux$	✓	cjt	$emnv$
$cjkt$	$sabcdfghijklopqrtux$	✓	$cjkt$	emv
emv	$sabdefghilmpqruvwx$	✓	cjt	$emnv$
cjt	$sabcdfghijlpqrtuwx$	✓	$cjkt$	emv
$emnv$	$sabdefghilmnpqruvwx$	✓	cjt	$emnv$
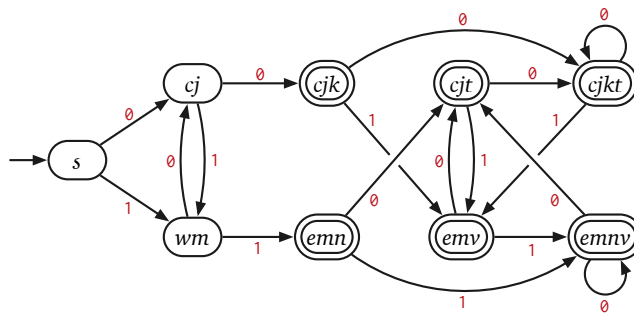
The DFA computed by the incremental subset algorithm from Thompson's NFA for $(0 + 1)^*(00 + 11)(0 + 1)^*$.

Finally, the DFA-minimization algorithm from the previous lecture note correctly discovers that all six accepting states of the incremental-subset DFA are equivalent, and thus reduces the DFA to just four states, as shown on the next page.
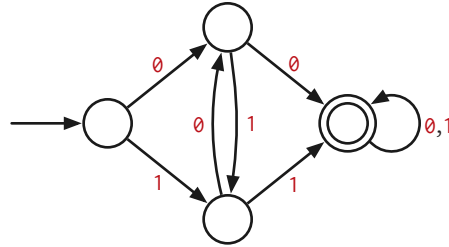
The minimal DFA that accepts the language $(0 + 1)^*(00 + 11)(0 + 1)^*$.

⋆4.8   NFA to Regular Expression: State Elimination

The only component of Kleene's theorem left to prove is that every language accepted by an NFA is regular. I'll describe a proof that is morally equivalent to Kleene's 1951 argument, but uses more modern standard notation. This formulation of the proof was first published in 1960 by Robert McNaughton and Hisao Yamada[3]; a similar formulation (with much uglier notation) was published by C. Y. Lee earlier in the same year.[4]

Recall that a standard NFA can be represented by a state-transition graph, whose vertices are the states and whose edges represent possible transitions. Each edge is labeled with a single symbol in Σ. A string $w \in \Sigma^*$ is accepted if and only if there is a sequence of transitions

$$s \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_\ell} q_\ell$$

where the final state q_ℓ is accepting and $a_1 a_2 \cdots a_\ell = w$.

We've already seen that NFAs can be generalized to include ε-transitions; we can push this generalization further. A **string NFA** allows each transition $p{\to}q$ to be labeled with an *arbitrary string* $x(p{\to}q) \in \Sigma^*$. We are allowed to transition from state p to state q if the label $x(p{\to}q)$ is a *prefix* of the remaining input. Thus, a string $w \in \Sigma^*$ is accepted if and only if there is a sequence of transitions

$$s \xrightarrow{x_1} q_1 \xrightarrow{x_2} q_2 \xrightarrow{x_3} \cdots \xrightarrow{x_\ell} q_\ell$$

where the final state q_ℓ is accepting, and $x_1 \bullet x_2 \bullet \cdots \bullet x_\ell = w$. Thus, an NFA with ε-transitions is just a string NFA where every label has length 0 or 1. Any string NFA can be converted into an equivalent standard NFA, by subdividing each edge $p{\to}q$ into a path of length $|x(p{\to}q)|$ (unless $x(p{\to}q) = \varepsilon$).

Finally, McNaughton and Yamada define an **expression automaton** as a finite-state machine where each transition $p{\to}q$ is labeled with an arbitrary *regular expression* $R(p{\to}q)$. We can transition from state p to state q if *any* prefix of the remaining input matches the regular expression $R(p{\to}q)$. Thus, a string $w \in \Sigma^*$ is accepted by an expression automaton if and only if there is a sequence of transitions

$$s \xrightarrow{R_1} q_1 \xrightarrow{R_2} q_2 \xrightarrow{R_3} \cdots \xrightarrow{R_\ell} q_\ell$$

where the final state q_ℓ is accepting, and we can write $w = x_1 \bullet x_2 \bullet \cdots \bullet x_\ell = w$, where each substring x_i matches the corresponding regular expression R_i.

More formally, an expression automaton consists of the following five components:

- A finite set Σ called the **input alphabet**

---

[3]Robert McNaughton and Hisao Yamada. Regular Expressions and State Graphs for Automata. *IRE Trans. Electron. Comput.* 9(1): 39-47, 1960.

[4]C. Y. Lee. Automata and finite automata. *Bell System Technical Journal* 39(2), 1267–1295, 1960.

- Another finite set Q whose elements are called **states**

- A unique **start state** $s \in Q$

- A unique **target state** $t \in Q \setminus \{s\}$

- A **transition function** $R: Q \times Q \rightarrow Reg(\Sigma)$, where $Reg(\Sigma)$ is the set of regular expressions over Σ.

The requirement that the target state is unique and distinct from s is not essential to the model, but imposed only to simplify the equivalence proof. Similarly, we assume for convenience that $R(q \rightarrow s) = R(t \rightarrow q) = \varnothing$ for every state q. Both of these assumptions can be enforced if necessary by adding artificial start and target states connected to other states by ε-transitions.

Expression automata are even more nondeterministic than NFAs. A single string could match several (even infinitely many) transition sequences from s to t, and it could match each of those sequences in several (even infinitely many) different ways. A string w is accepted if *any* decomposition of w into a sequence of substrings matches *any* sequence of transitions from s to t. Conversely, a string might match *no* state sequences, in which case the string is rejected.

Two extreme special cases of expression automata are already familiar. First, every regular language is clearly the language of an expression automaton with exactly two states. Second, with only minor modifications, any DFA or NFA can be converted into an expression automaton with trivial transition expressions. Thompson's algorithm can be used to transform any expression automaton into a standard NFA (with ε-transitions), by recursively expanding any nontrivial transition expression. To complete the proof of Kleene's theorem, we show how to convert an arbitrary expression automaton into a regular expression, by repeatedly deleting vertices.
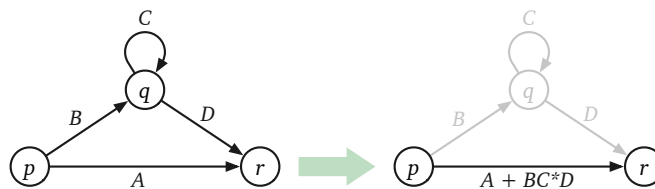
Lemma 4.3. *Every expression automaton accepts a regular language.*

Proof: Let $E = (Q, \Sigma, R, s, t)$ be an arbitrary expression automaton. Assume that any expression automaton with fewer states than E accepts a regular language. There are two cases to consider, depending on the number of states in Q:

- If $Q = \{s, t\}$, then E trivially accepts the regular language $R(s \rightarrow t)$.

- On the other hand, suppose Q has more than two states; fix an arbitrary state $q \in Q \setminus \{s, t\}$. We modify the automaton, without changing its language, so that state q is redundant and can be removed. Define a new transition function $R': Q \times Q \rightarrow Reg(\Sigma)$ by setting

$$R'(p \rightarrow r) := R(p \rightarrow r) + R(p \rightarrow q) R(q \rightarrow q)^* R(q \rightarrow r).$$

With this modified transition function in place, any string w that matches the sequence $p \rightarrow q \rightarrow q \rightarrow \cdots \rightarrow q \rightarrow r$, with any number of q's, also matches the single transition $p \rightarrow r$. Thus, by induction, if w matches a sequence of states, it also matches the subsequence obtained by removing all q's.


One step in Kleene's/Lee's/McNaughton and Yamada's reduction algorithm.

Let E' be the expression automaton with states $Q' = Q \setminus \{q\}$ that uses this modified transition function R'. This new automaton accepts exactly the same strings as the original automaton E. Because E' has fewer states than E, the inductive hypothesis implies E' accepts a regular language.

In both cases, we conclude that E accepts a regular language.   □

This proof can be mechanically translated into an algorithm to convert any DFA or NFA into an equivalent regular expression, via a sequence of expression automata with fewer and fewer states, but increasingly complex transition expressions.

The figure on the next page shows McNaughton and Yamada's reduction algorithm in action, starting with a DFA that accepts the binary representations of non-negative integers divisible by 3, possibly with extra leading 0s. (State i means the binary number we've read so far is congruent to i mod 3.) We add an artificial target state t with an ε-transition from the unique accepting state 0. (We don't need an artificial start state, because s already has no incoming transitions.) We then remove the numbered states one by one, updating the transition expressions between any remaining states at each iteration. For the sake of clarity, the figure omits all edges $p \rightarrow q$ where $R(p \rightarrow q) = \varnothing$. The final regular expression $00^* + (00^*1 + 1)(10^*1 + 01^*0)^*10^*$ can be slightly simplified to $0^*0 + 0^*1(10^*1 + 01^*0)^*10^*$, which is precisely the regular expression we gave for this language back in Lecture Note 2!
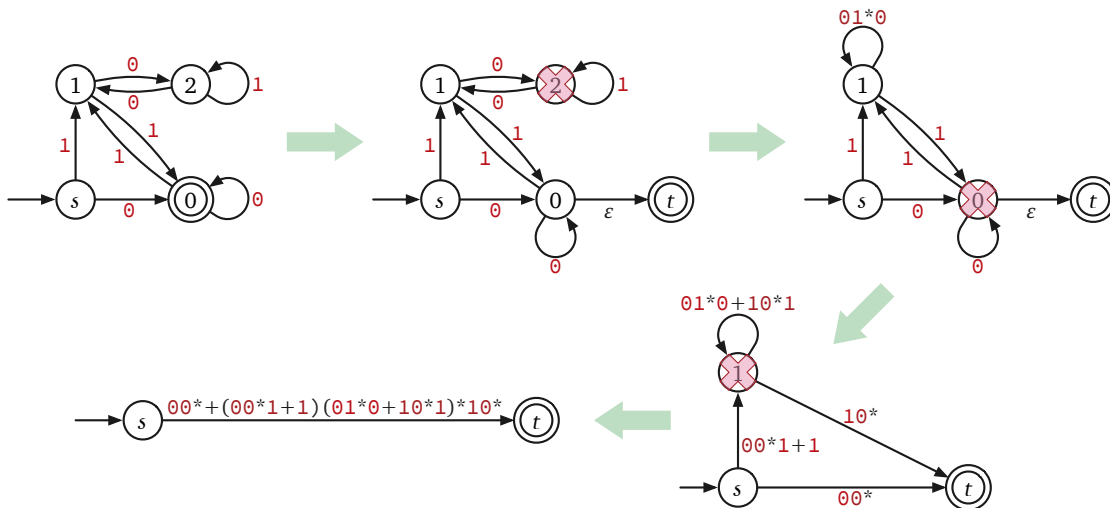
Converting a DFA into an equivalent regular expression using McNaughton and Yamada's algorithm.

Given an NFA with n states (including s and t), McNaughton and Yamada's algorithm iteratively removes $n - 2$ states, updating $O(n^2)$ transition expressions in each iteration. If the concatenation and Kleene star operations could be performed in constant time, the resulting algorithm would run in $O(n^3)$ time. However, in the worst case, the transition expressions grows in length by roughly a factor of 4 in each iteration, so the final expression has length $\Theta(4^n)$. If we insist on representing the expressions as explicit strings, the worst-case running time is actually $\Theta(4^n)$.

4.9 Regular Language Transformations

We have already seen that many functions of regular languages are themselves regular: unions, concatenations, and Kleene closure by definition. and intersections and differences by product

construction on DFA. However, the set of regular languages is closed under a much richer class of functions.

Suppose we wanted to prove that regular languages are closed under some function f; that is, for every regular language L, we want to prove that the language $f(L)$ is also regular. There are two general techniques to prove such a statement:

- Describe an algorithm that transforms an arbitrary regular expression R into a new regular expression R' such that $L(R') = f(L(R))$.

- Describe an algorithm that transforms an arbitrary DFA M into a new NFA M' such that $L(M') = f(L(M))$.

The equivalence between regular expressions and finite automata implies that *in principle* we can always use either technique, but in practice, the second one is far more powerful and usually simpler. The asymmetry in the second technique is important. We start with a DFA for L to impose as much structure as possible in the input; we aim for an NFA with ε-transitions to give ourselves as much freedom as possible in the output.[5]

For our first example, I'll describe proofs using both techniques.

Lemma 4.4. *For any regular language L, the language $L^R = \{w^R \mid w \in L\}$ is also regular.*

Proof (regular expression to regular expression): Let R be an arbitrary regular expression such that $L = L(R)$. Assume for any proper subexpression S of R that $L(S)^R$ is regular. There are five cases to consider, mirroring the recursive definition of regular expressions:

- If $R = \varnothing$, then $L^R = L = \varnothing$, so $L(R) = L^R$.

- Suppose R consists of a single word w. Let $R' = w^R$. Then $L(R') = \{w^R\} = L^R$.

- Suppose $R = A + B$. The inductive hypothesis implies that there are regular expressions A' and B' such that $L(A') = L(A)^R$ and $L(B') = L(B)^R$. Let $R' = A' + B'$. Then $L(R') = L(A') \cup L(B') = L(A)^R \cup L(B)^R = (L(A) \cup L(B))^R = L^R$.

- Suppose $R = A \bullet B$. The inductive hypothesis implies that there are regular expressions A' and B' such that $L(A') = L(A)^R$ and $L(B') = L(B)^R$. Let $R' = B' \bullet A'$. Then $L(R') = L(B') \bullet L(A') = L(B)^R \bullet L(A)^R = (L(A) \bullet L(B))^R = L^R$.

- Finally, suppose $R = A^*$. The inductive hypothesis implies that there is a regular expression A' such that $L(A') = L(A)^R$. Let $R' = (A')^*$. Then $L(R') = L(A')^* = (L(A)^R)^* = (L(A)^*)^R = L^R$.

In all cases, we have constructed a regular expression R' such that $L(R') = L^R$. We conclude that L^R is regular. $\qquad\square$

For example, if L is described by the regular expression $(10 + 1)^*(01 + 100)$, then L^R is described by the regular expression $(10 + 001)(0 + 01)^*$.

Careful readers may be unsatisfied with the previous proof, because it assumes several "obvious" properties of string and language reversal. Specifically, for all strings x and y and all languages L and L', we assumed the following identities:

- $(x \bullet y)^R = y^R \bullet x^R$

---

[5] We could give ourselves even more freedom by constructing an *expression* automaton, but creativity thrives on constraint.

- $(L \cdot L')^R = (L')^R \cdot L^R$.
- $(L \cup L')^R = L^R \cup (L')^R$.
- $(L^*)^R = (L^R)^*$.

All of these claims are all easy to prove by inductive definition-chasing.

However, in general, it is much easier to argue regularity of the transformed language by constructing an NFA.

Proof (DFA to NFA): Let $M = (\Sigma, Q, s, A, \delta)$ be an arbitrary DFA that accepts L. We construct an NFA $M^R = (\Sigma, Q^R, s^R, A^R, \delta^R)$ with ε-transitions that accepts L^R, intuitively by reversing every transition in M, and swapping the roles of the start state and the accepting states. Because M does not have a unique accepting state, we introduce a new start state s^R, with ε-transitions to each accepting state in M. These are the only ε-transitions in M^R.

$$Q^R = Q \cup \{s^R\}$$
$$s^R \text{ is a new state}$$
$$A^R = \{s\}$$
$$\delta^R(s^R, \varepsilon) = A$$
$$\delta^R(s^R, a) = \varnothing \qquad\qquad \text{for all } a \in \Sigma$$
$$\delta^R(q, \varepsilon) = \varnothing \qquad\qquad \text{for all } q \in Q$$
$$\delta^R(q, a) = \{p \mid q \in \delta(p, a)\} \qquad\qquad \text{for all } q \in Q \text{ and } a \in \Sigma$$

Routine inductive definition-chasing now implies that the reversal of any sequence $s \to q_1 \to \cdots \to q_\ell$ of transitions in M is a valid sequence $q_\ell \to \cdots \to q_1 \to s$ of transitions in M^R. Because the transitions retain their labels (but reverse directions), it follows that M accepts any string w if and only if M^R accepts w^R.

We conclude that the NFA M^R accepts L^R, and therefore L^R must be regular. $\qquad\square$

The following figure shows an example of this transformation in action. On the left is a DFA that accepts binary strings whose last two symbols are equal. Applying our reversal transformation yields the NFA in the middle. Finally, applying the incremental subset construction to this reversed NFA yields the DFA on the right. The final DFA clearly accepts binary strings whose *first* two symbols are equal, and therefore so does the equivalent middle NFA.
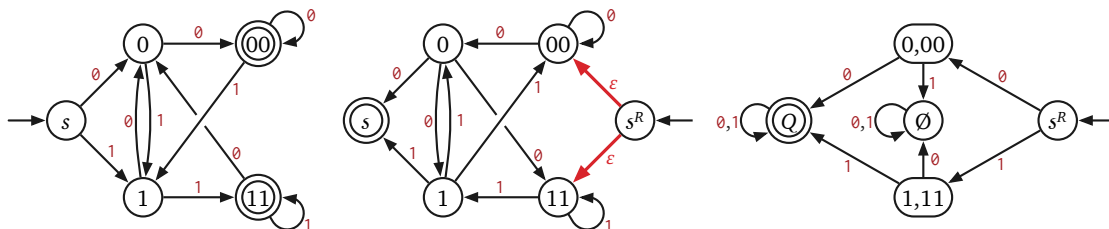
Reversing a DFA, and then transforming the resulting NFA into an equivalent DFA.

4.9.1 A More Interesting Example

Lemma 4.5. *For any regular language L, the language $palin(L) := \{w \mid ww^R \in L\}$ is also regular.*

The language *palin(L)* contains the first half of every even-length palindrome in L. For example, if $L = \{\varepsilon, \text{AHA}, \text{POOP}, \text{TACOCAT}, \text{PALINDROME}\}$, then *palin(L)* $= \{\varepsilon, \text{PO}\}$.

As a more interesting example, suppose L is the language $(0+1)^*(00+11)$ of strings whose last two symbols are equal. Every palindrome in L either starts and ends with 00, or starts and ends with 11. Thus, *palin(L)* is actually the language $0 + 1 + (00+11)(0+1)^*$ of strings either with length one or whose first two symbols are equal.

Proof: Let $M = (\Sigma, Q, s, A, \delta)$ be an arbitrary DFA that accepts L.

Intuitively, we construct an NFA M' that reads its input string w and simulates the original DFA M reading the input string ww^R. The NFA M' is a product construction of the original DFA M and its reversal NFA M^R constructed in our second proof of Lemma 4.4. Just like any other product construction, we must explicitly specify the accepting states to complete the description of M'.

Let $q = \delta^*(s, w)$ be the state that M reaches on reading input string w. Then M accepts the palindrome ww^R if and only if

$$\delta^*(s, ww^R) \;=\; \delta^*(\delta^*(s, w), w^R) \;=\; \delta^*(q, w^R) \;\in\; A.$$

But $\delta^*(q, w^R) \in A$ if and only if the reversed machine M^R reaches state q on reading the same input string w. Thus, our new NFA M' should accept w if and only if both M and M^R end at the same state. In other words, we need $A' = \{(q, q) \mid q \in Q\}$.

Summing up, our new NFA $M' = (\Sigma, Q', s', A', \delta')$ is formally defined as follows. All unspecified transitions go to $\varnothing$.

$$
\begin{aligned}
Q' &= Q \times (Q \cup \{s^R\}) && \text{where } s^R \text{ is a new state}\\
s &= (s, s^R)\\
A' &= \{(q, q) \mid q \in Q\}\\[4pt]
\delta'((s, s^R), \varepsilon) &= \big\{(s, q) \;\big|\; q \in A\big\}\\
\delta'((q, r), a) &= \big\{(\delta(q, a), p) \;\big|\; q \in \delta(p, a)\big\} && \text{for all } q, r \in Q \text{ and } a \in \Sigma \qquad \square
\end{aligned}
$$

The NFAs constructed by this algorithm can be quite large. For example, if M is the five-state DFA for the language $L = (0+1)^*(00+11)$ that we used to illustrate the reversal transformation, our algorithm constructs an NFA M' for *palin(L)* with 30 states, which is far too complex to draw. It turns out that 11 of those 30 states either cannot be reached from the start state (s, S^R) or cannot reach an accepting state (q, q); more significantly, the language *palin(L)* $= 0+1+(00+11)(0+1)^*$ can actually be decided by a five-state DFA! But that extra complexity from our algorithm is completely okay—we're not even trying to construct an *efficient* machine; we just need to prove than one NFA exists.

4.9.2 A Harder Example

In the next example, we make more fundamental use of nondeterminism to *guess* the future behavior of the DFA we are simulating.

Lemma 4.6. *For any regular language L, the language half(L) := \{w \mid ww \in L\}$ is also regular.*

Proof: Let $M = (\Sigma, Q, s, A, \delta)$ be an arbitrary DFA that accepts L.

Intuitively, we construct an NFA M' that reads its input string w and simulates the original DFA M reading the input string ww. Our overall strategy has three parts:

- First M' non-deterministically *guesses* the state $h = \delta^*(s, w)$ that M will reach after reading input w. (We can't just run M on input w to compute the correct state h, because that would consume the input string!)

- Then M' runs two copies of M in parallel (using a product construction): a "left" copy starting at s and a "right" copy starting at the (guessed) halfway state h.

- Finally, when M' is done reading w, it accepts if and only if the first copy of M actually stopped in state h (so our initial guess was correct) and the second copy of M stopped in an accepting state. That is, M' accepts if and only if $\delta^*(s, w) = h$ and $\delta^*(h, w) \in A$.

To implement this strategy, M' needs to maintain *three* states of M: the state of the left copy of M, the guess h for the halfway state, and the state of the right copy of M. The first and third states evolve according to the transition function δ, but the second state never changes. Finally, to implement the non-deterministic guessing, M' includes a special start state s' with ε-transitions to every triple of the form (s, h, h).

Summing up, our new NFA $M' = (\Sigma, Q', s', A', \delta')$ is formally defined as follows.

$$Q' = (Q \times Q \times Q) \cup \{s'\}$$
$$A' = \left\{(h, h, q) \mid h \in Q \text{ and } q \in A\right\}$$
$$\delta'(s', \varepsilon) = \left\{(s, h, h) \mid h \in Q\right\}$$
$$\delta'(s', a) = \varnothing \qquad\qquad\qquad \text{for all } a \in \Sigma$$
$$\delta'((p, h, q), \varepsilon) = \varnothing \qquad\qquad\qquad \text{for all } p, h, q \in Q$$
$$\delta'((p, h, q), a) = \left\{(\delta(p, a), h, \delta(q, a))\right\} \qquad \text{for all } p, h, q \in Q \text{ and } a \in \Sigma \qquad \square$$

Exercises

1. For each of the following regular expressions, describe or draw two finite-state machines:

   - An NFA that accepts the same language, constructed using Thompson's algorithm.
   - An equivalent DFA, built from the previous NFA using the incremental subset construction. For each state in your DFA, identify the corresponding subset of states in your NFA. Your DFA should have no unreachable states.

   (a) $(01 + 10)^*(0 + 1 + \varepsilon)$
   (b) $(\varepsilon + 1)(01)^*(\varepsilon + 0)$
   (c) $1^* + (10)^* + (100)^*$
   (d) $(\varepsilon + 0 + 00)(1 + 10 + 100)^*$
   (e) $((0 + 1)(0 + 1))^*$
   (f) $\varepsilon + 0(0 + 1)^* + 1(1 + 0)^*$

2. The accepting language of an NFA $M = (\Sigma, Q, s, A, \delta)$ is defined as follows:
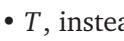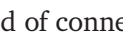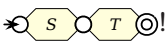
   $$L(M) := \left\{w \in \Sigma^* \mid \delta^*(s, w) \cap A \neq \varnothing\right\}.$$

   Kleene's theorem (described here as McNaughton and Yamada's state-reduction algorithm) implies that $L(M)$ is regular. Prove that the following languages associated with M are also regular:

(a) $L^\forall(M) := \{w \in \Sigma^* \mid A \subseteq \delta^*(s, w)\}$. That is, a string w is in the language $L^\forall(M)$ if and only if $\delta^*(s, w)$ contains *every* accepting states.

(b) $L^\subseteq(M) := \{w \in \Sigma^* \mid \delta^*(s, w) \subseteq A\}$. That is, a string w is in the language $L^\subseteq(M)$ if and only if $\delta^*(s, w)$ contains *only* accepting state.

(c) $L^\equiv(M) := \{w \in \Sigma^* \mid \delta^*(s, w) = A\}$. That is, a string w is in the language $L^=(M)$ if and only if $\delta^*(s, w)$ is exactly the set of accepting states.

[Hint: These are easier than they look!]

3. A certain professor who really should know better once woke up in the middle of the night with a startling revelation—Thompson's algorithm doesn't need all those ε-transitions! Filled with the certainty that only sleep deprivation can bring, he ran to his laptop and quickly changed two cases in his description of Thompson's algorithm.

   - When $R = S \bullet T$, instead of connecting the accept state of ⊸⬡ s ◎ to the start state of ⊸⬡ T ◎ with an ε-transition, we can **just** identify those two states to build the simpler NFA ⊸⬡ s ⬡ T ◎!

   - When $R = S^*$, instead of introducing two new states and four ε-transitions, we can **just** add two ε-transitions between the start and accept states of ⊸⬡ s ◎ to build the simpler NFA ⊸⬡ s ◎!

   Satisfied with his simplification, he thanked the penguin who gave him the idea, and then flew his hat back into the ocean marshmallows, where a giant man with the head of a dog gave him the power of bread.[6] The next morning, while he was proudly teaching his new simplified proof for the first time, he realized his embarrassing mistake.

   Prove that *neither* of the professor's optimizations is actually correct.

   (a) Find a regular expression R, such that the NFA constructed from R by Thompson's algorithm **with only the first modification** accepts strings that are not in $L(R)$.

   (b) Find a regular expression R, such that the NFA constructed from R by Thompson's algorithm **with only the second modification** accepts strings that are not in $L(R)$.

   The moral of this story: Whenever you use the word "***just***", you're probably missing something important.

4. A **Moore machine** is a variant of a finite-state automaton that produces output; Moore machines are sometimes called finite-state *transducers*. For purposes of this problem, a Moore machine formally consists of six components:

   - A finite set Σ called the input alphabet
   - A finite set Γ called the output alphabet
   - A finite set Q whose elements are called states

---

[6]Eid ma clack shaw supoven del ba. Mertepy ven seinur cofally ragdah.

- A start state $s \in Q$
- A transition function $\delta \colon Q \times \Sigma \to Q$
- An output function $\omega \colon Q \to \Gamma$

More intuitively, a Moore machine is a graph with a special start vertex, where every node (state) has one outgoing edge labeled with each symbol from the input alphabet, and each node (state) is additionally labeled with a symbol from the output alphabet.

The Moore machine reads an input string $w \in \Sigma^*$ one symbol at a time. For each symbol, the machine changes its state according to the transition function δ, and then outputs the symbol $\omega(q)$, where q is the new state. Formally, we recursively define a *transducer* function $\omega^* \colon \Sigma^* \times Q \to \Gamma^*$ as follows:

$$\omega^*(w,q) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \omega(\delta(a,q)) \cdot \omega^*(x, \delta(a,q)) & \text{if } w = ax \end{cases}$$

Given the input string $w \in \Sigma^*$, the machine outputs the string $\omega^*(w,s) \in \Gamma^*$. To simplify notation, we define $M(w) = \omega^*(w,s)$.

Finally, the **output language $L^\circ(M)$** of a Moore machine M is the set of all strings that the machine can output:
$$L^\circ(M) := \big\{ M(w) \mid w \in \Sigma^* \big\}$$

(a) Let M be an arbitrary Moore machine. Prove that $L^\circ(M)$ is a regular language.

(b) Let M be an arbitrary Moore machine whose input alphabet Σ and output alphabet Γ are identical. Prove that the language

$$L^=(M) = \big\{ w \in \Sigma^* \mid M(w) = w \big\}$$

is regular. Strings in $L^=(M)$ are also called *fixed points* of the function $M \colon \Sigma^* \to \Sigma^*$.

*(c) As in part (b), let M be an arbitrary Moore machine whose input and output alphabets are identical. Prove that the language $\big\{ w \in \Sigma^* \mid M(M(w)) = w \big\}$ is regular.

[Hint: Parts (a) and (b) are easier than they look!]

5. A **Mealy machine** is a variant of a finite-state automaton that produces output; Mealy machines are sometimes called finite-state *transducers*. For purposes of this problem, a Mealy machine formally consists of six components:

- A finite set Σ called the input alphabet
- A finite set Γ called the output alphabet
- A finite set Q whose elements are called states
- A start state $s \in Q$
- A transition function $\delta \colon Q \times \Sigma \to Q$
- An output function $\omega \colon Q \times \Sigma \to \Gamma$

More intuitively, a Mealy machine is a graph with a special start vertex, where every node (state) has one outgoing edge labeled with each symbol from the input alphabet, and each edge (transition) is additionally labeled with a symbol from the output alphabet. (Mealy machines are closely related to *Moore* machines, which produce output at each *state* instead of at each transition.)

The Mealy machine reads an input string $w \in \Sigma^*$ one symbol at a time. For each symbol, the machine changes its state according to the transition function δ, and simultaneously outputs a symbol according the output function ω. Formally, we recursively define a *transducer* function $\omega^* \colon Q \times \Sigma^* \to \Gamma^*$ as follows:

$$\omega^*(q, w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \omega(q, a) \cdot \omega^*(\delta(q, a), x) & \text{if } w = ax \end{cases}$$

Given any input string $w \in \Sigma^*$, the machine outputs the string $\omega^*(w, s) \in \Gamma^*$. To simplify notation, we define $M(w) = \omega^*(w, s)$.

Finally, the **output language $L^\circ(M)$** of a Mealy machine M is the set of all strings that the machine can output:

$$L^\circ(M) := \{M(w) \mid w \in \Sigma^*\}$$

(a) Let M be an arbitrary Mealy machine. Prove that $L^\circ(M)$ is a regular language.

(b) Let M be an arbitrary Mealy machine whose input alphabet Σ and output alphabet Γ are identical. Prove that the language

$$L^=(M) = \{w \in \Sigma^* \mid w = \omega^*(s, w)\}$$

is regular. $L^=(M)$ consists of all strings w such that M outputs w when given input w; these are also called *fixed points* for the transducer function ω^*.

*(c) As in part (b), let M be an arbitrary Mealy machine whose input and output alphabets are identical. Prove that the language $\{w \in \Sigma^* \mid M(M(w)) = w\}$ is regular.

[Hint: Parts (a) and (b) are easier than they look!]

6. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular. Assume $\# \in \Sigma$.

(a) $censor(L) := \{\#^{|w|} \mid w \in L\}$

(b) $dehash(L) = \{dehash(w) \mid w \in L\}$, where $dehash(w)$ is the subsequence of w obtained by deleting every $\#$.

(c) $insert\#(L) := \{x\#y \mid xy \in L\}$.

(d) $delete\#(L) := \{xy \mid x\#y \in L\}$.

(e) $prefix(L) := \{x \in \Sigma^* \mid xy \in L \text{ for some } y \in \Sigma^*\}$

(f) $suffix(L) := \{y \in \Sigma^* \mid xy \in L \text{ for some } x \in \Sigma^*\}$

(g) $substring(L) := \{y \in \Sigma^* \mid xyz \in L \text{ for some } x, z \in \Sigma^*\}$

(h) $superstring(L) := \{xyz \mid y \in L \text{ and } x, z \in \Sigma^*\}$

(i) $cycle(L) := \{xy \mid x, y \in \Sigma^* \text{ and } yx \in L\}$

(j) $prefmax(L) := \{x \in L \mid xy \in L \iff y = \varepsilon\}.$

(k) $sufmin(L) := \{xy \in L \mid y \in L \iff x = \varepsilon\}.$

(l) $minimal(L) := \{w \in L \mid \text{no proper substring of } w \text{ is in } L\}.$

(m) $maximal(L) := \{w \in L \mid \text{no proper superstring of } w \text{ is in } L\}.$

(n) $evens(L) := \{evens(w) \mid w \in L\}$, where $even(w)$ is the subsequence of w containing every even-indexed symbol. For example, $evens(\text{EVENINDEX}) = \text{VNNE}$.

(o) $evens^{-1}(L) := \{w \in \Sigma^* \mid evens(w) \in L\}.$

(p) $subseq(L) := \{x \in \Sigma^* \mid x \text{ is a subsequence of some } y \in L\}$

(q) $superseq(L) := \{x \in \Sigma^* \mid \text{some } y \in L \text{ is a subsequence of } x\}$

(r) $swap(L) := \{swap(w) \mid w \in L\}$, where $swap(w)$ is defined recursively as follows:

$$swap(w) = \begin{cases} w & \text{if } |w| \le 1 \\ ba \bullet swap(x) & \text{if } w = abx \text{ for some } a, b \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

(s) $oneswap(L) := \{xbay \mid xaby \in L \text{ where } a, b \in \Sigma \text{ and } x, y \in \Sigma^*\}.$

(t) $left(L) := \{x \in \Sigma^* \mid xy \in L \text{ for some } y \in \Sigma^* \text{ where } |x| = |y|\}$

(u) $right(L) := \{y \in \Sigma^* \mid xy \in L \text{ for some } x \in \Sigma^* \text{ where } |x| = |y|\}$

(v) $middle(L) := \{y \in \Sigma^* \mid xyz \in L \text{ for some } x, z \in \Sigma^* \text{ where } |x| = |y| = |z|\}$

(w) $halfseq(L) := \{w \in \Sigma^* \mid w \text{ is a subsequence of some string } x \in L \text{ where } |x| = 2 \cdot |w|\}$

(x) $third(L) := \{w \in \Sigma^* \mid www \in L\}$

(y) $drome(L) := \{w \in \Sigma^* \mid w^R w \in L\}$

7. Let L and L' be arbitrary regular languages over the alphabet $\{0, 1\}$. Prove that the following languages are also regular:

(a) $L \sqcap L' := \{x \sqcap y \mid x \in L \text{ and } y \in L' \text{ and } |x| = |y|\}$, where $x \sqcap y$ denotes bitwise-and. For example, $0011 \sqcap 0101 = 0001$.

(b) $L \sqcup L' := \{x \sqcup y \mid x \in L \text{ and } y \in L' \text{ with } |x| = |y|\}$, where $x \sqcup y$ denotes bitwise-or. For example, $0011 \sqcup 0101 = 0111$.

(c) $L \boxplus L' := \{x \boxplus y \mid x \in L \text{ and } y \in L' \text{ with } |x| = |y|\}$, where $x \boxplus y$ denotes bitwise-exclusive-or. For example, $0011 \boxplus 0101 = 0110$.

(d) $faro(L, L') := \{faro(x, z) \mid x \in L \text{ and } z \in L' \text{ with } |x| = |z|\}$, where

$$faro(x, z) := \begin{cases} z & \text{if } x = \varepsilon \\ a \cdot faro(z, y) & \text{if } x = ay \end{cases}$$

For example, $faro(0011, 0101) = 00011011$.

(e) $shuffles(L, L') := \bigcup_{w \in L, y \in L'} shuffles(w, y)$, where $shuffles(w, y)$ is the set of all strings obtained by shuffling w and y, or equivalently, all strings in which w and y are complementary subsequences. Formally:

$$shuffles(w, y) = \begin{cases} \{y\} & \text{if } w = \varepsilon \\ \{w\} & \text{if } y = \varepsilon \\ \{a\} \bullet shuffles(x, y) \cup \{b\} \bullet shuffles(w, z) & \text{if } w = ax \text{ and } y = bz \end{cases}$$

For example, $shuffles(01, 10) = \{0101, 0110, 1001, 1010\}$ and $shuffles(00, 11) = \{0011, 0101, 1001, 0110, 1010, 1100\}$.

8. (a) Let $inc: \{0, 1\}^* \to \{0, 1\}^*$ denote the *increment* function, which transforms the binary representation of an arbitrary integer n into the binary representation of $n + 1$, truncated to the same number of bits. For example:

$$inc(0010) = 0011 \qquad inc(0111) = 1000 \qquad inc(1111) = 0000 \qquad inc(\varepsilon) = \varepsilon$$

Let $L \subseteq \{0, 1\}^*$ be an arbitrary regular language. Prove that $inc(L) = \{inc(w) \mid w \in L\}$ is also regular.

(b) Let $dec: \{0, 1\}^* \to \{0, 1\}^*$ denote the *decrement* function, which transforms the binary representation of an arbitrary integer n into the binary representation of $n - 1$, truncated to the same number of bits. More simply, dec is the inverse function of inc. For example:

$$dec(0011) = 0010 \qquad dec(1000) = 0111 \qquad dec(0000) = 1111 \qquad dec(\varepsilon) = \varepsilon$$

Let $L \subseteq \{0, 1\}^*$ be an arbitrary regular language. Prove that $dec(L) = \{dec(w) \mid w \in L\}$ is also regular.

(c) Let $dub: \{0, 1\}^* \to \{0, 1\}^*$ denote the *doubling* function, which transforms the binary representation of an arbitrary integer n into the binary representation of $2n$, truncated to the same number of bits. For example:

$$dub(0010) = 0100 \qquad dub(0111) = 1110 \qquad dub(1111) = 1110 \qquad dub(\varepsilon) = \varepsilon$$

Let $L \subseteq \{0, 1\}^*$ be an arbitrary regular language. Prove that $dub(L) = \{dub(w) \mid w \in L\}$ is also regular.

⋆(d) Let $trip: \{0, 1\}^* \to \{0, 1\}^*$ denote the *tripling* function, which transforms the binary representation of an arbitrary integer n into the binary representation of $3n$, truncated to the same number of bits. For example:

$$trip(0010) = 0110 \qquad trip(0111) = 0101 \qquad trip(1111) = 1101 \qquad trip(\varepsilon) = \varepsilon$$

Let $L \subseteq \{0, 1\}^*$ be an arbitrary regular language. Prove that $trip(L) = \{trip(w) \mid w \in L\}$ is also regular. *[Hint: It may be easier to consider the language $trip(L^R)^R$ first.]*

⋆9. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular.

(a) $sqrt(L) := \{x \in \Sigma^* \mid xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = |x|^2\}$

(b) $log(L) := \{x \in \Sigma^* \mid xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = 2^{|x|}\}$

(c) $flog(L) := \{x \in \Sigma^* \mid xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = F_{|x|}\}$, where F_n is the nth Fibonacci number.

$\star$10. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular.

(a) $somerep(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for some } n \geq 0\}$

(b) $allreps(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for every } n \geq 0\}$

(c) $manyreps(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for infinitely many } n \geq 0\}$

(d) $fewreps(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for finitely many } n \geq 0\}$

(e) $powers(L) := \{w \in \Sigma^* \mid w^{2^n} \in L \text{ for some } n \geq 0\}$

$\bigstar$ (f) $whatthe_N(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for some } n \in N\}$, where N is an **arbitrary** fixed set of non-negative integers. *[Hint: You only have to prove that an accepting NFA exists; you don't have to describe how to construct it.]*

[Hint: For each of these languages, there is an accepting NFA with at most q^q states, where q is the number of states in some DFA that accepts L.]

11. For any string $w \in (0 + 1)^*$, let $sort(w)$ be the string obtained by sorting the symbols in w. More succinctly, let $sort(w) = 0^{\#(0,w)} 1^{\#(1,w)}$.

(a) Prove that there is a regular language $L \subseteq (0 + 1)^*$ such that the language $sort(L) = \{sort(w) \mid w \in L\}$ is *not* regular.

$\star$(b) Prove that for any regular language $L \subseteq (0 + 1)^*$, the language $sort^{-1}(L) = \{w \mid sort(w) \in L\}$ is regular.

$\star$12. For any string $w \in (0 + 1)^*$, let $\langle w \rangle_2$ denote the integer represented by w in binary. For example:
$$\langle \varepsilon \rangle_2 = 0 \qquad \langle 0010 \rangle_2 = 2 \qquad \langle 0111 \rangle_2 = 7 \qquad \langle 1111 \rangle_2 = 15$$

Let L and L' be arbitrary regular languages over the alphabet $\{0, 1\}$. Prove that the following language is also regular:

$$\{w \in (0 + 1)^* \mid \langle w \rangle_2 = \langle x \rangle_2 + \langle y \rangle_2 \text{ for some strings } x \in L \text{ and } y \in L'\}$$

$\bigstar$13. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular.

(a) $repsqrt(L) = \{w \in \Sigma^* \mid w^{|w|} \in L\}$.

(b) $replog(L) = \{w \in \Sigma^* \mid w^{2^{|w|}} \in L\}$.

(c) $repflog(L) = \left\{ w \in \Sigma^* \mid w^{F_{|w|}} \in L \right\}$, where F_n is the nth Fibonacci number.

*[Hint: The NFAs for these languages use a **LOT** of states. Let $M = (\Sigma, Q, s, A, \delta)$ be a DFA that accepts L. Imagine that you somehow know $\delta^*(q, w)$ in advance, for every state $q \in Q$. Ha, ha, ha! Mine is an evil laugh!]*