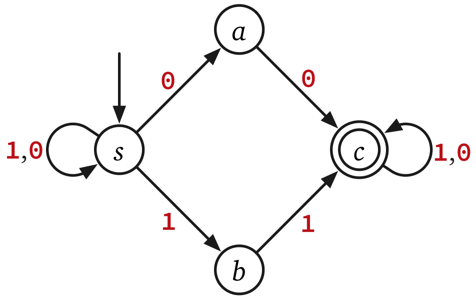
The art of art, the glory of expression
and the sunshine of the light of letters is simplicity.
Nothing is better than simplicity . . . .
nothing can make up for excess or for the lack of definiteness.
— Walt Whitman, Preface to *Leaves of Grass* (1855)

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 0 and 1—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***.

© Copyright 2016 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

- An arbitrary ***transition*** function $\boldsymbol{\delta} \colon Q \times \Sigma \to 2^Q$.
- A ***start state*** $\boldsymbol{s} \in Q$.
- A subset $\boldsymbol{A} \subseteq Q$ 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 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 $\bigcup_{q \in C} \delta(q, a)$. 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.

More formally, we define the function $\delta^* \colon Q \times \Sigma^* \to 2^Q$ that transitions on *strings* as follows:

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

The NFA $(Q, \Sigma, \delta, s, A)$ ***accepts*** $w \in \Sigma^*$ if and only if $\delta^*(s, w) \cap A \neq \varnothing$.

We can equivalently define an NFA as a directed graph whose vertices are the states Q, whose edges are labeled with symbols from Σ. We no longer 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 w.

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. As the BSD fortune file put it, "Nondeterminism means never having to say you're wrong."[1] 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.

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). Similarly, if the NFA finds itself in a non-accepting state when the input ends, the NFA destroys the universe. Thus, when the input is gone, only universes in which

[1] 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 for that one time it was parodied on the Simpsons.

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.

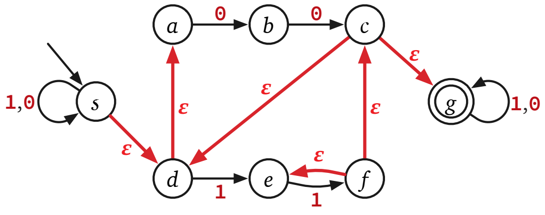
Proofs/oracles. Finally, we can treat NFAs not as a mechanism for *computing* something, but as a mechanism for *verifying proofs*. If we want to *prove* that a string w contains one of the suffixes `00` or `11`, it suffices to demonstrate a single walk in our example NFA that starts at s and ends at 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 $a_1 a_2 \cdots a_\ell = w$.

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.



An 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.

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

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 ***ε-reach*** of a state $q \in Q$ consists of all states r that satisfy one of the following 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 example, in the example NFA above, the ε-reach of state f is $\{a, c, d, f, g\}$.

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

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

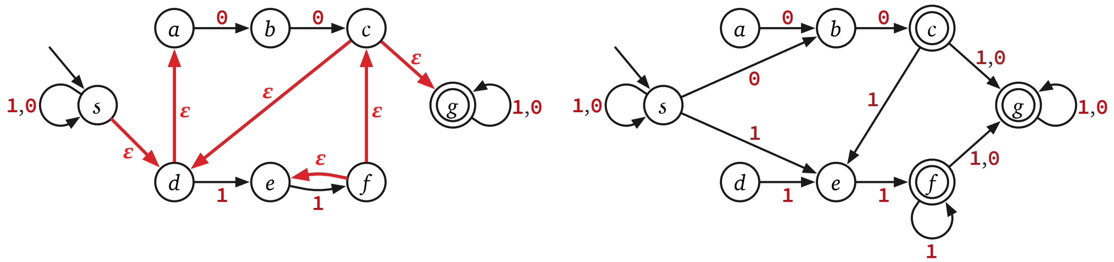
Finally, as usual an NFA with ε-transitions accepts a string w if $\delta^*(s,w)$ contains at least one accepting state.

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:

$$\begin{aligned} Q' &:= Q \\ s' &= s \\ A' &= \big\{ q \in Q \;\big|\; \varepsilon\text{-reach}(q) \cap A \neq \varnothing \big\} \\ \delta'(q,a) &= \delta(\varepsilon\text{-reach}(q),a) \end{aligned}$$

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 would be transformed into the following equivalent NFA without ε-transitions. The NFA on the right has two unreachable states a and d, but whatever.



An NFA with ε-transitions, and an equivalent NFA without ε-transitions

4.4 Kleene's Theorem

We are now finally in a position to prove the following fundamental fact, first observed by Steven 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 DFA from NFAs: 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 $\delta' \colon 2^Q \times \Sigma \to 2^Q$, 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:

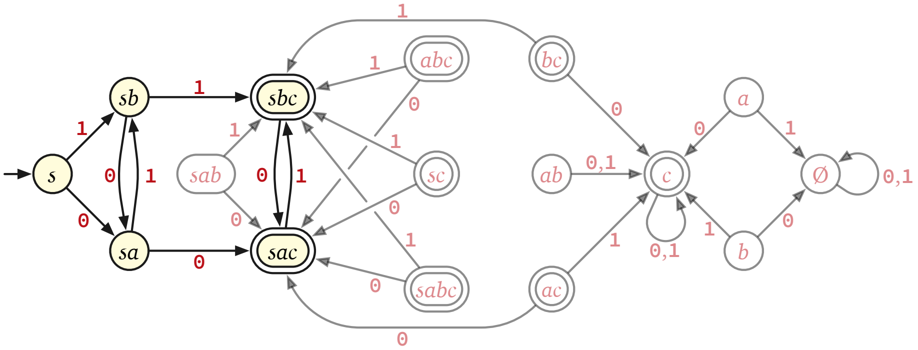
$$\begin{aligned}
Q' &:= 2^Q \\
s' &:= \{s\} \\
A' &:= \left\{ S \subseteq Q \mid S \cap A \neq \varnothing \right\} \\
\delta'(q', a) &:= \bigcup_{p \in q'} \delta(p, a) \qquad \text{for all } q' \subseteq Q \text{ and } a \in \Sigma.
\end{aligned}$$

Similarly, any NFA with ε-transitions is equivalent to a DFA with the transition function

$$\delta'(q', a) = \bigcup_{p \in q'} \; \bigcup_{r \in \varepsilon\text{-reach}(p)} \delta(r, a)$$

for all $q' \subseteq Q$ 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.

For example, given the NFA with ε-transitions from Section **??**, the standard subset construction would produce a DFA with 256 states, but the incremental subset construction produces an eleven-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.

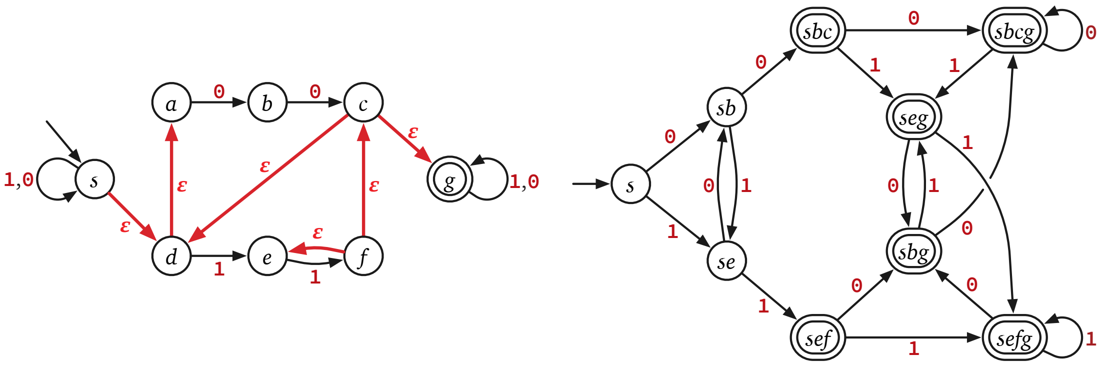
- None of the NFA states $\{s, a, d\}$ is an accepting state, so s is not an accepting state of the DFA; we *don't* check the second column.

- Next, $\delta'(\{s, a, d\}, \mathtt{0}) = \delta(s, \mathtt{0}) \cup \delta(a, \mathtt{0}) \cup \delta(d, \mathtt{0}) = \{s\} \cup \{b\} \cup \varnothing = \{s, b\}$, so we write sb in the third column, and start a new row for the DFA state sb.

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

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', \mathtt{0})$	$\delta'(q', \mathtt{1})$
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.



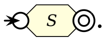
An eight-state NFA with ε-transitions, and the output of the incremental subset construction for that NFA.

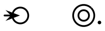
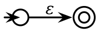
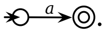
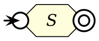
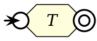
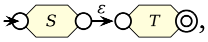
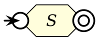
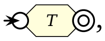
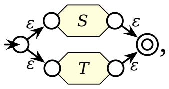
4.6 NFAs from Regular Expressions: Thompson's Algorithm

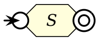
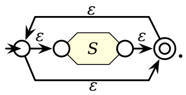
We now turn to the core of Kleene's theorem, which claims that regular languages (described by regular expressions) and automatic languages (accepted by finite-state automata) are the same.

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.

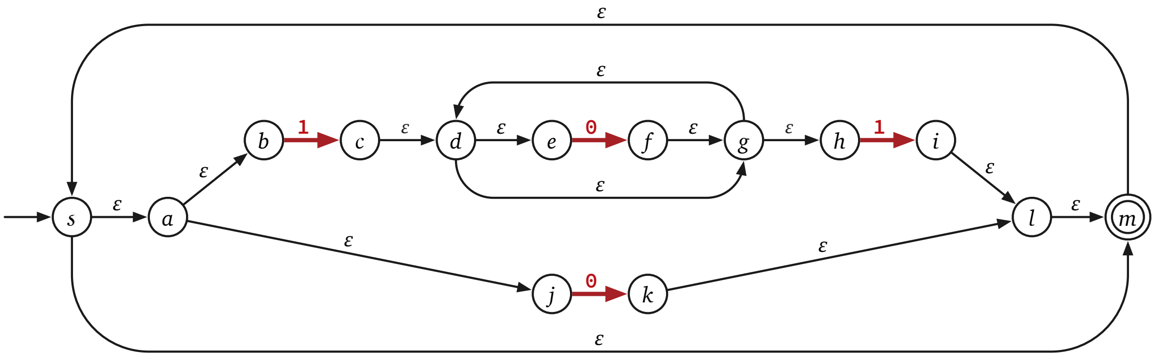
Let R be an arbitrary regular expression over an arbitrary finite alphabet Σ. Assume that for any sub-expression S of R, the language described by S is accepted by an NFA with one accepting state distinct from its start state, which we denote pictorially by . There are six cases to consider—three base 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 .
- If $R = a$ for some symbol $a \in \Sigma$, then $L(R) = \{a\}$ is accepted by the NFA . (The case where R is a single *string* with length greater than 1 reduces to the single-symbol case by concatenation, as described in the next case.)
- 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  and , respectively. Then $L(R) = L(ST) = L(S) \bullet L(T)$ is accepted by the NFA , 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 language $L(S)$ and $L(T)$ are accepted by NFAs  and , respectively. Then $L(R) = L(S + T) = L(S) \cup L(T)$ is accepted by the NFA , built by connecting the two component NFAs in parallel with new start and accept states.

- Finally, suppose $R = S^*$ for some regular expression S. The inductive hypothesis implies that the language $L(S)$ is accepted by an NFA . Then the language $L(R) = L(S^*) = L(S)^*$ is accepted by the NFA .

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. □

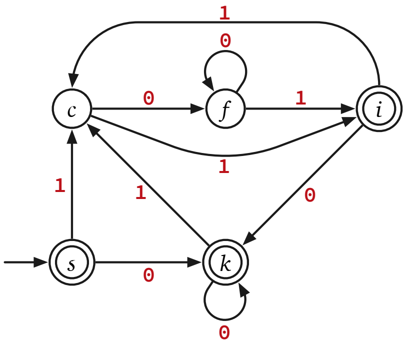
As an example, given the regular expression $(0+10^*1)^*$ of strings containing an even number of 1s, Thompson's algorithm produces a 14-state NFA shown on the next page. As this example shows, Thompson's algorithm tends to produce NFAs with many redundant states. Fortunately, just as there are for DFAs, 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 NFA shown above, for example, the incremental subset construction yields a DFA for the language $(0+10^*1)^*$ 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

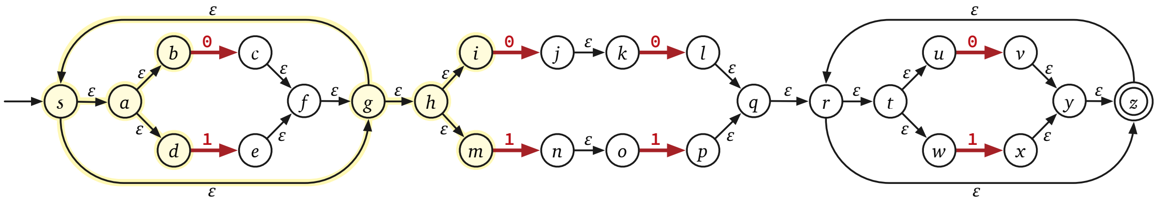


The DFA computed by the incremental subset algorithm from Thompson's NFA for $(0+10^*1)^*$.

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 $2^{13} = 8096$ states that the naïve subset construction would yield!

4.7 Another Example

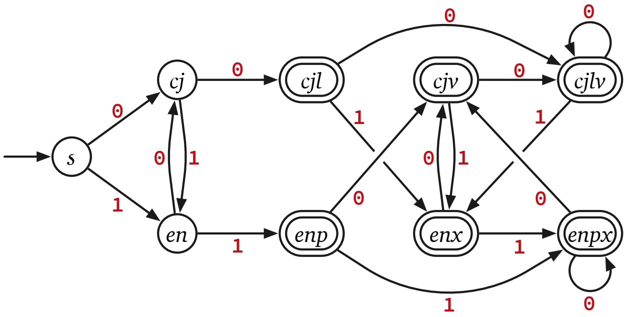
Here is another example of all the algorithms we've seen so far, starting with the regular expression $(\mathtt{0}+\mathtt{1})^*(\mathtt{00}+\mathtt{11})(\mathtt{0}+\mathtt{1})^*$, which describes the language accepted by our very first example NFA. Thompson's algorithm constructs the following 26-state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



Thompson's NFA for the regular expression $(\mathtt{0}+\mathtt{1})^*(\mathtt{00}+\mathtt{11})(\mathtt{0}+\mathtt{1})^*$,
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 Thompson's NFA. As usual, the resulting DFA has far more states that necessary.

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q',\mathtt{0})$	$\delta'(q',\mathtt{1})$
s	$sabdghim$		cj	en
cj	$sabcdfghijkm$		cjl	en
en	$sabedfghimno$		cj	enp
cjl	$sabcdfghijklmqrtuwz$	✓	$cjlv$	enx
enp	$sabdefghimnopqrtuwz$	✓	cjv	$enpx$
$cjlv$	$sabcdfghijklmqrtuvwyz$	✓	$cjlv$	enx
enx	$sabdefghmnopqrtuwxyz$	✓	cjv	$enpx$
cjv	$sabcdfghijkmrtuvwyz$	✓	$cjlv$	enx
$enpx$	$sabdefghmnopqrtuwxyz$	✓	cjv	$enpx$



The DFA computed by the incremental subset algorithm from Thompson's NFA for $(\mathtt{0}+\mathtt{1})^*(\mathtt{00}+\mathtt{11})(\mathtt{0}+\mathtt{1})^*$.

⋆4.8 Regular Expressions from NFAs: Han and Wood's Algorithm

The only component of Kleene's theorem we still have to prove is that every language accepted by a DFA or NFA is regular. I'll describe a relatively recent argument that is (at least morally) equivalent to Kleene's 1951 proof, but using more modern standard notation.

As often happens, it is actually easier to prove a stronger result. We consider a natural generalization of NFAs called ***expression automata***, introduced by Yo-Sub Han and Derick Wood in 2005.[2] Formally, an expression automaton consists of the following components:

- A finite set Σ called the ***input alphabet***
- Another finite set Q whose elements are called ***states***
- A ***start state*** $s \in Q$
- A single ***terminal state*** $t \in Q \setminus \{s\}$
- A ***transition function*** $R\colon (Q \setminus \{t\}) \times (Q \setminus \{s\}) \to \mathit{Reg}(\Sigma)$, where $\mathit{Reg}(\Sigma)$ is the set of regular expressions over Σ

Less formally, an expression automaton is a directed graph that includes a directed edge $p{\to}q$ labeled with a regular expression $R(p{\to}q)$, from *every* vertex p to *every* vertex q (including $q = p$), where by convention, we require that $R(q{\to}s) = R(t{\to}q) = \varnothing$ for every vertex q.

We say that string w ***matches*** a transition $p{\to}q$ if w matches the regular expression $R(p{\to}q)$. In particular, if $R(p{\to}q) = \varnothing$, then no string matches $p{\to}q$. More generally, w matches a sequence of states $q_0{\to}q_1{\to}\cdots{\to}q_k$ if w matches the regular expression $R(q_0{\to}q_1) \bullet R(q_1{\to}q_2) \bullet \cdots \bullet R(q_{k-1}{\to}q_k)$. Equivalently, w matches the sequence $q_0{\to}q_1{\to}\cdots{\to}q_k$ if either

- $w = \varepsilon$ and the sequence has only one state ($k = 0$), or
- $w = xy$ for some string x that matches the regular expression $R(q_0{\to}q_1)$ and some string y that matches the remaining sequence $q_1{\to}\cdots{\to}q_k$.

An expression automaton ***accepts*** any string that matches at least one sequence of states that starts at s and ends at t. The ***language*** of an expression automaton E is the set of all strings that E accepts.

Expression automata are nondeterministic. A single string could match several (even infinitely many) state sequences that start with s, and it could match each of those state sequences in several different ways. A string is accepted if *at least one* of the state sequences it matches ends at t. Conversely, a string might match *no* state sequences; all such strings are rejected.

Two 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 an NFA, by recursively expanding any nontrivial transition. To complete the proof of Kleene's theorem, we show how to convert any 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 trivially, E accepts the regular language $R(s{\to}t)$.

[2] Yo-Sub Han* and Derick Wood. The generalization of generalized automata: Expression automata. *International Journal of Foundations of Computer Science* 16(3):499–510, 2005.

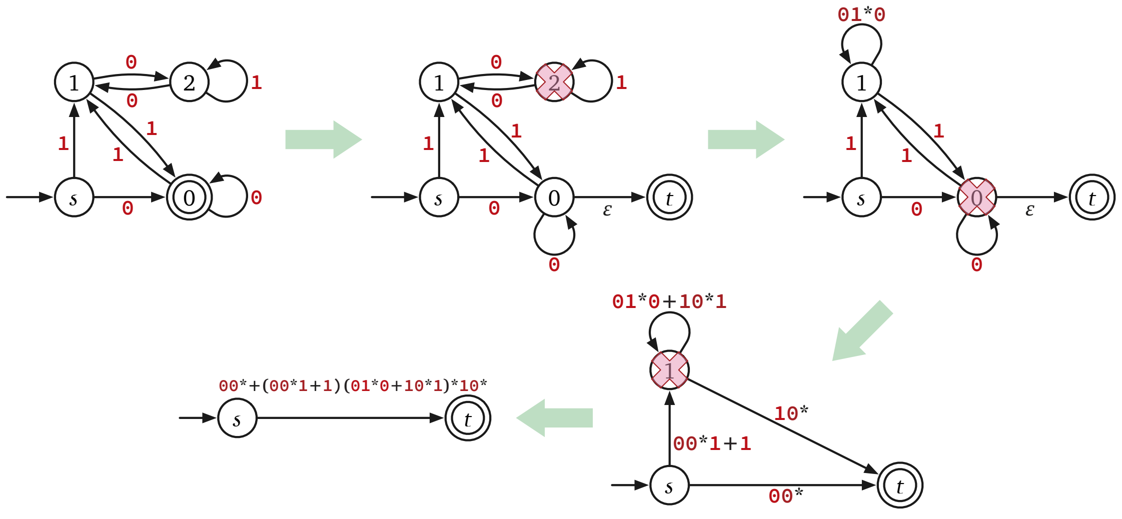
- On the other hand, suppose there is a state $q \in Q \setminus \{s,t\}$. We modify the automaton, without changing its language, so that state q is redundant and can be removed. Define a new transition function $R' \colon Q \times Q \to \mathit{Reg}(\Sigma)$ by setting

$$R'(p{\to}r) := R(p{\to}r) + R(p{\to}q)\,R(q{\to}q)^*\,R(q{\to}r).$$

  With this modified transition function in place, any string w that matches the sequence $p{\to}q{\to}q{\to}\cdots{\to}q{\to}r$ with any number of q's also matches the single transition $p{\to}r$. Thus, by induction, if w matches a sequence of states, it also matches the subsequence obtained by removing all q's. 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 NFA—in particular, any DFA—into an equivalent regular expression. The figure below shows this algorithm in action, on a DFA that accepts the binary representations of non-negative integers divisible by 3, possibly with extra leading `0`s. (State i means the binary number we've read so far is congruent to $i \bmod 3$.) First we convert the DFA into an expression automaton by adding a new accept state. (We don't need to add a new start state, because there are no transitions the original start state s.) Then we remove state 2, then state 0, and finally state 1, updating the transition expressions between any remaining states at each iteration. For the sake of clarity, edges $p{\to}q$ with $R(p{\to}q) = \varnothing$ are omitted from the figures. The final regular expression $\texttt{00}^* + (\texttt{00}^*\texttt{1} + \texttt{1})(\texttt{10}^*\texttt{1} + \texttt{01}^*\texttt{0})^*\texttt{10}^*$ can be slightly simplified to $\texttt{00}^* + \texttt{0}^*\texttt{1}(\texttt{10}^*\texttt{1} + \texttt{01}^*\texttt{0})^*\texttt{10}^*$, 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 Han and Wood's algorithm.

Given an NFA with n states (including s and t), Han and Wood'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 Functions of Regular Languages

★★★ English!

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

Lemma 4.5. *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, our new NFA M' reads its input string w and somehow needs to simulate 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 reaches 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' simultaneously run two copies of M (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 $h = \delta^*(s, w)$ 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.

$$\begin{aligned}
Q' &= (Q \times Q \times Q) \cup \{s'\} \\
A' &= \big\{(h,h,q) \bigm| h \in Q \text{ and } q \in A\big\}
\end{aligned}$$

$$\begin{aligned}
\delta'(s', \varepsilon) &= \big\{(s,h,h) \bigm| h \in Q\big\} \\
\delta'(s', a) &= \varnothing && \text{for all } a \in \Sigma \\
\delta'((p,h,q), \varepsilon) &= \varnothing && \text{for all } p,h,q \in Q \\
\delta'((p,h,q), a) &= \big\{(\delta(p,a), h, \delta(q,a))\big\} && \text{for all } p,h,q \in Q \text{ and } a \in \Sigma
\end{aligned}$$

□

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) $(\mathtt{01} + \mathtt{10})^*(\mathtt{0} + \mathtt{1} + \varepsilon)$

(b) $(\varepsilon + \mathtt{1})(\mathtt{01})^*(\varepsilon + \mathtt{0})$

(c) $\mathtt{1}^* + (\mathtt{10})^* + (\mathtt{100})^*$

(d) $(\varepsilon + \mathtt{0} + \mathtt{00})(\mathtt{1} + \mathtt{10} + \mathtt{100})^*$

(e) $((\mathtt{0} + \mathtt{1})(\mathtt{0} + \mathtt{1}))^*$

(f) $\varepsilon + \mathtt{0}(\mathtt{0} + \mathtt{1})^* + \mathtt{1}(\mathtt{1} + \mathtt{0})^*$

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

(a) $\mathit{censor}(L) := \left\{\mathtt{\#}^{|w|} \;\middle|\; w \in L\right\}$

(b) $\mathit{dehash}(L) = \left\{\mathit{dehash}(w) \;\middle|\; w \in L\right\}$, where $\mathit{dehash}(w)$ is the subsequence of w obtained by deleting every $\mathtt{\#}$.

(c) $\mathit{insert}(L) := \left\{x\mathtt{\#}y \;\middle|\; xy \in L\right\}$.

(d) $\mathit{delete}(L) := \left\{xy \;\middle|\; x\mathtt{\#}y \in L\right\}$.

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

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

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

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

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

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

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

(l) $\mathit{everyother}(L) := \{\mathit{everyother}(w) \mid w \in L\}$, where $\mathit{everyother}(w)$ is the subsequence of w containing every other symbol. For example, $\mathit{everyother}(\mathtt{EVERYOTHER}) = \mathtt{VROHR}$.

(m) $\mathit{rehtoyreve}(L) := \{w \in \Sigma^* \mid \mathit{everyother}(w) \in L\}$.

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

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

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

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

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

(s) $\mathit{half}(L) := \{w \in \Sigma^* \mid ww \in L\}$

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

(u) $\mathit{palin}(L) := \left\{w \in \Sigma^* \;\middle|\; ww^R \in L\right\}$

(v) $\mathit{drome}(L) := \left\{w \in \Sigma^* \;\middle|\; w^Rw \in L\right\}$

*(w) $\mathit{sqrt}(L) := \left\{x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = |x|^2\right\}$

- ⋆(x) *log*$(L) := \left\{x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = 2^{|x|}\right\}$
- ⋆(y) *flog*$(L) := \left\{x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = F_{|x|}\right\}$, where F_n is the nth Fibonacci number.

⋆3. 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*$(L) := \left\{w \in \Sigma^* \;\middle|\; w^{2^n} \in L \text{ for some } n \geq 0\right\}$

★(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.]

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

(a) *repsqrt*$(L) = \left\{w \in \Sigma^* \;\middle|\; w^{|w|} \in L\right\}$.

(b) *replog*$(L) = \left\{w \in \Sigma^* \;\middle|\; w^{2^{|w|}} \in L\right\}$.

(c) *repflog*$(L) = \left\{w \in \Sigma^* \;\middle|\; w^{F_{|w|}} \in L\right\}$, 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!]*

5. 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
- A start state $s \in Q$
- A transition function $\delta : Q \times \Sigma \to Q$
- An output function $\omega : Q \to \Gamma$

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 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*** $\boldsymbol{L^\circ(M)}$ of a Moore machine M is the set of all strings that the machine can output:

$$L^\circ(M) := \left\{ M(w) \;\middle|\; w \in \Sigma^* \right\}$$

(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

$$L^=(M) = \left\{ w \in \Sigma^* \;\middle|\; M(w) = w \right\}$$

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 $\left\{ w \in \Sigma^* \;\middle|\; M(M(w)) = w \right\}$ is regular.

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