Life only avails, not the having lived. Power ceases in the instant of repose;
it resides in the moment of transition from a past to a new state,
in the shooting of the gulf, in the darting to an aim.
— Ralph Waldo Emerson, "Self Reliance", *Essays, First Series* (1841)

O Marvelous! what new configuration will come next?
I am bewildered with multiplicity.
— William Carlos Williams, "At Dawn" (1914)

3 Finite-State Machines

3.1 Intuition

Suppose we want to determine whether a given string $w[1..n]$ of bits represents a multiple of 5 in binary. After a bit of thought, you might realize that you can read the bits in w one at a time, from left to right, keeping track of the value modulo 5 of the prefix you have read so far.

```
MultipleOf5($w[1..n]$):
  $rem \leftarrow 0$
  for $i \leftarrow 1$ to $n$
      $rem \leftarrow (2 \cdot rem + w[i]) \bmod 5$
  if $rem = 0$
      return True
  else
      return False
```

Aside from the loop index i, which we need just to read the entire input string, this algorithm has a single local variable *rem*, which has only four different values: 0, 1, 2, 3, or 4.

For example, given the 11-bit input string `00101110110`, your algorithm proceeds as follows, eventually returning False. For purposes of illustration, I'm including the actual binary *value* of the prefix read so far, without any modular arithmetic. The algorithm does not actually maintain this value, only its remainder $rem = value \bmod 5$.

i	$w[1..i]$	*value*	*rem*
0	ε	0	0
1	`0`	0	0
2	`00`	0	0
3	`001`	1	1
4	`0010`	2	2
5	`00101`	5	0
6	`001011`	11	1
7	`0010111`	23	3
8	`00101110`	46	1
9	`001011101`	93	3
10	`0010111011`	187	2
11	`00101110110`	374	4

This algorithm already runs in $O(n)$ time, which is the best we can hope for—after all, we have to read every bit in the input—but we can speed up the algorithm *in practice*. Let's define a

change or ***transition*** function $\delta\colon \{0,1,2,3,4\} \times \{\mathtt{0},\mathtt{1}\} \to \{0,1,2,3,4\}$ as follows:

$$\delta(q,a) = (2q + a) \bmod 5.$$

(Here I'm implicitly converting the symbols `0` and `1` to the corresponding integers 0 and 1.) Since we already know all values of the transition function, we can store them in a precomputed table, and then replace the computation in the main loop of MultipleOf5 with a simple array lookup.

We can also modify the return condition to check for different values modulo 5. To be completely general, we replace the final if-then-else lines with another array lookup, using an array $A[0..4]$ of booleans describing which final mod-5 values are "acceptable".

After both of these modifications, our algorithm looks like one of the following, depending on whether we want something iterative or recursive (with $q = 0$ in the initial call):

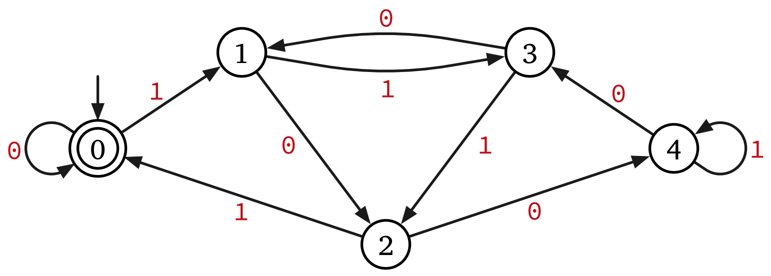
```
DoSomethingCool(w[1..n]):
  $q \leftarrow 0$
  for $i \leftarrow 1$ to $n$
      $q \leftarrow \delta[q, w[i]]$
  return $A[q]$
```

```
DoSomethingCool(q, w):
  if $w = \varepsilon$
      return $A[q]$
  else
      decompose $w = a \cdot x$
      return DoSomethingCool($\delta(q,a), x$)
```

If we want to use our new DoSomethingCool algorithm to implement MultipleOf5, we can give the arrays δ and A the following hard-coded values:

q	$\delta[q,\mathtt{0}]$	$\delta[q,\mathtt{1}]$	$A[q]$
0	0	1	True
1	2	3	False
2	4	0	False
3	1	2	False
4	3	4	False

We can also visualize the behavior of DoSomethingCool by drawing a directed graph, whose vertices represent possible values of the variable q—the possible ***states*** of the algorithm—and whose edges are labeled with input symbols to represent transitions between states. Specifically, the graph includes the labeled directed edge $p \xrightarrow{a} q$ if and only if $\delta(p,a) = q$. To indicate the proper return value, we draw the "acceptable" final states using doubled circles. Here is the resulting graph for MultipleOf5:



State-transition graph for MultipleOf5

Again, if we run the MultipleOf5 algorithm on the string `00101110110` (representing the number 374 in binary), the algorithm performs the following sequence of transitions:

$$0 \xrightarrow{\mathtt{0}} 0 \xrightarrow{\mathtt{0}} 0 \xrightarrow{\mathtt{1}} 1 \xrightarrow{\mathtt{0}} 2 \xrightarrow{\mathtt{1}} 0 \xrightarrow{\mathtt{1}} 1 \xrightarrow{\mathtt{1}} 3 \xrightarrow{\mathtt{0}} 1 \xrightarrow{\mathtt{1}} 3 \xrightarrow{\mathtt{1}} 2 \xrightarrow{\mathtt{0}} 4$$

Because the final state is not the "acceptable" state 0, the algorithm correctly returns FALSE. We can also think of this sequence of transitions as a walk in the graph, which is completely determined by the start state 0 and the sequence of edge labels; the algorithm returns TRUE if and only if this walk ends at an "acceptable" state.

★★★

Here's another bit of intuition that *might* be closer to Kleene's.

Consider the regular sequences of events that can be produced by a structured piece of code: sequencing = concatenation, branching = alternation, looping = Kleene closure. We can model the code as a directed graph, whose nodes correspond to points in the code ("states"), where the edges ("transitions") leaving each node are labeled by events caused by executing that line of code. (For example: read, write, math, successful comparison, unsuccessful comparison, return.) Walking through the graph produces a regular language of event streams. Turning that process on its head, if we're given an event stream, we can check whether a given piece of code could produce it by walking through the graph.

Kleene's insight is that allowing *arbitrary* transition graphs does not let us capture more languages than insisting on structured graphs. In short, goto is unnecessary!

3.2 Formal Definitions

The object we have just described is an example of a ***finite-state machine***. A finite-state machine is a formal model of any system/machine/algorithm that can exist in a finite number of ***states*** and that transitions among those states based on sequence of ***input*** symbols.

Finite-state machines are also known as ***deterministic finite-state automata***, abbreviated ***DFAs***. The word "deterministic" means that the behavior of the machine is completely *determined* by the input string; we'll discuss nondeterministic automata in the next lecture. The word "automaton" (the singular of "automata") comes from ancient Greek αὐτόματος meaning "self-acting", from the roots αὐτό- ("self") and -ματος ("thinking, willing", the root of Latin *mentus*).

Formally, every finite-state machine consists of five components:

- An arbitrary finite set Σ, called the ***input alphabet***.
- Another arbitrary finite set $\boldsymbol{Q}$, whose elements are called ***states***.[1]
- An arbitrary ***transition*** function $\boldsymbol{\delta} : Q \times \Sigma \to Q$.
- A ***start state*** $\boldsymbol{s} \in Q$.
- A subset $\boldsymbol{A} \subseteq Q$ of ***accepting states***.

The behavior of a finite-state machine is governed by an ***input string*** w, which is a finite sequence of symbols from the input alphabet Σ. The machine ***reads*** the symbols in w one at a time in order (from left to right). At all times, the machine has a *current state* q; initially q is the machine's start state s. Each time the machine reads a symbol a from the input string, its current state ***transitions*** from q to $\delta(q, a)$. After all the characters have been read, the machine ***accepts*** w if the current state is in A and ***rejects*** w otherwise. In other words, every finite state machine runs the algorithm DOSOMETHINGCOOL!

[1]It's unclear why we use the letter Q to refer to the state set, and lower-case q to refer to a generic state, but that is now the firmly-established notational standard. Although the formal study of finite-state automata began much earlier, its modern formulation was established in a 1959 paper by Michael Rabin and Dana Scott, for which they won the Turing award. Rabin and Scott called the set of states S, used lower-case s for a generic state, and called the start state s_0. On the other hand, in the 1936 paper for which the Turing award was named, Alan Turing used $q_1, q_2, \ldots, q_R$ to refer to states (or "m-configurations") of a generic Turing machine. Turing may have been mirroring the standard notation Q for configuration (or "qonfiguration") spaces in classical mechanics, also of uncertain origin.

More formally, we extend the transition function $\delta\colon Q \times \Sigma \to Q$ of any finite-state machine to a function $\delta^*\colon Q \times \Sigma^* \to Q$ that transitions on *strings* as follows:

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

Finally, a finite-state machine ***accepts*** a string w if and only if $\delta^*(s,w) \in A$, and ***rejects*** w otherwise. (Compare this definition with the recursive formulation of DoSomethingCool!)

For example, our final MultipleOf5 algorithm is a DFA with the following components:

- input alphabet: $\Sigma = \{\mathtt{0}, \mathtt{1}\}$
- state set: $Q = \{0,1,2,3,4\}$
- transition function: $\delta(q,a) = (2q + a) \bmod 5$
- start state: $s = 0$
- accepting states: $A = \{0\}$

This machine rejects the string `00101110110`, because

$$\begin{aligned}
\delta^*(0,\mathtt{00101110110}) &= \delta^*(\delta(0,\mathtt{0}),\mathtt{0101110110}) \\
= \delta^*(0,\mathtt{0101110110}) &= \delta^*(\delta(0,\mathtt{0}),\mathtt{101110110}) \\
= \delta^*(0,\mathtt{101110110}) &= \delta^*(\delta(0,\mathtt{1}),\mathtt{01110110}) = \cdots \\
&\vdots \\
\cdots = \delta^*(1,\mathtt{110}) &= \delta^*(\delta(1,\mathtt{1}),\mathtt{10}) \\
= \delta^*(3,\mathtt{10}) &= \delta^*(\delta(3,\mathtt{1}),\mathtt{0}) \\
= \delta^*(2,\mathtt{0}) &= \delta^*(\delta(3,\mathtt{0}),\varepsilon) \\
= \delta^*(4,\varepsilon) &= 4 \notin A.
\end{aligned}$$

We have already seen a more graphical representation of this entire sequence of transitions:

$$0 \xrightarrow{\mathtt{0}} 0 \xrightarrow{\mathtt{0}} 0 \xrightarrow{\mathtt{1}} 1 \xrightarrow{\mathtt{0}} 2 \xrightarrow{\mathtt{1}} 0 \xrightarrow{\mathtt{1}} 1 \xrightarrow{\mathtt{1}} 3 \xrightarrow{\mathtt{0}} 1 \xrightarrow{\mathtt{1}} 3 \xrightarrow{\mathtt{1}} 2 \xrightarrow{\mathtt{0}} 4$$

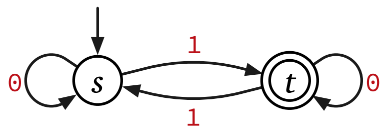
The arrow notation is easier to read and write for specific examples, but surprisingly, most people actually find the more formal functional notation easier to use in formal proofs. Try them both!

We can equivalently define a DFA as a directed graph whose vertices are the states Q, whose edges are labeled with symbols from Σ, such that every vertex has exactly one outgoing edge with each label. In our drawings of finite state machines, the start state s is always indicated by an incoming arrow, and the accepting states A are always indicted by doubled circles. By induction, for any string $w \in \Sigma^*$, this graph contains a unique walk that starts at s and whose edges are labeled with the symbols in w in order. The machine accepts w if this walk ends at an accepting state. This graphical formulation of DFAs is incredibly useful for developing intuition and even designing DFAs. For proofs, it's largely a matter of taste whether to write in terms of extended transition functions or labeled graphs, but (as much as I wish otherwise) I actually find it easier to write ***correct*** proofs using the functional formulation.

3.3 Another Example

The following drawing shows a finite-state machine with input alphabet $\Sigma = \{0, 1\}$, state set $Q = \{s, t\}$, start state s, a single accepting state t, and the transition function

$$\delta(s, 0) = s, \quad \delta(s, 1) = t, \quad \delta(t, 0) = t, \quad \delta(t, 1) = s.$$



A simple finite-state machine.

For example, the two-state machine M at the top of this page accepts the string 00101110100 after the following sequence of transitions:

$$s \xrightarrow{0} s \xrightarrow{0} s \xrightarrow{1} t \xrightarrow{0} t \xrightarrow{1} s \xrightarrow{1} t \xrightarrow{1} s \xrightarrow{0} s \xrightarrow{1} t \xrightarrow{0} t \xrightarrow{0} t.$$

The same machine M rejects the string 11101101 after the following sequence of transitions:

$$s \xrightarrow{1} t \xrightarrow{1} s \xrightarrow{1} t \xrightarrow{0} t \xrightarrow{1} s \xrightarrow{1} t \xrightarrow{0} t \xrightarrow{1} s.$$

Finally, M rejects the empty string, because the start state s is not an accepting state.

From these examples and others, it is easy to conjecture that the language of M is the set of all strings of 0s and 1s with an odd number of 1s. So let's prove it!

Proof (tedious case analysis): Let $\#(a, w)$ denote the number of times symbol a appears in string w. We will prove the following stronger claims by induction, for any string w.

$$\delta^*(s, w) = \begin{cases} s & \text{if } \#(1, w) \text{ is even} \\ t & \text{if } \#(1, w) \text{ is odd} \end{cases} \qquad \text{and} \qquad \delta^*(t, w) = \begin{cases} t & \text{if } \#(1, w) \text{ is even} \\ s & \text{if } \#(1, w) \text{ is odd} \end{cases}$$

Let's begin. Let w be an arbitrary string. Assume that for any string x that is shorter than w, we have $\delta^*(s, x) = s$ and $\delta^*(t, x) = t$ if x has an even number of 1s, and $\delta^*(s, x) = t$ and $\delta^*(t, x) = s$ if x has an odd number of 1s. There are five cases to consider.

- If $w = \varepsilon$, then w contains an even number of 1s and $\delta^*(s, w) = s$ and $\delta^*(t, w) = t$ by definition.

- Suppose $w = 1x$ and $\#(1, w)$ is even. Then $\#(1, x)$ is odd, which implies

$$\begin{aligned} \delta^*(s, w) &= \delta^*(\delta(s, 1), x) && \text{by definition of } \delta^* \\ &= \delta^*(t, x) && \text{by definition of } \delta \\ &= s && \text{by the inductive hypothesis} \end{aligned}$$

$$\begin{aligned} \delta^*(t, w) &= \delta^*(\delta(t, 1), x) && \text{by definition of } \delta^* \\ &= \delta^*(s, x) && \text{by definition of } \delta \\ &= T && \text{by the inductive hypothesis} \end{aligned}$$

Since the remaining cases are similar, I'll omit the line-by-line justification.

- If $w = 1x$ and $\#(1, w)$ is odd, then $\#(1, x)$ is even, so the inductive hypothesis implies

$$\delta^*(s, w) = \delta^*(\delta(s, 1), x) = \delta^*(t, x) = t$$
$$\delta^*(t, w) = \delta^*(\delta(t, 1), x) = \delta^*(s, x) = s$$

- If $w = 0x$ and $\#(1, w)$ is even, then $\#(1, x)$ is even, so the inductive hypothesis implies

$$\delta^*(s, w) = \delta^*(\delta(s, 0), x) = \delta^*(s, x) = s$$
$$\delta^*(t, w) = \delta^*(\delta(t, 0), x) = \delta^*(t, x) = t$$

- Finally, if $w = 0x$ and $\#(1, w)$ is odd, then $\#(1, x)$ is odd, so the inductive hypothesis implies

$$\delta^*(s, w) = \delta^*(\delta(s, 0), x) = \delta^*(s, x) = t$$
$$\delta^*(t, w) = \delta^*(\delta(t, 0), x) = \delta^*(t, x) = s \qquad \square$$

Notice that this proof contains $|Q|^2 \cdot |\Sigma| + |Q|$ separate inductive arguments. For every pair of states p and q, we must argue about the language of all strings w such that $\delta^*(p, w) = q$, and we must consider every possible first symbol in w. We must also argue about $\delta(p, \varepsilon)$ for every state p. Each of those arguments is typically straightforward, but it's easy to get lost in the deluge of cases.

For this particular proof, however, we can reduce the number of cases by switching from tail recursion to *head* recursion. The following identity holds for all strings $x \in \Sigma^*$ and symbols $a \in \Sigma$:

$$\boxed{\delta^*(q, xa) = \delta(\delta^*(q, x), a)}$$

We leave the inductive proof of this identity as a straightforward exercise (hint, hint).

Proof (clever renaming, head induction): Let's rename the states with the integers 0 and 1 instead of s and t. Then the transition function can be described concisely as $\boldsymbol{\delta(q, a) = (q + a) \bmod 2}$. We claim that for every string w, we have $\delta^*(0, w) = \#(1, w) \bmod 2$.

Let w be an arbitrary string, and assume that for any string x that is shorter than w that $\delta^*(0, x) = \#(1, x) \bmod 2$. There are only two cases to consider: either w is empty or it isn't.

- If $w = \varepsilon$, then $\delta^*(0, w) = 0 = \#(1, w) \bmod 2$ by definition.

- Otherwise, $w = xa$ for some string x and some symbol a, and we have

$$\begin{aligned}
\delta^*(0, w) &= \delta(\delta^*(0, x), a) && \text{by definition of } \delta^* \\
&= \delta(\#(1, x) \bmod 2, a) && \text{by the inductive hypothesis} \\
&= (\#(1, x) \bmod 2 + a) \bmod 2 && \text{by definition of } \delta \\
&= (\#(1, x) + a) \bmod 2 && \text{by definition of mod 2} \\
&= (\#(1, x) + \#(1, a)) \bmod 2 && \text{because } \#(1, 0) = 0 \text{ and } \#(1, 1) = 1 \\
&= (\#(1, xa)) \bmod 2 && \text{by definition of } \# \\
&= (\#(1, w)) \bmod 2 && \text{because } w = xa \qquad \square
\end{aligned}$$

Hmmm. This "clever" proof is certainly shorter than the earlier brute-force proof, but is it actually *better*? Simpler? More intuitive? Easier to understand? I'm skeptical. Sometimes brute force really is more effective.

3.4 Real-World Examples

Finite-state machines were first formally defined in the mid-20th century, but people have been building automata for centuries, if not millennia. Many of the earliest records about automata are clearly mythological—for example, the brass giant Talus created by Hephaestus to guard Crete against intruders—but others are more believable, such as King-Shu's construction of a flying magpie from wood and bamboo in China around 500BCE.

Perhaps the most common examples of finite-state automata are ***clocks***. For example, the Swiss railway clock designed by Hans Hilfiker in 1944 has hour and minute hands that can indicate any time between 1:00 and 12:59. The minute hands advance discretely once per minute when they receive an electrical signal from a central master clock.[2] Thus, a Swiss railway clock is a finite-state machine with 720 states, one input symbol, and a simple transition function:

$$\begin{aligned}
Q &= \{(h,m) \mid 0 \le h 11 \text{ and } 0 \le m \le 59\}\\
\Sigma &= \{\mathsf{tick}\}\\
\delta((h,m),\mathsf{tick}) &= \begin{cases} (h, m+1) & \text{if } m < 59\\ (h+1, 0) & \text{if } h < 11 \text{ and } m = 59\\ (0,0) & \text{if } h = 11 \text{ and } m = 59 \end{cases}
\end{aligned}$$

This clock doesn't *quite* match our abstraction, because there's no "start" state or "accepting" states, unless perhaps you consider the "accepting" state to be the time when your train arrives.

Three finite-state machines.

A more playful example of a finite-state machine is the ***Rubik's cube***, a well-known mechanical puzzle invented independently by Ernő Rubik in Hungary and Terutoshi Ishigi in Japan in the mid-1970s. This puzzle has precisely 519,024,039,293,878,272,000 distinct configurations. In the unique *solved* configuration, each of the six faces of the cube shows exactly one color. We can change the configuration of the cube by rotating one of the six faces of the cube by 90 degrees, either clockwise or counterclockwise. The cube has six faces (front, back, left, right, up, and down), so there are exactly twelve possible turns, typically represented by the symbols $\mathsf{R}, \mathsf{L}, \mathsf{F}, \mathsf{B}, \mathsf{U}, \mathsf{D}, \bar{\mathsf{R}}, \bar{\mathsf{L}}, \bar{\mathsf{F}}, \bar{\mathsf{B}}, \bar{\mathsf{U}}, \bar{\mathsf{D}}$, where the letter indicates which face to turn and the presence or absence of a bar over the letter indicates turning counterclockwise or clockwise, respectively. Thus, we can represent a Rubik's cube as a finite-state machine with 519,024,039,293,878,272,000 states and an input alphabet with 12 symbols; or equivalently, as a directed graph with 519,024,039,293,878,272,000 vertices,

[2]A second hand was added to the Swiss Railway clocks in the mid-1950s, which sweeps continuously around the clock in approximately 58½ seconds and then pauses at 12:00 until the next minute signal "to bring calm in the last moment and ease punctual train departure". Let's ignore that.

each with 12 outgoing edges. In practice, the number of states is *far* too large for us to actually draw the machine or explicitly specify its transition function; nevertheless, the number of states is still finite. If we let the start state s and the sole accepting state be the solved state, then the language of this finite state machine is the set of all move sequences that leave the cube unchanged.

3.5 A Brute-Force Design Example

As usual in algorithm design, there is no purely mechanical recipe—no *automatic* method—no *algorithm*—for building DFAs in general. Here I'll describe one systematic approach that works reasonably well, although it tends to produce DFAs with many more states than necessary.

3.5.1 DFAs are Algorithms

The basic approach is to try to *construct an algorithm* that looks like MultipleOf5: A simple for-loop through the symbols, using a *constant* number of variables, where each variable (except the loop index) has only a *constant* number of possible values. Here, "constant" means an actual number that is not a function of the input size n. You should be able to compute the number of possible values for each variable *at compile time*.

For example, the following algorithm determines whether a given string in $\Sigma = \{0, 1\}$ contains the substring 11.

```
Contains11($w[1..n]$):
  $found \leftarrow$ False
  for $i \leftarrow 1$ to $n$
      if $i = 1$
          $last2 \leftarrow w[1]$
      else
          $last2 \leftarrow w[i-1] \cdot w[i]$
      if $last2 = 11$
          $found \leftarrow$ True
  return $found$
```

Aside from the loop index, this algorithm has exactly two variables.

- A boolean flag *found* indicating whether we have seen the substring 11. This variable has exactly two possible values: True and False.
- A string *last2* containing the last (up to) three symbols we have read so far. This variable has exactly 7 possible values: ε, 0, 1, 00, 01, 10, and 11.

Thus, altogether, the algorithm can be in at most $2 \times 7 = 14$ possible states, one for each possible pair $(found, last2)$. Thus, we can encode the behavior of Contains11 as a DFA with fourteen states, where the start state is $(\text{False}, \varepsilon)$ and the accepting states are all seven states of the form $(\text{True}, *)$. The transition function is described in the following table (split into two parts to save space):

q	$\delta[q,0]$	$\delta[q,1]$
(False, ε)	(False, 0)	(False, 1)
(False, 0)	(False, 00)	(False, 01)
(False, 1)	(False, 10)	(**True**, 11)
(False, 00)	(False, 00)	(False, 01)
(False, 01)	(False, 10)	(**True**, 11)
(False, 10)	(False, 00)	(False, 01)
(False, 11)	(False, 10)	(**True**, 11)

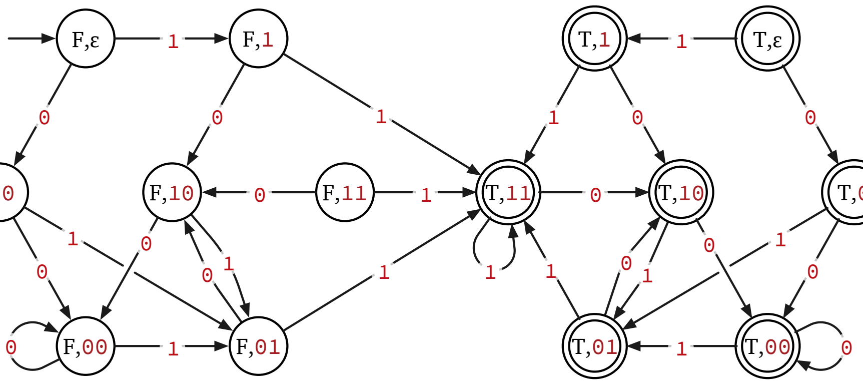
q	$\delta[q,0]$	$\delta[q,1]$
(True, ε)	(True, 0)	(True, 1)
(True, 0)	(True, 00)	(True, 01)
(True, 1)	(True, 10)	(True, 11)
(True, 00)	(True, 00)	(True, 01)
(True, 01)	(True, 10)	(True, 11)
(True, 10)	(True, 00)	(True, 01)
(True, 11)	(True, 10)	(True, 11)

For example, given the input string 10010$\underline{11}$100, this DFA performs the following sequence of transitions and then accepts.

$$(\textsc{False},\varepsilon) \xrightarrow{1} (\textsc{False},1) \xrightarrow{0} (\textsc{False},10) \xrightarrow{0} (\textsc{False},00) \xrightarrow{1}$$
$$(\textsc{False},01) \xrightarrow{0} (\textsc{False},10) \xrightarrow{1} (\textsc{False},01) \xrightarrow{1}$$
$$(\textsc{True},11) \xrightarrow{1} (\textsc{True},11) \xrightarrow{0} (\textsc{True},10) \xrightarrow{0} (\textsc{True},00)$$

3.5.2 . . . but Algorithms can be Wasteful

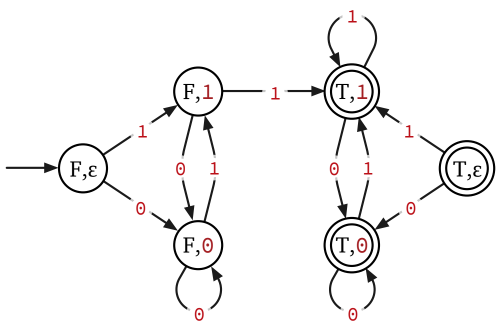
You can probably guess that the brute-force DFA we just constructed has considerably more states than necessary, especially after seeing its transition graph:



Our brute-force DFA for strings containing the substring 11

For example, the state (False, 11) has no incoming transitions, so we can just delete it. (This state would indicate that we’ve never read 11, but the last two symbols we read were 11, which is impossible!) More significantly, we don’t need actually to remember both of the last two symbols, but only the penultimate symbol, because the last symbol is the one we’re currently reading. This observation allows us to reduce the number of states from fourteen to only six.

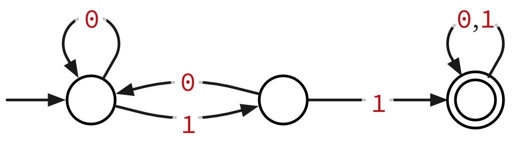
But even this DFA has more states than necessary. Once the flag part of the state is set to True, we know the machine will eventually accept, so we might as well merge all the accepting states together. More subtly, because both transitions out of (False, 0) and (False, ε) lead to the same states, we can merge those two states together as well. After all these optimizations, we obtain the following DFA with just three states:



A less brute-force DFA for strings containing the substring 11

- The start state, which indicates that the machine has not read the substring 11 and did not just read the symbol 1.
- An intermediate state, which indicates that the machine has not read the substring 11 but just read the symbol 1.
- A unique accept state, which indicates that the machine has read the substring 11.

This is the smallest possible DFA for this language.



A minimal DFA for superstrings of 11

While it is important not to use an excessive number of states when we design DFAs—too many states makes a DFA hard to understand—there is really no point in trying to reduce DFAs *by hand* to the absolute minimum number of states. Clarity is much more important than brevity (especially in this class), and DFAs with too *few* states can *also* be hard to understand. At the end of this note, I'll describe an efficient algorithm that automatically transforms any given DFA into an equivalent DFA with the fewest possible states.

3.6 Combining DFAs: The Product Construction

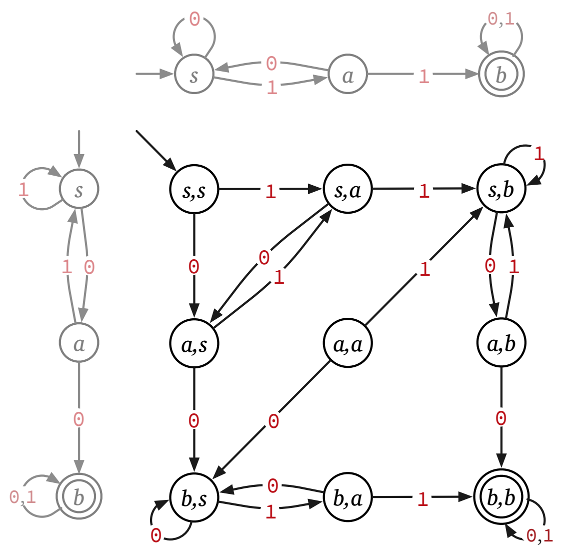
Now suppose we want to accept all strings that contain both 00 and 11 as substrings, in either order. Intuitively, we'd like to run two DFAs in parallel—the DFA M_{00} to detect superstrings of 00 and a similar DFA M_{11} obtained from M_{00} by swapping $0 \longleftrightarrow 1$ everywhere—and then accept the input string if and only if *both* of these DFAs accept.

In fact, we can encode precisely this "parallel computation" into a single DFA using the following *product construction* first proposed by Edward Moore in 1956:

- The states of the new DFA are all ordered pairs (p,q), where p is a state in M_{00} and q is a state in M_{11}.
- The start state of the new DFA is the pair (s,s'), where s is the start state of M_{00} and s' is the start state of M_{11}.
- The new DFA includes the transition $(p,q) \xrightarrow{a} (p',q')$ if and only if M_{00} contains the transition $p \xrightarrow{a} p'$ and M_{11} contains the transition $q \xrightarrow{a} q'$.

- Finally, (p,q) is an accepting state of the new DFA if and only if p is an accepting state in M_{00} and q is an accepting state in M_{11}.

The resulting nine-state DFA is shown on the next page, with the two factor DFAs M_{00} and M_{11} shown in gray for reference. (The state (a,a) can be removed, because it has no incoming transition, but let's not worry about that now.)



Building a DFA for the language of strings containing both 00 and 11.

More generally, let $M_1 = (\Sigma, Q_1, \delta_1, s_1, A_1)$ be an arbitrary DFA that accepts some language L_1, and let $M_2 = (\Sigma, Q_2, \delta_2, s_2, A_2)$ be an arbitrary DFA that accepts some language L_2 (over the same alphabet Σ). We can construct a third DFA $M = (\Sigma, Q, \delta, s, A)$ that accepts the intersection language $L_1 \cap L_2$ as follows.

$$\begin{aligned} Q &:= Q_1 \times Q_2 = \{(p,q) \mid p \in Q_1 \text{ and } q \in Q_2\} \\ \delta((p,q),a) &:= \big(\delta_1(p,a),\ \delta_2(q,a)\big) \\ s &:= (s_1, s_2) \\ A &:= A_1 \times A_2 = \{(p,q) \mid p \in A_1 \text{ and } q \in A_2\} \end{aligned}$$

To convince ourselves that this product construction is actually correct, let's consider the extended transition function $\delta^* \colon (Q \times Q') \times \Sigma^* \to (Q \times Q')$, which acts on strings instead of individual symbols. Recall that this function is defined recursively as follows:

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

This function behaves exactly as we should expect:

Lemma 3.1. *$\delta^*((p,q),w) = \big(\delta_1^*(p,w),\ \delta_2^*(q,w)\big)$ for any string w.*

Proof: Let w be an arbitrary string. Assume $\delta^*((p,q),x) = \big(\delta_1^*(p,x),\ \delta_2^*(q,x)\big)$ for every string x that is shorter than w. As usual, there are two cases to consider.

- First suppose $w = \varepsilon$:

$$\begin{aligned}
\delta^*\big((p,q),\varepsilon\big) &= (p,q) && \text{by the definition of } \delta^* \\
&= \big(\delta_1^*(p,\varepsilon),\, q\big) && \text{by the definition of } \delta_1^* \\
&= \big(\delta_1^*(p,e),\, \delta_2^*(q,\varepsilon)\big) && \text{by the definition of } \delta_2^*
\end{aligned}$$

- Now suppose $w = ax$ for some symbol a and some string x:

$$\begin{aligned}
\delta^*\big((p,q),ax\big) &= \delta^*\big(\delta((p,q),a),\ x\big) && \text{by the definition of } \delta^* \\
&= \delta^*\big((\delta_1(p,a),\, \delta_2(q,a)),\ x\big) && \text{by the definition of } \delta \\
&= \big(\delta_1^*((\delta_1(p,a),\, x),\ \delta_2^*(\delta_2(q,a),x)\big) && \text{by the induction hypothesis} \\
&= \big(\delta_1^*(p,ax),\ \delta_2^*(q,ax)\big) && \text{by the definitions of } \delta_1^* \text{ and } \delta_2^*.
\end{aligned}$$

In both cases, we conclude that $\delta^*((p,q),w) = \big(\delta_1^*(p,w),\, \delta_2^*(q,w)\big)$. □

An immediate consequence of this lemma is that for every string w, we have $\delta^*(s,w) \in A$ if and only if both $\delta_1^*(s_1,w) \in A_1$ and $\delta_2^*(s_2,w) \in A_2$. In other words, M accepts w if and only if ***both*** M_1 accepts w ***and*** M_2 accept w, as required.

As usual, this construction technique does not necessarily yield *minimal* DFAs. For example, in our first example of a product DFA, illustrated above, the central state (a,a) cannot be reached by any other state and is therefore redundant. Whatever.

Similar product constructions can be used to build DFAs that accept any other boolean combination of languages; in fact, the only part of the construction that changes is the choice of accepting states. For example:

- To accept the union $L_1 \cup L_2$, define $A = \big\{(p,q) \bigm| p \in A_1 \textbf{\textit{ or }} q \in A_2\big\}$.
- To accept the difference $L_1 \setminus L_2$, define $A = \big\{(p,q) \bigm| p \in A_1 \textbf{\textit{ but }} q \notin A_2\big\}$.
- To accept the symmetric difference $L_1 \oplus L_2$, define $A = \big\{(p,q) \bigm| p \in A_1 \textbf{\textit{ xor }} q \in A_2\big\}$.

Examples of these constructions are shown on the next page.

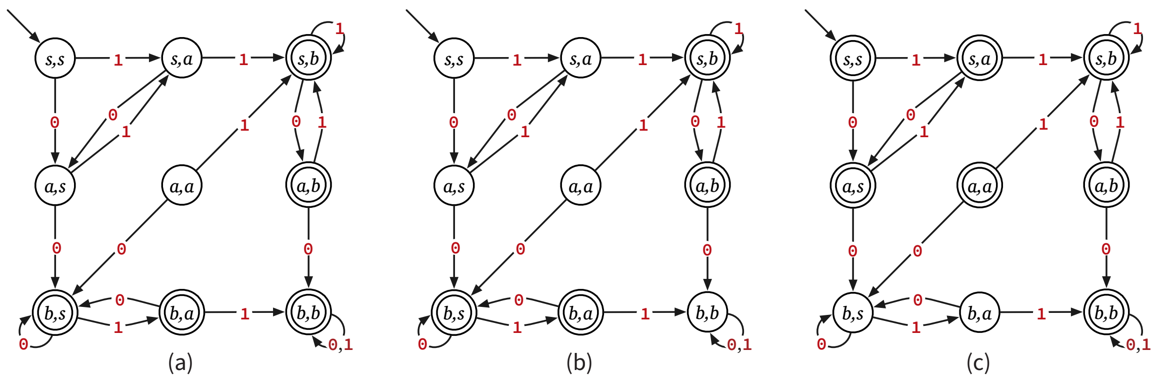
Moreover, by cascading this product construction, we can construct DFAs that accept arbitrary boolean combinations of arbitrary finite collections of regular languages.

3.7 Automatic Languages and Closure Properties

The ***language*** of a finite state machine M, denoted $\boldsymbol{L(M)}$, is the set of all strings in Σ^* that M accepts. More formally, if $M = (\Sigma, Q, \delta, s, A)$, then

$$L(M) \ :=\ \big\{w \in \Sigma^* \bigm| \delta^*(s,w) \in A\big\}.$$

We call a language ***automatic*** if it is the language of some finite state machine. Our product construction examples let us prove that the set of automatic languages is ***closed*** under simple boolean operations.



DFAs for (a) strings that contain `00` or `11`, (b) strings that contain either `00` or `11` but not both, and (c) strings that contain `11` if they contain `00`. These DFAs are identical except for their choices of accepting states.

Theorem 3.2. *Let L and L' be arbitrary automatic languages over an arbitrary alphabet Σ.*

- *$\overline{L} = \Sigma^* \setminus L$ is automatic.*
- *$L \cup L'$ is automatic.*
- *$L \cap L'$ is automatic.*
- *$L \setminus L'$ is automatic.*
- *$L \oplus L'$ is automatic.*

Eager students may have noticed that a Google search for the phrase "automatic language" turns up ***no*** results that are relevant for this class, except perhaps this lecture note. That's because "automatic" is just a synonym for "regular"! This equivalence was first observed by Stephen Kleene (the inventor of regular expressions) in 1956.

Theorem 3.3 (Kleene). *For any regular expression R, there is a DFA M such that $L(R) = L(M)$. For any DFA M, there is a regular expression R such that $L(M) = L(R)$.*

Unfortunately, we don't yet have all the tools we need to prove Kleene's theorem; we'll return to the proof in the next lecture note, after we have introduced *nondeterministic* finite-state machines. The proof is actually constructive—there are explicit algorithms that transform arbitrary DFAs into equivalent regular expressions and vice versa.[3]

This equivalence between regular and automatic languages implies that the set of ***regular*** languages is also closed under simple boolean operations. The union of two regular languages is regular *by definition*, but it's much less obvious that *every* boolean combination of regular languages can also be described by regular expressions.

Corollary 3.4. *Let L and L' be arbitrary* **regular** *languages over an arbitrary alphabet Σ.*

- *$\overline{L} = \Sigma^* \setminus L$ is regular.*
- *$L \cap L'$ is regular.*
- *$L \setminus L'$ is regular.*
- *$L \oplus L'$ is regular.*

Conversely, because concatenations and Kleene closures of regular languages are regular *by definition*, we can immediately conclude that concatenations and Kleene closures of automatic languages are automatic.

[3]These conversion algorithms run in exponential time in the worst case, but that's unavoidable. There are regular languages whose smallest accepting DFA is exponentially larger than their smallest regular expression, and there are regular languages whose smallest regular expression is exponentially larger than their smallest accepting DFA.

Corollary 3.5. *Let L and L' be arbitrary automatic languages.*

- *$L \bullet L'$ is automatic.*
- *L^* is automatic.*

These results give us several options to prove that a given languages is regular or automatic. We can either (1) build a regular expression that describes the language, (2) build a DFA that accepts the language, or (3) build the language from simpler pieces from other regular/automatic languages. (Later we'll see a fourth option, and possibly even a fifth.)

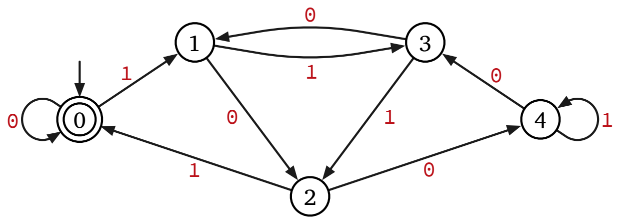
3.8 Proving a Language is Not Regular

But now suppose we're faced with a language L where none of these techniques seem to work. How would we prove L is *not* regular? By Theorem 3.3, it suffices to prove that there is no finite-state automaton that accepts L. Equivalently, we need to prove that any automaton that accepts L requires infinitely many states. That may sound tricky, what with the "infinitely many", but there's actually a fairly simple technique to prove exactly that.

3.8.1 Distinguishing Suffixes

Perhaps the single most important feature of DFAs is that they have no memory other than the current state. Once a DFA enters a particular state, all *future* transitions depend only on that state and *future* input symbols; *past* input symbols are simply forgotten.

For example, consider our very first DFA, which accepts the binary representations of integers divisible by 5.



DFA accepting binary multiples of 5.

The strings `0010` and `11011` both lead this DFA to state 2, although they follow different transitions to get there. Thus, for any string z, the strings `0010`z and `11011`z also lead to the same state in this DFA. In particular, `0010`z leads to the accepting state if and only if `11011`z leads to the accepting state. It follows that `0010`z is divisible by 5 if and only if `11011`z is divisible by 5.

More generally, any DFA $M = (\Sigma, Q, s, A, \delta)$ defines an equivalence relation over Σ^*, where two strings x and y are equivalent if and only if they lead to the same state, or more formally, if $\delta^*(s,x) = \delta^*(s,y)$. If x and y are equivalent strings, then for any string z, the strings xz and yz are also equivalent. In particular, M accepts xz if and only if M accepts yz. Thus, if L is the language accepted by M, then $xz \in L$ if and only if $yz \in L$. In short, if the *machine* can't distinguish between x and y, then the *language* can't distinguish between xz and yz for any suffix z.

Now let's turn the previous argument on its head. Let L be an arbitrary language, and let x and y be arbitrary strings. A ***distinguishing suffix*** for x and y (with respect to L) is a third string z such that *exactly one* of the strings xz and yz is in L. If x and y have a distinguishing suffix z, then in *any* DFA that accepts L, the strings xz and yz must lead to different states, and therefore the strings x and y must lead to different states!

For example, let L_5 denote the the set of all strings over $\{\texttt{0},\texttt{1}\}$ that represent multiples of 5 in binary. Then the strings $x = \texttt{01}$ and $y = \texttt{0011}$ are distinguished by the suffix $z = \texttt{01}$:

$$xz = \texttt{01} \bullet \texttt{01} = \texttt{0101} \in L_5 \qquad \text{(because } \texttt{0101}_2 = 5\text{)}$$
$$yz = \texttt{0011} \bullet \texttt{01} = \texttt{001101} \notin L_5 \qquad \text{(because } \texttt{001101}_2 = 13\text{)}$$

It follows that in ***every*** DFA that accepts L_5, the strings `01` and `0011` lead to different states. Moreover, since neither `01` nor `0011` belong to L_5, every DFA that accepts L_5 must have at least two *non-accepting* states, and therefore at least three states overall.

3.8.2 Fooling Sets

A ***fooling set*** for a language L is a set F of strings such that *every* pair of strings in F has a distinguishing suffix. For example, $F = \{\texttt{0},\texttt{1},\texttt{10},\texttt{11},\texttt{100}\}$ is a fooling set for the language L_5 of binary multiples of 5, because each pair of strings in F has a distinguishing suffix:

- `0` distinguishes `0` and `1`;
- `0` distinguishes `0` and `10`;
- `0` distinguishes `0` and `11`;
- `0` distinguishes `0` and `100`;
- `1` distinguishes `1` and `10`;
- `01` distinguishes `1` and `11`;
- `01` distinguishes `1` and `100`;
- `1` distinguishes `10` and `11`;
- `1` distinguishes `10` and `100`;
- `11` distinguishes `11` and `100`.

So in for *every* DFA M that accepts L_5, each pair of these five strings leads to two different states. In other words, each string leads M to a different state. It follows that *every* DFA that accepts the language L_5 has at least five states. And hey, look, we already have a DFA for L_5 with five states, so that's the best we can do!

More generally, for *every* language L, and for *every* fooling set F for L, *every* DFA that accepts L must have at least $|F|$ states.

Theorem 3.6. *Let L be an arbitrary language, let M be an arbitrary DFA that accepts L, and let F be an arbitrary fooling set for L. The number of states in M is greater than or equal to the number of strings in F.*

This theorem has a very important consequence: Suppose we can find an *infinite* fooling set for some language L. Then *every* DFA that accepts L must have an *infinite* number of states. But there's no such thing as a ***finite***-state machine with an ***infinite*** number of states! So it's impossible for any DFA to accept L. In other words, L is not regular.

$$\boxed{\textit{If L has an infinite fooling set, then L is not regular.}}$$

Finding an infinite fooling set is arguably both the simplest and most powerful method for proving that a language is non-regular.

3.8.3 Examples

Here are a few canonical examples of the fooling-set technique in action. I will write out the first proof in more explicit detail. I strongly recommend following the line-by-line format of the first two proofs in your homeworks and exams.

Lemma 3.7. *The language $L = \{0^n1^n \mid n \geq 0\}$ is not regular.*

Proof: Consider the infinite set $F = \{0^n \mid n \geq 0\}$, or more simply $F = 0^*$.
Let x and y be arbitrary distinct strings in F.
The definition of F implies $x = 0^i$ and $y = 0^j$ for some integers $i \neq j$.
Let z be the string 1^i.
Then $xz = 0^i1^i \in L$.
But $yz = 0^j1^i \notin L$, because $i \neq j$.
So z is a distinguishing suffix for x and y.
Because x and y are *arbitrary* strings in F, *every* pair of strings in F has a distinguishing suffix.
In other words, F is a fooling set for L.
In fact, F is an *infinite* fooling set for L.
We conclude that L cannot be regular. □

Lemma 3.8. *The language $L = \{ww^R \mid w \in \Sigma^*\}$ of even-length palindromes is not regular.*

Proof: Let F denote the infinite set 0^*1.
Let x and y be arbitrary distinct strings in F.
We must have $x = 0^i1$ and $y = 0^j1$ for some integers $i \neq j$.
Let $z = 10^i$.
Then $xz = 0^i110^i \in L$.
But $yz = 0^i110^j \notin L$, because $i \neq j$.
We conclude that F is a fooling set for L.
Because F is infinite, L cannot be regular. □

Lemma 3.9. *The language $L = \{0^{2^n} \mid n \geq 0\}$ is not regular.*

Proof ($F = L$): Let x and y be arbitrary distinct strings in L. Then we must have $x = 0^{2^i}$ and $y = 0^{2^j}$ for some integers $i \neq j$. The suffix $z = 0^{2^i}$ distinguishes x and y, because $xz = 0^{2^i+2^i} = 0^{2^{i+1}} \in L$, but $yz = 0^{2^i+2^j} \notin L$. We conclude that L itself is a fooling set for L. Because L is infinite, L cannot be regular. □

Proof ($F = 0^*$): Let x and y be arbitrary distinct strings in 0^*. Then we must have $x = 0^i$ and $y = 0^j$ for some integers $i \neq j$. Without loss of generality, assume $i < j$; otherwise swap the variable. Let k be any positive integer such that $2^k > j$. Consider the suffix $z = 0^{2^k-i}$. We have $xz = 0^{i+(2^k-i)} = 0^{2^k} \in L$, but $yz = 0^{j+(2^k-i)} = 0^{2^k-i+j} \notin L$, because

$$2^k < 2^k - i + j < 2^k + j < 2^k + 2^k = 2^{k+1}.$$

Thus, z is a distinguishing suffix for x and y. We conclude that 0^* is a fooling set for L. Because L is infinite, L cannot be regular. □

Proof ($F = \mathtt{0}^*$ again): Let x and y be arbitrary distinct strings in $\mathtt{0}^*$. Then we must have $x = \mathtt{0}^i$ and $y = \mathtt{0}^j$ for some integers $i \neq j$; without loss of generality, assume $i < j$. Let k be any positive integer such that $2^{k-1} > j$. Consider the suffix $z = \mathtt{0}^{2^k - j}$. We have $xz = \mathtt{0}^{i+(2^k-j)} = \mathtt{0}^{2^k-j+i} \notin L$, because

$$2^{k-1} < 2^k - 2^{k-1} + i < 2^k - j + i < 2^k.$$

On the other hand, $yz = \mathtt{0}^{j+(2^k-j)} = \mathtt{0}^{2^k} \in L$. Thus, z is a distinguishing suffix for x and y. We conclude that $\mathtt{0}^*$ is a fooling set for L. Because L is infinite, L cannot be regular. □

The previous examples show the flexibility of this proof technique; a single non-regular language can have many different infinite fooling sets,[4] and each pair of strings in any fooling set can have many different distinguishing suffixes. Fortunately, we only have to find *one* infinite set F and *one* distinguishing suffix for each pair of strings in F.

Lemma 3.10. *The language $L = \{\mathtt{0}^p \mid p \text{ is prime}\}$ is not regular.*

Proof ($F = \mathtt{0}^*$): Again, we use $\mathtt{0}^*$ as our fooling set, but but the actual argument is somewhat more complicated than in our earlier examples.

Let x and y be arbitrary distinct strings in $\mathtt{0}^*$. Then we must have $x = \mathtt{0}^i$ and $y = \mathtt{0}^j$ for some integers $i \neq j$; without loss of generality, assume that $i < j$. Let p be any prime number larger than i. Because $p + 0(j-i)$ is prime and $p + p(j-i) > p$ is not, there must be a positive integer $k \le p$ such that $p + (k-1)(j-i)$ is prime but $p + k(j-i)$ is not. Then I claim that the suffix $z = \mathtt{0}^{p+(k-1)j-ki}$ distinguishes x and y:

$$xz = \mathtt{0}^i\,\mathtt{0}^{p+(k-1)j-ki} = \mathtt{0}^{p+(k-1)(j-i)} \in L \qquad \text{because } p + (k-1)(j-i) \text{ is prime;}$$
$$yz = \mathtt{0}^j\,\mathtt{0}^{p+(k-1)j-ki} = \mathtt{0}^{p+k(j-i)} \notin L \qquad \text{because } p + k(j-i) \text{ is not prime.}$$

(Because $i < j$ and $i < p$, the suffix $\mathtt{0}^{p+(k-1)j-ki} = \mathtt{0}^{(p-i)+(k-1)(j-i)}$ has positive length and therefore *actually exists!*) We conclude that $\mathtt{0}^*$ is indeed a fooling set for L, which implies that L is not regular. □

Proof ($F = L$): Let x and y be arbitrary distinct strings in L. Then we must have $x = \mathtt{0}^p$ and $y = \mathtt{0}^q$ for some primes $p \neq q$; without loss of generality, assume $p < q$.

Now consider strings of the form $\mathtt{0}^{p+k(q-p)}$. Because $p + 0(q-p)$ is prime and $p + p(q-p) > p$ is not prime, there must be a non-negative integer $k < p$ such that $p + k(p-q)$ is prime but $p + (k+1)(p-q)$ is not prime. I claim that the suffix $z = \mathtt{0}^{k(q-p)}$ distinguishes x and y:

$$xz = \mathtt{0}^p\,\mathtt{0}^{k(q-p)} = \mathtt{0}^{p+k(p-q)} \in L \qquad \text{because } p + k(p-q) \text{ is prime;}$$
$$yz = \mathtt{0}^q\,\mathtt{0}^{k(q-p)} = \mathtt{0}^{p+(k+1)(q-p)} \notin L \qquad \text{because } p + (k+1)(p-q) \text{ is not prime.}$$

We conclude that L is a fooling set for itself!! Because L is infinite, L cannot be regular! □

3.8.4 Choosing the Fooling Set

Obviously the most difficult part of this technique is coming up with an appropriate fooling set. Fortunately, *most* languages L—in particular, almost all languages that students are asked to prove non-regular on homeworks or exams—fall into one of two categories:

[4] At some level, this observation is trivial. If F is an infinite fooling set for L, then every infinite subset of F is also an infinite fooling set for L!

- Some simple regular language like $\mathtt{0}^*$ or $\mathtt{10}^*\mathtt{1}$ or $(\mathtt{01})^*$ is a fooling set for L. In particular, the fooling set is a regular language with exactly one Kleene star and no $+$.
- The language L itself is a fooling set for L.

The most important point to remember is that ***you choose*** the fooling set F, and you can use that fooling set to effectively impose additional structure on the language L.

★★★

I'm not sure yet how to express this effectively, but here is some more intuition about choosing fooling sets and distinguishing suffixes.

As a sanity check, try to write an *algorithm* to recognize strings in L, as described at the start of this note, where the only variable that can take on an unbounded number of values is the loop index i. (I should probably rewrite that template as a while-loop or tail recursion, but anyway....) If you succeed, the language is regular. But if you fail, it's probably because there are counters of string variables that you can't get rid of. ***One of those unavoidable counters is the basis for your fooling set.***

For example, any algorithm that recognizes the language $\{\mathtt{0}^n\mathtt{1}^n\mathtt{2}^n \mid n \geq 0\}$ "obviously" has to count $\mathtt{0}$s and $\mathtt{1}$s in the input string. (We can avoid counting $\mathtt{2}$s by decrementing the $\mathtt{0}$ counter.) Because the $\mathtt{0}$s come first in the string, this intuition suggests using strings of the form $\mathtt{0}^n$ as our fooling set and matching strings of the form $\mathtt{1}^n\mathtt{2}^n$ as distinguishing suffixes. (This is a rare example of an "obvious" fact that is actually true.)

★★★

It's also important to remember that when you choose the fooling set, you can effectively impose additional structure that isn't present in the language already. For example, to prove that the language $L = \{w \in (\mathtt{0}+\mathtt{1})^* \mid \#(\mathtt{0}, w) = (\mathtt{1}, w)\}$ is not regular, we can use strings of the form $\mathtt{0}^n$ as our fooling set and matching strings of the form $\mathtt{1}^n$ as distinguishing suffixes, ***exactly*** as we did for $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$. The fact that L contains strings that start with $\mathtt{1}$ is irrelevant. There may be more equivalence classes that our proof doesn't find, but since we found an infinite set of equivalence class, we don't care.

At some level, this fooling set proof is implicitly considering the simpler language $L \cap \mathtt{0}^*\mathtt{1}^* = \{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$. If L were regular, then $L \cap \mathtt{0}^*\mathtt{1}^*$ would also be regular, because regular languages are closed under intersection.

★★★

Finally, every non-regular language has infinitely many infinite fooling sets, but some of these are easier to prove than others. In particular, the language $\mathtt{0}^*$ works as a fooling set surprisingly often, but finding distinguishing suffixes for any two strings $\mathtt{0}^i$ and $\mathtt{0}^j$ is often more difficult, either conceptually or just because of additional case work, than finding distinguishing suffixes for strings in some other fooling set with more structure.

*3.9 The Myhill-Nerode Theorem

The fooling set technique implies a *necessary* condition for a language to be accepted by a DFA—the language must have no infinite fooling sets. In fact, this condition is also *sufficient*. The following powerful theorem was first proved by Anil Nerode in 1958, strengthening a 1957 result of John Myhill.[5] We write $x \equiv_L y$ if $xz \in L \iff yz \in L$ for all strings z.

[5]Myhill considered the finer equivalence relation $x \sim_L y$, meaning $wxz \in L$ if and only if $wyz \in L$ for all strings w and z, and proved that L is regular if and only if $\sim_L$ defines a finite number of equivalence classes. Like most of Myhill's early automata research, this result appears in an unpublished Air Force technical report. The modern Myhill-Nerode theorem appears (in an even more general form) as a minor lemma in Nerode's 1958 paper, which (not surprisingly) does not cite Myhill.

The Myhill-Nerode Theorem. *For any language L, the following are equal:*
(a) the minimum number of states in a DFA that accepts L,
(b) the maximum size of a fooling set for L, and
(c) the number of equivalence classes of $\equiv_L$.
In particular, L is accepted by a DFA if and only if every fooling set for L is finite.

Proof: Let L be an arbitrary language.

We have already proved that the size of any fooling set for L is at most the number of states in any DFA that accepts L, so (a)≥(b). It also follows directly from the definitions that $F \subseteq \Sigma^*$ is a fooling set for L if and only if F contains at most one string in each equivalence class of $\equiv_L$; thus, (b)=(c). We complete the proof by showing that (a)≤(c).

We have already proved that if $\equiv_L$ has an infinite number of equivalence classes, there is no DFA that accepts L, so assume that the number of equivalence classes is finite. For any string w, let $[w]$ denote its equivalence class. We define a DFA $M_\equiv = (\Sigma, Q, s, A, \delta)$ as follows:

$$\begin{aligned} Q &:= \big\{[w] \bigm| w \in \Sigma^*\big\} \\ s &:= [\varepsilon] \\ A &:= \big\{[w] \bigm| w \in L\big\} \\ \delta([w], a) &:= [w \bullet a] \end{aligned}$$

We claim that this DFA accepts the language L; this claim completes the proof of the theorem.

But before we can prove anything about this DFA, we first need to verify that it is actually well-defined. Let x and y be two strings such that $[x] = [y]$. By definition of L-equivalence, for any string z, we have $xz \in L$ if and only if $yz \in L$. It immediately follows that for any symbol $a \in \Sigma$ and any string z', we have $xaz' \in L$ if and only if $yaz' \in L$. Thus, by definition of L-equivalence, we have $[xa] = [ya]$ for every symbol $a \in \Sigma$. We conclude that the function δ is indeed well-defined.

An easy inductive proof implies that $\delta^*([\varepsilon], x) = [x]$ for every string x. Thus, M accepts string x if and only if $[x] = [w]$ for some string $w \in L$. But if $[x] = [w]$, then by definition (setting $z = \varepsilon$), we have $x \in L$ if and only if $w \in L$. So M accepts x if and only if $x \in L$. In other words, M accepts L, as claimed, so the proof is complete. □

*3.10 Minimal Automata

Given a DFA $M = (\Sigma, Q, s, A, \delta)$, suppose we want to find another DFA $M' = (\Sigma, Q', s', A', \delta')$ with the fewest possible states that accepts the same language. In this final section, we describe an efficient algorithm to minimize DFAs, first described (in slightly different form) by Edward Moore in 1956. We analyze the running time of Moore's in terms of two parameters: $n = |Q|$ and $\sigma = |\Sigma|$.

In the preprocessing phase, we find and remove any states that cannot be reached from the start state s; this filtering can be performed in $O(n\sigma)$ time using any graph traversal algorithm. So from now on we assume that all states are reachable from s.

Now we recursively define two states p and q in the remaining DFA to be ***distingushable***, written $\boldsymbol{p \not\sim q}$, if at least one of the following conditions holds:

- $p \in A$ and $q \notin A$,
- $p \notin A$ and $q \in A$, or
- $\delta(p, a) \not\sim \delta(q, a)$ for some $a \in \Sigma$.

Equivalently, $p \not\sim q$ if and only if there is a string z such that exactly one of the states $\delta^*(p,z)$ and $\delta^*(q,z)$ is accepting. (Sound familiar?) Intuitively, the main algorithm assumes that all states are equivalent until proven otherwise, and then repeatedly looks for state pairs that can be proved distinguishable.

The main algorithm maintains a two-dimensional table, indexed by the states, where $Dist[p,q] = \text{True}$ indicates that we have proved states p and q are distinguishable. Initially, for all states p and q, we set $Dist[p,q] \leftarrow \text{True}$ if $p \in A$ and $q \notin A$ or vice versa, and $Dist[p,q] = \text{False}$ otherwise. Then we repeatedly consider each pair of states and each symbol to find more distinguishable pairs, until we make a complete pass through the table without modifying it. The table-filling algorithm can be summarized as follows:

```
MinDFATable($\Sigma, Q, s, A, \delta$):
  for all $p \in Q$
      for all $q \in Q$
          if ($p \in A$ and $q \notin A$) or ($p \notin A$ and $q \in A$)
              $Dist[p,q] \leftarrow$ True
          else
              $Dist[p,q] \leftarrow$ False
  $notdone \leftarrow$ True
  while $notdone$
      $notdone \leftarrow$ False
      for all $p \in Q$
          for all $q \in Q$
              if $Dist[p,q] =$ False
              for all $a \in \Sigma$
                  if $Dist[\delta(p,a), \delta(q,a)]$
                      $Dist[p,q] \leftarrow$ True
                      $notdone \leftarrow$ True
  return $Dist$
```

The algorithm must eventually halt, because there are only a finite number of entries in the table that can be marked. In fact, the main loop is guaranteed to terminate after at most n iterations, which implies that the entire algorithm runs in $\boldsymbol{O(\sigma n^3)}$ time. Once the table is filled,[6] any two states p and q such that $Dist(p,q) = \text{False}$ are equivalent and can be merged into a single state. The remaining details of constructing the minimized DFA are straightforward.

★★★ Need to prove that the main loop terminates in at most n iterations.

With more care, Moore's minimization algorithm can be modified to run in $O(\sigma n^2)$ time. A faster DFA minimization algorithm, due to John Hopcroft, runs in $O(\sigma n \log n)$ time.

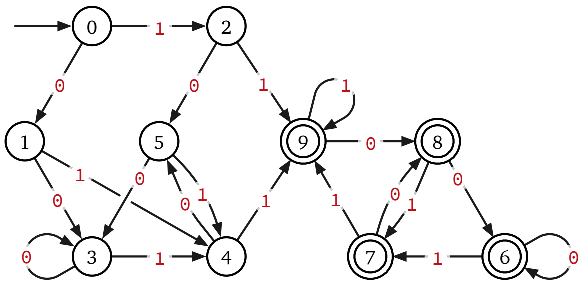
[6]More experienced readers should be enraged by the mere suggestion that any algorithm merely *fills in a table*, as opposed to *evaluating a recurrence*. This algorithm is no exception. Consider the boolean function $Dist(p,q,k)$, which equals True if and only if p and q can be distinguished by some string of length at most k. This function obeys the following recurrence:

$$Dist(p,q,k) = \begin{cases} (p \in A) \oplus (q \in A) & \text{if } k = 0, \\ Dist(p,q,k-1) \vee \bigvee_{a \in \Sigma} Dist\big(\delta(p,a), \delta(q,a), k-1\big) & \text{otherwise.} \end{cases}$$

Moore's "table-filling" algorithm is just a space-efficient dynamic programming algorithm to evaluate this recurrence.

Example

To get a better idea how this algorithm works, let's visualize its execution on our earlier brute-force DFA for strings containing the substring 11. This DFA has four unreachable states: (False, 11), (True, ε), (True, 0), and (True, 1). We remove these states, and relabel the remaining states for easier reference. (In an actual implementation, the states would almost certainly be represented by indices into an array anyway, not by mnemonic labels.)



Our brute-force DFA for strings containing the substring 11, after removing all four unreachable states

The main algorithm initializes (the bottom half of) a 10×10 table as follows. (In the following figures, cells marked $\times$ have value True and blank cells have value False.)

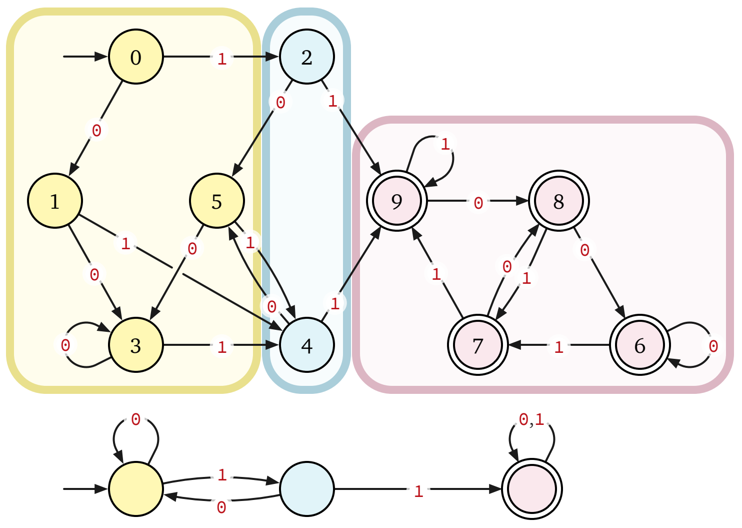
	0	1	2	3	4	5	6	7	8
1									
2									
3									
4									
5									
6	×	×	×	×	×	×			
7	×	×	×	×	×	×			
8	×	×	×	×	×	×			
9	×	×	×	×	×	×			

In the first iteration of the main loop, the algorithm discovers several distinguishable pairs of states. For example, the algorithm sets $\mathit{Dist}[0,2] \leftarrow$ True because $\mathit{Dist}[\delta(0,1), \delta(2,1)] = \mathit{Dist}[2,9] =$ True. After the iteration ends, the table looks like this:

	0	1	2	3	4	5	6	7	8
1									
2	×	×							
3			×						
4	×	×		×					
5			×		×				
6	×	×	×	×	×	×			
7	×	×	×	×	×	×			
8	×	×	×	×	×	×			
9	×	×	×	×	×	×			

The second iteration of the while loop makes no further changes to the table—We got lucky!—so the algorithm terminates.

The final table implies that the 10 states of our DFA fall into exactly three equivalence classes: $\{0,1,3,5\}$, $\{2,4\}$, and $\{6,7,8,9\}$. Replacing each equivalence class with a single state gives us the three-state DFA that we already discovered.



Equivalence classes of states in our DFA, and the resulting minimal equivalent DFA.

Exercises

1. For each of the following languages in $\{0,1\}^*$, describe a deterministic finite-state machine that accepts that language. There are infinitely many correct answers for each language. "Describe" does not necessarily mean "draw".

   (a) Only the string 0110.

   (b) Every string except 0110.

   (c) Strings that contain the substring 0110.

   (d) Strings that do not contain the substring 0110.

   ⋆(e) Strings that contain an even number of occurrences of the substring 0110. (For example, this language contains the strings 0110110 and 01011.)

   (f) Strings that contain the *subsequence* 0110.

   (g) Strings that do not contain the *subsequence* 0110.

   ⋆(h) Strings that contain an even number of occurrences of the *subsequence* 0110.

   (i) Strings that contain an even number of 1s and an odd number of 0s.

   (j) Every string that represents a number divisible by 7 in binary.

   (k) Every string whose reversal represents a number divisible by 7 in binary.

   (l) Strings in which the substrings 01 and 10 appear the same number of times.

   (m) Strings such that in every prefix, the number of 0s and the number of 1s differ by at most 1.

   (n) Strings such that in every prefix, the number of 0s and the number of 1s differ by at most 4.

   (o) Strings that end with 0^{10} = 0000000000.

   (p) All strings in which the number of 0s is even if and only if the number of 1s is *not* divisible by 3.

   (q) All strings that are both the binary representation of an integer divisible by 3 and the ternary (base-3) representation of an integer divisible by 4.

(r) Strings in which the number of 1s is even, the number of 0s is divisible by 3, the overall length is divisible by 5, the binary value is divisible by 7, the binary value of the reversal is divisible by 11, and does not contain thirteen 1s in a row. *[Hint: This is more tedious than difficult.]*

⋆(s) Strings w such that $\binom{|w|}{2} \bmod 6 = 4$.

⋆(t) Strings w such that $F_{\#(10,w)} \bmod 10 = 4$, where $\#(10, w)$ denotes the number of times 10 appears as a substring of w, and as usual F_n is the nth Fibonacci number:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

★(u) Strings w such that $F_{\#(1\cdots0,w)} \bmod 10 = 4$, where $\#(1\cdots0, w)$ denotes the number of times 10 appears as a *subsequence* of w, and as usual F_n is the nth Fibonacci number:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

2. (a) Let $L \subseteq 0^*$ be an arbitrary *unary* language. Prove that L^* is regular.

   (b) Prove that there is a binary language $L \subseteq (0 + 1)^*$ such that L^* is not regular.

3. Prove that none of the following languages is automatic.

   (a) $\{0^{n^2} \mid n \geq 0\}$

   (b) $\{0^{n^3} \mid n \geq 0\}$

   (c) $\{0^{f(n)} \mid n \geq 0\}$, where $f(n)$ is *any* fixed polynomial in n with degree at least 2.

   (d) $\{0^n \mid n \text{ is composite}\}$

   (e) $\{0^n10^n \mid n \geq 0\}$

   (f) $\{0^m1^n \mid m \neq n\}$

   (g) $\{0^m1^n \mid m < 3n\}$

   (h) $\{0^{2n}1^n \mid n \geq 0\}$

   (i) $\{w \in (0 + 1)^* \mid \#(0, w) = \#(1, w)\}$

   (j) $\{w \in (0 + 1)^* \mid \#(0, w) < \#(1, w)\}$

   (k) $\{0^m1^n \mid m/n \text{ is an integer}\}$

   (l) $\{0^m1^n \mid m \text{ and } n \text{ are relatively prime}\}$

   (m) $\{0^m1^n \mid n - m \text{ is a perfect square}\}$

   (n) $\{w\#w \mid w \in (0 + 1)^*\}$

   (o) $\{ww \mid w \in (0 + 1)^*\}$

   (p) $\{w\#0^{|w|} \mid w \in (0 + 1)^*\}$

   (q) $\{w0^{|w|} \mid w \in (0 + 1)^*\}$

(r) $\{xy \mid x, y \in (0+1)^* \text{ and } |x| = |y| \text{ but } x \neq y\}$

(s) $\{0^m 1^n 0^{m+n} \mid m, n \geq 0\}$

(t) $\{0^m 1^n 0^{mn} \mid m, n \geq 0\}$

(u) Strings in which the substrings 00 and 11 appear the same number of times.

(v) Strings of the form $w_1\#w_2\#\cdots\#w_n$ for some $n \geq 2$, where $w_i \in \{0,1\}^*$ for every index i, and $w_i = w_j$ for some indices $i \neq j$.

(w) The set of all palindromes in $(0+1)^*$ whose length is divisible by 7.

(x) $\{w \in (0+1)^* \mid w \text{ is the binary representation of a perfect square}\}$

★(y) $\{w \in (0+1)^* \mid w \text{ is the binary representation of a prime number}\}$

4. For each of the following languages over the alphabet $\Sigma = \{0,1\}$, either prove that the language is regular (by constructing an appropriate DFA or regular expression) or prove that the language is not regular (using fooling sets). Recall that Σ^+ denotes the set of all *nonempty* strings over Σ. *[Hint: Believe it or not, most of these languages **are** actually regular.]*

(a) $\{0^n w 1^n \mid w \in \Sigma^* \text{ and } n \geq 0\}$

(b) $\{0^n 1^n w \mid w \in \Sigma^* \text{ and } n \geq 0\}$

(c) $\{w 0^n 1^n x \mid w, x \in \Sigma^* \text{ and } n \geq 0\}$

(d) $\{0^n w 1^n x \mid w, x \in \Sigma^* \text{ and } n \geq 0\}$

(e) $\{0^n w 1 x 0^n \mid w, x \in \Sigma^* \text{ and } n \geq 0\}$

(f) $\{0^n w 0^n \mid w \in \Sigma^+ \text{ and } n > 0\}$

(g) $\{w 0^n w \mid w \in \Sigma^+ \text{ and } n > 0\}$

(h) $\{wxw \mid w, x \in \Sigma^*\}$

(i) $\{wxw \mid w, x \in \Sigma^+\}$

(j) $\{wxw^R \mid w, x \in \Sigma^+\}$

(k) $\{wwx \mid w, x \in \Sigma^+\}$

(l) $\{ww^R x \mid w, x \in \Sigma^+\}$

(m) $\{wxwy \mid w, x, y \in \Sigma^+\}$

(n) $\{wxw^R y \mid w, x, y \in \Sigma^+\}$

(o) $\{xwwy \mid w, x, y \in \Sigma^+\}$

(p) $\{xww^R y \mid w, x, y \in \Sigma^+\}$

(q) $\{wxxw \mid w, x \in \Sigma^+\}$

⋆(r) $\{wxw^R x \mid w, x \in \Sigma^+\}$

(s) All strings w such that no prefix of w is a palindrome.

(t) All strings w such that no prefix of w with length at least 3 is a palindrome.

(u) All strings w such that no substring of w with length at least 3 is a palindrome.

(v) All strings w such that no prefix of w with positive even length is a palindrome.

(w) All strings w such that no substring of w with positive even length is a palindrome.

(x) Strings in which the substrings `00` and `11` appear the same number of times.

(y) Strings in which the substrings `01` and `10` appear the same number of times.

5. Let F and L be arbitrary infinite languages in $\{0,1\}^*$.

   (a) Suppose for any two distinct strings $x, y \in F$, there is a string $w \in \Sigma^*$ such that $wx \in L$ and $wy \notin L$. (We can reasonably call w a *distinguishing* ***prefix*** for x and y.) Prove that L cannot be regular. *[Hint: The reversal of a regular language is regular.]*

   *(b) Suppose for any two distinct strings $x, y \in F$, there are two (possibly equal) strings $w, z \in \Sigma^*$ such that $wxz \in L$ and $wyz \notin L$. Prove that L cannot be regular.