Caveat lector! This is the first edition of this lecture note. Some topics are incomplete, and there are almost certainly a few serious errors. Please send bug reports and suggestions to jeffe@illinois.edu.

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)

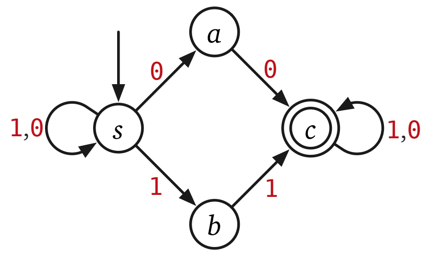
Nondeterminism means never having to say you are wrong.

— BSD 4.3 fortune(6) file (c.1985)

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***.

- An arbitrary ***transition*** function $\boldsymbol{\delta} : Q \times \Sigma \to 2^Q$.
- 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 $\delta^* : Q \times \Sigma^* \to 2^Q$ 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,Σ,δ,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 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 only 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

[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.

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 $(w, \omega) \in (\Sigma \times \Gamma)^*$ 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 $a_1 a_2 \cdots a_\ell = w$.

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:

- $r = q$
- $r \in \delta(q', \varepsilon)$ 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. Now we redefine the extended transition function $\delta^*\colon Q \times \Sigma^* \to 2^Q$, which transitions on arbitrary strings, as follows:

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

As usual, the modified NFA accepts a string w if and only if $\delta^*(s, w) \cap A \neq \varnothing$.

Given an NFA $M = (\Sigma, Q, s, A, \delta)$ with ε-transitions, we can easily construct an equivalent NFA $M' = (\Sigma, Q', s', A', \delta')$ without ε-transitions as follows:

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

Straightforward definition-chasing 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.

4.4 Kleene's Theorem

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

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 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.

One disadvantage of this "construction" is that it usually leads to DFAs with far more states than necessary, in part because most of those states are unreachable. These unreachable states can be avoided by constructing the DFA incrementally, essentially by performing a breadth-first search of the DFA graph, starting at its start state.

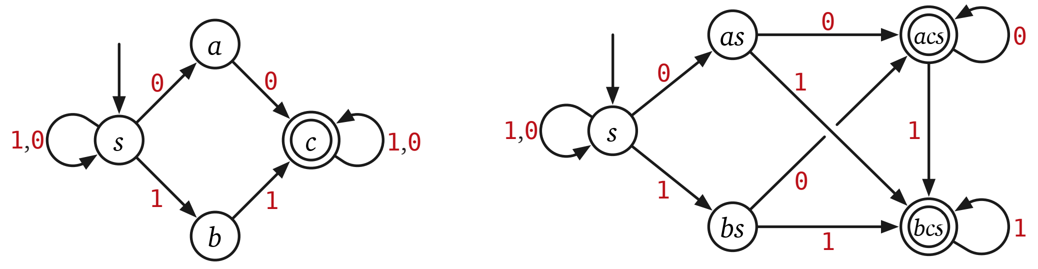
To execute this algorithm by hand, we prepare a table with $|\Sigma| + 3$ columns, with one row for each DFA state we discover. In order, these columns record the following information:

- 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 on the first page of this note, this incremental algorithm produces the following table, yielding a five-state DFA. For this example, the second column is redundant, because the NFA has no ε-transitions, but we will see another example with ε-transitions in the next subsection. To simplify notation, we write each set of states as a simple string, omitting braces and commas.

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	s		as	bs
as	as		acs	bs
bs	bs		as	bcs
acs	acs	✓	acs	bcs
bcs	bcs	✓	acs	bcs



Our example NFA, and the output of the incremental subset construction for that NFA.

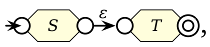
4.6 NFAs from Regular Expressions: Thompson's Algorithm

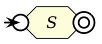
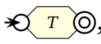
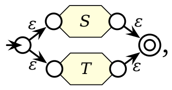
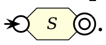
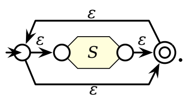
Lemma 4.2. *Every regular language is accepted by a non-deterministic finite automaton.*

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 $\to\bigcirc\boxed{S}\circledcirc$. There are six cases to consider—three base cases and three recursive cases—mirroring the recursive definition of a regular expression.

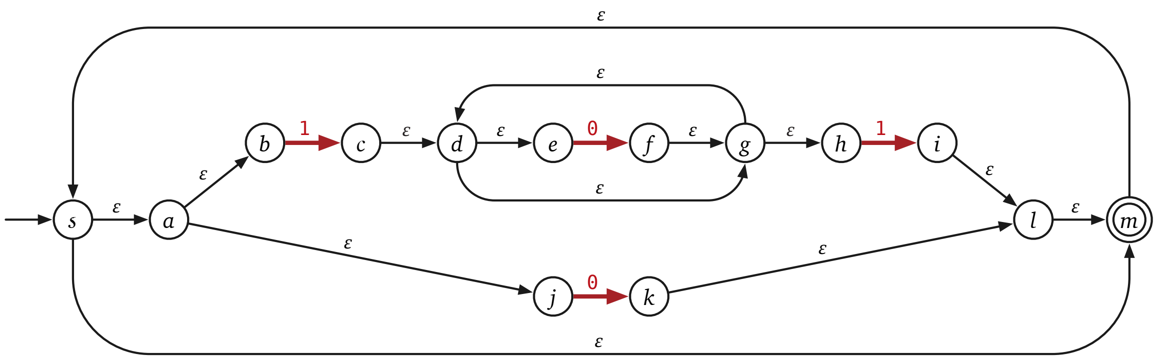
- If $R = \varnothing$, then $L(R) = \varnothing$ is accepted by the empty NFA: $\to\bigcirc\quad\circledcirc$.
- If $R = \varepsilon$, then $L(R) = \{\varepsilon\}$ is accepted by the NFA $\to\bigcirc\xrightarrow{\varepsilon}\circledcirc$.
- If $R = a$ for some symbol $a \in \Sigma$, then $L(R) = \{a\}$ is accepted by the NFA $\to\bigcirc\xrightarrow{a}\circledcirc$. (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 $\to\bigcirc\boxed{S}\circledcirc$ and $\to\bigcirc\boxed{T}\circledcirc$, 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.

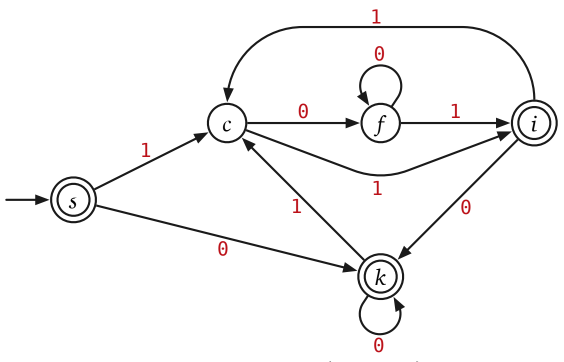


The NFA constructed by Thompson's algorithm for the regular expression $(0 + 10^*1)^*$.
The four non-ε-transitions are drawn with with bold red arrows for emphasis.

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:

This DFA can be further simplified to just two states, by observing that all three accepting states are all 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!

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	$degh$		f	i
i	$sabjilm$	✓	k	c



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

*4.7 NFAs from Regular Expressions: Glushkov's Algorithm

Thompson's algorithm is actually a modification of an earlier algorithm, which was independently discovered by V. I. Glushkov in 1961 and Robert McNaughton and Hisao Yamada in 1960. Given a regular expression containing n symbols (not counting the parentheses and pluses and stars), Glushkov's algorithm produces an NFA with exactly $n + 1$ states.

Glushkov's algorithm combines six functions on regular expressions:

- *index*(R) is the regular expression obtained by replacing the symbols in R with the integers 1 through n, in order from left to right. For example, *index*$((0 + 10^*1)^*) = (1 + 23^*4)^*$.
- *symbols*(R) denotes the string obtained by removing all non-symbols from R. For example, *symbols*$((0 + 10^*1)^*) = 0101$.
- *has-ε*(R) is TRUE if $\varepsilon \in L(R)$ and FALSE otherwise.
- *first*(R) is the set of all initial symbols of strings in $L(R)$.
- *last*(R) is the set of all final symbols of strings in $L(R)$.
- *middle*(R) is the set of all pairs (a, b) such that ab is a substring of some string in $L(R)$.

The last four functions obey the following recurrences:

$$\begin{aligned}
\textit{has-}\varepsilon(\varnothing) &= \varnothing \\
\textit{has-}\varepsilon(w) &= \begin{cases} \textsc{True} & \text{if } w = \varepsilon \\ \textsc{False} & \text{otherwise} \end{cases} \\
\textit{has-}\varepsilon(S + T) &= \textit{has-}\varepsilon(S) \vee \textit{has-}\varepsilon(T) \\
\textit{has-}\varepsilon(ST) &= \textit{has-}\varepsilon(S) \wedge \textit{has-}\varepsilon(T) \\
\textit{has-}\varepsilon(S^*) &= \textsc{True}
\end{aligned}$$

$$\begin{aligned}
\textit{first}(\varnothing) &= \varnothing \\
\textit{first}(w) &= \begin{cases} \varnothing & \text{if } w = \varepsilon \\ \{a\} & \text{if } w = ax \end{cases} \\
\textit{first}(S + T) &= \textit{first}(S) \cup \textit{first}(T) \\
\textit{first}(ST) &= \begin{cases} \textit{first}(S) \cup \textit{first}(T) & \text{if } \textit{has-}\varepsilon(S) \\ \textit{first}(T) & \text{otherwise} \end{cases} \\
\textit{first}(S^*) &= \textit{first}(S)
\end{aligned}$$

$$\begin{aligned}
\textit{last}(\varnothing) &= \varnothing \\
\textit{last}(w) &= \begin{cases} \varnothing & \text{if } w = \varepsilon \\ \{a\} & \text{if } w = xa \end{cases} \\
\textit{last}(S + T) &= \textit{last}(S) \cup \textit{last}(T) \\
\textit{last}(ST) &= \begin{cases} \textit{last}(S) \cup \textit{last}(T) & \text{if } \textit{has-}\varepsilon(T) \\ \textit{last}(T) & \text{otherwise} \end{cases} \\
\textit{last}(S^*) &= \textit{last}(S)
\end{aligned}$$

$$\begin{aligned}
\mathit{middle}(\varnothing) &= \varnothing \\
\mathit{middle}(w) &= \begin{cases} \varnothing & \text{if } |w| \le 1 \\ \{(a,b)\} \cup \mathit{middle}(bx) & \text{if } w = abx \end{cases} \\
\mathit{middle}(S+T) &= \mathit{middle}(S) \cup \mathit{middle}(T) \\
\mathit{middle}(ST) &= \mathit{middle}(S) \cup (\mathit{last}(S) \times \mathit{first}(T)) \cup \mathit{middle}(T) \\
\mathit{middle}(S^*) &= \mathit{middle}(S) \cup (\mathit{last}(S) \times \mathit{first}(S))
\end{aligned}$$

For example, the set $\mathit{middle}((1+23^*4)^*)$ can be computed recursively as follows. If we're doing this by hand, we can skip many of the steps in this derivation, because we know what the functions *first*, *middle*, *last*, and *has-ε* actually mean, but a mechanical recursive evaluation would necessarily evaluate every step.

$$\begin{aligned}
&\mathit{middle}((1+23^*4)^*) \\
&\quad = \mathit{middle}(1+23^*4) \cup \big(\mathit{last}(1+23^*4) \times \mathit{first}(1+23^*4)\big) \\
&\quad = \mathit{middle}(1) \cup \mathit{middle}(23^*4) \;\cup\; \big(\mathit{last}(1+23^*4) \times \mathit{first}(1+23^*4)\big) \\
&\quad = \varnothing \cup \mathit{middle}(23^*4) \;\cup\; \big(\mathit{last}(1+23^*4) \times \mathit{first}(1+23^*4)\big) \\
&\quad = \mathit{middle}(2) \cup \big(\mathit{last}(2) \times \mathit{first}(3^*4)\big) \cup \mathit{middle}(3^*4) \;\cup\; \big(\mathit{last}(1+23^*4) \times \mathit{first}(1+23^*4)\big) \\
&\quad = \varnothing \cup \big(\{2\} \times \mathit{first}(3^*4)\big) \;\cup\; \mathit{middle}(3^*4) \;\cup\; \big(\mathit{last}(1+23^*4) \times \mathit{first}(1+23^*4)\big) \\
&\quad = \big(\{2\} \times \big(\mathit{first}(3^*) \cup \mathit{first}(4)\big)\big) \;\cup\; \mathit{middle}(3^*4) \;\cup\; \big(\mathit{last}(1+23^*4) \times \mathit{first}(1+23^*4)\big) \\
&\quad = \big(\{2\} \times \big(\mathit{first}(3) \cup \mathit{first}(4)\big)\big) \;\cup\; \mathit{middle}(3^*4) \;\cup\; \big(\mathit{last}(1+23^*4) \times \mathit{first}(1+23^*4)\big) \\
&\quad = \big(\{2\} \times \{3,4\}\big) \;\cup\; \mathit{middle}(3^*4) \;\cup\; \big(\mathit{last}(1+23^*4) \times \mathit{first}(1+23^*4)\big) \\
&\quad = \big\{(2,3),(3,4)\big\} \;\cup\; \mathit{middle}(3^*4) \;\cup\; \big(\mathit{last}(1+23^*4) \times \mathit{first}(1+23^*4)\big) \\
&\quad \vdots \\
&\quad = \big\{(1,1),(1,2),(2,3),(2,4),(3,3),(3,4),(4,1),(4,2)\big\}
\end{aligned}$$

Finally, given any regular expression R, Glushkov's algorithm constructs the NFA $M_R = (\Sigma, Q, s, A, \delta)$ that accepts exactly the language $L(R)$ as follows:

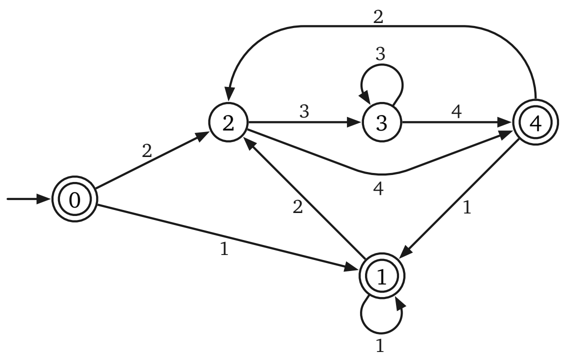
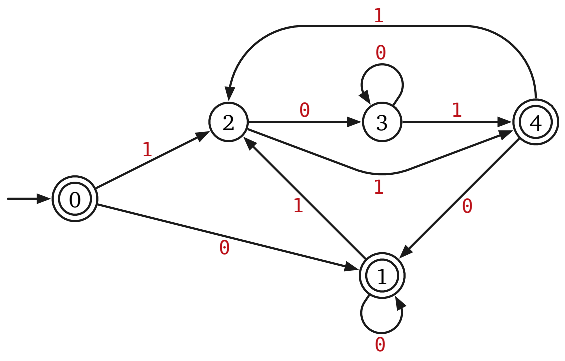
$$\begin{aligned}
Q &= \big\{0, 1, \ldots, |\mathit{symbols}(R)|\big\} \\
s &= 0 \\
A &= \begin{cases} \{0\} \cup \mathit{last}(\mathit{index}(R)) & \text{if } \mathit{has}\text{-}\varepsilon(R) \\ \mathit{last}(\mathit{index}(R)) & \text{otherwise} \end{cases} \\
\delta(0,a) &= \big\{ j \in \mathit{first}(\mathit{index}(R)) \;\big|\; a = \mathit{symbols}(R)[j] \big\} \\
\delta(i,a) &= \big\{ j \;\big|\; (i,j) \in \mathit{middle}(\mathit{index}(R)) \text{ and } a = \mathit{symbols}(R)[j] \big\}
\end{aligned}$$

There are a few natural ways to think about Glushkov's algorithm that are somewhat less impenetrable than the wall of definitions. One viewpoint is that Glushkov's algorithm first computes a *DFA* for the indexed regular expression *index*(R)—in fact, a DFA with the fewest possible states, except for an extra start state—and then replaces each index with the corresponding symbol in *symbols*(R) to get an NFA for the original expression R. Another useful observation is that Glushkov's NFA is identical to the result of removing all ε-transitions from Thompson's NFA for the same regular expression.

For example, given the regular expression $R = (\mathtt{0} + \mathtt{10}^*\mathtt{1})^*$, Glushkov's algorithm computes

$$\begin{aligned}
\mathit{index}(R) &= (1 + 23^*4)^* \\
\mathit{symbols}(R) &= \mathtt{0101} \\
\mathit{has}\text{-}\varepsilon(R) &= \textsc{True} \\
\mathit{first}(\mathit{index}(R)) &= \{1, 2\} \\
\mathit{last}(\mathit{index}(R)) &= \{1, 4\} \\
\mathit{middle}(\mathit{index}(R)) &= \big\{(1,1), (1,2), (2,3), (2,4), (3,3), (3,4), (4,1), (4,2)\big\}
\end{aligned}$$

and then constructs the following five-state NFA.

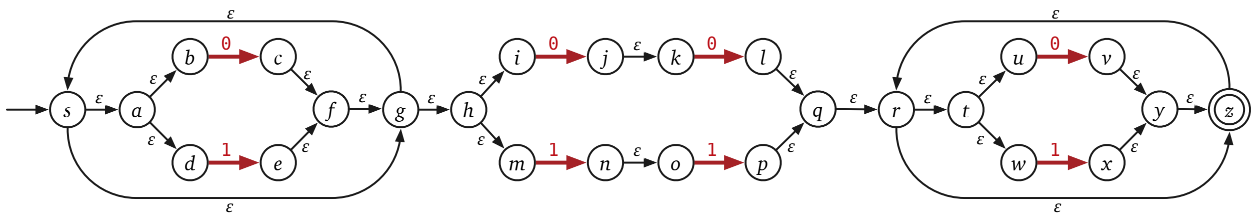



Glushkov's DFA for the index expression $(1+23^*4)^*$ and Glushkov's NFA for the regular expression $(\mathtt{0}+\mathtt{10}^*\mathtt{1})^*$.

Hey, look, Glushkov's algorithm actually gave us a DFA! In fact, it gave us *precisely* the same DFA that we constructed earlier by sending Thompson's NFA through the incremental subset algorithm! Unfortunately, that's just a coincidence; in general the output of Glushkov's algorithm is *not* deterministic. We'll see a more typical example in the next section.

4.8 Another Example

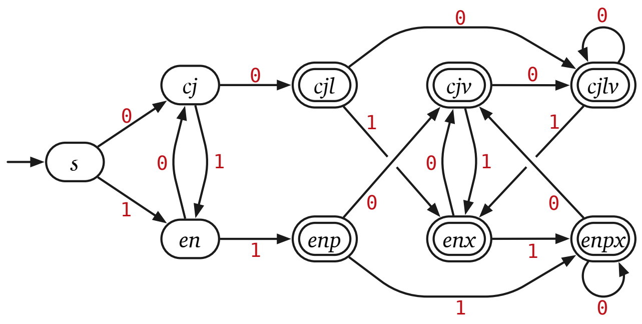
Here is another example of all the algorithms we've seen so far, starting with the regular expression $(\mathtt{0}+\mathtt{1})^*(\mathtt{00}+\mathtt{11})(\mathtt{0}+\mathtt{1})^*$, which describes the language accepted by our very first example NFA. Thompson's algorithm constructs the following 26-state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



Thompson's NFA for the regular expression $(\mathtt{0}+\mathtt{1})^*(\mathtt{00}+\mathtt{11})(\mathtt{0}+\mathtt{1})^*$

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.

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	$sabdghim$		cj	en
cj	$sabdfghijkm$		cjl	en
en	$sabdfghmno$		cj	enp
cjl	$sabdfghijklmqrtuwz$	✓	$cjlv$	enx
enp	$sabdfghmnopqrtuwz$	✓	cjv	$enpx$
$cjlv$	$sabdfghijklmqrtuvwyz$	✓	$cjlv$	enx
enx	$sabdfghmnopqrtuwxyz$	✓	cjv	$enpx$
cjv	$sabdfghijkmrtuvwyz$	✓	$cjlv$	enx
$enpx$	$sabdfghmnopqrtuwxyz$	✓	cjv	$enpx$



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

This DFA has far more states that necessary, intuitively because it keeps looking for 00 and 11 substrings even after it's already found one. After all, when Thompson's NFA finds a00 and 11 substring, it doesn't kill all the other parallel threads, because it *can't*. NFAs often have significantly fewer states than equivalent DFAs, but that efficiency also makes them kind of stupid.

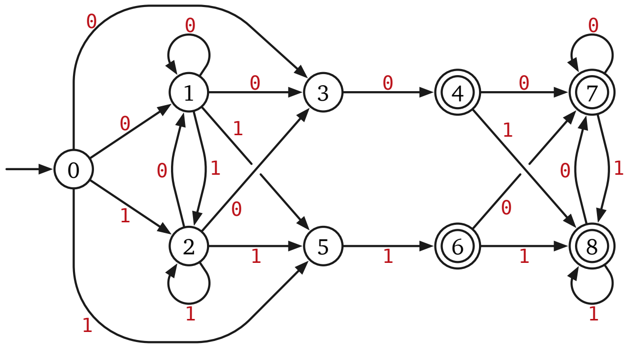
Glushkov's algorithm recursively computes the following values for the same regular expression $R = (0+1)^*(00+11)(0+1)^*$:

$$\begin{aligned}
\mathit{index}(R) &= (1+2)^*(34+56)(7+8)^*\\
\mathit{symbols}(R) &= \mathtt{01001101}\\
\mathit{has}\text{-}\varepsilon(R) &= \textsc{False}\\
\mathit{first}(\mathit{index}(R)) &= \{1,2,3,5\}\\
\mathit{last}(\mathit{index}(R)) &= \{4,6,7,8\}\\
\mathit{middle}(\mathit{index}(R)) &= \big\{(1,1),(1,2),(2,1),(2,2),(1,3),(1,5),(2,3),(2,5),(3,4),\\
&\qquad (5,6),(4,7),(4,8),(6,7),(6,8),(7,7),(7,8),(8,7),(8,8)\big\}
\end{aligned}$$

These values imply the nine-state NFA shown on the next page. Careful readers should confirm that running the incremental subset construction on this NFA yields exactly the same DFA (with different state names) as it did for Thompson's NFA.

*4.9 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. As usual, it is actually easier to prove a stronger result. We consider a more general class of finite-state machines called ***expression automata***, introduced by Yo-Sub Han and Derick Wood


Glushkov's NFA for $(0+1)^*(00+11)(0+1)^*$

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.

[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.

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 $R(s \to t)$.
- On the other hand, suppose there is a state $q \in Q \setminus \{s, a\}$. We can modify the expression automaton so that state q is redundant and can be removed. Define a new transition function $R' : Q \times Q \to Reg(\Sigma)$ 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, any string w that matches the sequence $p \to q \to q \to \cdots \to q \to r$ (for 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. Given an NFA with n states (including s and a), the 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 each iteration, the transition expressions grows in length by roughly a factor of 4 in the worst case, 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 $\Theta(4^n)$.

A figure on the next page shows this conversion algorithm in action for a simple DFA. First we convert the DFA into an expression automaton by adding new start and accept states and merging two transitions, and then we remove each of the three original states, 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.

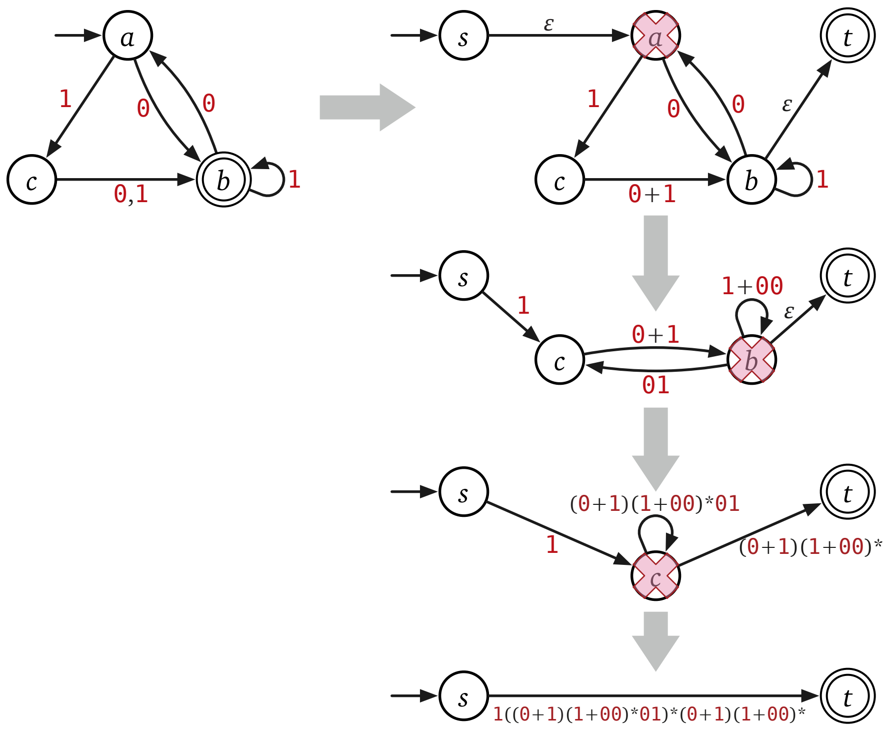
*4.10 Recursive Automata

★★★ Move to separate starred note?

All the flavors of finite-state automata we have seen so far describe/encode/accept/compute *regular* language; these are precisely the languages that can be constructed from individual strings by union, concatenation, and unbounded repetition. The more general class of *context-free* languages can be constructed from individual strings by union, concatenation, and *recursion*. Just as context-free grammars are recursive generalizations of regular expressions, we can define a class of machines called *recursive automata*, which generalize (nondeterministic) finite-state automata.

Formally, a ***recursive automaton*** consists of the following components:

- A non-empty finite set Σ, called the ***input alphabet***
- Another non-empty finite set N, disjoint from Σ, whose elements are called ***module names***
- A ***start name*** $S \in N$



Converting a DFA into an equivalent regular expression.

- A set $M = \{M_A \mid A \in N\}$ of NFAs over the alphabet $\Sigma \cup N$ called ***modules***, each with a single accepting state. Each module M_A has the following components:
  - A finite set Q_A of ***states***, such that $Q_A \cap Q_B = \varnothing$ for all $A \neq B$
  - A ***start*** state $s_A \in Q_A$
  - A ***terminal*** or ***accepting*** state $t_A \in Q_A$
  - A ***transition function*** $\delta_A \colon Q_A \times (\Sigma \cup \{\varepsilon\} \cup N) \to 2^{Q_A}$.

Equivalently, we have a single global transition function $\delta \colon Q \times (\Sigma \cup \{\varepsilon\} \cup N) \to 2^Q$, where $Q = \bigcup_{A \in N} Q_A$, such that for any name A and any state $q \in Q_A$ we have $\delta(q) \subseteq Q_A$. Machine M_S is called the ***main module***.

A ***configuration*** of a recursive automaton is a triple (w, q, s), where w is a string in Σ^* called the ***input***, q is a state in Q called the ***local state***, and σ is a string in Q^* called the ***stack***. The module containing the local state q is called the ***active module***. A configuration can be changed by three types of transitions.

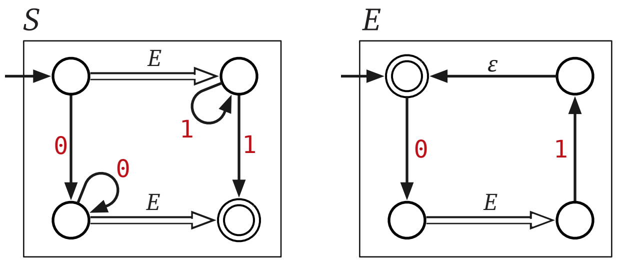
- A ***read*** transition consumes the first symbol in the input and changes the local state within the current module, just like a standard NFA.
- An ***epsilon*** transition changes the local state within the current module, without consuming any input characters, just like a standard NFA.
- A ***call*** transition chooses an arbitrary name A, changes the current state q_0 to some state in $\delta(q, A)$, and pushes the corresponding start state s_A onto the stack (thereby changing the active module to M_A), without consuming any input characters.
- Finally, if the current state is the terminal state of some module *and* the stack is non-empty, a ***return*** transition pops the top state off the stack and makes it the new local state (thereby possibly changing the active module), without consuming any input characters.

Symbolically, we can describe these transitions as follows:

read:	$(ax, q, \sigma) \longmapsto (x, q', \sigma)$	for some $q' \in \delta(q, a)$
epsilon:	$(w, q, \sigma) \longmapsto (w, q', \sigma)$	for some $q' \in \delta(q, \varepsilon)$
call:	$(w, q, \sigma) \longmapsto (w, s_A, q' \cdot \sigma)$	for some $A \in N$ and $q' \in \delta(q, A)$
return:	$(w, t_A, q \cdot \sigma) \longmapsto (w, q, \sigma)$	

A recursive automaton ***accepts*** a string w if there is a *finite* sequence of transitions starting at the start configuration (w, s_S, ε) and ending at the terminal configuration $(\varepsilon, t_S, \varepsilon)$.

For example, the following recursive automaton accepts the language $\{0^m1^n \mid m \neq n\}$. The recursive automaton has two component machines; the start machine named S and a "subroutine" named E (for "equal") that accepts the language $\{0^n1^n \mid n \geq 0\}$. White arrows indicate recursive transitions.



A recursive automaton for the language $\{0^m1^n \mid m \neq n\}$

Lemma 4.4. *Every context-free language is accepted by a recursive automaton.*

Proof:

★★★ Direct construction from the CFG, with one module per nonterminal.

□

For example, the context-free grammar

$$\begin{aligned} S &\to 0A \mid B1 \\ A &\to 0A \mid E \\ B &\to B1 \mid E \\ E &\to \varepsilon \mid 0E0 \end{aligned}$$

leads to the following recursive automaton with four modules:

★★★ Figure!

Lemma 4.5. *Every recursive automaton accepts a context-free language.*

Proof (sketch): Let $R = (\Sigma, N, S, \delta, M)$ be an arbitrary recursive automaton. We define a context-free grammar G that describes the language accepted by R as follows.

The set of nonterminals in G is isomorphic the state set Q; that is, for each state $q \in Q$, the grammar contains a corresponding nonterminal $[q]$. The language of $[q]$ will be the set of strings w such that there is a finite sequence of transitions starting at the start configuration (w, q, ε) and ending at the terminal configuration $(\varepsilon, t, \varepsilon)$, where t is the terminal state of the module containing q.

The grammar has four types of production rules, corresponding to the four types of transitions:

- ***read:*** For each symbol a and each pair of states p and q such that $p \in \delta(q, a)$, the grammar contains the production rule $[q] \rightarrow a[p]$.
- ***epsilon:*** For any two states p and q such that $p \in \delta(q, \varepsilon)$, the grammar contains the production rule $[q] \rightarrow [p]$.
- ***call:*** Each name A and each pair of states states p and q such that $p \in \delta(q, A)$, the grammar contains the production rule $[q] \rightarrow [s_A][p]$.
- ***return:*** Each name A, the grammar contains the production rule $[t_A] \rightarrow \varepsilon$.

Finally, the starting nonterminal of G is $[s_S]$, which corresponds to the start state of the main module.

We can now argue inductively that the grammar G and the recursive automaton R describe the same language. Specifically, any sequence of transitions in R from (w, s_S, ε) to $(\varepsilon, t_S, \varepsilon)$ can be transformed mechanically into a derivation of w from the nonterminal $[s_S]$ in G. Symmetrically, the ***leftmost*** derivation of any string w in G can be mechanically transformed into an accepting sequence of transitions in R. We omit the straightforward but tedious details. □

For example, the recursive automaton on the previous page gives us the following context-free grammar. To make the grammar more readable, I've renamed the nonterminals corresponding to start and terminal states: $S = [s_S]$, $T = [t_S]$, and $E = [s_E] = [t_E]$:

$$\begin{array}{ll} S \rightarrow EA \mid 0B & E \rightarrow \varepsilon \mid 0X \\ A \rightarrow 1A \mid 1T & X \rightarrow EY \\ B \rightarrow 0B \mid ET & Y \rightarrow 1Z \\ T \rightarrow \varepsilon & Z \rightarrow E \end{array}$$

Our earlier proofs imply that we can forbid ε-transitions or even allow regular-expression transitions in our recursive automata without changing the set of languages they accept.

Exercises

1. For each of the following NFAs, describe an equivalent DFA. ("Describe" does not necessarily mean "draw"!)

★★★ Half a dozen examples.

2. For each of the following regular expressions, draw an equivalent NFA.

★★★ Half a dozen examples.

3. For each of the following regular expressions, describe an equivalent DFA. ("Describe" does not necessarily mean "draw"!)

★★★ Half a dozen examples.

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

   (a) $\mathit{ones}(L) := \left\{ w \in 1^* \;\middle|\; |w| = |x| \text{ for some } x \in L \right\}$

   (b) $\mathit{reverse}(L) := \left\{ w \in \Sigma^* \;\middle|\; w^R \in L \right\}$. (Recall that w^R denotes the reversal of string w.)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

*5. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular. *[Hint: For each language, there is an accepting NFA with at most q^q states, where q is the number of states in some DFA that accepts L.]*

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

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

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

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

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

★(f) $\mathit{whatthe}_N(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for some } n \in N\}$, 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.]*

6. For each of the following expression automata, describe an equivalent DFA *and* an equivalent regular expression.

★★★ Half a dozen examples.

7. Describe recursive automata that accept the following languages. ("Describe" does not necessarily mean "draw"!)

★★★ Half a dozen examples.