Finite State machines

Jim Hefferon

University of Vermont

hefferon.net

Finite State machines

Motivation

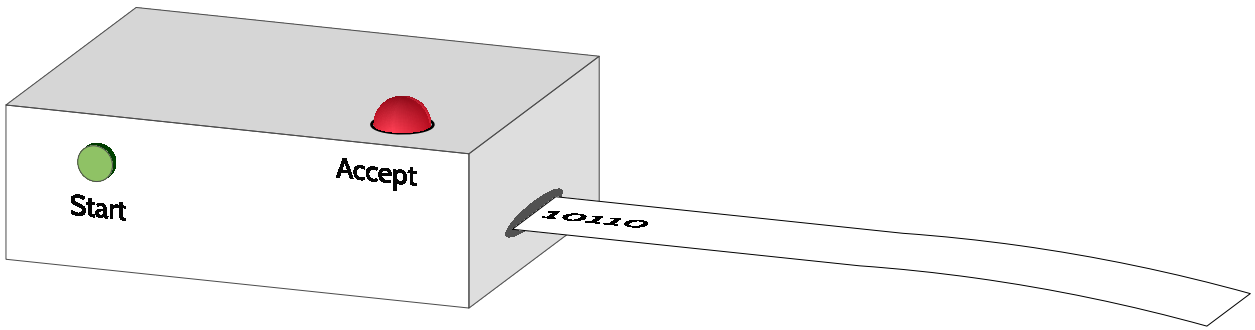
Turing and other researchers tried to define what is computable in principle. Church's Thesis is that their definitions do indeed capture the intuitive idea of effectively computable, of computable by an unbounded mechanism.

A person could object, "Although a Turing machine has only finitely many states, it can write to the tape. Consequently there are Turing machines that write arbitrarily many times to the tape, and so have unboundedly many configurations. But the physical universe cannot support unboundedly many configurations."

We will next study machines that are modifications of Turing machines to have a bounded number of configurations. These have finitely many states and can read, but they cannot write. Instead, they just consume their input and then indicate either that the input string was accepted or that it was not accepted.

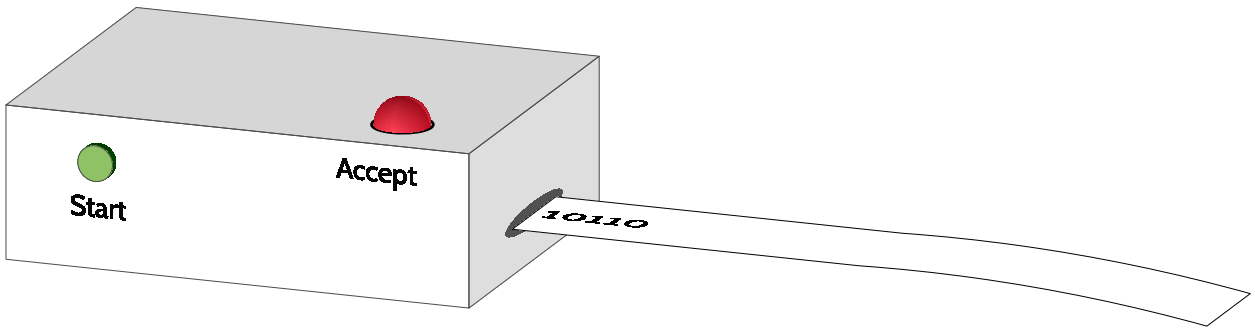
Definitions

Definition A Finite State machine or Finite State automata $\langle Q, q_0, F, \Sigma, \Delta \rangle$ has a finite set of states Q, one of which is the start state q_0, a subset $F \subseteq Q$ of final states or accepting states, a finite input alphabet set Σ, and a next-state function or transition function $\Delta : Q \times \Sigma \to Q$.



Definitions

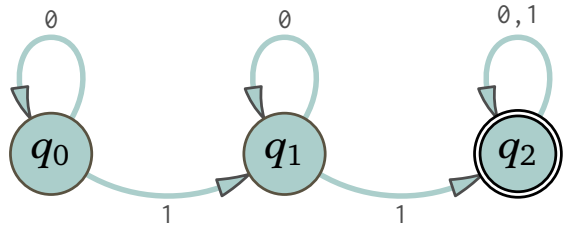
DEFINITION A Finite State machine or Finite State automata $\langle Q, q_0, F, \Sigma, \Delta \rangle$ has a finite set of states Q, one of which is the start state q_0, a subset $F \subseteq Q$ of final states or accepting states, a finite input alphabet set Σ, and a next-state function or transition function $\Delta : Q \times \Sigma \to Q$.



Remark: Recall that Turing machines are defined to be sets of four-tuple instructions $q_p T_p T_n q_n$, subject to the restriction that the set is deterministic, that no two instructions can begin with the same pair $q_p T_p$. Consequently, over all $q_p T_p T_n q_n \in \mathcal{P}$ the association of present pair $q_p T_p$ with next pair $T_n q_n$, defines a function, the next-state function $\Delta : Q \times \Sigma \to \Sigma \cup \{\mathsf{L}, \mathsf{R}\} \times Q$.

Finite State machines work the same way, and in the definition above we skipped four-tuples and went straight to the function.

EXAMPLE This machine has input alphabet $\Sigma = \mathbb{B} = \{0, 1\}$. The set of states is $Q = \{q_0, q_1, q_2\}$. It has one accepting state $F = \{q_2\}$, marked in the diagram with a double circle.



The transition function is a tabular version of the diagram. For instance, $\Delta(q_1, 1) = q_2$.

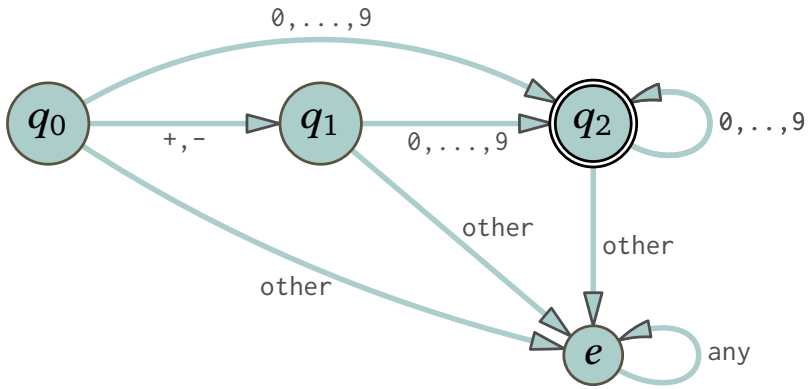
Δ	0	1
q_0	q_0	q_1
q_1	q_1	q_2
+ q_2	q_2	q_2

The + next to q_2 marks it as an accepting state.

This machine accepts the string `01010`. It does not accept the string `001`. In general, it accepts strings that contain at least two 1's.

EXAMPLE This machine accepts strings that are valid decimal representations of integers. So it accepts the strings `21` and `-707` but does not accept `501-`.

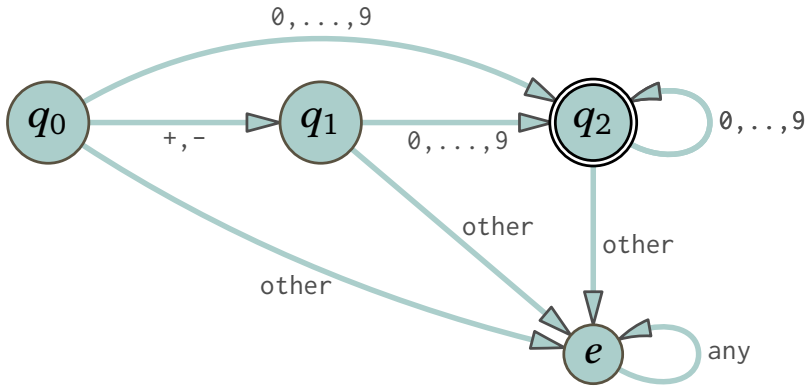
The transition graph and the table both group some inputs together when they result in the same action. For instance, when in state q_0 this machine does the same thing whether the input is `+` or `-`, namely it passes into q_1.



Δ	+, -	0, ... 9	*other*
q_0	q_1	q_2	e
q_1	e	q_2	e
+ q_2	e	q_2	e
e	e	e	e

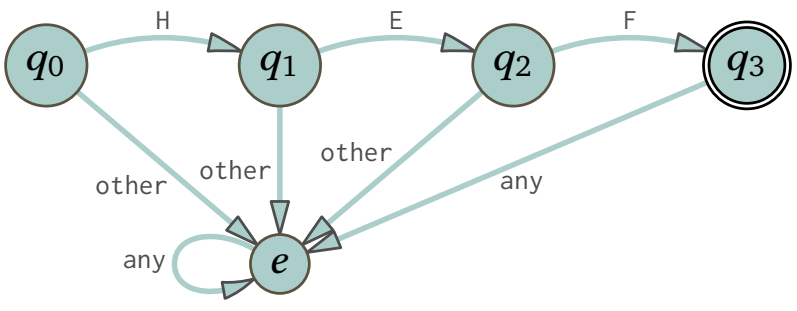
EXAMPLE This machine accepts strings that are valid decimal representations of integers. So it accepts the strings `21` and `-707` but does not accept `501-`.

The transition graph and the table both group some inputs together when they result in the same action. For instance, when in state q_0 this machine does the same thing whether the input is `+` or `-`, namely it passes into q_1.

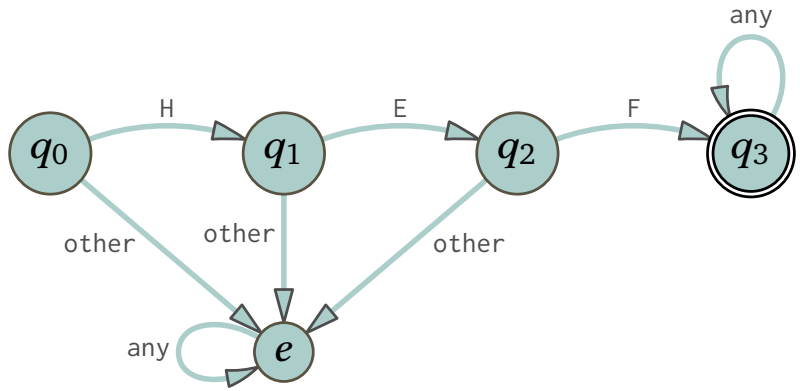


Δ	+, -	0, ... 9	*other*
q_0	q_1	q_2	e
q_1	e	q_2	e
+ q_2	e	q_2	e
e	e	e	e

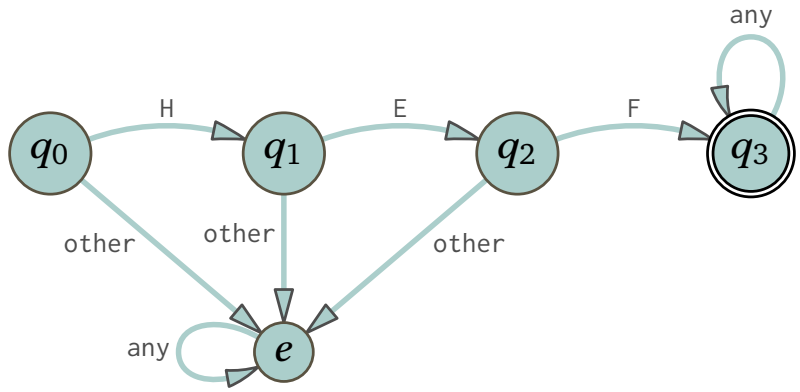
EXAMPLE This machine accepts only the single string `HEF`.



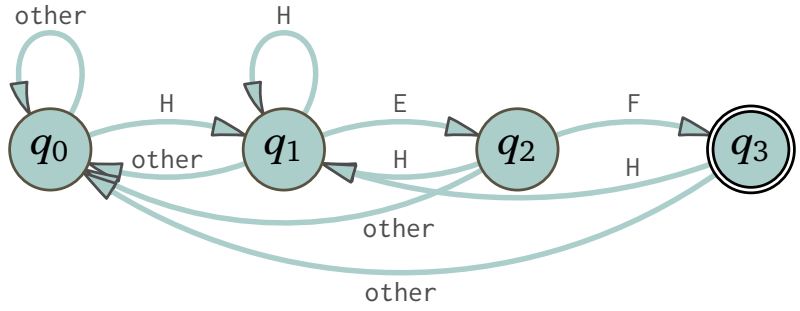
EXAMPLE This machine accepts only strings where HEF is the prefix.



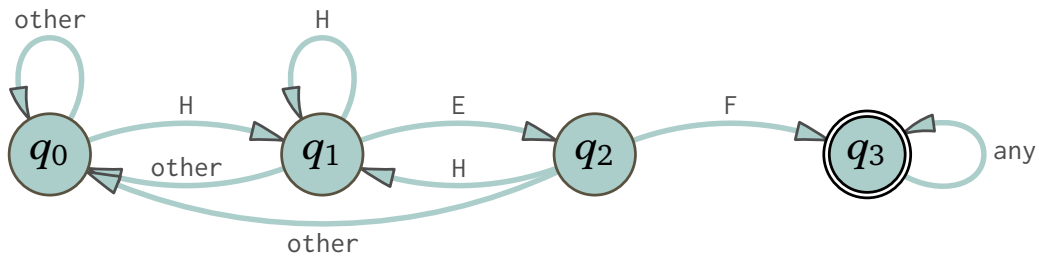
EXAMPLE This machine accepts only strings where HEF is the prefix.



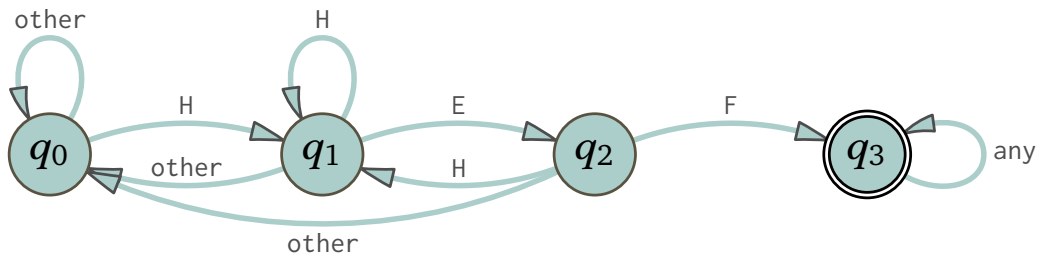
EXAMPLE This machine accepts only strings where HEF is the suffix.



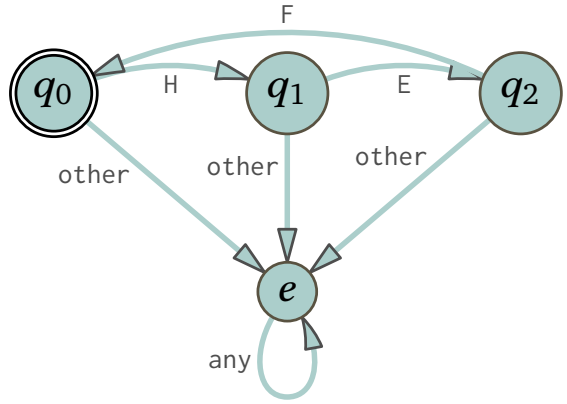
EXAMPLE This machine accepts only strings where HEF is a substring.



EXAMPLE This machine accepts only strings where HEF is a substring.

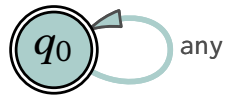


EXAMPLE This machine accepts only strings consisting of zero or more repetitions of HEF.



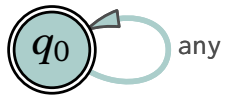
The string consisting of zero-many repetitions of HEF is the empty string, ε.

EXAMPLE This machine accepts all strings starting with zero or more copies of HEF.



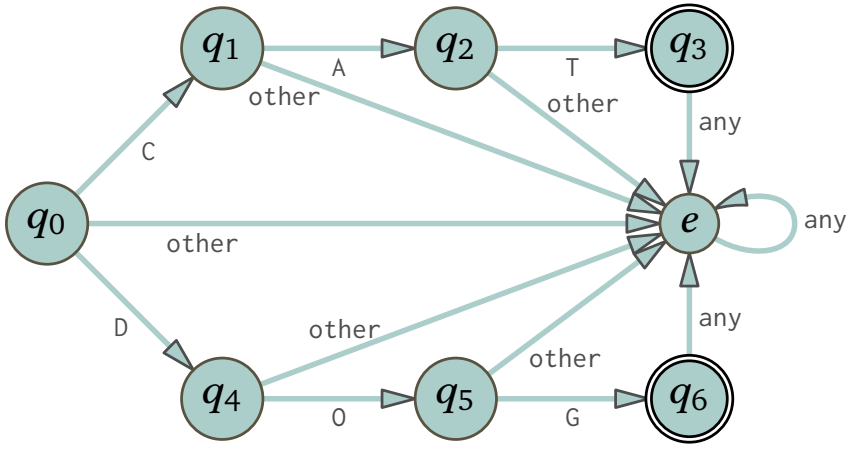
(It is a gotcha question; all strings begin with zero-many copies of HEF.)

EXAMPLE This machine accepts all strings starting with zero or more copies of HEF.

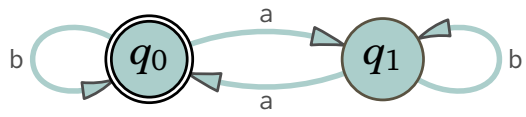


(It is a gotcha question; all strings begin with zero-many copies of HEF.)

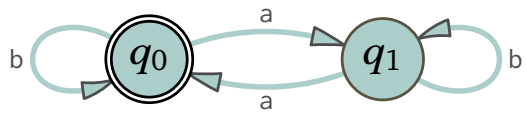
EXAMPLE This machine accepts only the strings CAT or DOG.



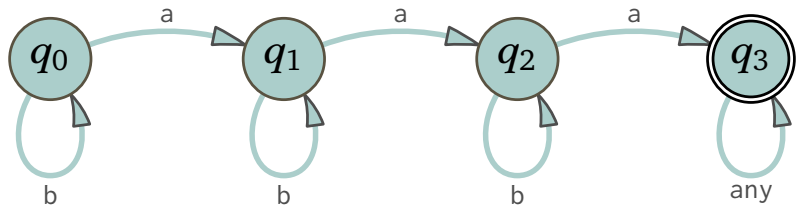
EXAMPLE This machine, with input alphabet $\sigma = \{\mathsf{a}, \mathsf{b}\}$, accepts only strings with an even number of a's.



EXAMPLE This machine, with input alphabet $\sigma = \{\mathsf{a}, \mathsf{b}\}$, accepts only strings with an even number of a's.



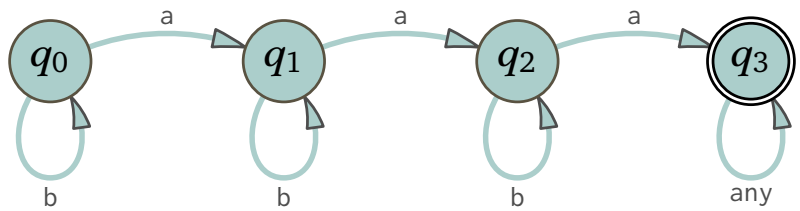
EXAMPLE This machine, with the same alphabet, accepts strings with at least three a's.



The action of a Finite State machine

As with Turing machines, we can give a definition of how Finite State machines act that is mathematically precise. A configuration of a Finite State machine $\mathcal{M}$ is a pair $\mathcal{C} = \langle q, \tau \rangle$, with $q \in Q$ and $\tau \in \Sigma^*$. Where $\mathcal{C}_0 = \langle q_0, \tau_0 \rangle$ is the initial configuration, τ_0 is the input.

EXAMPLE We can trace through this machine when given the input $\tau_0 = \mathsf{abababb}$.



The transition function Δ determines how the machine moves step-by-step, in response to the input (read '$\vdash$' aloud as "yields").

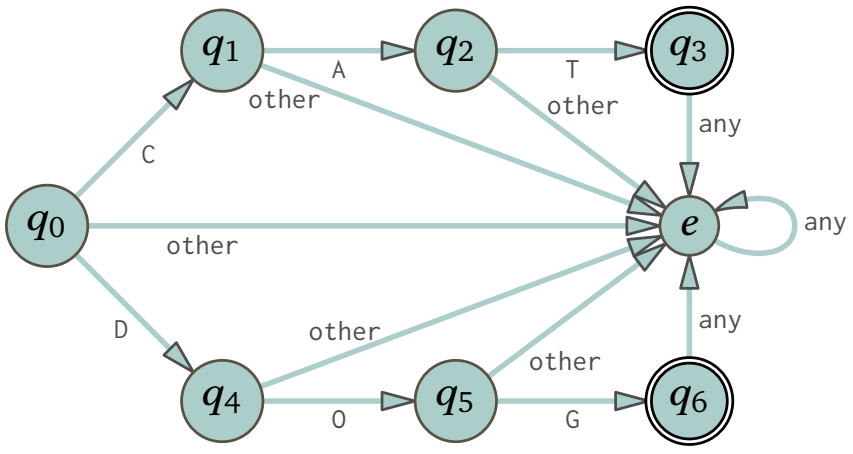
$$\begin{aligned}
\mathcal{C}_0 = \langle q_0, \mathsf{abababb} \rangle &\vdash \mathcal{C}_1 = \langle q_1, \mathsf{bababb} \rangle \\
&\vdash \mathcal{C}_2 = \langle q_1, \mathsf{ababb} \rangle \\
&\vdash \mathcal{C}_3 = \langle q_2, \mathsf{babb} \rangle \\
&\vdash \mathcal{C}_4 = \langle q_2, \mathsf{abb} \rangle \\
&\vdash \mathcal{C}_5 = \langle q_3, \mathsf{bb} \rangle \\
&\vdash \mathcal{C}_6 = \langle q_3, \mathsf{b} \rangle \\
&\vdash \mathcal{C}_7 = \langle q_3, \varepsilon \rangle
\end{aligned}$$

The machine accepts τ_0 because when $\tau = \varepsilon$ then the state q_3 is accepting.

Language of a machine

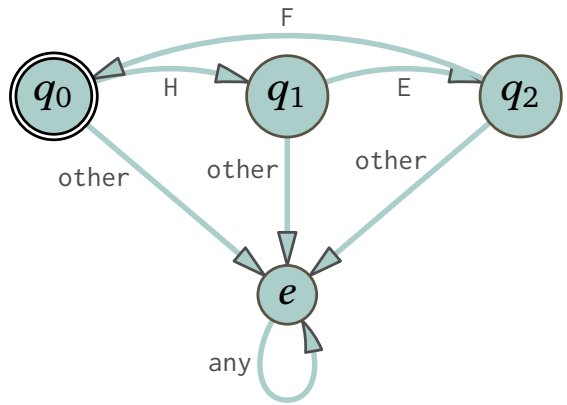
DEFINITION The set of strings accepted by a Finite State machine $\mathcal{M}$ is the language of that machine, $\mathcal{L}(\mathcal{M})$, or the language recognized or decided, or accepted, by the machine.

EXAMPLE This machine



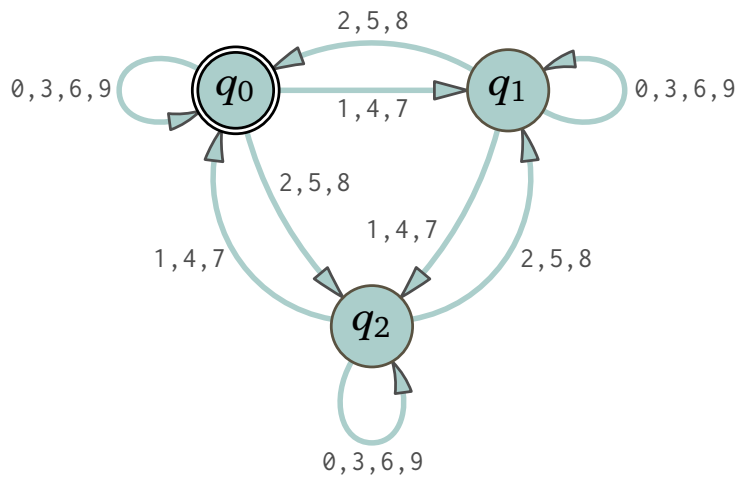
has the language $\mathcal{L}(M) = \{\texttt{CAT}, \texttt{DOG}\}$.

Example This machine



has the language $\mathcal{L} = \{\varepsilon, \texttt{HEF}, \texttt{HEFHEF}, (\texttt{HEF})^3, \ldots\}$.

Example This machine recognizes a natural number that is a multiple of three, such as `15` or `5013`.



Where $\Sigma = \{\texttt{0}, \ldots \texttt{9}\}$, the language is this.

$$\mathcal{L}(\mathcal{M}) = \{\sigma \in \Sigma^* \;\big|\; \sigma \text{ is the decimal representation of a multiple of three}\}$$

Finite State machines translate easily to code

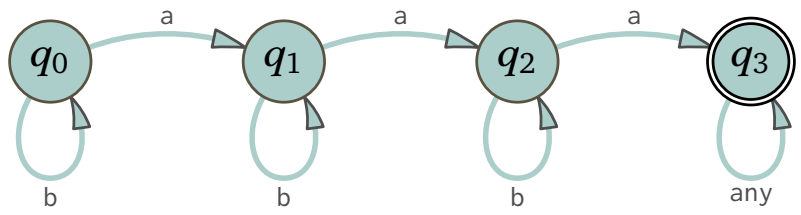
Here is a Python implementation of the above multiple of three machine.

```
def multiple_of_three_fsm(s):
    state = 0
    for ch in s:
        if state == 0:
            if ch in set([0,3,6,9]):
                state = 0
            elif ch in set([1,4,7]):
                state = 1
            else:
                state = 2
        elif state == 1:
            if ch in set([0,3,6,9]):
                state = 1
            elif ch in set([1,4,7]):
                state = 2
            else:
                state = 0
        else:
            if ch in set([0,3,6,9]):
                state = 2
            elif ch in set([1,4,7]):
                state = 0
            else:
                state = 1
    if state == 0:
        return True
    else:
        return False
```

Extended transition function

DEFINITION For any Finite State machine, the extended transition function $\hat{\Delta}\colon \Sigma^* \to Q$ gives the state in which the machine ends after starting in the start state and consuming the given string.

EXAMPLE For this machine, compute the extended transition function on all strings of length less than three.



The alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ has two elements so there are seven strings of length less than three.

Input σ	ε	a	b	aa	ab	ba	bb
$\hat{\Delta}(\sigma)$	q_0	q_1	q_0	q_2	q_1	q_1	q_0

Nondeterminism

Recall: determinism

The Turing machine definition gives that the set of instructions $q_p T_p T_n q_n$ must be deterministic, meaning that no two instructions can begin with the same pair $q_p T_p$. For instance the definition outlaws that both $q_0 1 0 q_1$ and $q_0 1 1 q_2$ are in the same machine because they both start with $q_0 1$.

Consequently, over the four-tuples $q_p T_p T_n q_n$ in a machine, the association of present pair $q_p T_p$ with next pair $T_n q_n$ defines a function, the next-state function.

$$\Delta : Q \times \Sigma \to \Sigma \cup \{\mathtt{L}, \mathtt{R}\} \times Q$$

It is this function that governs the sequence of transitions from configuration to configuration.

Finite State machines work the same way. The definition gives them as also deterministic.

Recall: determinism

The Turing machine definition gives that the set of instructions $q_p T_p T_n q_n$ must be deterministic, meaning that no two instructions can begin with the same pair $q_p T_p$. For instance the definition outlaws that both $q_0 1 0 q_1$ and $q_0 1 1 q_2$ are in the same machine because they both start with $q_0 1$.

Consequently, over the four-tuples $q_p T_p T_n q_n$ in a machine, the association of present pair $q_p T_p$ with next pair $T_n q_n$ defines a function, the next-state function.

$$\Delta : Q \times \Sigma \to \Sigma \cup \{\mathtt{L}, \mathtt{R}\} \times Q$$

It is this function that governs the sequence of transitions from configuration to configuration.

Finite State machines work the same way. The definition gives them as also deterministic.

We now drop determinism. We will do it here for Finite State machines and in the next chapter for Turing machines.

Nondeterminism is very important. But it can give the initial impression of being unnatural. So before the definition we will see two examples of computations that, in some sense, do more than one thing at a time. These will lead to two mental models, which we will rely on for the rest of the course.

Nondeterminism: deriving a string from a grammar

From this grammar

$$\begin{aligned} S &\rightarrow TbU \\ T &\rightarrow aT \mid \varepsilon \\ U &\rightarrow aU \mid bU \mid \varepsilon \end{aligned}$$

we can derive, for example, $\sigma_0 = aabab$

$$\begin{aligned} S \Rightarrow T\,b\,U \Rightarrow a\,T\,b\,U \Rightarrow a\,a\,T\,b\,U \Rightarrow a\,a\,\varepsilon\,b\,U = a\,a\,b\,U \\ \Rightarrow a\,a\,b\,a\,U \Rightarrow a\,a\,b\,a\,b\,U \Rightarrow a\,a\,b\,a\,b\,\varepsilon = a\,a\,b\,a\,b \end{aligned}$$

or $\sigma_1 = baab$.

$$\begin{aligned} S \Rightarrow T\,b\,U \Rightarrow \varepsilon\,b\,U = b\,U \Rightarrow b\,a\,U \\ \Rightarrow b\,a\,a\,U \Rightarrow b\,a\,a\,b\,U \Rightarrow b\,a\,a\,b\,\varepsilon = b\,a\,a\,b \end{aligned}$$

Derivation is a computation in that it is pushing symbols by following rigid rules.

Nondeterminism: deriving a string from a grammar

From this grammar

$$\begin{aligned} S &\rightarrow TbU \\ T &\rightarrow aT \mid \varepsilon \\ U &\rightarrow aU \mid bU \mid \varepsilon \end{aligned}$$

we can derive, for example, $\sigma_0 = aabab$

$$\begin{aligned} S \Rightarrow T\,b\,U \Rightarrow a\,T\,b\,U \Rightarrow a\,a\,T\,b\,U \Rightarrow a\,a\,\varepsilon\,b\,U = a\,a\,b\,U \\ \Rightarrow a\,a\,b\,a\,U \Rightarrow a\,a\,b\,a\,b\,U \Rightarrow a\,a\,b\,a\,b\,\varepsilon = a\,a\,b\,a\,b \end{aligned}$$

or $\sigma_1 = baab$.

$$\begin{aligned} S \Rightarrow T\,b\,U \Rightarrow \varepsilon\,b\,U = b\,U \Rightarrow b\,a\,U \\ \Rightarrow b\,a\,a\,U \Rightarrow b\,a\,a\,b\,U \Rightarrow b\,a\,a\,b\,\varepsilon = b\,a\,a\,b \end{aligned}$$

Derivation is a computation in that it is pushing symbols by following rigid rules.

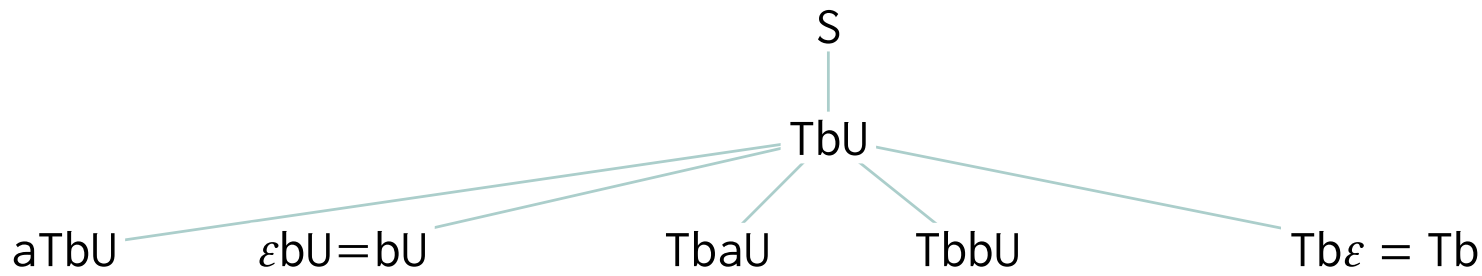
This computation is nondeterministic in the sense that while doing a sequence of $\Rightarrow$ steps, we often find that more than one $\rightarrow$ rule applies. So there is more than one next thing to do, not just a unique next string.

Suppose that we have a grammar where there is sometimes more than one applicable rule. Give a target string σ, how to derive it?

One way is brute force: perform all applicable production rules to the start symbol, then for each result perform all applicable rules, etc. If σ is in the language of the grammar then we will eventually derive it. This is a breadth-first search of the $\Rightarrow$ sequences; here are the first few levels for the grammar on the prior slide.

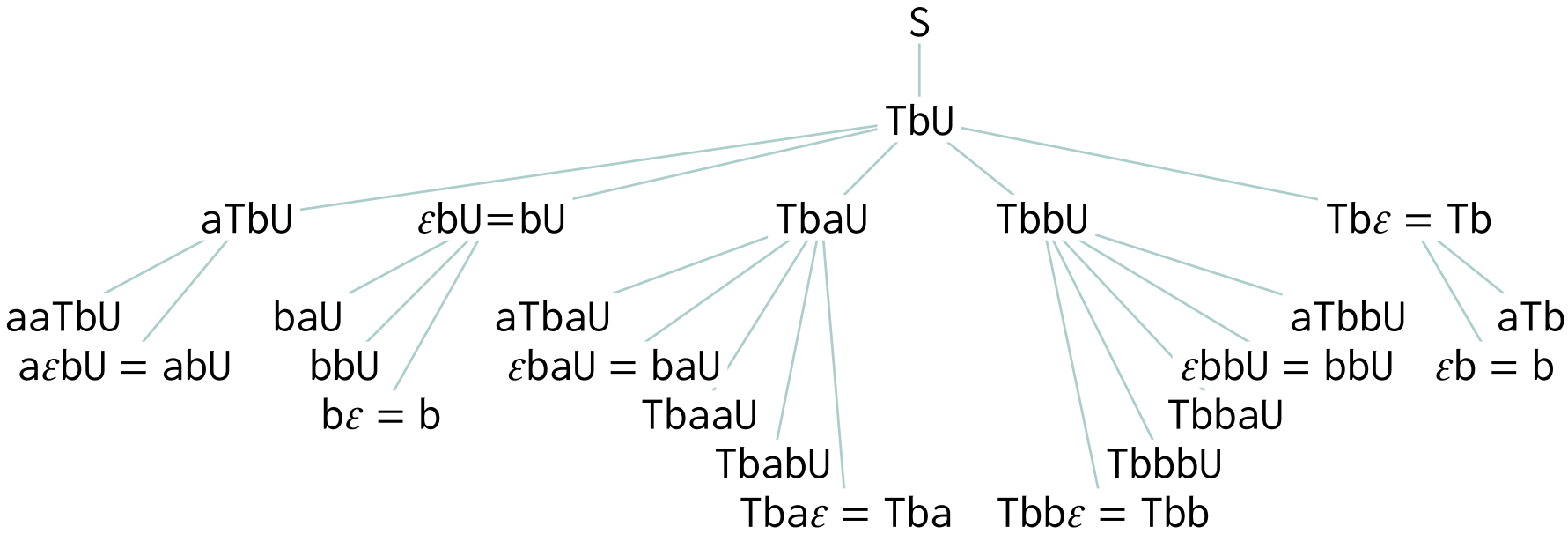
Suppose that we have a grammar where there is sometimes more than one applicable rule. Give a target string σ, how to derive it?

One way is brute force: perform all applicable production rules to the start symbol, then for each result perform all applicable rules, etc. If σ is in the language of the grammar then we will eventually derive it. This is a breadth-first search of the $\Rightarrow$ sequences; here are the first few levels for the grammar on the prior slide.



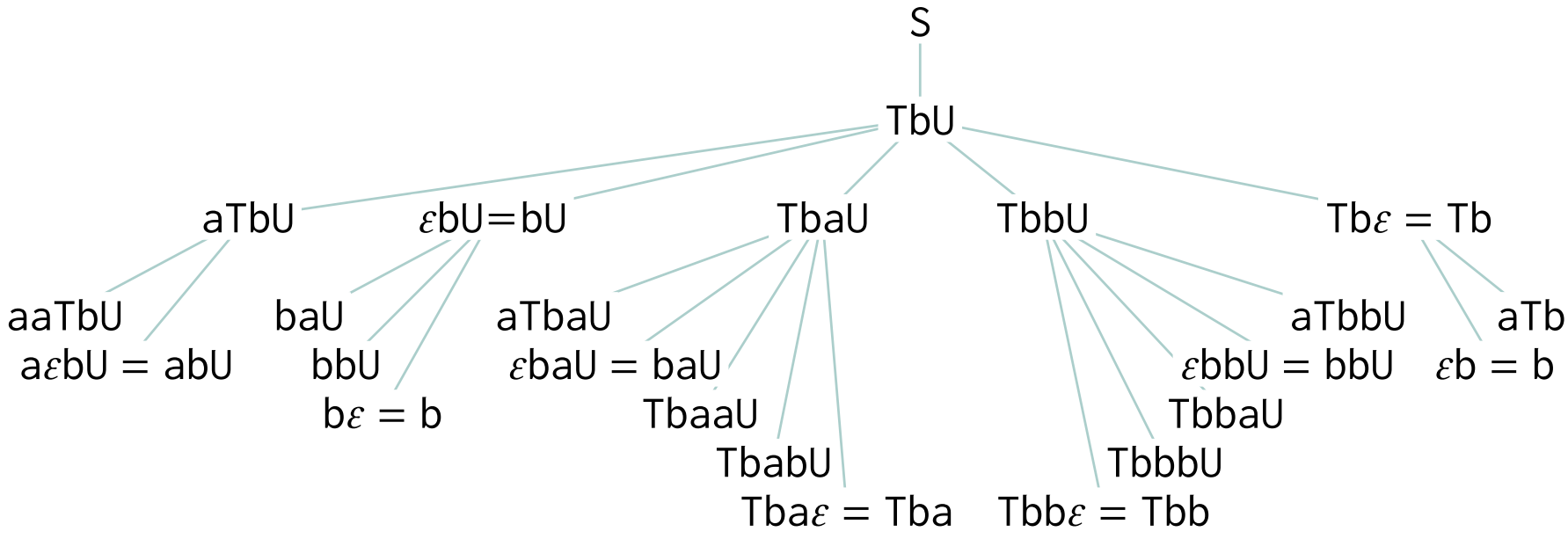
Suppose that we have a grammar where there is sometimes more than one applicable rule. Give a target string σ, how to derive it?

One way is brute force: perform all applicable production rules to the start symbol, then for each result perform all applicable rules, etc. If σ is in the language of the grammar then we will eventually derive it. This is a breadth-first search of the $\Rightarrow$ sequences; here are the first few levels for the grammar on the prior slide.



Suppose that we have a grammar where there is sometimes more than one applicable rule. Give a target string σ, how to derive it?

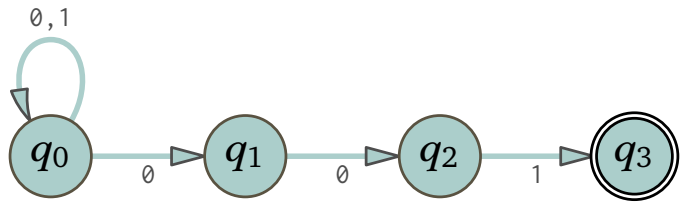
One way is brute force: perform all applicable production rules to the start symbol, then for each result perform all applicable rules, etc. If σ is in the language of the grammar then we will eventually derive it. This is a breadth-first search of the $\Rightarrow$ sequences; here are the first few levels for the grammar on the prior slide.



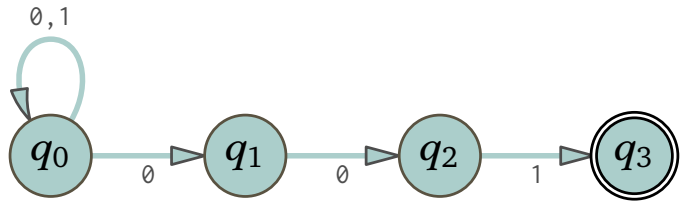
The second way, the way that we actually solve these problems, for instance on the Grammar section's homework, is to guess. Then we verify that guess by exhibiting a derivation.

Nondeterminism: a machine

This machine is nondeterministic: for a given pair ⟨present state, input character⟩ there may be one next state, or more than one, or none. For instance, leaving q_0 are two different arrows labeled `0`. And there is no arrow at all leaving q_1 with the label `1`.



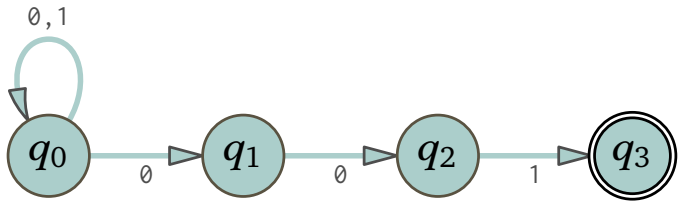
Suppose that this machine gets the input `00001`. Just as with the grammar, we will have two ways to think of what happens. First, we can trace out every possible branch.



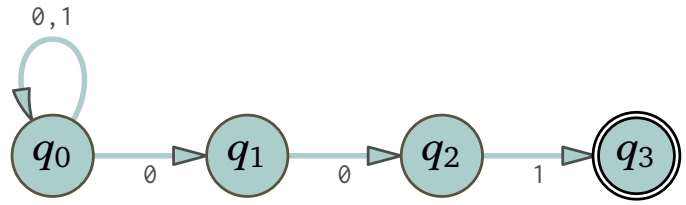
Input:	0	0	0	0	ɪ

q_0

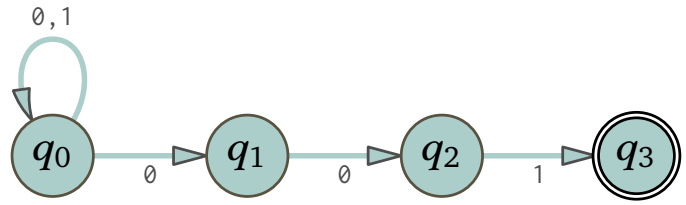
Step:	0	1	2	3	4	5



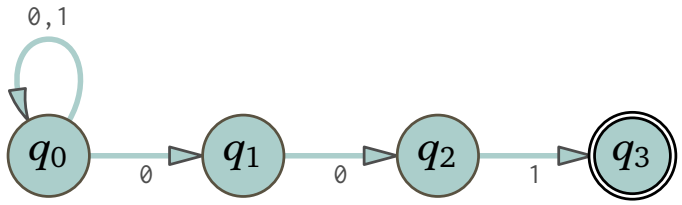
Input:		0	0	0	0	1
			q_0			
	q_0	⊢	⊢			
			q_1			
Step:	0		1	2	3	4	5



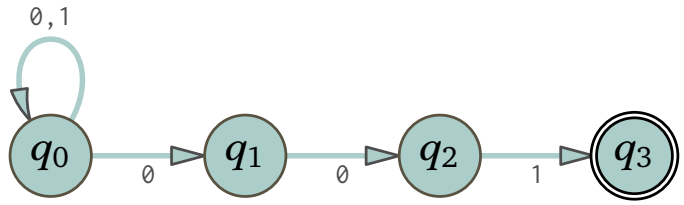
Input:	0	0	0	0	1
Step:	0	1	2	3	4	5
	q_0	q_0	q_0			
		⊢	q_1			
			⊢	q_2		



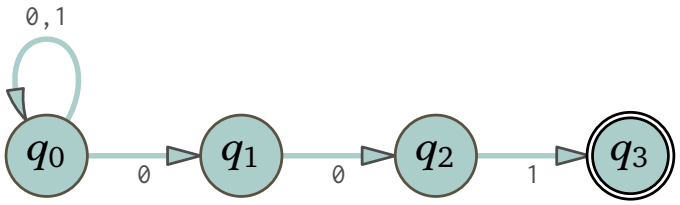
Input:	0	0	0	0	1
Step:	0	1	2	3	4 5
	q_0	q_0	q_0	q_0	
	⊢	q_1	q_1	q_1	
	⊢	⊢	q_2	q_2	
	⊢	⊢	⊢		



Input:	0	0	0	0	1
	q_0	q_0	q_0	q_0	q_0
	⊢	⊢	⊢	⊢	⊢
		q_1	q_1	q_1	q_1
			q_2	q_2	q_2
Step:	0	1	2	3	4	5



Input:	0	0	0	0	1	
Step:	0	1	2	3	4	5
	q_0	⊢ q_0	⊢ ⊢ q_0	⊢ ⊢ ⊢ q_0	⊢ ⊢ q_0	⊢ q_0
		⊢ q_1	⊢ ⊢ q_1	⊢ ⊢ q_1	⊢ q_1	q_3
			⊢ q_2	⊢ q_2	q_2	



Input:	0	0	0	0	1	
	q_0	⊢	⊢	⊢	⊢	q_0
		q_0	⊢	⊢	⊢	⊢
		q_1	q_0	⊢	⊢	⊢
			q_1	q_0	⊢	⊢
			q_2	q_1	q_0	⊢
				q_2	q_1	q_0
					q_2	q_1
						q_2
						q_3
Step:	0	1	2	3	4	5

The highlighted branch ends in an accepting state.

As with the grammar, we are picturing that when the machine is asked to do two things, it does them both. And when there is no next state, the branch dies. So the computation history is not straight-line but instead is a tree.

After the first step, the machine is in multiple states simultaneously, namely q_0 and q_1. The same also happens at all later steps; in all of those steps the machine is in a set of states.

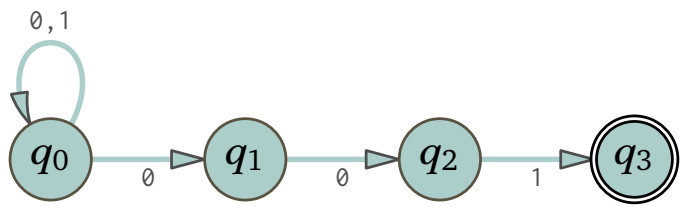
There is a branch that gets to the end and accepts the input string `00001`. Branches can fail to accept the string because they die, or because they get to the end and they just are not in an accepting state.

As with the grammar, we are picturing that when the machine is asked to do two things, it does them both. And when there is no next state, the branch dies. So the computation history is not straight-line but instead is a tree.

After the first step, the machine is in multiple states simultaneously, namely q_0 and q_1. The same also happens at all later steps; in all of those steps the machine is in a set of states.

There is a branch that gets to the end and accepts the input string `00001`. Branches can fail to accept the string because they die, or because they get to the end and they just are not in an accepting state.

As with the grammar, there is a second way to think of what is happening. This machine,



when presented with an input string `00001`, must decide: when should it stop going around q_0's loop and start following the tail to the right? Above we've seen the machine accepting `00001`, so it has solved this problem. We could say that the machine has correctly guessed.

First mental model for nondeterminism: spawning

For the grammar example, we made a branch each time a production rule applied to a string. For the machine example, we also branched each time there was a possible thing to do.

Another example of this branching is the Travelling Salesman problem. Suppose that we want a cyclic trip that visits every state capitol in the forty eight contiguous US states in less than 16 000 kilometers. If we start at Montpelier, Vermont then there are forty seven possible next cities. A reasonable approach is to spawn forty seven child processes. The process assigned Albany, for instance, would know that the trip so far is 126 kilometers. After that, each child would fork its own set of children, forty six of them. At the end if there is a trip of less than 16 000 kilometers then some branch has found it. Note that we consider the overall computation a success if there exists even one process that is a success.

The first mental model for nondeterminism is that when presented with multiple things to do, the machine does them all. Think of it as an unboundedly parallelizable device.

This model is natural. In everyday computing, we are all used to having lots of windows open on the screen, doing lots of things. Under the hood, the operating system forks child processes to cover each thing that it must do.

Second mental model for nondeterminism: guessing

For both the grammar example and the machine example, we have also framed that the system guessed the answer.

For example, we described the nondeterministic machine as guessing when to stop going around q_0's loop and start following the tail to the right.

Second mental model for nondeterminism: guessing

For both the grammar example and the machine example, we have also framed that the system guessed the answer.

For example, we described the nondeterministic machine as guessing when to stop going around q_0's loop and start following the tail to the right.

Saying "guessing" is jarring because our experience with computers in programming class is that they don't guess. An alternate way to state this model is that the machine is furnished with the answer ("go around twice, then off to the right") and it only has to verify that answer. Often the furnisher is personified as a demon.

Flauros, the Duke of Hell.

What the two views share

In both mental models, the computation succeeds if there is a way for it to succeed. More precisely, the machine accepts the input if there exists a sequence of configurations that ends in an accepting state.

Very important: 'if there exists' doesn't mean that there must be an algorithm for finding it, such as backtracking or dynamic programming, etc. This is pure existence — it just requires that there exist a way for the computation to succeed. That's why the demon is here; it is a being so powerful that it goes beyond needing ordinary computation to know the answer.

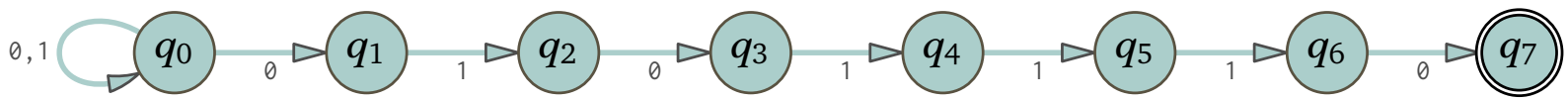
Definition of a Nondeterministic Finite State machine

To make the definition of a nondeterministic Finite State machine, we change one thing from the definition of Finite State machine, namely we change the codomain of Δ from Q to the power set $\mathscr{P}(Q)$.

DEFINITION A nondeterministic Finite State machine $\langle Q, q_{\text{start}}, F, \Sigma, \Delta \rangle$ consists of a finite set of states Q, one of which is the start state q_{start}, a subset $F \subseteq Q$ of accepting states or final states, a finite input alphabet set Σ, and a next-state function $\Delta : Q \times \Sigma \to \mathscr{P}(Q)$.

The machine accepts its input if there exists at least one sequence of configurations that ends in acceptance.

EXAMPLE This nondeterministic listener waits to hear the remote signal `0101110`. That is, it accepts the language $\{\sigma \frown \texttt{0101110} \mid \sigma \in \mathbb{B}^*\}$.



For example, if this listener hears a string $\sigma = \texttt{010101110}$, with three `01`'s in the prefix instead of two then it ends in an accepting state. In terms of our second framing, it guesses that it should ignore the first `01` but not the second.

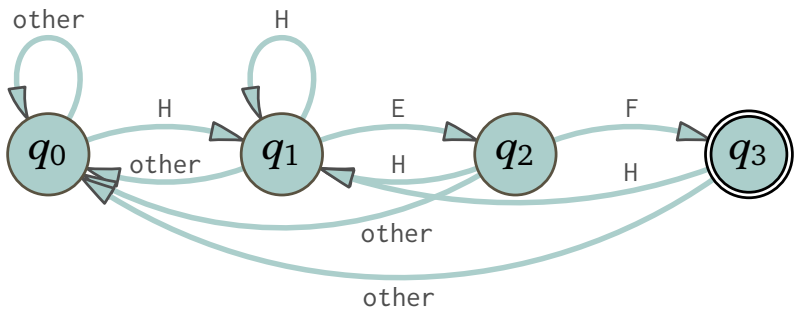
Nondeterminism can simplify thinking about machines

EXAMPLE This nondeterministic Finite State machine accepts only those strings ending in HEF. (The alphabet Σ is the capital ASCII letters.)

Nondeterminism can simplify thinking about machines

EXAMPLE This nondeterministic Finite State machine accepts only those strings ending in HEF. (The alphabet Σ is the capital ASCII letters.)

Contrast that with what we got when we did this job using a deterministic machine.



The nondeterministic machine is easier to understand and more obviously matches its specification.

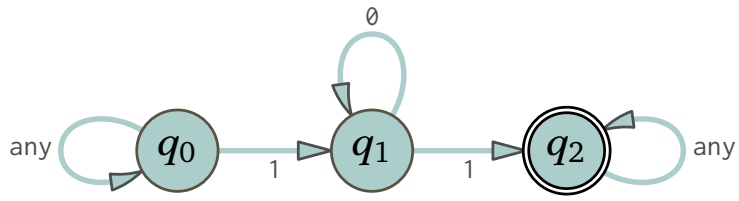
EXAMPLE This nondeterminstic Finite State machine $\mathcal{M}$ with alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ accepts an input string only if it has a in the third position from the end.

Thus $\mathcal{L}(\mathcal{M}) = \{\mathsf{aaa}, \mathsf{aab}, \mathsf{aba}, \mathsf{abb}, \mathsf{aaaa}, \mathsf{aaab}, \mathsf{aaba}, \mathsf{aabb}, \mathsf{baaa}, \mathsf{baab}, \ldots\}$.

EXAMPLE This nondeterminstic Finite State machine $\mathcal{M}$ with alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ accepts an input string only if it has a in the third position from the end.

Thus $\mathcal{L}(\mathcal{M}) = \{\mathsf{aaa}, \mathsf{aab}, \mathsf{aba}, \mathsf{abb}, \mathsf{aaaa}, \mathsf{aaab}, \mathsf{aaba}, \mathsf{aabb}, \mathsf{baaa}, \mathsf{baab}, \ldots\}$.

EXAMPLE This nondeterministic machine $\hat{\mathcal{M}}$ with alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ accepts an input string only if it contains a substring with a 1 followed by any number of 0's, including possibly zero-many 0's, followed by a second 1.



Thus $\mathcal{L}(\hat{\mathcal{M}}) = \{\mathtt{11}, \mathtt{011}, \mathtt{110}, \mathtt{111}, \mathtt{0011}, \mathtt{0101}, \mathtt{0110}, \mathtt{0111}, \ldots\}$.

ε transitions

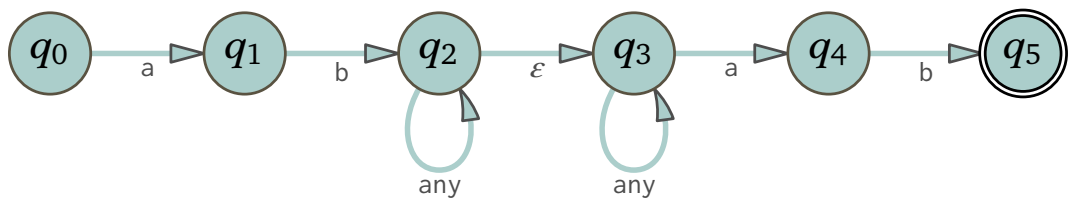
Another extension, beyond nondeterminism, is to allow ε transitions or ε moves. We alter the definition of a nondeterministic Finite State machine so that instead of $\Delta\colon Q \times \Sigma \to \mathcal{P}(Q)$ the transition function's signature is $\Delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$. The associated behavior is that the machine can transition spontaneously, without consuming any input.

ε transitions

Another extension, beyond nondeterminism, is to allow ε transitions or ε moves. We alter the definition of a nondeterministic Finite State machine so that instead of $\Delta\colon Q \times \Sigma \to \mathcal{P}(Q)$ the transition function's signature is $\Delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$. The associated behavior is that the machine can transition spontaneously, without consuming any input.

EXAMPLE The machine on the left accepts strings with prefix ab. The one on the right accepts strings with suffix ab.

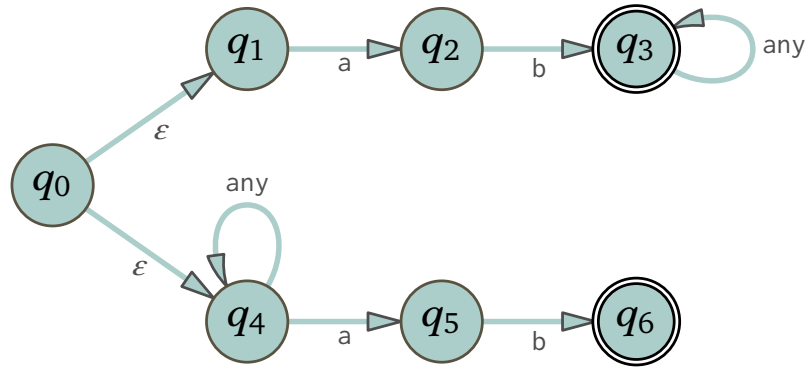
This ε transition joins them together, serially.



The joined machine accepts strings that begin with ab and end with ab.

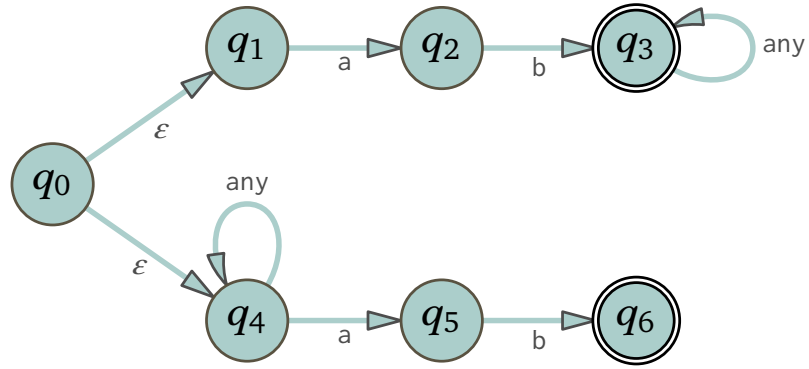
EXAMPLE We can take the same two machines, one that accepts strings with prefix ab and one that accepts strings with suffix ab.

and join them in parallel. The result accepts strings that begin with ab or end with ab.

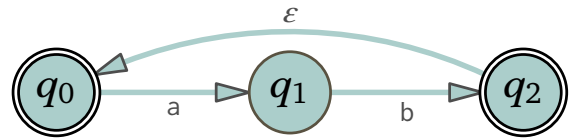


Example We can take the same two machines, one that accepts strings with prefix ab and one that accepts strings with suffix ab.

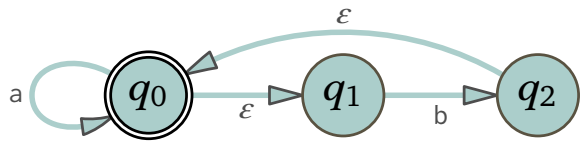
and join them in parallel. The result accepts strings that begin with ab or end with ab.



Example Without the ε edge this machine's language is $\mathcal{L} = \{\varepsilon, \mathsf{ab}\}$, while with it the language is $\mathcal{L}^* = \{(\mathsf{ab})^n \mid n \in \mathbb{N}\}$.



EXAMPLE For this nondeterministic Finite State machine with ε transitions



the computation tree on input aab is below. The ε moves are shown vertically, within the white stripes.

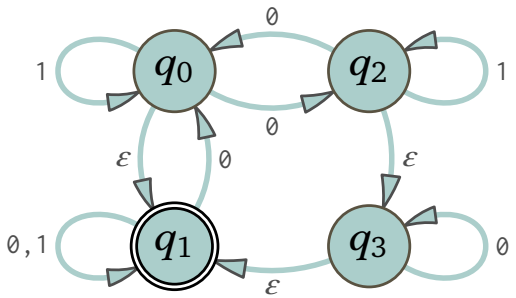
Input:	a		a		b	
						q_1
						$\perp$
						q_0
						$\perp$
q_1		q_1		q_1	$\vdash$	q_2
$\perp$		$\perp$		$\perp$		
q_0	$\vdash$	q_0	$\vdash$	q_0		
Step: 0		1		2		3

There is an accepting branch, highlighted.

ε Closure

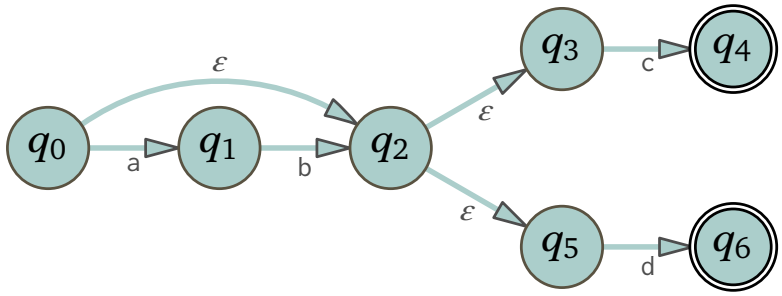
The ε closure function $\hat{E}\colon Q \to \mathcal{P}(Q)$ inputs a state q and returns the set of states that are reachable from q without consuming any input, that is, via ε moves alone.

EXAMPLE For this machine, these are the ε closures.



state	q_0	q_1	q_2	q_3
ε *closure*	$\{q_0, q_1\}$	$\{q_1\}$	$\{q_1, q_2, q_3\}$	$\{q_1, q_3\}$

EXAMPLE We can compute a state's ε closure by proceeding in steps. The table below uses this machine.



	$m = 0$	1	2	3	$\hat{E}(q)$
q_0	$\{q_0\}$	$\{q_0, q_2\}$	$\{q_0, q_2, q_3, q_5\}$	$\{q_0, q_2, q_3, q_5\}$	$\{q_0, q_2, q_3, q_5\}$
q_1	$\{q_1\}$	$\{q_1\}$	$\{q_1\}$	$\{q_1\}$	$\{q_1\}$
q_2	$\{q_2\}$	$\{q_2, q_3, q_5\}$	$\{q_2, q_3, q_5\}$	$\{q_2, q_3, q_5\}$	$\{q_2, q_3, q_5\}$
q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$
q_4	$\{q_4\}$	$\{q_4\}$	$\{q_4\}$	$\{q_4\}$	$\{q_4\}$
q_5	$\{q_5\}$	$\{q_5\}$	$\{q_5\}$	$\{q_5\}$	$\{q_5\}$

The initial column's set contains only the row's state. To move to column 1, add the states reachable with one ε move from the initial state. Thus for the top row, that machine has an ε move from q_0 to q_2 so the $m = 1$ set adds q_2 to the $m = 0$ set. To move to column 2, add states reachable with an ε move from any element of column 1's set (which are thus at most two ε moves away from the initial state). Doing the same for the third column does not change the sets in any row, so we are done.

The action of a nondeterministic Finite State machine with ε moves

A configuration is a pair $\mathcal{C} = \langle q, \tau \rangle \in Q \times \Sigma^*$. These machines start in an initial configuration $\mathcal{C}_0 = \langle q_0, \tau_0 \rangle$, with initial state q_0 and input τ_0. As earlier, we will specify when two configurations are related by '$\vdash$' and then we will say that a machine accepts its input if there is at least one sequence of related configurations that ends in a halting configuration whose state is accepting.

Consider a configuration $\mathcal{C}_s = \langle q, \tau_s \rangle$, in order to describe under what circumstances $\mathcal{C}_s \vdash \hat{\mathcal{C}}$ for some next $\hat{\mathcal{C}} = \langle \hat{q}, \hat{\tau} \rangle$. There are two possibilities. The first is the same as for any nondeterministic machine: τ is not empty and $\hat{\tau}$ comes from popping τ's leading character, $\tau = c^\frown \hat{\tau}$ where $c \in \Sigma$, and $\hat{q}$ is a member of $\Delta(q, c)$. The second possibility differs from the description for machines without ε moves: $\hat{\tau} = \tau$ and $\hat{q}$ is a member of the ε closure $\hat{E}(q)$ with $\hat{q} \neq q$ (the significance of $\hat{\tau} = \tau$ is that the machine doesn't consume any input characters).

With that, a computation of a nondeterministic Finite State machine with ε transitions is the set containing all of the sequences of transitions that are as described. A sequence ends in a halting configuration if its final pair has an empty tape string, so that the sequence has the form $\mathcal{C}_0 = \langle q_0, \tau_0 \rangle \vdash \mathcal{C}_1 \vdash \cdots \mathcal{C}_h = \langle q_h, \varepsilon \rangle$. The machine accepts τ_0 if there is at least one such sequence where q_h is an element of F. If there is no such sequence then the machine rejects τ_0.

Equivalence of machine types

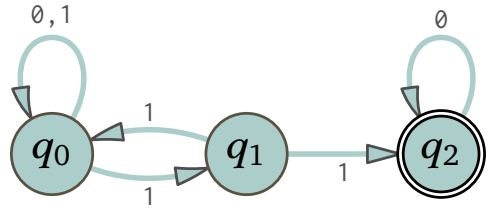
We will show how to convert any nondeterministic Finite State machine into a deterministic machine doing the same job. This may help obviate qualms over 'guessing'.

THEOREM The class of languages recognized by nondeterministic Finite State machines equals the class of languages recognized by deterministic Finite State machines. This remains true if we allow the nondeterministic machines to have ε transitions.

Inclusion in one direction is easy because any deterministic machine is, essentially, a nondeterministic machine. In a deterministic machine the next-state function outputs single states and to make it a nondeterministic machine, just convert those states into singleton sets. Thus the set of languages recognized by deterministic machines is a subset of the set recognized by nondeterministic machines.

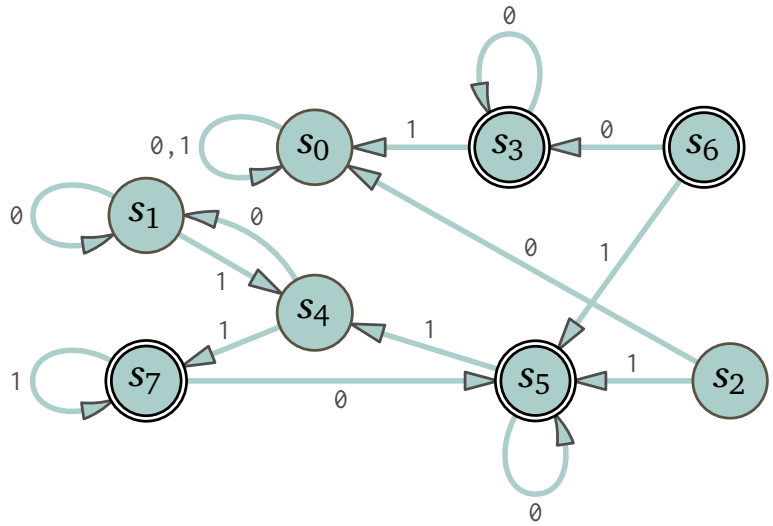
We will demonstrate inclusion in the other direction constructively, starting with nondeterministic machines and building deterministic machines that recognize the same language. The two examples below show the powerset construction.

EXAMPLE Here is a nondeterministic Finite State machine $\mathcal{M}$.

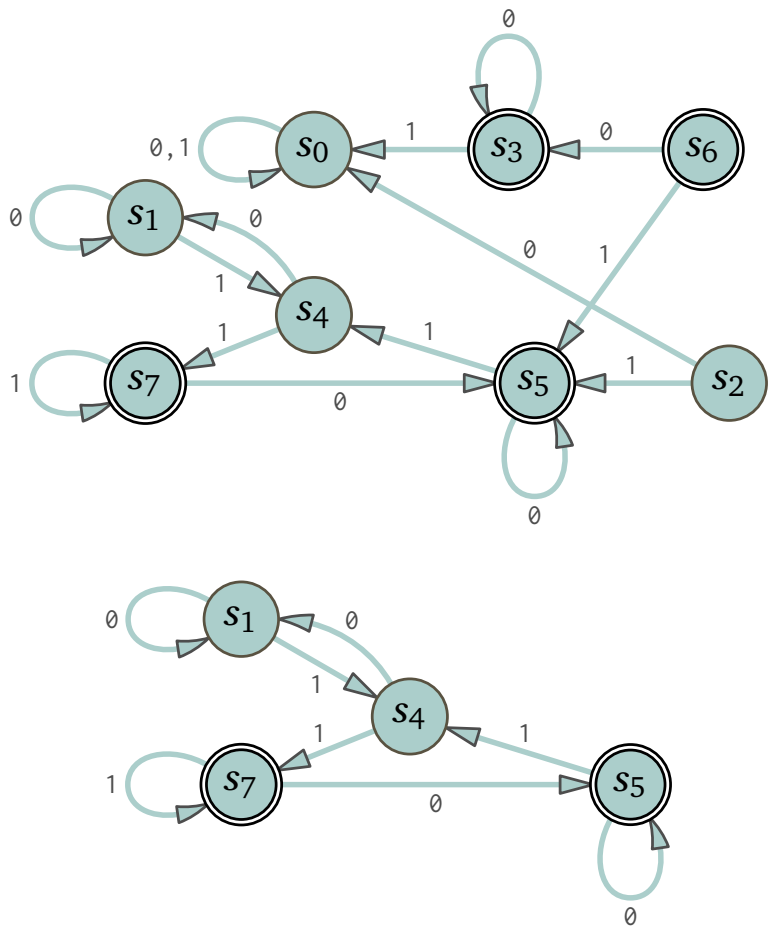


The equivalent deterministic machine's states are the sets of $\mathcal{M}$'s states. The starting state is $s_1 = \{q_0\}$. A state is accepting if it contains any accepting state of the nondeterministic machine.

	0	1
$s_0 = \{\ \}$	s_0	s_0
$s_1 = \{q_0\}$	s_1	s_4
$s_2 = \{q_1\}$	s_0	s_5
$+\ s_3 = \{q_2\}$	s_3	s_0
$s_4 = \{q_0, q_1\}$	s_1	s_7
$+\ s_5 = \{q_0, q_2\}$	s_5	s_4
$+\ s_6 = \{q_1, q_2\}$	s_3	s_5
$+\ s_7 = \{q_0, q_1, q_2\}$	s_5	s_7



Here is that diagram again, along with the one showing only reachable states.



Algorithm for converting to a deterministic machine

Now we give the algorithm for starting with a nondeterministic machine $\mathcal{M}_N$ and constructing a deterministic machine $\mathcal{M}_D$ with the same behavior, whether or not $\mathcal{M}_N$ has ε transitions. Elements of $\mathcal{M}_D$ are sets of states from $\mathcal{M}_N$. Define the ε closure of a set of states to be the union of the ε closures of its members.

The transition function Δ_D inputs a state $s_i \in \mathcal{M}_D$, that is, $s_i = \{q_{k_0}, \ldots q_{k_i}\}$, along with a tape character $c \in \Sigma$. First apply $\mathcal{M}_N$'s next state function to s_i's elements to get $\hat{s}_{i,c} = \Delta_N(q_{k_0}, c) \cup \cdots \Delta_N(q_{k_i}, c)$. Second, where $\hat{s}_{i,c} = \{q_{j_0}, \ldots q_{j_i}\}$, finish by taking the ε closure of all of $\hat{s}$'s elements, $\Delta_D(s_i, c) = \hat{E}(q_{j_0}) \cup \cdots \hat{E}(q_{j_i})$. (For machines without ε transitions this second part has no effect.)

The start state of $\mathcal{M}_D$ is the ε closure of q_0 (for machines without ε moves this is $\{q_0\}$). A state of $\mathcal{M}_D$ is accepting if it contains any of $\mathcal{M}_N$'s accepting states.

Algorithm for converting to a deterministic machine

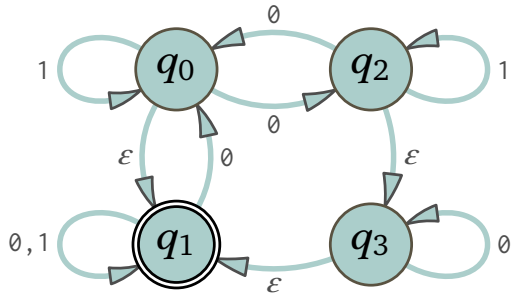
Now we give the algorithm for starting with a nondeterministic machine $\mathcal{M}_N$ and constructing a deterministic machine $\mathcal{M}_D$ with the same behavior, whether or not $\mathcal{M}_N$ has ε transitions. Elements of $\mathcal{M}_D$ are sets of states from $\mathcal{M}_N$. Define the ε closure of a set of states to be the union of the ε closures of its members.

The transition function Δ_D inputs a state $s_i \in \mathcal{M}_D$, that is, $s_i = \{q_{k_0}, \ldots q_{k_i}\}$, along with a tape character $c \in \Sigma$. First apply $\mathcal{M}_N$'s next state function to s_i's elements to get $\hat{s}_{i,c} = \Delta_N(q_{k_0}, c) \cup \cdots \Delta_N(q_{k_i}, c)$. Second, where $\hat{s}_{i,c} = \{q_{j_0}, \ldots q_{j_i}\}$, finish by taking the ε closure of all of $\hat{s}$'s elements, $\Delta_D(s_i, c) = \hat{E}(q_{j_0}) \cup \cdots \hat{E}(q_{j_i})$. (For machines without ε transitions this second part has no effect.)

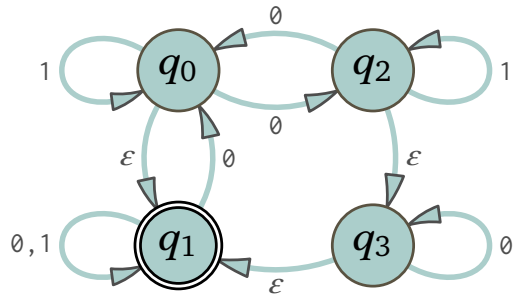
The start state of $\mathcal{M}_D$ is the ε closure of q_0 (for machines without ε moves this is $\{q_0\}$). A state of $\mathcal{M}_D$ is accepting if it contains any of $\mathcal{M}_N$'s accepting states.

As an example, the next slide does the conversion for this machine.

$\mathcal{M}_N$:



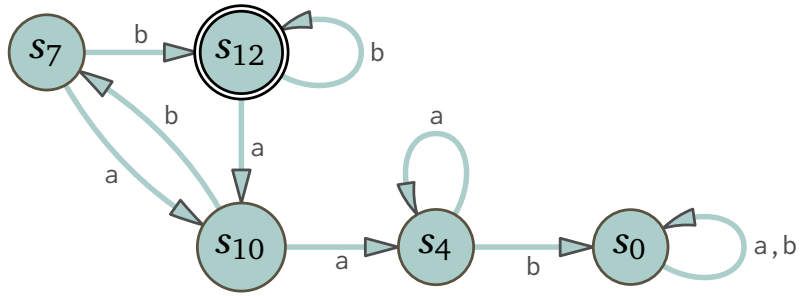
$\mathcal{M}_N$:



The start state is $\hat{E}(\{q_0\}) = s_7$. A state of $\mathcal{M}_D$ is accepting if and only if it contains q_1.

	$\hat{s}_{i,\mathsf{a}}$	$\Delta_D(s_i, \mathsf{a})$	$\hat{s}_{i,\mathsf{b}}$	$\Delta_D(s_i, \mathsf{b})$
$s_0 = \{\,\}$	$\{\,\}$	$\{\,\} = s_0$	$\{\,\}$	$\{\,\} = s_0$
$s_1 = \{q_0\}$	$\{q_2\}$	$\{q_2\} = s_3$	$\{q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$
$+\ s_2 = \{q_1\}$	$\{\,\}$	$\{\,\} = s_0$	$\{\,\}$	$\{\,\} = s_0$
$s_3 = \{q_2\}$	$\{\,\}$	$\{\,\} = s_0$	$\{q_0\}$	$\{q_0, q_3\} = s_7$
$s_4 = \{q_3\}$	$\{q_3\}$	$\{q_3\} = s_4$	$\{\,\}$	$\{\,\} = s_0$
$+\ s_5 = \{q_0, q_1\}$	$\{q_2\}$	$\{q_2\} = s_3$	$\{q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$
$s_6 = \{q_0, q_2\}$	$\{q_2\}$	$\{q_2\} = s_3$	$\{q_0, q_1\}$	$\{q_0.q_1, q_3\} = s_{12}$
$s_7 = \{q_0, q_3\}$	$\{q_2, q_3\}$	$\{q_2, q_3\} = s_{10}$	$\{q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$
$+\ s_8 = \{q_1, q_2\}$	$\{\,\}$	$\{\,\} = s_0$	$\{q_0\}$	$\{q_0, q_3\} = s_7$
$+\ s_9 = \{q_1, q_3\}$	$\{q_3\}$	$\{q_3\} = s_4$	$\{\,\}$	$\{\,\} = s_0$
$s_{10} = \{q_2, q_3\}$	$\{q_3\}$	$\{q_3\} = s_4$	$\{q_0\}$	$\{q_0, q_3\} = s_7$
$+\ s_{11} = \{q_0, q_1, q_2\}$	$\{q_2\}$	$\{q_2\} = s_3$	$\{q_0, q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$
$+\ s_{12} = \{q_0, q_1, q_3\}$	$\{q_2, q_3\}$	$\{q_2, q_3\} = s_{10}$	$\{q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$
$s_{13} = \{q_0, q_2, q_3\}$	$\{q_2, q_3\}$	$\{q_2, q_3\} = s_{10}$	$\{q_0, q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$
$+\ s_{14} = \{q_1, q_2, q_3\}$	$\{q_3\}$	$\{q_3\} = s_4$	$\{q_0\}$	$\{q_0, q_3\} = s_7$
$+\ s_{15} = \{q_0, q_1, q_2, q_3\}$	$\{q_2, q_3\}$	$\{q_2, q_3\} = s_{10}$	$\{q_0, q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$

Here is that deterministic machine as a graph, showing only the reachable states.



Regular expressions

Examples

The languages accepted by a Finite State machine have patterns. We exploit these patterns to describe them with regular expressions. We will start with some examples and then give a definition.

EXAMPLE The regular expression `p(a|e|i|o|u)t` describes the language of strings that start with `p`, have a vowel in the middle, and end with `t`. That is, this regular expression describes the language consisting of five words, $\mathcal{L} = \{\texttt{pat}, \texttt{pet}, \texttt{pit}, \texttt{pot}, \texttt{put}\}$.

The pipe operator, '|', which is a kind of 'or', and the parentheses, which provide grouping, are not part of the strings being described; they are metacharacters.

Besides the pipe and parentheses, the regular expression in that example also describes concatenation since the language consists of strings where the initial `p` is concatenated with a vowel, which in turn is concatenated with `t`.

Examples

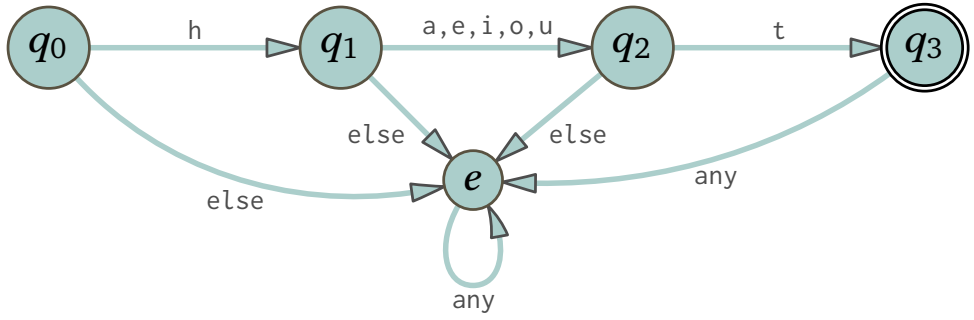
The languages accepted by a Finite State machine have patterns. We exploit these patterns to describe them with regular expressions. We will start with some examples and then give a definition.

EXAMPLE The regular expression `p(a|e|i|o|u)t` describes the language of strings that start with `p`, have a vowel in the middle, and end with `t`. That is, this regular expression describes the language consisting of five words, $\mathcal{L} = \{\texttt{pat}, \texttt{pet}, \texttt{pit}, \texttt{pot}, \texttt{put}\}$.

The pipe operator, '|', which is a kind of 'or', and the parentheses, which provide grouping, are not part of the strings being described; they are metacharacters.

Besides the pipe and parentheses, the regular expression in that example also describes concatenation since the language consists of strings where the initial `p` is concatenated with a vowel, which in turn is concatenated with `t`.

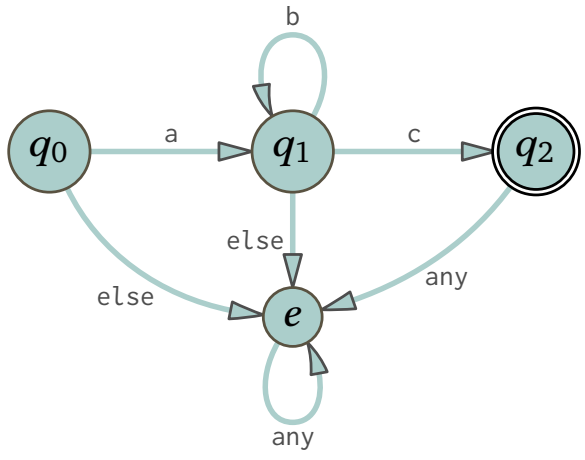
Those strings are accepted by this Finite State machine.



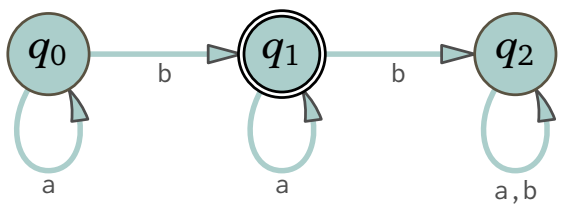
EXAMPLE The regular expression ab*c describes the language whose words begin with an a, followed by any number of b's (including possibly zero-many b's), and ending with a c. Thus, '*' means 'repeat the prior thing any number of times, including possibly zero-many times'. This regular expression describes the language
$\mathcal{L} = \{\mathsf{ac}, \mathsf{abc}, \mathsf{abbc}, \ldots\}$.

EXAMPLE The regular expression ab*c describes the language whose words begin with an a, followed by any number of b's (including possibly zero-many b's), and ending with a c. Thus, '*' means 'repeat the prior thing any number of times, including possibly zero-many times'. This regular expression describes the language
$\mathcal{L} = \{\text{ac}, \text{abc}, \text{abbc}, \ldots\}$.

That is the language of this machine.



EXAMPLE The strings that are accepted by this machine match a*ba*.



Definition: syntax

DEFINITION Let Σ be an alphabet not containing the metacharacters ')', '(', '|', or '*'. A regular expression over Σ is a string that can be derived from the grammar

$$\begin{aligned}
\langle regex \rangle &\rightarrow \langle concat \rangle \\
&\quad | \ (\ \langle regex \rangle\ \text{'|'}\ \langle concat \rangle\ ) \\
\langle concat \rangle &\rightarrow \langle simple \rangle \\
&\quad | \ (\ \langle concat \rangle\ \langle simple \rangle\ ) \\
\langle simple \rangle &\rightarrow \langle char \rangle \\
&\quad | \ (\ \langle simple \rangle\ *\ ) \\
\langle char \rangle &\rightarrow \varnothing \ | \ \varepsilon \ | \ x_0 \ | \ x_1 \ | \ \ldots
\end{aligned}$$

where the x_i characters are members of Σ.

Semantics

The language described by the regular expression $\varnothing$ is the empty language, $\mathcal{L}(\varnothing) = \varnothing$. The language described by the regular expression consisting of only the character ε is the one-element language containing only the empty string, $\mathcal{L}(\varepsilon) = \{\varepsilon\} = \{\text{''}\}$. If the regular expression consists of a single character from the alphabet Σ then the language that it describes contains only one string and that string has only that single character, as in $\mathcal{L}(\mathsf{a}) = \{\mathsf{a}\}$.

We finish by doing the operations. Start with regular expressions R_0 and R_1 describing languages $\mathcal{L}(R_0)$ and $\mathcal{L}(R_1)$. Then the pipe symbol describes the union of the languages, so that $\mathcal{L}(R_0|R_1) = \mathcal{L}(R_0) \cup \mathcal{L}(R_1)$. Concatenation of the regular expressions describes concatenation of the languages, $\mathcal{L}(R_0R_1) = \mathcal{L}(R_0)^\frown \mathcal{L}(R_1)$. And $\mathcal{L}({R_0}^*) = \mathcal{L}(R_0)^*$, so the Kleene star of the regular expression describes the star of the language.

Examples

EXAMPLE The language consisting of strings of a's whose length is a multiple of three, $\mathcal{L} = \{ a^{3k} \mid k \in \mathbb{N} \} = \{ \varepsilon, \mathtt{aaa}, \mathtt{aaaaaa}, \ldots \}$, is described by $(\mathtt{aaa})*$.

Examples

EXAMPLE The language consisting of strings of a's whose length is a multiple of three, $\mathcal{L} = \{ a^{3k} \mid k \in \mathbb{N} \} = \{ \varepsilon, \mathsf{aaa}, \mathsf{aaaaaa}, \ldots \}$, is described by $(\mathsf{aaa})*$.

EXAMPLE The empty string character ε is handy to mark things as optional. Thus $\mathsf{a}*(\varepsilon|\mathsf{b})$ describes the language of strings that have any number of a's and optionally end in one b, $\mathcal{L} = \{ \varepsilon, \mathsf{b}, \mathsf{a}, \mathsf{ab}, \mathsf{aa}, \mathsf{aab}, \ldots \}$. Similarly, to describe the language consisting of words with between three and five a's, $\mathcal{L} = \{ \mathsf{aaa}, \mathsf{aaaa}, \mathsf{aaaaa} \}$, we can use $\mathsf{aaa}(\varepsilon|\mathsf{a}|\mathsf{aa})$.

Examples

EXAMPLE The language consisting of strings of a's whose length is a multiple of three, $\mathcal{L} = \{ a^{3k} \mid k \in \mathbb{N} \} = \{ \varepsilon, \mathsf{aaa}, \mathsf{aaaaaa}, \ldots \}$, is described by $(\mathsf{aaa})*$.

EXAMPLE The empty string character ε is handy to mark things as optional. Thus $\mathsf{a}*(\varepsilon|\mathsf{b})$ describes the language of strings that have any number of a's and optionally end in one b, $\mathcal{L} = \{ \varepsilon, \mathsf{b}, \mathsf{a}, \mathsf{ab}, \mathsf{aa}, \mathsf{aab}, \ldots \}$. Similarly, to describe the language consisting of words with between three and five a's, $\mathcal{L} = \{ \mathsf{aaa}, \mathsf{aaaa}, \mathsf{aaaaa} \}$, we can use $\mathsf{aaa}(\varepsilon|\mathsf{a}|\mathsf{aa})$.

EXAMPLE To match any character we can list them all. The language over $\Sigma = \{ \mathsf{a}, \mathsf{b}, \mathsf{c} \}$ of three-letter words ending in bc is $\{ \mathsf{abc}, \mathsf{bbc}, \mathsf{cbc} \}$. The regular expression $(\mathsf{a}|\mathsf{b}|\mathsf{c})\mathsf{bc}$ describes it. Another regular expression that describes this language is $(\mathsf{abc})|(\mathsf{bbc})|(\mathsf{cbc})$.

Examples

EXAMPLE The language consisting of strings of a's whose length is a multiple of three, $\mathcal{L} = \{ a^{3k} \mid k \in \mathbb{N} \} = \{ \varepsilon, \mathsf{aaa}, \mathsf{aaaaaa}, \ldots \}$, is described by $(\mathsf{aaa})*$.

EXAMPLE The empty string character ε is handy to mark things as optional. Thus $\mathsf{a}*(\varepsilon|\mathsf{b})$ describes the language of strings that have any number of a's and optionally end in one b, $\mathcal{L} = \{ \varepsilon, \mathsf{b}, \mathsf{a}, \mathsf{ab}, \mathsf{aa}, \mathsf{aab}, \ldots \}$. Similarly, to describe the language consisting of words with between three and five a's, $\mathcal{L} = \{ \mathsf{aaa}, \mathsf{aaaa}, \mathsf{aaaaa} \}$, we can use $\mathsf{aaa}(\varepsilon|\mathsf{a}|\mathsf{aa})$.

EXAMPLE To match any character we can list them all. The language over $\Sigma = \{ \mathsf{a}, \mathsf{b}, \mathsf{c} \}$ of three-letter words ending in bc is $\{ \mathsf{abc}, \mathsf{bbc}, \mathsf{cbc} \}$. The regular expression $(\mathsf{a}|\mathsf{b}|\mathsf{c})\mathsf{bc}$ describes it. Another regular expression that describes this language is $(\mathsf{abc})|(\mathsf{bbc})|(\mathsf{cbc})$.

EXAMPLE The language $\{ \mathsf{b}, \mathsf{bc}, \mathsf{bcc}, \mathsf{ab}, \mathsf{abc}, \mathsf{abcc}, \mathsf{aab}, \ldots \}$ has words starting with any number of a's (including zero-many a's), followed by a single b, and then ending in fewer than three c's. To describe it we can use $\mathsf{a}*\mathsf{b}(\varepsilon|\mathsf{c}|\mathsf{cc})$.

More examples

EXAMPLE Produce a regular expression to match each description. For each, take the alphabet to be the set of bits, $\mathbb{B} = \{\mathtt{0}, \mathtt{1}\}$.

1. The set of bitstrings that end in three consecutive 1's.

More examples

EXAMPLE Produce a regular expression to match each description. For each, take the alphabet to be the set of bits, $\mathbb{B} = \{\texttt{0}, \texttt{1}\}$.

1. The set of bitstrings that end in three consecutive 1's.
2. The set of bitstrings with at least one `0`.

More examples

EXAMPLE Produce a regular expression to match each description. For each, take the alphabet to be the set of bits, $\mathbb{B} = \{\texttt{0}, \texttt{1}\}$.

1. The set of bitstrings that end in three consecutive 1's.
2. The set of bitstrings with at least one 0.
3. The bitstrings containing at least three 1's.

More examples

EXAMPLE Produce a regular expression to match each description. For each, take the alphabet to be the set of bits, $\mathbb{B} = \{\texttt{0}, \texttt{1}\}$.

1. The set of bitstrings that end in three consecutive 1's.
2. The set of bitstrings with at least one 0.
3. The bitstrings containing at least three 1's.
4. The set of bitstrings starting with a digit other than the one that it ends with.

Kleene's Theorem

THEOREM (KLEENE'S THEOREM) A language is recognized by a Finite State machine if and only if that language is described by a regular expression.

Kleene's Theorem

THEOREM (KLEENE'S THEOREM) A language is recognized by a Finite State machine if and only if that language is described by a regular expression.

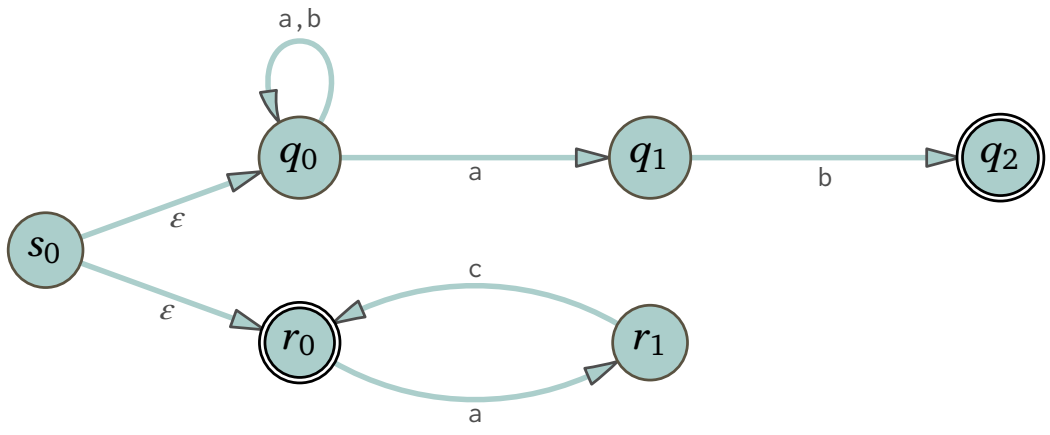
LEMMA If a language is described by a regular expression then there is a Finite State machine recognizing that language.

PF We will show that for any regular expression R there is a machine that accepts strings matching that expression. We give nondeterministic machines, but of course we can convert them to deterministic ones if desired.

Start with regular expressions consisting of a single character. If $R = \varnothing$ then $\mathcal{L}(R) = \{\ \}$ and the machine on the left below recognizes this language. If $R = \varepsilon$ then $\mathcal{L}(R) = \{\varepsilon\}$ and the machine in the middle recognizes it. If the regular expression is a character from the alphabet such as $R = \mathtt{a}$ then the machine on the right works.

We finish by handling the three operations. Let R_0 and R_1 be regular expressions. The inductive hypothesis gives a machine $\mathcal{M}_0$ whose language is described by R_0 and a machine $\mathcal{M}_1$ whose language is described by R_1.

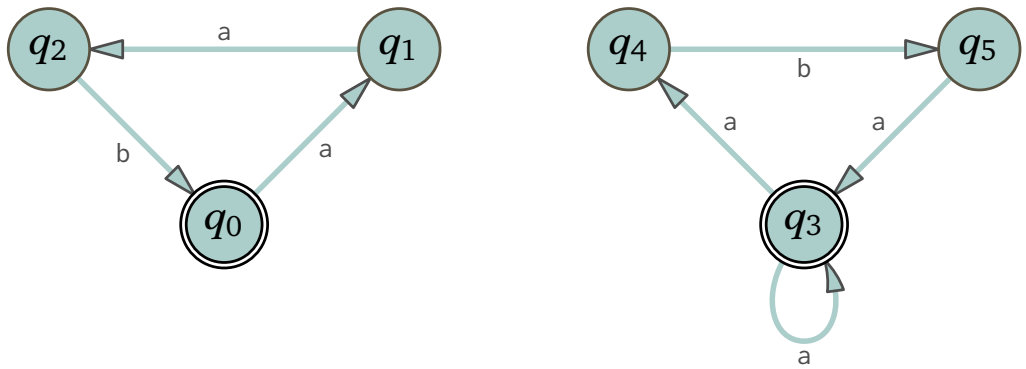
First consider alternation, $R = R_0 \,|\, R_1$. Create the machine recognizing the language described by R by joining those two machines in parallel: introduce a new state s and use ε transitions to connect s to the start states of $\mathcal{M}_0$ and $\mathcal{M}_1$. For example, this shows two machines combined



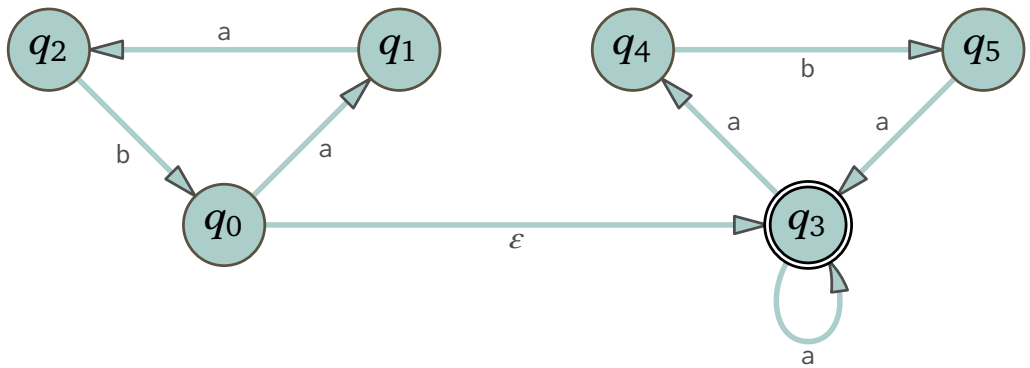
The top nondeterministic machine's language is $\{\sigma \in \Sigma^* \mid \sigma \text{ ends in ab}\}$ and the bottom machine's language is $\{\sigma \in \Sigma^* \mid \sigma = (\mathsf{ac})^n \text{ for some } n \in \mathbb{N}\}$. The language for the entire machine is the union.

$$\mathcal{L} = \{\sigma \in \Sigma^* \mid \text{either } \sigma \text{ ends in ab or } \sigma = (\mathsf{ac})^n \text{ for } n \in \mathbb{N}\}$$

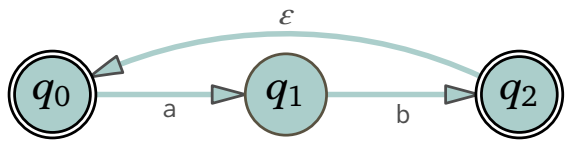
Next consider concatenation, $R = R_0 \frown R_1$. Join the two machines serially: for each accepting state in $\mathcal{M}_0$, make an ε transition to the start state of $\mathcal{M}_1$ and then convert all of the accepting states of $\mathcal{M}_0$ to be non-accepting states. For example, the machine on the left below recognizes the language of repetitions of the string aab, $\mathcal{L}_0 = \{(\mathsf{aab})^i \mid i \in \mathbb{N}\}$. The machine on the right recognizes repetitions of either a or aba, $\mathcal{L}_1 = \{\sigma_0 \frown \cdots \sigma_{j-1} \mid j \in \mathbb{N} \text{ and } \sigma_k = \mathsf{a} \text{ or } \sigma_k = \mathsf{aba} \text{ for } 0 \leq k \leq j-1\}$.



The combined machine accepts strings in the concatenation of those languages, $\mathcal{L}(\mathcal{M}) = \mathcal{L}_0 \frown \mathcal{L}_1$.



Finally consider Kleene star, $R = (R_0)*$. For each accepting state in the machine $\mathcal{M}_0$ that is not the start state, make an ε transition to the start state and then make the start state an accepting state. For example, without the ε edge this machine's language is $\{\varepsilon, \mathsf{ab}\}$, while with it the language is $\{(\mathsf{ab})^n \mid n \in \mathbb{N}\}$.



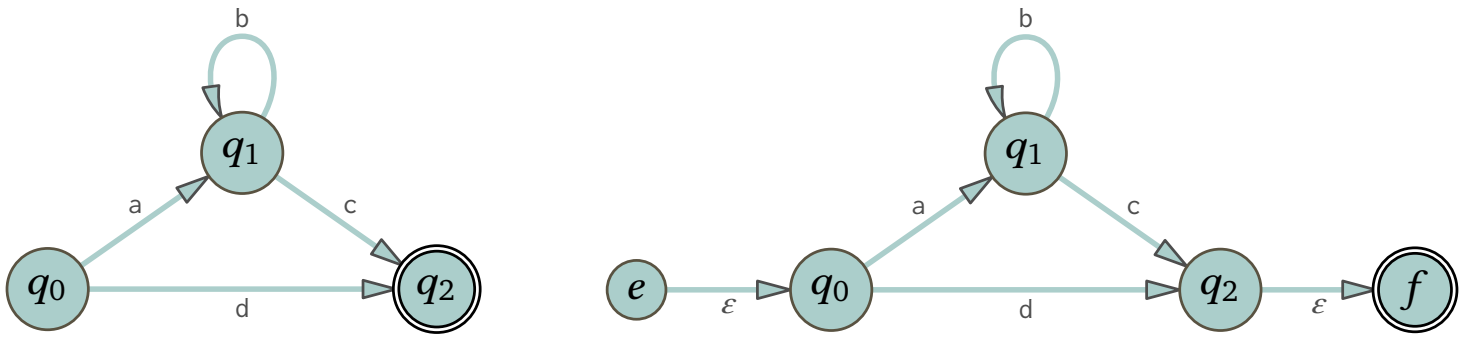
Those are all the allowed operations, and so that ends the proof. ∎

LEMMA Any language recognized by a Finite State machine is described by a regular expression.

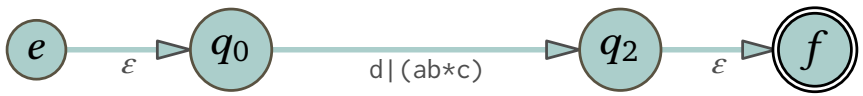
Here is the idea: start with a Finite State machine and eliminate its states one at a time, keeping the accepted language the same.

For this, we generalize the transition graphs to allow edge labels that are regular expressions.

A bit more detail: take a, b, c, and d to be regular expressions. Start by introducing a new start state guaranteed to have no incoming edges, e, and a new final state guaranteed to be unique, f.

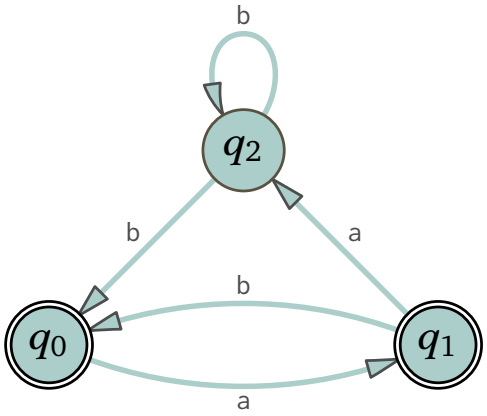


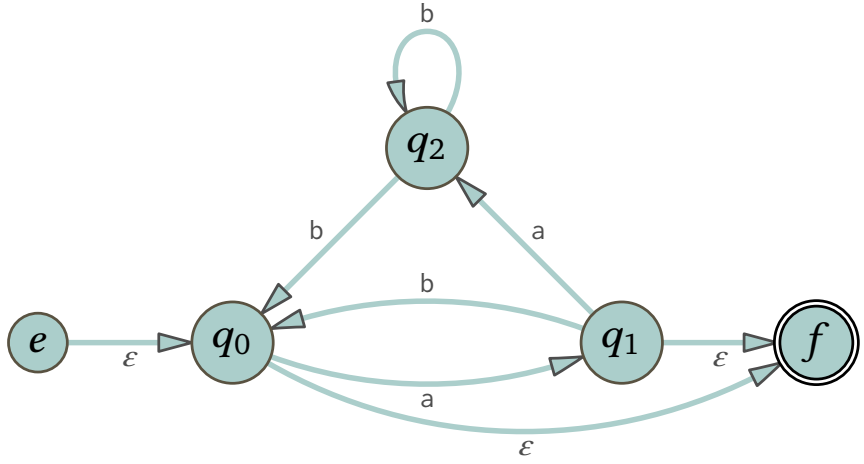
Then eliminate q_1 as below.



The proof is in the book. Here is an illustration.

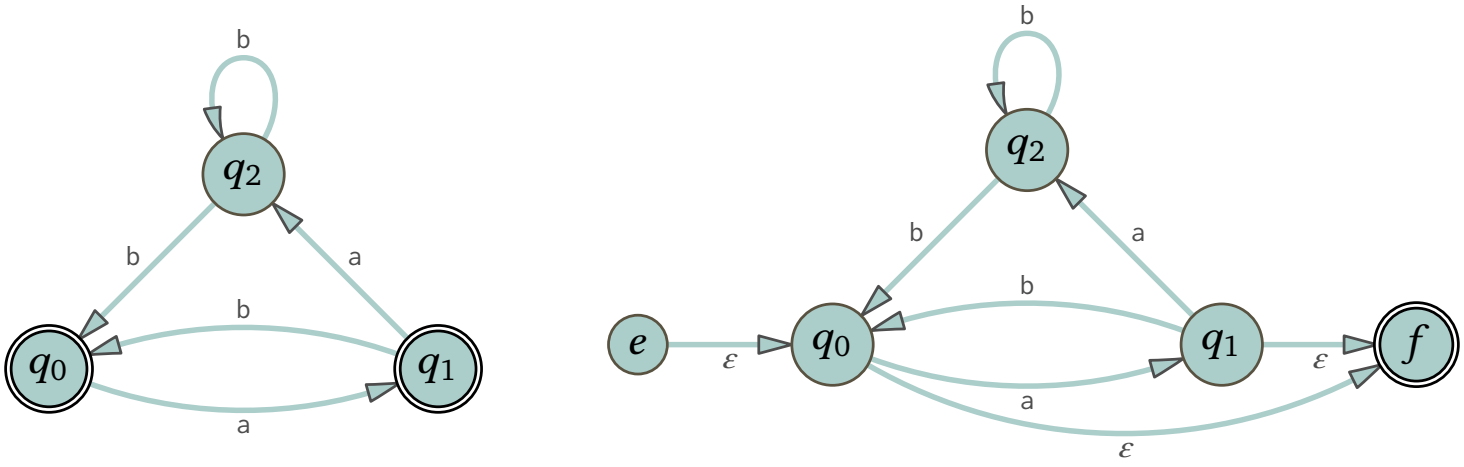
EXAMPLE Consider $\mathcal{M}$ on the left. Introduce e and f to get $\hat{\mathcal{M}}$ on the right.



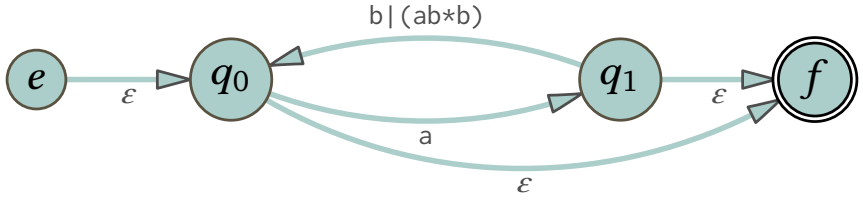


The proof is in the book. Here is an illustration.

EXAMPLE Consider $\mathcal{M}$ on the left. Introduce e and f to get $\hat{\mathcal{M}}$ on the right.



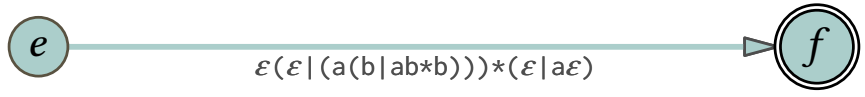
Start by eliminating q_2. That gives this machine.



Next eliminate q_1.

Next eliminate q_1.

Finally, eliminate q_0.



That regular expression simplifies. For instance, $a\varepsilon = a$. The result is `(a(b|ab*b))*(ε|a)`.

Regular languages

Definition

DEFINITION A regular language is one that is recognized by some Finite State machine or equivalently, described by a regular expression.

EXAMPLE The set $\{ \mathsf{ab}^n \mid n \in \mathbb{N} \} = \{ \mathsf{a}, \mathsf{ab}, \mathsf{abb} \ldots \}$ is a regular language. Another language that is regular is any finite set of strings.

Definition

DEFINITION A regular language is one that is recognized by some Finite State machine or equivalently, described by a regular expression.

EXAMPLE The set $\{ \mathsf{ab}^n \mid n \in \mathbb{N} \} = \{ \mathsf{a}, \mathsf{ab}, \mathsf{abb} \ldots \}$ is a regular language. Another language that is regular is any finite set of strings.

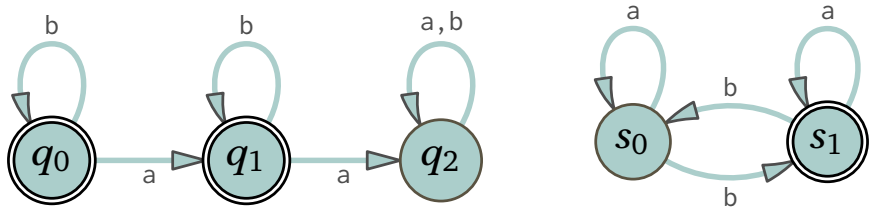
In the proof of Kleene's Theorem, we saw that if two languages are regular then their union is also regular. We say that a structure is closed under an operation if performing that operation on its members always yields another member.

LEMMA The collection of regular languages is closed under the union of two languages, the concatenation of two languages, and the Kleene star of a language.

PF. In the first lemma for the proof of Kleene's Theorem we did the regular expression operations of pipe, concatenation, and Kleene star. Those correspond to set operations; for instance, if R_0 is a regular expression describing the language $\mathcal{L}_0$, and R_1 describes $\mathcal{L}_1$, then the regular expression $R_0 \,|\, R_1$ describes $\mathcal{L}_0 \cup \mathcal{L}_1$. ∎

Product construction

The machine on the left, $\mathcal{M}_0$, accepts strings with fewer than two a's. The one on the right, $\mathcal{M}_1$, accepts strings with an odd number of b's.



The transition tables contain the same information as the pictures.

	Δ_0	a	b
+	q_0	q_1	q_0
+	q_1	q_2	q_1
	q_2	q_2	q_2

	Δ_1	a	b
	s_0	s_0	s_1
+	s_1	s_1	s_0

This machine $\mathcal{M}$ has states that are the members of the cross product $Q_0 \times Q_1$ and transitions that are given by $\Delta\big((q_i, r_j)\big) = \big(\Delta_0(q_i), \Delta_1(r_j)\big)$. It starts in (q_0, r_0).

Δ	a	b
(q_0, s_0)	(q_1, s_0)	(q_0, s_1)
(q_0, s_1)	(q_1, s_1)	(q_0, s_0)
(q_1, s_0)	(q_2, s_0)	(q_1, s_1)
(q_1, s_1)	(q_2, s_1)	(q_1, s_0)
(q_2, s_0)	(q_2, s_0)	(q_2, s_1)
(q_2, s_1)	(q_2, s_1)	(q_2, s_0)

If we feed the string aba to $\mathcal{M}$ then the machine's states go from (q_0, s_0) to (q_1, s_0), to (q_1, s_1), and then to (q_2, s_1). This is simply because $\mathcal{M}_0$ passes from q_0 to q_1, to q_1 again, and then to q_2, while $\mathcal{M}_1$ does s_0 to s_0, to s_1, and to s_1. That is, we can view that $\mathcal{M}$ runs $\mathcal{M}_0$ and $\mathcal{M}_1$ in parallel.

We have not yet specified which states are accepting. We can vary the language accepted by a product machine by varying the set of accepting states. One possibility is to stipulate, as on the left below, that the accepting states (q_i, s_j) are the ones where both q_i and s_j are accepting. That means $\mathcal{M}$ accepts a string σ if and only if both $\mathcal{M}_0$ and $\mathcal{M}_1$ accept it.

		a	b
	(q_0, s_0)	(q_1, s_0)	(q_0, s_1)
+	(q_0, s_1)	(q_1, s_1)	(q_0, s_0)
	(q_1, s_0)	(q_2, s_0)	(q_1, s_1)
+	(q_1, s_1)	(q_2, s_1)	(q_1, s_0)
	(q_2, s_0)	(q_2, s_0)	(q_2, s_1)
	(q_2, s_1)	(q_2, s_1)	(q_2, s_0)

		a	b
+	(q_0, s_0)	(q_1, s_0)	(q_0, s_1)
	(q_0, s_1)	(q_1, s_1)	(q_0, s_0)
+	(q_1, s_0)	(q_2, s_0)	(q_1, s_1)
	(q_1, s_1)	(q_2, s_1)	(q_1, s_0)
	(q_2, s_0)	(q_2, s_0)	(q_2, s_1)
	(q_2, s_1)	(q_2, s_1)	(q_2, s_0)

Another possibility is to stipulate, as on the right, that the accepting states (q_i, s_j) are the ones where q_i is accepting but s_j is not. That means the machine accepts strings that are in the language of $\mathcal{M}_0$ but not that of $\mathcal{M}_1$.

THEOREM The collection of regular languages is closed under the intersection of two languages, the set difference of two languages, and the set complement of a language.

PF. Start with two Finite State machines, $\mathcal{M}_0$ and $\mathcal{M}_1$, which accept languages $\mathcal{L}_0$ and $\mathcal{L}_1$ over some alphabet Σ. Perform the product construction to get $\mathcal{M}$. If the accepting states of $\mathcal{M}$ are those pairs where both the first and second component states are accepting, in $\mathcal{M}_0$ and $\mathcal{M}_1$, then $\mathcal{M}$ accepts the intersection of the languages, $\mathcal{L}_0 \cap \mathcal{L}_1$. If the accepting states of $\mathcal{M}$ are those pairs where the first component state is accepting but the second is not, then $\mathcal{M}$ accepts the set difference of the languages, $\mathcal{L}_0 - \mathcal{L}_1$. A special case of this second one is when $\mathcal{L}_0$ is the set of all strings, Σ^*, and in this case $\mathcal{M}$ accepts the complement, ${\mathcal{L}_1}^{\mathsf{c}}$. ∎

THEOREM The collection of regular languages is closed under the intersection of two languages, the set difference of two languages, and the set complement of a language.

PF. Start with two Finite State machines, $\mathcal{M}_0$ and $\mathcal{M}_1$, which accept languages $\mathcal{L}_0$ and $\mathcal{L}_1$ over some alphabet Σ. Perform the product construction to get $\mathcal{M}$. If the accepting states of $\mathcal{M}$ are those pairs where both the first and second component states are accepting, in $\mathcal{M}_0$ and $\mathcal{M}_1$, then $\mathcal{M}$ accepts the intersection of the languages, $\mathcal{L}_0 \cap \mathcal{L}_1$. If the accepting states of $\mathcal{M}$ are those pairs where the first component state is accepting but the second is not, then $\mathcal{M}$ accepts the set difference of the languages, $\mathcal{L}_0 - \mathcal{L}_1$. A special case of this second one is when $\mathcal{L}_0$ is the set of all strings, Σ^*, and in this case $\mathcal{M}$ accepts the complement, ${\mathcal{L}_1}^{\mathsf{c}}$. ∎

EXAMPLE Let $\Sigma = \mathbb{B} = \{ 0, 1 \}$. We could show that the language $\mathcal{L} = \{ \sigma 1 \mid \sigma \in \Sigma^* \}$ is regular by producing a Finite State machine to accept it, or by giving a regular expression that describes it. But we can also use closure. That language is the concatenation $\mathcal{L} = \mathcal{L}_0 \frown \mathcal{L}_1$ of $\mathcal{L}_0 = \Sigma^*$ and $\mathcal{L}_1 = \{ 1 \}$, which are both clearly regular. Regular languages are closed under concatenation, so $\mathcal{L}$ is regular.

EXAMPLE The language

$$\mathcal{L} = \{\sigma \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}^* \mid \sigma \text{ does not have the substring } \mathsf{acb}\}$$

it the complement of $\mathcal{L}^{\mathsf{c}} = \{\sigma \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}^* \mid \sigma \text{ has the substring } \mathsf{acb}\}$. The language $\mathcal{L}^{\mathsf{c}}$ is regular because it is described by a regular expression, `(a|b|c)*acb(a|b|c)*`.

EXAMPLE The language

$$\mathcal{L} = \{\sigma \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}^* \mid \sigma \text{ does not have the substring } \mathsf{acb}\}$$

it the complement of $\mathcal{L}^{\mathsf{c}} = \{\sigma \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}^* \mid \sigma \text{ has the substring } \mathsf{acb}\}$. The language $\mathcal{L}^{\mathsf{c}}$ is regular because it is described by a regular expression, `(a|b|c)*acb(a|b|c)*`.

EXAMPLE We can show that the language

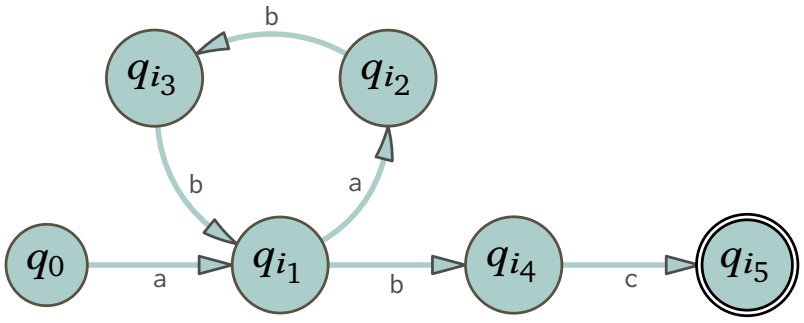
$$\mathcal{L} = \{\sigma \in \{\mathsf{a}, \mathsf{b}\}^* \mid \sigma \text{ contains both of the substrings } \mathsf{acb} \text{ and } \mathsf{cbc}\}$$

is regular by constructing a Finite State machine for each substring or give regular expressions, and then cite that the regular languages are closed under intersection. (By the way, note that the string $\sigma = \mathsf{acbc}$ is a member of $\mathcal{L}$.)

Languages that are not regular

Periodicity

The diagram below shows a machine that accepts aabbbc (it only shows some of the states, those that the machine traverses in processing this input).



Because of the cycle, in addition to aabbbc this machine must also accept $a(abb)^2bc$ since that string takes the machine tthrough the cycle twice. Likewise, this machine accepts $a(abb)^3bc$, and cycling more times pumps out more accepted strings.

Pumping lemma

THEOREM (PUMPING LEMMA) Let $\mathcal{L}$ be a regular language. Then there is a constant $p \in \mathbb{N}$, the pumping length for the language, such that every string $\sigma \in \mathcal{L}$ with $|\sigma| \geq p$ decomposes into three substrings $\sigma = \alpha^\frown \beta^\frown \gamma$ satisfying: (1) the first two components are short, $|\alpha\beta| \leq p$, (2) β is not empty, and (3) all of the strings $\alpha\gamma$, $\alpha\beta^2\gamma$, $\alpha\beta^3\gamma$, ... are also members of the language $\mathcal{L}$.

PF. Suppose that $\mathcal{L}$ is recognized by the Finite State machine $\mathcal{M}$. Take p to be the number of states in $\mathcal{M}$.

Consider a string $\sigma \in \mathcal{L}$ with $|\sigma| \geq p$. Finite State machines perform one transition per character so the number of characters in an input string equals the number of transitions. Thus the number of states, not necessarily distinct ones, that the machine visits is one more than the number of transitions. (For instance, with a one-character input a machine visits two states.) So in processing σ, the machine revisits at least one state. It cycles.

Of the states that are repeated as the machine processes σ, fix the one q that it revisits first. Also fix σ's shortest two prefixes $\langle s_0, \ldots s_i \rangle$ and $\langle s_0, \ldots s_i, \ldots s_j \rangle$ that take the machine to q. That is, i and j are minimal such that $i \neq j$ and the extended transition function gives $\hat{\Delta}(\langle s_0, \ldots s_i \rangle) = \hat{\Delta}(\langle s_0, \ldots s_j \rangle) = q$. Let $\alpha = \langle s_0, \ldots, s_i \rangle$, let $\beta = \langle s_{i+1}, \ldots s_j \rangle$, and let $\gamma = \langle s_{j+1}, \ldots s_k \rangle$. (Think of α as the initial substring of σ that transitions the machine up to the loop, think of β as the substring that takes the machine around the loop once, and γ is the rest of σ. Possibly α is empty. Possibly also γ is empty, or perhaps it includes a substring that transitions the machine around the loop a number of times after β has taken it around the first time. For the machine in $(*)$ above, with $\sigma = \mathsf{a}(\mathsf{abb})^2\mathsf{bc}$ we have $\alpha = \mathsf{a}$, $\beta = \mathsf{abb}$, and $\gamma = \mathsf{abbbc}$.)

These strings satisfy conditions (1) and (2). In particular, choosing q, i, and j to be minimal guarantees that that $|\alpha^\frown \beta| \leq p$ because the machine has p-many states and so a state must repeat by at most the p-th input character. For condition (3), this string

$$\alpha^\frown \gamma = \langle s_0, \ldots s_i, s_{j+1}, \ldots s_k \rangle$$

brings the machine from the start state q_0 to q, and then to the same ending state as did σ. That is, $\hat{\Delta}(\alpha\gamma) = \hat{\Delta}(\alpha\beta\gamma)$ and so the machine accepts $\alpha\gamma$. The other strings in (3) work the same way. For instance, for

$$\alpha^\frown \beta^2{}^\frown \gamma = \langle s_0, \ldots s_i, s_{i+1}, \ldots, s_j, s_{i+1}, \ldots s_j, s_{j+1}, \ldots s_k \rangle$$

the substring α brings the machine from q_0 to q, the first β brings it from q around to q again, the second β makes the machine cycle to q yet again, and finally γ brings it to the same ending state as did σ. ∎

Usually we use the Pumping Lemma to show that a language is not regular.

EXAMPLE Show that this language over $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ is not regular: $\mathcal{L} = \{\mathsf{a}^n\mathsf{b}\mathsf{a}^n \mid n \in \mathbb{N}\}$.

First, here are five members of $\mathcal{L}$: aba, aabaa, aaabaaa, $\mathsf{a}^5\mathsf{b}\mathsf{a}^5$, and b. Five non-members are aabaaa, $\mathsf{a}^5\mathsf{b}\mathsf{a}^4$, a, aa, and ab.

The proof is by contradiction. Suppose that $\mathcal{L}$ is regular. Then it has a pumping length, which we denote p. Consider the string $\sigma = \mathsf{a}^p b \mathsf{a}^p$. Observe that $\sigma \in \mathcal{L}$.

Because $|\sigma| \geq p$, that string decomposes as $\sigma = \alpha^\frown \beta^\frown \gamma$, subject to the three conditions.

Usually we use the Pumping Lemma to show that a language is not regular.

EXAMPLE Show that this language over $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ is not regular: $\mathcal{L} = \{\mathsf{a}^n\mathsf{b}\mathsf{a}^n \mid n \in \mathbb{N}\}$.

First, here are five members of $\mathcal{L}$: aba, aabaa, aaabaaa, $\mathsf{a}^5\mathsf{b}\mathsf{a}^5$, and b. Five non-members are aabaaa, $\mathsf{a}^5\mathsf{b}\mathsf{a}^4$, a, aa, and ab.

The proof is by contradiction. Suppose that $\mathcal{L}$ is regular. Then it has a pumping length, which we denote p. Consider the string $\sigma = \mathsf{a}^p b \mathsf{a}^p$. Observe that $\sigma \in \mathcal{L}$.

Because $|\sigma| \geq p$, that string decomposes as $\sigma = \alpha^\frown \beta^\frown \gamma$, subject to the three conditions. Condition (1) says that the initial two substrings together have length no greater than the pumping length, $|\alpha\beta| \leq p$. Therefore, because the first p-many characters in σ are all a's, we conclude that substring α and substring β consist solely of a's.

Usually we use the Pumping Lemma to show that a language is not regular.

EXAMPLE Show that this language over $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ is not regular: $\mathcal{L} = \{\mathsf{a}^n\mathsf{b}\mathsf{a}^n \mid n \in \mathbb{N}\}$.

First, here are five members of $\mathcal{L}$: aba, aabaa, aaabaaa, $\mathsf{a}^5\mathsf{b}\mathsf{a}^5$, and b. Five non-members are aabaaa, $\mathsf{a}^5\mathsf{b}\mathsf{a}^4$, a, aa, and ab.

The proof is by contradiction. Suppose that $\mathcal{L}$ is regular. Then it has a pumping length, which we denote p. Consider the string $\sigma = \mathsf{a}^p b \mathsf{a}^p$. Observe that $\sigma \in \mathcal{L}$.

Because $|\sigma| \geq p$, that string decomposes as $\sigma = \alpha^\frown \beta^\frown \gamma$, subject to the three conditions. Condition (1) says that the initial two substrings together have length no greater than the pumping length, $|\alpha\beta| \leq p$. Therefore, because the first p-many characters in σ are all a's, we conclude that substring α and substring β consist solely of a's. Condition (2) says that β is not the empty string and so β consists of at least one a.

Usually we use the Pumping Lemma to show that a language is not regular.

EXAMPLE Show that this language over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ is not regular: $\mathcal{L} = \{\mathtt{a}^n\mathtt{b}\mathtt{a}^n \mid n \in \mathbb{N}\}$.

First, here are five members of $\mathcal{L}$: aba, aabaa, aaabaaa, $\mathtt{a}^5\mathtt{b}\mathtt{a}^5$, and b. Five non-members are aabaaa, $\mathtt{a}^5\mathtt{b}\mathtt{a}^4$, a, aa, and ab.

The proof is by contradiction. Suppose that $\mathcal{L}$ is regular. Then it has a pumping length, which we denote p. Consider the string $\sigma = \mathtt{a}^p b \mathtt{a}^p$. Observe that $\sigma \in \mathcal{L}$.

Because $|\sigma| \geq p$, that string decomposes as $\sigma = \alpha^\frown \beta^\frown \gamma$, subject to the three conditions. Condition (1) says that the initial two substrings together have length no greater than the pumping length, $|\alpha\beta| \leq p$. Therefore, because the first p-many characters in σ are all a's, we conclude that substring α and substring β consist solely of a's. Condition (2) says that β is not the empty string and so β consists of at least one a. With that, consider condition (3). It says that all of the strings $\alpha\gamma$, $\alpha\beta^2\gamma$, $\alpha\beta^3\gamma$, etc., are members of $\mathcal{L}$. We will find the contradiction in $\alpha\gamma$. Compared with σ, the string $\alpha\gamma$ has at least one fewer a before the b (because it omits the string β), but the same number of a's after the b. Therefore, $\alpha\gamma$ does not have the same number of a's before the b as after. It is thus not a member of $\mathcal{L}$, which is a contradiction.

EXAMPLE Consider this language.

$$\mathcal{L} = \{ \tau^\frown \tau \mid \tau \in \mathbb{B}^* \}$$

Name five members and five nonmembers. Then use the Pumping Lemma to prove that it is not a regular language.

EXAMPLE Consider this language.

$$\mathcal{L} = \{ \tau^\frown \tau \mid \tau \in \mathbb{B}^* \}$$

Name five members and five nonmembers. Then use the Pumping Lemma to prove that it is not a regular language.

Five members are `001001`, `11011101`, `00`, `1111`, and ε. Five nonmembers are `000`, `1`, `0110`, `001000`, and `01`.

For the proof, assume that $\mathcal{L}$ is regular. Then it has a pumping length, which we will denote p. Consider the string $\sigma = \texttt{0}^p\texttt{10}^p\texttt{1}$.

It is a member of $\mathcal{L}$ and $|\sigma| \geq p$ so it decomposes into three substrings $\sigma = \alpha\beta\gamma$ in a way that satisfies the three conditions.

Condition (1) is that α and β together have length no greater than the pumping length, $|\alpha\beta| \leq p$. Because the first p-many characters in σ are all a's, this means that substring α and substring β consist solely of `0`'s. Condition (2) says that β is not empty, and so β consists of at least one `0`. Condition (3) says that the strings $\alpha\gamma$, $\alpha\beta^2\gamma$, $\alpha\beta^3\gamma$, etc., are members of $\mathcal{L}$. But compared with σ, the string $\alpha\beta^2\gamma$ has at least one more `0` before its first 1, but the same number of `0`'s before its second1. Therefore, $\alpha\beta^2\gamma$ does not have the form $\tau^\frown\tau$ and is not a member of $\mathcal{L}$. That's a contradiction.

EXAMPLE Consider this language.

$$\mathcal{L} = \{ \tau^\frown \tau \mid \tau \in \mathbb{B}^* \}$$

Name five members and five nonmembers. Then use the Pumping Lemma to prove that it is not a regular language.

Five members are `001001`, `11011101`, `00`, `1111`, and ε. Five nonmembers are `000`, `1`, `0110`, `001000`, and `01`.

For the proof, assume that $\mathcal{L}$ is regular. Then it has a pumping length, which we will denote p. Consider the string $\sigma = \texttt{0}^p\texttt{10}^p\texttt{1}$.

It is a member of $\mathcal{L}$ and $|\sigma| \geq p$ so it decomposes into three substrings $\sigma = \alpha\beta\gamma$ in a way that satisfies the three conditions.

Condition (1) is that α and β together have length no greater than the pumping length, $|\alpha\beta| \leq p$. Because the first p-many characters in σ are all a's, this means that substring α and substring β consist solely of `0`'s. Condition (2) says that β is not empty, and so β consists of at least one `0`. Condition (3) says that the strings $\alpha\gamma$, $\alpha\beta^2\gamma$, $\alpha\beta^3\gamma$, etc., are members of $\mathcal{L}$. But compared with σ, the string $\alpha\beta^2\gamma$ has at least one more `0` before its first 1, but the same number of `0`'s before its second1. Therefore, $\alpha\beta^2\gamma$ does not have the form $\tau^\frown\tau$ and is not a member of $\mathcal{L}$. That's a contradiction.

EXAMPLE Show that each language over $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ is not regular.

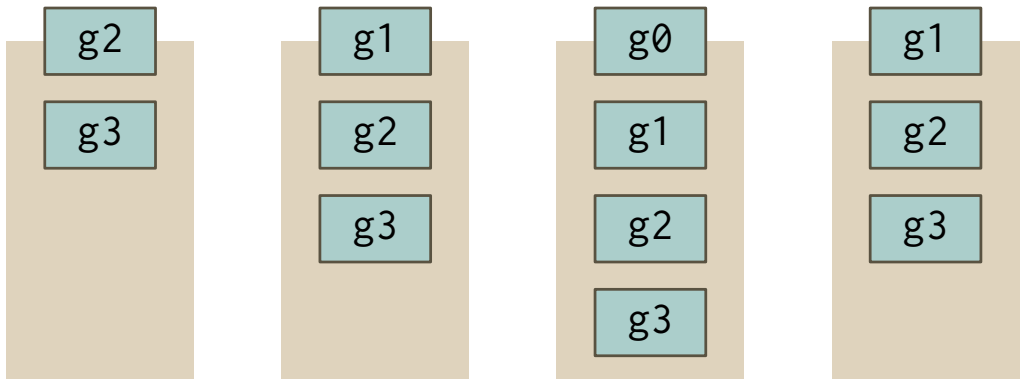
1. $\{ \mathsf{a}^n\mathsf{b}^m \mid n = m + 1 \}$
2. the set of palindromes

Pushdown machines

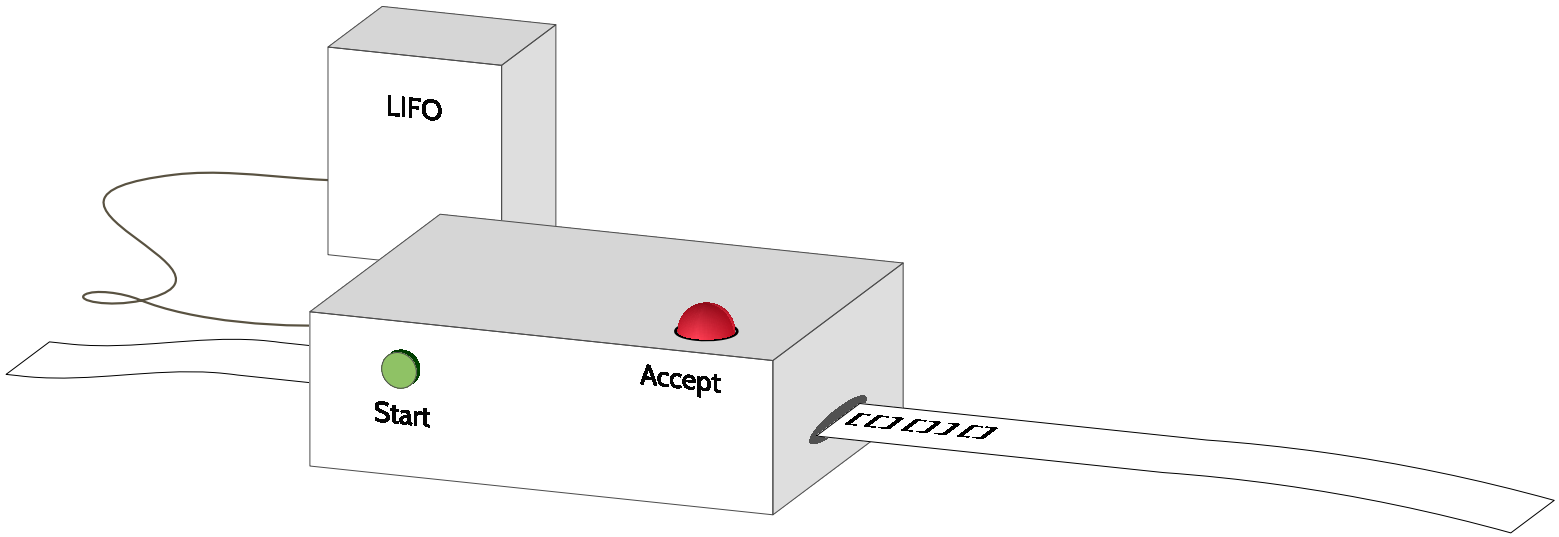
Introduction

No Finite State machine can recognize the language of balanced parentheses. So this machine model is not powerful enough to, for instance, decide whether input strings are valid programs in most programming languages. To handle nested parentheses, the natural data structure is a pushdown stack. We now supplement a Finite State machine by giving it access to a stack.

Below on the right is a sequence of views of a stack. Initially the stack has two characters, `g3` and `g2`. We push `g1` on the stack, and then `g0`. Now, although `g1` is on the stack, we don't have immediate access to it. To get at `g1` we must first pop off `g0`, as in the last stack shown.



DEFINITION A nondeterministic Pushdown machine $\langle Q, q_0, F, \Sigma, \Gamma, \Delta \rangle$ consists of a finite set of states $Q = \{q_0, \ldots q_{n-1}\}$, including a start state q_0, a subset $F \subseteq Q$ of accepting states, a nonempty input alphabet Σ, a nonempty stack alphabet Γ, and a transition function $\Delta : Q \times (\Sigma \cup \{\mathsf{B}, \varepsilon\}) \times (\Gamma \cup \{\perp\}) \to \mathcal{P}\big(Q \times (\Gamma \cup \{\perp\})^*\big)$.



The book has details. We will give a couple of examples.

EXAMPLE This gives the machine as a set of instructions. (We could instead take it to give the inputs and outputs for Δ.)

Instruction number	*Input*	*Output*
0	q_0, `[`, $\perp$	q_0, ‘`g0` $\perp$’
1	q_0, `[`, `g0`	q_0, ‘`g0 g0`’
2	q_0, `]`, `g0`	q_0, ‘’
3	q_0, `]`, $\perp$	q_0, ‘’
4	q_0, `B`, $\perp$	q_1, ‘$\perp$’
5	q_0, `B`, `g0`	q_0, ‘`g0`’

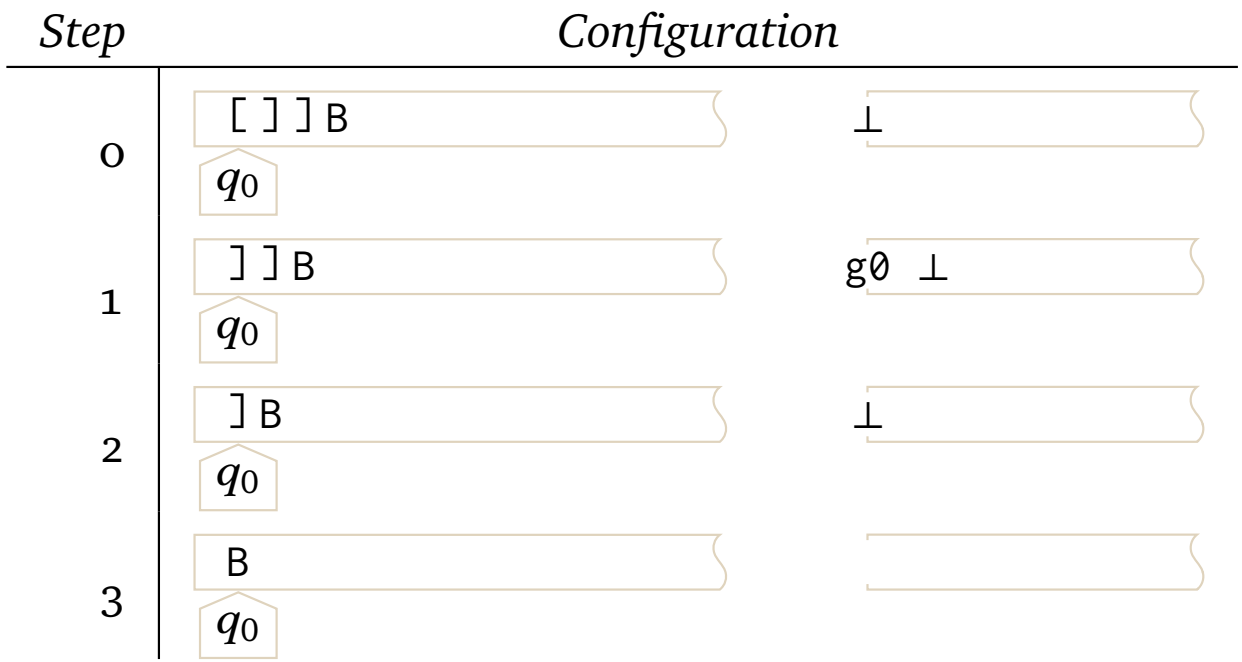
The only accepting state is q_1.

These machines always pop the stack before they make a transition. So Instruction 0 must push $\perp$ back on the stack, and then it adds `g0` on top of that. Instruction 2 pops the `g0` off and doesn't replace it.

These machines sometimes need to do some action when the tape is exhausted, so we put a `B` to mark the end of the input (and assume the input does not contain that character).

Step	*Configuration*		
0	[[ ]] [ ] B	q_0	⊥
1	[ ]] [ ] B	q_0	g0 ⊥
2	]] [ ] B	q_0	g0 g0 ⊥
3	] [ ] B	q_0	g0 ⊥
4	[ ] B	q_0	⊥
5	] B	q_0	g0 ⊥
6	B	q_0	⊥
7		q_1	⊥

Here is a rejection example, whose initial string does not have balanced parentheses.



At the end, although the tape still has content, the stack is empty. The machine cannot start the next step by popping the top stack character because there is no such character. The computation dies, without accepting the input.

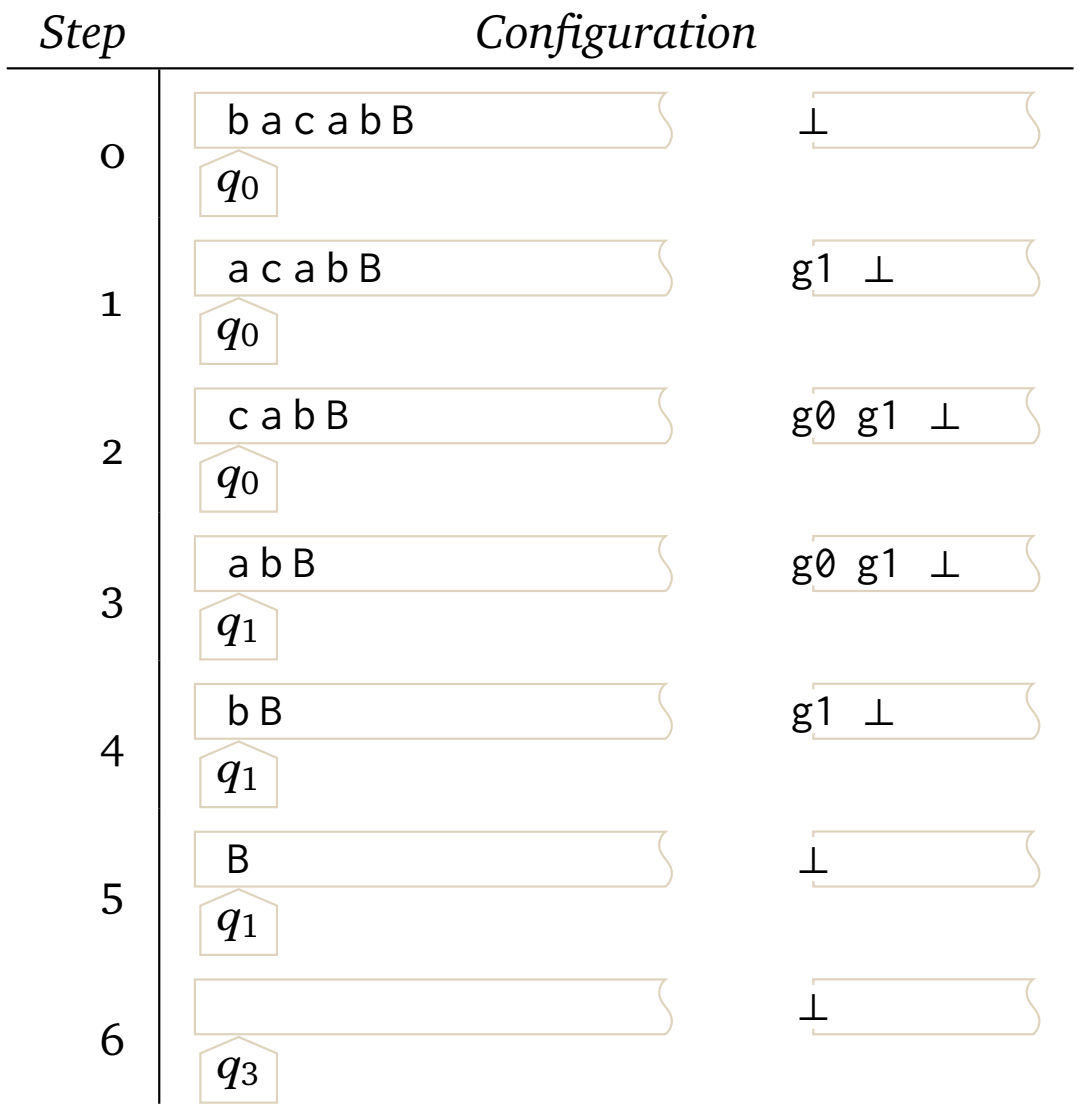
EXAMPLE Palindromes that are odd length have a character in the middle that acts as its own reverse. That is, they have the form $\sigma \frown s \frown \sigma^{\mathsf{R}}$ for $s \in \Sigma$. The language $\mathcal{L}_{\text{MM}}$ uses $\sigma \in \{\mathsf{a}, \mathsf{b}\}^*$ along with the character $s = \mathsf{c}$ as a middle marker so that $\mathcal{L}_{\text{MM}} = \{\sigma \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}^* \mid \sigma = \tau \frown \mathsf{c} \frown \tau^{\mathsf{R}} \text{ for some } \tau \in \{\mathsf{a}, \mathsf{b}\}^*\}$.

The machine below accepts this language. It has $Q = \{q_0, q_1, q_2\}$, $F = \{q_2\}$, $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, and $\Gamma = \{\mathsf{g0}, \mathsf{g1}\}$.

Inst	*Input*	*Output*
0	$q_0, \mathsf{a}, \perp$	q_0, ‘g0 $\perp$’
1	$q_0, \mathsf{b}, \perp$	q_0, ‘g1 $\perp$’
2	$q_0, \mathsf{c}, \perp$	q_1, ‘$\perp$’
3	$q_0, \mathsf{B}, \perp$	q_3, ‘$\perp$’
4	$q_0, \mathsf{a}, \mathsf{g0}$	q_0, ‘g0 g0’
5	$q_0, \mathsf{a}, \mathsf{g1}$	q_0, ‘g0 g1’
6	$q_0, \mathsf{b}, \mathsf{g0}$	q_0, ‘g1 g0’
7	$q_0, \mathsf{b}, \mathsf{g1}$	q_0, ‘g1 g1’

Inst	*Input*	*Output*
8	$q_0, \mathsf{c}, \mathsf{g0}$	q_1, ‘g0’
9	$q_0, \mathsf{c}, \mathsf{g1}$	q_1, ‘g1’
10	$q_1, \mathsf{a}, \mathsf{g0}$	q_1, ‘’
11	$q_1, \mathsf{b}, \mathsf{g1}$	q_1, ‘’
12	$q_1, \mathsf{B}, \perp$	q_2, ‘$\perp$’

Here is an example computation accepting the input bacab.



Pushdown machines with ε transitions

EXAMPLE This machine accepts the language of even length palindromes.

$$\mathcal{L}_{\text{ELP}} = \{\, \sigma\sigma^{\mathsf{R}} \;\big|\; \sigma \in \mathbb{B}^* \,\} = \{\, \varepsilon, \mathtt{00}, 11, \mathtt{0000}, \mathtt{0110}, \mathtt{1001}, 1111, \ldots \,\}$$

It has $F = \{\, q_2 \,\}$ as well as $\Gamma = \{\, \mathtt{g0}, \mathtt{g1} \,\}$.

Inst	*Input*	*Output*
0	$q_0, \mathtt{0}, \bot$	q_0, ‘g0⊥’
1	$q_0, 1, \bot$	q_0, ‘g1⊥’
2	$q_0, \varepsilon, \bot$	$q_1, \bot$
3	$q_0, \mathtt{0}, \mathtt{g0}$	q_0, ‘g0g0’
4	$q_0, \mathtt{0}, \mathtt{g1}$	q_0, ‘g0g1’
5	$q_0, 1, \mathtt{g0}$	q_0, ‘g1g0’
6	$q_0, 1, \mathtt{g1}$	q_0, ‘g1g1’

Inst	*Input*	*Output*
7	$q_0, \varepsilon, \mathtt{g0}$	q_1, ‘g0’
8	$q_0, \varepsilon, \mathtt{g1}$	q_1, ‘g1’
9	$q_1, \mathtt{0}, \mathtt{g0}$	q_1, ‘’
10	$q_1, 1, \mathtt{g1}$	q_1, ‘’
11	$q_1, \varepsilon, \bot$	$q_2, \bot$

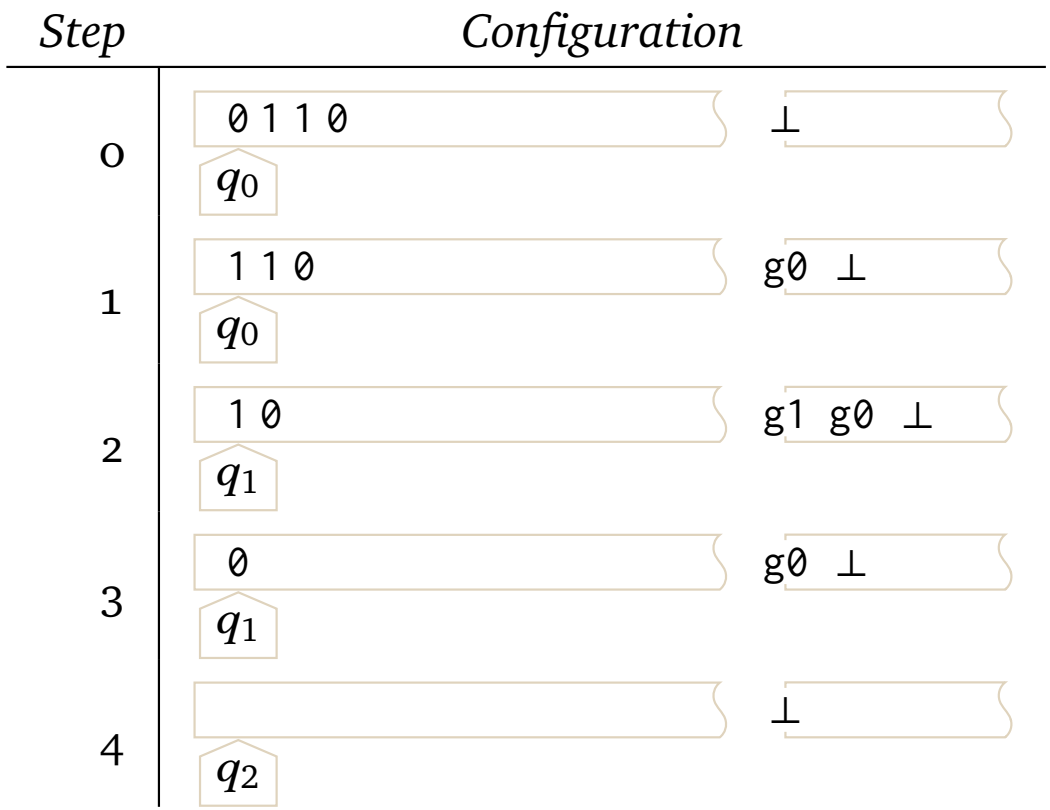
Because this machine can guess, we can have it guess whether the input is finished. So we will omit the terminating B that we used in the two prior examples.

The machine runs in two phases. Where the input is $\sigma\sigma^{\mathsf{R}}$, the first phase works with σ. If the tape character is `0` then the machine pushes the token `g0` onto the stack, and if it is `1` then the machine pushes `g1`. This is done while in state q_0.

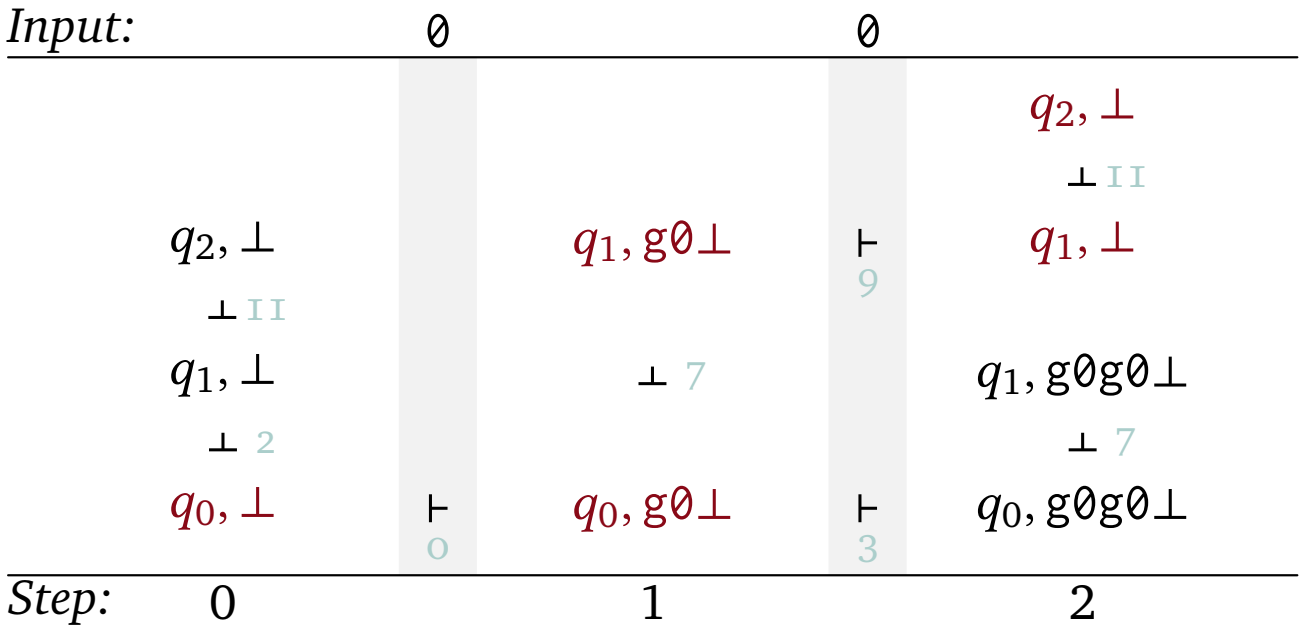
The second phase works with σ^{R}. If `0` is on the tape and `g0` tops the stack, or `1` and `g1`, then the machine proceeds. Otherwise there is no matching instruction and the computation branch dies. This is done while in state q_1.

Without a middle marker how does the machine know when to change from phase one to two, from pushing to popping? It is nondeterministic — it guesses. That happens in lines 7 and 8. The ε character in the input means that the machine can spontaneously transition from q_0 to q_1.

Here is the branch of the computation tree where the machine accepts the even-length palindrome string `0110`.

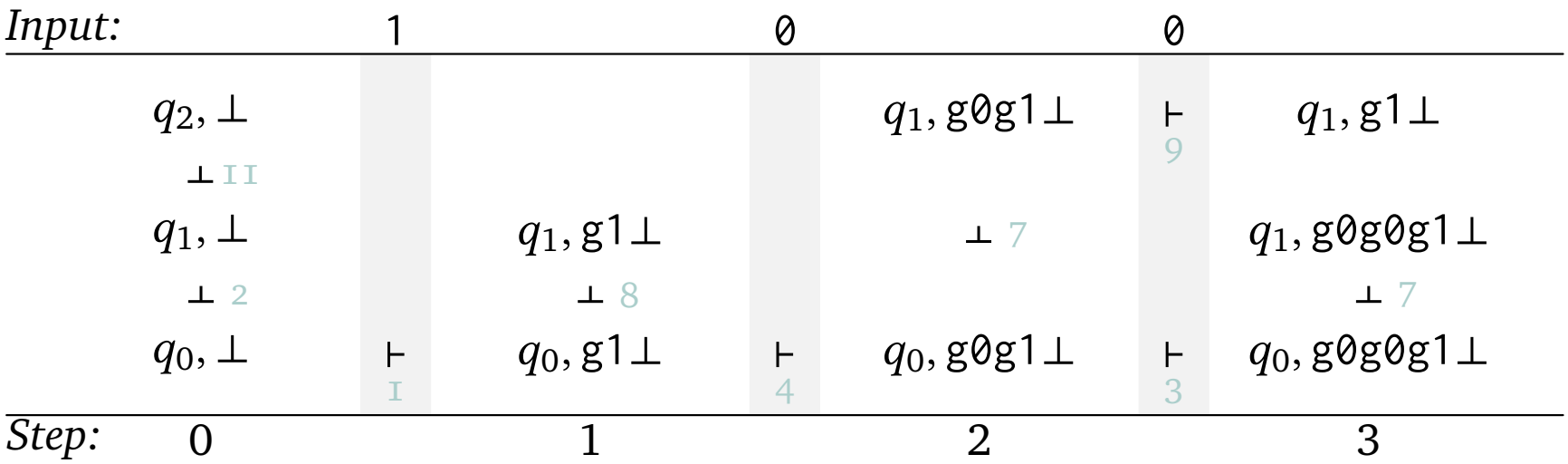


Here is the machine's computation tree on input `00`. The ε transitions go up. (The relevant instructions are written next to the $\vdash$ symbols.)



As the highlighted branch shows, it accepts.

Here is the machine's full computation tree on input `100`.



None of the branches end both with an empty string and in an accepting state, so it rejects the input.

Deterministic Pushdown machines

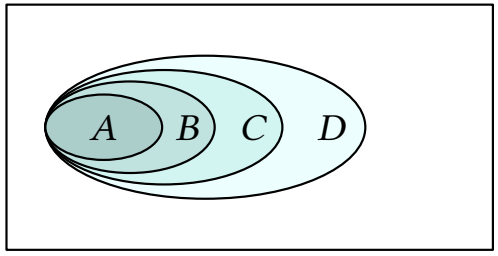
The definition that we gave earlier is of a nondeterministic Pushdown machine. For the next slide we need to describe deterministic ones.

In contradistinction to a nondeterministic machine, in a deterministic machine at any step there is exactly one legal move. So to adjust the definition we have for nondeterministic Pushdown machines to one for deterministic ones we eliminate situations where the machine has choices. There are two situations. One is that $\Delta(q_i, t, g)$ is a set and so we will require that in a deterministic machine that set must have exactly one element. The other is evident in the tree diagrams above: nondeterministic machines can have that $\Delta(q_i, \varepsilon, g)$ is a nonempty set and also that $\Delta(q_i, t, g)$ is nonempty for $t \neq \varepsilon$ (see for instance the the prior example's machine in lines 0–2). So we outlaw the possibility that both are nonempty.

This section's first two example Pushdown machines are both deterministic.

Relation between machine types and languages

Below, the box includes all languages. That is, fix an alphabet Σ and then the universe consists of every subset of Σ^*.



Class	*Machine type*
A	Finite State, including nondeterministic
B	deterministic Pushdown
C	nondeterministic Pushdown
D	Turing

Class *A* consists of the languages that are regular, those decided by some Finite State machine. Class *B* holds the languages that are decided by some deterministic Pushdown machine. Class *C* is the languages decided by some nondeterministic Pushdown machine. Finally, class *D* consists of the languages decided by some Turing machine. (These are all proper subsets but proof of that lies outside of our scope.)

An important point to notice is that although for Finite State machines adding nondeterminism does not change what languages the collection of machines can recognize, for Pushdown machines it does. In particular, there is a nondeterministic Pushdown machine that recognizes the language of palindromes over $\Sigma = \{\mathsf{a}, \mathsf{b}\}$. But no deterministic Pushdown machine can do that — the intuition is that it has no way to know when to change from pushing to popping.

Regexes

Regular expressions in practice

Regular expressions are so useful that they have escaped the theory and are widely used for symbol manipulation in the wild.

Along the way, they acquired a large number of extensions. Some of these are conveniences, things that you could do with the regular expression tools that we have already seen. But some of these are genuine extensions, which allow you to do more than is possible with the regular expressions that we have studied.

We shall refer to the in-practice expressions as regexes.

Convenience extensions

Our theory alphabets had two or three characters. In practice we need at least ASCII's printable characters: `a`–`z`, `A`–`Z`, `0`–`9`, space, tab, period, dash, exclamation point, percent sign, dollar sign, open and closed parenthesis, open and closed curly braces, etc. The alphabet may even contain all of Unicode's more than one hundred thousand characters.

Regexes allow matching a digit with `[0123456789]`, using square brackets as metacharacters. We can shorten it further to `[0-9]`. Similarly, `[A-Za-z]` matches a single English letter.

To invert the set of matched characters, put a caret '`^`' as the first thing inside the bracket. Thus, `[^0-9]` matches a non-digit and `[^A-Za-z]` matches a character that is not an ASCII letter.

The most common lists have short abbreviations. Another abbreviation for the ASCII digits is `\d`. Use `\D` for the ASCII non-digits, `\s` for the whitespace characters (space, tab, newline, formfeed, and line return) and `\S` for ASCII characters that are non-whitespace. Cover the alphanumeric characters (upper and lower case ASCII letters, digits, and underscore) with `\w` and cover the complement with `\W`.

Earlier, to match ‘at most one a’ we used ε|a. So we can write something like (|a). But depicting the empty string by just putting nothing there can be confusing. Modern languages allow you to write a? for ‘at most one a’.

For ‘at least one a’ modern languages use a+, so the plus sign is another metacharacter. More generally, we often want to specify quantities. For instance, to match five a’s extended regular expressions use the curly braces as metacharacters, with a{5}. Match between two and five of them with a{2,5} and match at least two with a{2,}. Thus, a+ is shorthand for a{1,}.

Earlier, to match 'at most one a' we used ε`|a`. So we can write something like `(|a)`. But depicting the empty string by just putting nothing there can be confusing. Modern languages allow you to write `a?` for 'at most one a'.

For 'at least one a' modern languages use `a+`, so the plus sign is another metacharacter. More generally, we often want to specify quantities. For instance, to match five a's extended regular expressions use the curly braces as metacharacters, with `a{5}`. Match between two and five of them with `a{2,5}` and match at least two with `a{2,}`. Thus, `a+` is shorthand for `a{1,}`.

To match a metacharacter, escape it with a backslash, '\'. Thus, to look for the string '`(Note`' put a backslash before the open parentheses: `\(Note`. Similarly, `\|` matches a pipe and `\[` matches an open square bracket. And, `To be or not to be\?` matches the famous question. Match backslash itself with `\\`.

Example US ZIP codes are five digits. Match them with `\d{5}`.

Example North American phone numbers match `\d{3} \d{3}-\d{4}`.

Example Canadian postal codes have seven characters: the fourth is a space, the first, third, and sixth are letters, and the others are digits. The regular expression `[a-zA-Z]\d[a-zA-Z] \d[a-zA-Z]\d` describes them.

Example Dates are often given in the 'dd/mm/yy' format. This matches: `\d\d/\d\d/\d\d`.

Example In the twelve hour time format some typical times strings are '8:05 am' or '10:15 pm'. You could use this (note the empty string at the start).

```
(|0|1)\d:\d\d\s(am|pm)
```

Note that it matches some strings that you don't want, such as '18:05 am'.

EXAMPLE The regex `(-|\+)?\d+` matches an integer, positive or negative. The question mark makes the sign optional. The plus sign makes sure there is at least one digit.

EXAMPLE A natural number represented in hexadecimal can contain the usual digits, along with the additional characters ‘a’ through ‘f’ (sometimes capitalized). Programmers often prefix such a representation with `0x`, so the expression is `(0x)?[a-fA-F0-9]+`.

EXAMPLE A C language identifier begins with an ASCII letter or underscore and then can have arbitrarily many more letters, digits, or underscores: `[a-zA-Z_]\w*`.

EXAMPLE Match a user name of between three and twelve letters, digits, underscores, or periods with `[\w\.]{3,12}`. Match a password that is at least eight characters long with `.{8,}`.

EXAMPLE The regex `(-|\+)?\d+` matches an integer, positive or negative. The question mark makes the sign optional. The plus sign makes sure there is at least one digit.

EXAMPLE A natural number represented in hexadecimal can contain the usual digits, along with the additional characters 'a' through 'f' (sometimes capitalized). Programmers often prefix such a representation with `0x`, so the expression is `(0x)?[a-fA-F0-9]+`.

EXAMPLE A C language identifier begins with an ASCII letter or underscore and then can have arbitrarily many more letters, digits, or underscores: `[a-zA-Z_]\w*`.

EXAMPLE Match a user name of between three and twelve letters, digits, underscores, or periods with `[\w\.]{3,12}`. Match a password that is at least eight characters long with `.{8,}`.

EXAMPLE For email addresses, `\S+@\S+` is an often used extended expression, as a sanity check.

EXAMPLE Match the text inside a single set of parentheses with `\([^()]*\)`.

EXAMPLE We next match a URL, a web address such as `https://hefferon.net/computation`. The regex breaks URL's into a scheme such as 'http' along with a colon and two forward slashes, a host such as `hefferon.net` followed by a slash, and then a path such as `computing` (the standard also allows a trailing query string but this regex does not handle that).

```
(https?|ftp)://([^\s/?\.#]+\.?){0,3}[^\s/?\.#]+(/[^\s]*/?)?
```

As to lines, you can also match the start of a line and end of line with the metacharacters caret ‘^’ and dollar sign ‘$’.

Many programs that use regexes are line-oriented. They often, by default, match the regex anywhere in the line.

Sometimes, as in Python, there are two modes. One matches the give regex anywhere in the line, and one specifically requires you to put in the parts such as start of line and end of line.

Extensions beyond convenience alone

EXAMPLE The web language HTML uses tags such as `<b>boldface text</b>` and `<i>italicized text</i>`. Matching any one tag is straightforward, for instance `<b>[^<]*</b>`.

But for a single expression that matches them all, you would seem to have to do each as a separate case and then combine cases with a pipe. However, instead modern regexes allow the system to remember what it finds at the start and look for that again at the end.

Thus, Racket's regex `<([^>]+)>.*</\\1>` matches HTML tags like the ones given. Its second character is an open parenthesis, and the `\\1` refers to everything between that open parenthesis and the matching close parenthesis. (As you might guess from the `1`, you can also have a second match with `\\2`, etc.)

That is a back reference. It is very convenient, but it gives regexes more power than the theoretical regular expressions that we studied earlier.

EXAMPLE This is the language of squares over $\Sigma = \{\mathsf{a}, \mathsf{b}\}$.

$$\mathcal{L} = \{\sigma \in \Sigma^* \mid \sigma = \tau^\frown \tau \text{ for some } \tau \in \Sigma^*\}$$

Some members are aabaab, baaabaaa, and aa. The Pumping Lemma shows that the language of squares is not regular. Describe this language with the regex `(.+)\1`; note the back-reference.

EXAMPLE The regex `.?|(..+?)\\1+` matches a unary number if and only if it is a not prime. The group corresponds to a natural number of characters that are matched. This group then appears some natural number of times. If there is a match then it is possible to find a product of two numbers greater than or equal to 2 that match the string.