Chapter 4
State Machines

State machines are a method of modeling systems whose output depends on the entire history of their inputs, and not just on the most recent input. Compared to purely functional systems, in which the output is purely determined by the input, state machines have a performance that is determined by its history. State machines can be used to model a wide variety of systems, including:

- user interfaces, with typed input, mouse clicks, etc.;
- conversations, in which, for example, the meaning of a word "it" depends on the history of things that have been said;
- the state of a spacecraft, including which valves are open and closed, the levels of fuel and oxygen, etc.; and
- the sequential patterns in DNA and what they mean.

State machine models can either be *continuous time* or *discrete time*. In continuous time models, we typically assume continuous spaces for the range of values of inputs and outputs, and use differential equations to describe the system's dynamics. This is an interesting and important approach, but it is hard to use it to describe the desired behavior of our robots, for example. The loop of reading sensors, computing, and generating an output is inherently discrete and too slow to be well-modeled as a continuous-time process. Also, our control policies are often highly nonlinear and discontinuous. So, in this class, we will concentrate on discrete-time models, meaning models whose inputs and outputs are determined at specific increments of time, and which are synchronized to those specific time samples. Furthermore, in this chapter, we will make no assumptions about the form of the dependence of the output on the time-history of inputs; it can be an arbitrary function.

Generally speaking, we can think of the job of an embedded system as performing a *transduction* from a stream (infinite sequence) of input values to a stream of output values. In order to specify the behavior of a system whose output depends on the history of its inputs mathematically, you could think of trying to specify a mapping from $i_1, \ldots, i_t$ (sequences of previous inputs) to o_t (current output), but that could become very complicated to specify or execute as the history gets longer. In the state-machine approach, we try to find some set of *states* of the system, which capture the essential properties of the history of the inputs and are used to determine the current output of the system as well as its next state. It is an interesting and sometimes difficult modeling problem to find a good state-machine model with the right set of states; in this chapter we will explore how the ideas of PCAP can aid us in designing useful models.

One thing that is particularly interesting and important about state machine models is how many ways we can use them. In this class, we will use them in three fairly different ways:

1. **Synthetically**: State machines can specify a "program" for a robot or other system embedded in the world, with inputs being sensor readings and outputs being control commands.
2. **Analytically**: State machines can describe the behavior of the combination of a control system and the environment it is controlling; the input is generally a simple command to the entire system, and the output is some simple measure of the state of the system. The goal here is to analyze global properties of the coupled system, like whether it will converge to a steady state, or will oscillate, or will diverge.
3. **Predictively**: State machines can describe the way the environment works, for example, where I will end up if I drive down some road from some intersection. In this case, the inputs are control commands and the outputs are states of the external world. Such a model can be used to plan trajectories through the space of the external world to reach desirable states, by considering different courses of action and using the model to predict their results.

We will develop a single formalism, and an encoding of that formalism in Python classes, that will serve all three of these purposes.

Our strategy for building very complex state machines will be, abstractly, the same strategy we use to build any kind of complex machine. We will define a set of primitive machines (in this case, an infinite class of primitive machines) and then a collection of *combinators* that allow us to put primitive machines together to make more complex machines, which can themselves be abstracted and combined to make more complex machines.

4.1 Primitive state machines

We can specify a transducer (a process that takes as input a sequence of values which serve as inputs to the state machine, and returns as ouput the set of outputs of the machine for each input) as a state machine (SM) by specifying:

- a set of *states*, S,
- a set of *inputs*, I, also called the *input vocabulary*,
- a set of *outputs*, O, also called the *output vocabulary*,
- a *next-state function*, $n(i_t, s_t) \mapsto s_{t+1}$, that maps the input at time t and the state at time t to the state at time $t+1$,
- an *output function*, $o(i_t, s_t) \mapsto o_t$, that maps the input at time t and the state at time t to the output at time t; and
- an *initial state*, s_0, which is the state at time 0.

Here are a few state machines, to give you an idea of the kind of systems we are considering.

- A *tick-tock* machine that generates the sequence $1, 0, 1, 0, \ldots$ is a finite-state machine that ignores its input.
- The controller for a digital watch is a more complicated finite-state machine: it transduces a sequence of inputs (combination of button presses) into a sequence of outputs (combinations of segments illuminated in the display).
- The controller for a bank of elevators in a large office building: it transduces the current set of buttons being pressed and sensors in the elevators (for position, open doors, etc.) into commands to the elevators to move up or down, and open or close their doors.

The very simplest kind of state machine is a pure function: if the machine has no state, and the output function is purely a function of the input, for example, $o_t = i_t + 1$, then we have an immediate functional relationship between inputs and outputs on the same time step. Another simple class of SMs are *finite*-state machines, for which the set of possible states is finite. The elevator controller can be thought of as a finite-state machine, if elevators are modeled as being only at one floor or another (or possibly between floors); but if the controller models the exact position of the elevator (for the purpose of stopping smoothly at each floor, for example), then it will be most easily expressed using real numbers for the state (though any real instantiation of it can ultimately only have finite precision). A different, large class of SMs are describable as *linear, time-invariant* (LTI) systems. We will discuss these in detail chapter ??.

4.1.1 Examples

Let's look at several examples of state machines, with complete formal definitions.

4.1.1.1 Language acceptor

Here is a finite-state machine whose output is *true* if the input string adheres to a simple pattern, and *false* otherwise. In this case, the pattern has to be $a, b, c, a, b, c, a, b, c, \ldots$.

It uses the states 0, 1, and 2 to stand for the situations in which it is expecting an a, b, and c, respectively; and it uses state 3 for the situation in which it has seen an input that was not the one that was expected. Once the machine goes to state 3 (sometimes called a *rejecting state*), it never exits that state.

$$
\begin{aligned}
S &= \{0, 1, 2, 3\} \\
I &= \{a, b, c\} \\
O &= \{\mathit{true}, \mathit{false}\} \\
n(s, i) &= \begin{cases} 1 & \text{if } s = 0, i = a \\ 2 & \text{if } s = 1, i = b \\ 0 & \text{if } s = 2, i = c \\ 3 & \text{otherwise} \end{cases} \\
o(s, i) &= \begin{cases} \mathit{false} & \text{if } n(s, i) = 3 \\ \mathit{true} & \text{otherwise} \end{cases} \\
s_0 &= 0
\end{aligned}
$$

Figure 4.1 shows a state transition diagram for this state machine. Each circle represents a state. The arcs connecting the circles represent possible transitions the machine can make; the arcs are labeled with a pair i, o, which means that if the machine is in the state denoted by a circle with label s, and gets an input i, then the arc points to the next state, $n(s, i)$ and the output generated $o(s, i) = o$. Some arcs have several labels, indicating that there are many different inputs that will cause the same transition. Arcs can only be traversed in the direction of the arrow.

For a state transition diagram to be complete, there must be an arrow emerging from each state for each possible input i (if the next state is the same for some inputs, then we draw the graph more compactly by using a single arrow with multiple labels, as you will see below).

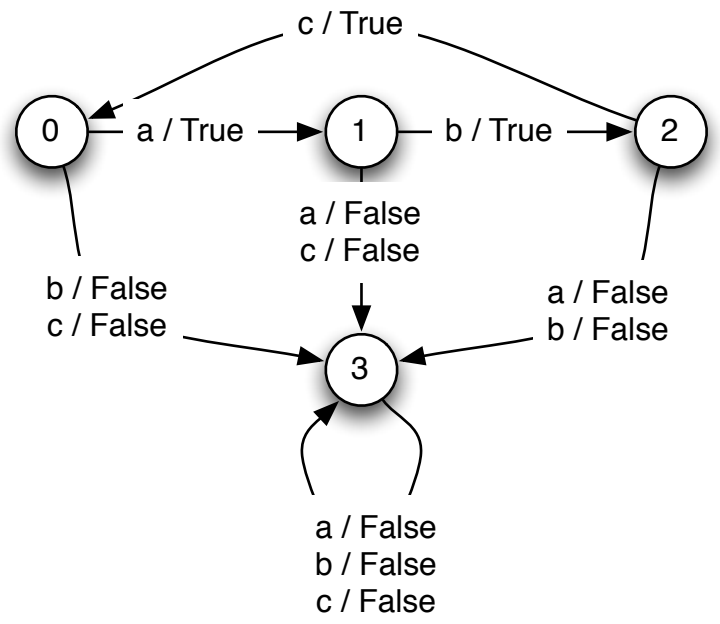


Figure 4.1 State transition diagram for language acceptor.

We will use tables like the following one to examine the evolution of a state machine:

time	0	1	2	...
input	i_0	i_1	i_2	...
state	s_0	s_1	s_2	...
output	o_1	o_2	o_3	...

For each column in the table, given the current input value and state we can use the output function o to determine the output in that column; and we use the n function applied to that input and state value to determine the state in the next column. Thus, just knowing the input sequence and s_0, and the next-state and output functions of the machine will allow you to fill in the rest of the table.

For example, here is the state of the machine at the initial time point:

time	0	1	2	...
input	i_0			...
state	s_0			...
output				...

Using our knowledge of the next state function n, we have:

time	0	1	2	...
input	i_0			...
state	s_0	s_1		...
output				...

and using our knowledge of the output function o, we have at the next input value

time	0	1	2	...
input	i_0	i_1		...
state	s_0	s_1		...
output	o_1			...

This completes one cycle of the statement machine, and we can now repeat the process.

Here is a table showing what the language acceptor machine does with input sequence ('a', 'b', 'c', 'a', 'c', 'a', 'b'):

time	0	1	2	3	4	5	6	7
input	'a'	'b'	'c'	'a'	'c'	'a'	'b'	
state	0	1	2	0	1	3	3	3
output	True	True	True	True	False	False	False	

The output sequence is (True, True, True, True, False, False, False).

Clearly we don't want to analyze a system by considering all input sequences, but this table helps us understand the state transitions of the system model.

To learn more: Finite-state machine language acceptors can be built for a class of patterns called *regular languages*. There are many more complex patterns (such as the set of strings with equal numbers of 1's and 0's) that cannot be recognized by finite-state machines, but can be recognized by a specialized kind of infinite-state machine called a *stack machine*. To learn more about these fundamental ideas in *computability theory*, start with the Wikipedia article on **Computability_theory_(computer_science)**

4.1.1.2 Up and down counter

This machine can count up and down; its state space is the countably infinite set of integers. It starts in state 0. Now, if it gets input u, it goes to state 1; if it gets u again, it goes to state 2. If it gets d, it goes back down to 1, and so on. For this machine, the output is always the same as the next state.

$$
\begin{aligned}
S &= \textit{integers} \\
I &= \{u, d\} \\
O &= \textit{integers} \\
n(s,i) &= \begin{cases} s+1 & \text{if } i = u \\ s-1 & \text{if } i = d \end{cases} \\
o(s,i) &= n(s,i) \\
s_0 &= 0
\end{aligned}
$$

Here is a table showing what the up and down counter does with input sequence u, u, u, d, d, u:

time	0	1	2	3	4	5	6
input	u	u	u	d	d	u	
state	0	1	2	3	2	1	2
output	1	2	3	2	1	2	

The output sequence is $1, 2, 3, 2, 1, 2$.

4.1.1.3 Delay

An even simpler machine just takes the input and passes it through to the output, but with one step of delay, so the kth element of the input sequence will be the $k+1$st element of the output sequence. Here is the formal machine definition:

$$
\begin{aligned}
S &= \textit{anything} \\
I &= \textit{anything} \\
O &= \textit{anything} \\
n(s,i) &= i \\
o(s,i) &= s \\
s_0 &= 0
\end{aligned}
$$

Given an input sequence $i_0, i_1, i_2, \ldots$, this machine will produce an output sequence $0, i_0, i_1, i_2, \ldots$. The initial 0 comes because it has to be able to produce an output before it has even seen an input, and that output is produced based on the initial state, which is 0. This very simple building block will come in handy for us later on.

Here is a table showing what the delay machine does with input sequence $3, 1, 2, 5, 9$:

time	0	1	2	3	4	5
input	3	1	2	5	9	
state	0	3	1	2	5	9
output	0	3	1	2	5	

The output sequence is $0, 3, 1, 2, 5$.

4.1.1.4 Accumulator

Here is a machine whose output is the sum of all the inputs it has ever seen.

$$\begin{aligned} S &= \textit{numbers} \\ I &= \textit{numbers} \\ O &= \textit{numbers} \\ n(s,i) &= s + i \\ o(s,i) &= n(s,i) \\ s_0 &= 0 \end{aligned}$$

Here is a table showing what the accumulator does with input sequence $100, -3, 4, -123, 10$:

time	0	1	2	3	4	5
input	100	-3	4	-123	10	
state	0	100	97	101	-22	-12
output	100	97	101	-22	-12	

4.1.1.5 Average2

Here is a machine whose output is the average of the current input and the previous input. It stores its previous input as its state.

$$\begin{aligned} S &= \textit{numbers} \\ I &= \textit{numbers} \\ O &= \textit{numbers} \\ n(s,i) &= i \\ o(s,i) &= (s + i)/2 \\ s_0 &= 0 \end{aligned}$$

Here is a table showing what the average2 machine does with input sequence $100, -3, 4, -123, 10$:

time	0	1	2	3	4	5
input	100	-3	4	-123	10	
state	0	100	-3	4	-123	10
output	50	48.5	0.5	-59.5	-56.5	

4.1.2 State machines in Python

Now, it is time to make computational implementations of state machine models. In this section we will build up some basic Python infrastructure to make it straightforward to define primitive machines; in later sections we will see how to combine primitive machines into more complex structures.

We will use Python's object-oriented facilities to make this convenient. We have an abstract class, `SM`, which will be the superclass for all of the particular state machine classes we define. It does not make sense to make an instance of `SM`, because it does not actually specify the behavior of the machine; it just provides some utility methods. To specify a new type of state machine, you define a new class that has `SM` as a superclass.

In any subclass of `SM`, it is crucial to define an attribute `startState`, which specifies the initial state of the machine, and a method `getNextValues` which takes the state at time t and the input at time t as input, and returns the state at time $t + 1$ and the output at time t. This is a choice that we have made as designers of our state machine model; we will rely on these two pieces of information in our underlying infrastructure for simulating state machines, as we will see shortly.

Here, for example, is the Python code for an accumulator state machine, which implements the definition given in **section 4.1.1.4**.[29]

```
class Accumulator(SM):
    startState = 0
    def getNextValues(self, state, inp):
        return (state + inp, state + inp)
```

It is important to note that `getNextValues` *does not change the state of the machine*, in other words, it does not mutate a state variable. Its job is to be a pure function: to answer the question of what the next state and output would be if the current state were `state` and the current input were `inp`. We will use the `getNextValues` methods of state machines later in the class to make plans by considering alternative courses of action, so they *must not have any side effects*. As we noted, this is our choice as designers of the state machine infrastructure We could have chosen to implement things differently, however this choice will prove to be very useful. Thus, in all our state machines, the function `getNextValues` will capture the transition from input and state to output and state, without mutating any stored state values.

To run a state machine, we make an instance of the appropriate state-machine class, call its `start` method (a built in method we will see shortly) to set the state to the starting state, and then ask it to take steps; each step consists of generating an output value (which is printed) and updating the state to the next state, based on the input. The abstract superclass `SM` defines the `start` and `step` methods. These methods are in charge of actually initializing and then changing the state of the machine as it is being executed. They do this by calling the `getNextValues` method for the class to which this instance belongs. The `start` method takes no arguments (but, even so, we have to put parentheses after it, indicating that we want to call the method, not to do something

[29] Throughout this code, we use `inp`, instead of `input`, which would be clearer. The reason is that the name `input` is used by Python as a function. Although it is legal to re-use it as the name of an argument to a procedure, doing so is a source of bugs that are hard to find (if, for instance, you misspell the name `input` in the argument list, your references to `input` later in the procedure will be legal, but will return the built-in function.)

with the method itself); the step method takes one argument, which is the input to the machine on the next step. So, here is a run of the accumulator, in which we feed it inputs 3, 4, and -2:

```
>>> a = Accumulator()
>>> a.start()
>>> a.step(3)
3
>>> a.step(4)
7
>>> a.step(-2)
5
```

The class `SM` specifies how state machines work in general; the class `Accumulator` specifies how accumulator machines work in general; and the instance `a` is a particular machine with a particular current state. We can make another instance of accumulator:

```
>>> b = Accumulator()
>>> b.start()
>>> b.step(10)
10
>>> b.state
10
>>> a.state
5
```

Now, we have two accumulators, `a`, and `b`, which remember, individually, in what states they exist. **Figure 4.2** shows the class and instance structures that will be in place after creating these two accumulators.

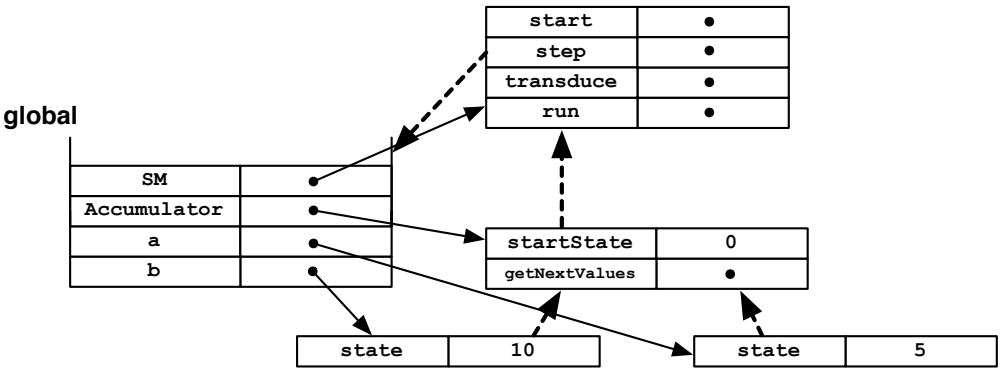


Figure 4.2 Classes and instances for an `Accumulator` `SM`. All of the dots represent procedure objects.

4.1.2.1 Defining a type of SM

Let's go back to the definition of the `Accumulator` class, and study it piece by piece.

First, we define the class and say that it is a subclass of `SM`:

```
class Accumulator(SM):
```

Next, we define an attribute of the class, `startState`, which is the starting state of the machine. In this case, our accumulator starts up with the value 0.

```
    startState = 0
```

Note that `startState` is required by the underlying `SM` class, so we must either define it in our subclass definition or use a default value from the `SM` superclass.

The next method defines both the next-state function and the output function, by taking the current state and input as arguments and returning a tuple containing both the next state and the output.

For our accumulator, the next state is just the sum of the previous state and the input; and the output is the same thing:

```
    def getNextValues(self, state, inp):
        return (state + inp, state + inp)
```

It is crucial that `getNextValues` *be a pure function; that is, that it not change the state of the object (by assigning to any attributes of* `self`*).* It must simply compute the necessary values and return them. We do not promise anything about how many times this method will be called and in what circumstances.

Sometimes, it is convenient to arrange it so that the class really defines a range of machines with slightly different behavior, which depends on some parameters that we set at the time we create an instance. So, for example, if we wanted to specify the initial value for an accumulator at the time the machine is created, we could add an `__init__` method[30] to our class, which takes an initial value as a parameter and uses it to set an attribute called `startState` of the instance.[31]

```
class Accumulator(SM):
    def __init__(self, initialValue):
        self.startState = initialValue
    def getNextValues(self, state, inp):
        return state + inp, state + inp
```

Now we can make an accumulator and run it like this:

```
>>> c = Accumulator(100)
>>> c.start()
>>> c.step(20)
120
>>> c.step(2)
122
```

[30] Remember that the `__init__` method is a special feature of the Python object-oriented system, which is called whenever an instance of the associated class is created.

[31] Note that in the original version of `Accumulator`, `startState` was an attribute of the class, since it was the same for every instance of the class; now that we want it to be different for different instances, we need `startState` to be an attribute of the instance rather than the class, which is why we assign it in the `__init__` method, which modifies the already-created instance.

4.1.2.2 The SM Class

The `SM` class contains generally useful methods that apply to all state machines. A state machine is an instance of any subclass of `SM`, that has defined the attribute `startState` and the method `getNextValues`, as we did for the `Accumulator` class. Here we examine these methods in more detail.

Running a machine

The first group of methods allows us to run a state machine. To run a machine is to provide it with a sequence of inputs and then sequentially go forward, computing the next state and generating the next output, as if we were filling in a state table.

To run a machine, we have to start by calling the `start` method. All it does is create an attribute of the instance, called `state`, and assign to it the value of the `startState` attribute. It is crucial that we have both of these attributes: if we were to just modify `startState`, then if we wanted to run this machine again, we would have permanently forgotten what the starting state should be. Note that `state` becomes a repository for the state of **this** instance; however we should not mutate it directly. This variable becomes the internal representation of state for each instance of this class.

```
class SM:
    def start(self):
        self.state = self.startState
```

Once it has started, we can ask it to take a step, using the `step` method, which, given an input, computes the output and updates the internal state of the machine, and returns the output value.

```
    def step(self, inp):
        (s, o) = self.getNextValues(self.state, inp)
        self.state = s
        return o
```

To run a machine on a whole sequence of input values, we can use the `transduce` method, which will return the sequence of output values that results from feeding the elements of the list `inputs` into the machine in order.

```
    def transduce(self, inputs):
        self.start()
        return [self.step(inp) for inp in inputs]
```

Here are the results of running `transduce` on our accumulator machine. We run it twice, first with a simple call that does not generate any debugging information, and simply returns the result. The second time, we ask it to be verbose, resulting in a print-out of what is happening on the intermediate steps.[32]

[32] In fact, the second machine trace, and all the others in this section were generated with a call like:

```
>>> a.transduce([100, -3, 4, -123, 10], verbose = True, compact = True)
```

See the *Infrastructure Guide* for details on different debugging options. To simplify the code examples we show in these notes, we have omitted parts of the code that are responsible for debugging printouts.

```
a = Accumulator()
>>> a.transduce([100, -3, 4, -123, 10])
[100, 97, 101, -22, -12]
>>> a.transduce([100, -3, 4, -123, 10], verbose = True)
Start state: 0
In: 100 Out: 100 Next State: 100
In: -3 Out: 97 Next State: 97
In: 4 Out: 101 Next State: 101
In: -123 Out: -22 Next State: -22
In: 10 Out: -12 Next State: -12
[100, 97, 101, -22, -12]
```

Some machines do not take any inputs; in that case, we can simply call the `SM run` method, which is equivalent to doing `transduce` on an input sequence of `[None, None, None, ...]`.

```
    def run(self, n = 10):
        return self.transduce([None]*n)
```

4.1.2.2.1 Default methods

This section is optional.

In order to make the specifications for the simplest machine types as short as possible, we have also provided a set of default methods in the `SM` class. These default methods say that, unless they are overridden in a subclass, as they were when we defined `Accumulator`, we will assume that a machine starts in state `None` and that its output is the same as its next state.

```
    startState = None
    def getNextValues(self, state, inp):
        nextState = self.getNextState(state, inp)
        return (nextState, nextState)
```

Because these methods are provided in `SM`, we can define, for example, a state machine whose output is always `k` times its input, with this simple class definition, which defines a `getNextState` procedure that simply returns a value that is treated as both the next state and the output.

```
class Gain(SM):
    def __init__(self, k):
        self.k = k
    def getNextState(self, state, inp):
        return inp * self.k
```

We can use this class as follows:

```
>>> g = Gain(3)
>>> g.transduce([1.1, -2, 100, 5])
[3.3000000000000003, -6, 300, 15]
```

The parameter `k` is specified at the time the instance is created. Then, the output of the machine is always just `k` times the input.

We can also use this strategy to write the `Accumulator` class even more succinctly:

```
class Accumulator(SM):
    startState = 0
    def getNextState(self, state, inp):
        return state + inp
```

The output of the `getNextState` method will be treated both as the output and the next state of the machine, because the inherited `getNextValues` function uses it to compute both values.

4.1.2.3 More examples

Here are Python versions of the rest of the machines we introduced in the first section.

Language acceptor

Here is a Python class for a machine that "accepts" the language that is any prefix of the infinite sequence `['a', 'b', 'c', 'a', 'b', 'c', ....]`.

```
class ABC(SM):
    startState = 0
    def getNextValues(self, state, inp):
        if state == 0 and inp == 'a':
            return (1, True)
        elif state == 1 and inp == 'b':
            return (2, True)
        elif state == 2 and inp == 'c':
            return (0, True)
        else:
            return (3, False)
```

It behaves as we would expect. As soon as it sees a character that deviates from the desired sequence, the output is `False`, and it will remain `False` for ever after.

```
>>> abc = ABC()
>>> abc.transduce(['a','a','a'], verbose = True)
Start state: 0
In: a Out: True Next State: 1
In: a Out: False Next State: 3
In: a Out: False Next State: 3
[True, False, False]

>>> abc.transduce(['a', 'b', 'c', 'a', 'c', 'a', 'b'], verbose = True)
Start state: 0
In: a Out: True Next State: 1
In: b Out: True Next State: 2
In: c Out: True Next State: 0
In: a Out: True Next State: 1
In: c Out: False Next State: 3
In: a Out: False Next State: 3
In: b Out: False Next State: 3
[True, True, True, True, False, False, False]
```

Count up and down

This is a direct translation of the machine defined in **section 4.1.1.2**.

```
class UpDown(SM):
    startState = 0
    def getNextState(self, state, inp):
        if inp == 'u':
            return state + 1
        else:
            return state - 1
```

We take advantage of the default `getNextValues` method to make the output the same as the next state.

```
>>> ud = UpDown()
>>> ud.transduce(['u','u','u','d','d','u'], verbose = True)
Start state: 0
In: u Out: 1 Next State: 1
In: u Out: 2 Next State: 2
In: u Out: 3 Next State: 3
In: d Out: 2 Next State: 2
In: d Out: 1 Next State: 1
In: u Out: 2 Next State: 2
[1, 2, 3, 2, 1, 2]
```

Delay

In order to make a machine that delays its input stream by one time step, we have to specify what the first output should be. We do this by passing the parameter, `v0`, into the `__init__` method of the `Delay` class. The state of a `Delay` machine is just the input from the previous step, and the output is the state (which is, therefore, the input from the previous time step).

```
class Delay(SM):
    def __init__(self, v0):
        self.startState = v0
    def getNextValues(self, state, inp):
        return (inp, state)
```

```
>>> d = Delay(7)
>>> d.transduce([3, 1, 2, 5, 9], verbose = True)
Start state: 7
In: 3 Out: 7 Next State: 3
In: 1 Out: 3 Next State: 1
In: 2 Out: 1 Next State: 2
In: 5 Out: 2 Next State: 5
In: 9 Out: 5 Next State: 9
[7, 3, 1, 2, 5]
```

```
>>> d100 = Delay(100)
>>> d100.transduce([3, 1, 2, 5, 9], verbose = True)
Start state: 100
In: 3 Out: 100 Next State: 3
```

```
In: 1 Out: 3 Next State: 1
In: 2 Out: 1 Next State: 2
In: 5 Out: 2 Next State: 5
In: 9 Out: 5 Next State: 9
[100, 3, 1, 2, 5]
```

We will use this machine so frequently that we put its definition in the `sm` module (file), along with the class.

R

We can use `R` as another name for the `Delay` class of state machines. It will be an important primitive in a compositional system of *linear time-invariant systems*, which we explore in the next chapter.

Average2

Here is a state machine whose output at time t is the average of the input values from times $t-1$ and t.

```
class Average2(SM):
    startState = 0
    def getNextValues(self, state, inp):
        return (inp, (inp + state) / 2.0)
```

It needs to remember the previous input, so the next state is equal to the input. The output is the average of the current input and the state (because the state is the previous input).

```
>>> a2 = Average2()
>>> a2.transduce([10, 5, 2, 10], verbose = True, compact = True)
Start state: 0
In: 10 Out: 5.0 Next State: 10
In: 5 Out: 7.5 Next State: 5
In: 2 Out: 3.5 Next State: 2
In: 10 Out: 6.0 Next State: 10
[5.0, 7.5, 3.5, 6.0]
```

Sum of last three inputs

Here is an example of a state machine where the state is actually a list of values. Generally speaking, the state can be anything (a dictionary, an array, a list); but it is important to be sure that the `getNextValues` method does not make direct changes to components of the state, instead returning a *new copy* of the state with appropriate changes. We may make several calls to the `getNextValues` function on one step (or, later in our work, call the `getNextValues` function with several different inputs to see what would happen under different choices); these function calls are made to find out a value of the next state, but if they actually change the state, then the same call with the same arguments may return a different value the next time.

This machine generates as output at time t the sum of i_{t-2}, i_{t-1} and i_t; that is, of the last three inputs. In order to do this, it has to remember the values of two previous inputs; so the state is a pair of numbers. We have defined it so that the initial state is `(0, 0)`. The `getNextValues`

method gets rid of the oldest value that it has been remembering, and remembers the current input as part of the state; the output is the sum of the current input with the two old inputs that are stored in the state. Note that the first line of the `getNextValues` procedure is a structured assignment (see **section 3.3**).

```
class SumLast3 (SM):
    startState = (0, 0)
    def getNextValues(self, state, inp):
        (previousPreviousInput, previousInput) = state
        return ((previousInput, inp),
                previousPreviousInput + previousInput + inp)
```

```
>>> sl3 = SumLast3()
>>> sl3.transduce([2, 1, 3, 4, 10, 1, 2, 1, 5], verbose = True)
Start state: (0, 0)
In: 2 Out: 2 Next State: (0, 2)
In: 1 Out: 3 Next State: (2, 1)
In: 3 Out: 6 Next State: (1, 3)
In: 4 Out: 8 Next State: (3, 4)
In: 10 Out: 17 Next State: (4, 10)
In: 1 Out: 15 Next State: (10, 1)
In: 2 Out: 13 Next State: (1, 2)
In: 1 Out: 4 Next State: (2, 1)
In: 5 Out: 8 Next State: (1, 5)
[2, 3, 6, 8, 17, 15, 13, 4, 8]
```

Selector

A simple functional machine that is very useful is the `Select` machine. You can make many different versions of this, but the simplest one takes an input that is a stream of lists or tuples of several values (or structures of values) and generates the stream made up only of the `k`th elements of the input values. Which particular component this machine is going to select is determined by the value `k`, which is passed in at the time the machine instance is initialized.

```
class Select (SM):
    def __init__(self, k):
        self.k = k
    def getNextState(self, state, inp):
        return inp[self.k]
```

4.1.3 Simple parking gate controller

As one more demonstration, here is a simple example of a finite-state controller for a gate leading out of a parking lot.

The gate has three sensors:

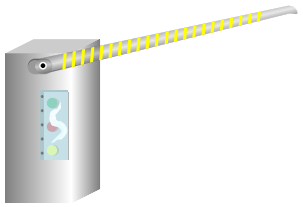


Image by MIT OpenCourseWare.

- `gatePosition` has one of three values `'top'`, `'middle'`, `'bottom'`, signifying the position of the arm of the parking gate.
- `carAtGate` is `True` if a car is waiting to come through the gate and `False` otherwise.
- `carJustExited` is `True` if a car has just passed through the gate; it is true for only one step before resetting to `False`.

> *Pause to try 4.1.* How many possible inputs are there?
>
> A: 12.

The gate has three possible outputs (think of them as controls to the motor for the gate arm): `'raise'`, `'lower'`, and `'nop'`. (Nop means "no operation.")

Roughly, here is what the gate needs to do:

- If a car wants to come through, the gate needs to raise the arm until it is at the top position.
- Once the gate is at the top position, it has to stay there until the car has driven through the gate.
- After the car has driven through the gate needs to lower the arm until it reaches the bottom position.

So, we have designed a simple finite-state controller with a state transition diagram as shown in **figure 4.3**. The machine has four possible states: `'waiting'` (for a car to arrive at the gate), `'raising'` (the arm), `'raised'` (the arm is at the top position and we're waiting for the car to drive through the gate), and `'lowering'` (the arm). To keep the figure from being too cluttered, we do not label each arc with every possible input that would cause that transition to be made: instead, we give a condition (think of it as a Boolean expression) that, if true, would cause the transition to be followed. The conditions on the arcs leading out of each state cover all the possible inputs, so the machine remains completely well specified.

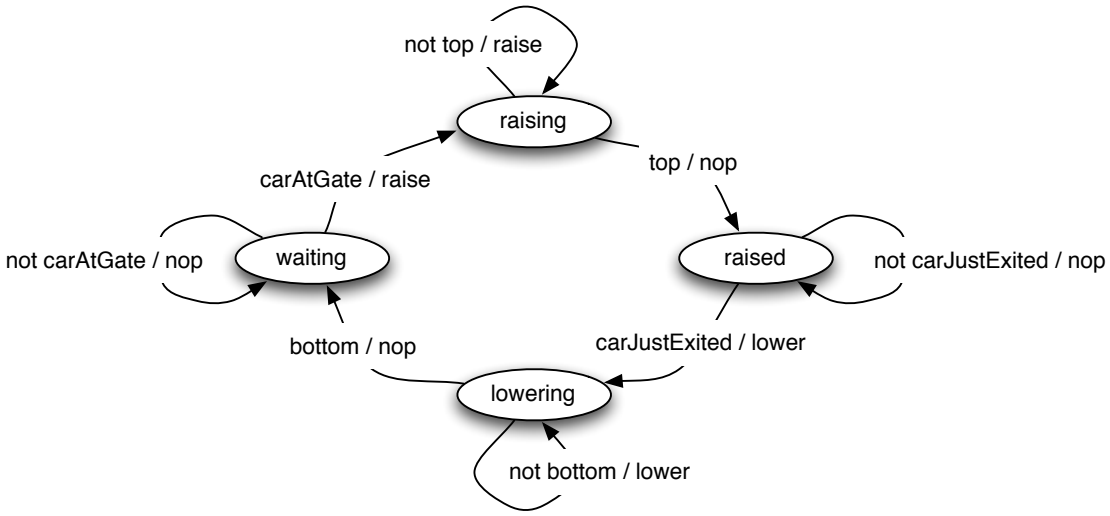


Figure 4.3 State transition diagram for parking gate controller.

Here is a simple state machine that implements the controller for the parking gate. For compactness, the `getNextValues` method starts by determining the next state of the gate. Then, depending on the next state, the `generateOutput` method selects the appropriate output.

```
class SimpleParkingGate (SM):
    startState = 'waiting'

    def generateOutput(self, state):
        if state == 'raising':
            return 'raise'
        elif state == 'lowering':
            return 'lower'
        else:
            return 'nop'

    def getNextValues(self, state, inp):
        (gatePosition, carAtGate, carJustExited) = inp
        if state == 'waiting' and carAtGate:
            nextState = 'raising'
        elif state == 'raising' and gatePosition == 'top':
            nextState = 'raised'
        elif state == 'raised' and carJustExited:
            nextState = 'lowering'
        elif state == 'lowering' and gatePosition == 'bottom':
            nextState = 'waiting'
        else:
            nextState = state
        return (nextState, self.generateOutput(nextState))
```

In the situations where the state does not change (that is, when the arcs lead back to the same state in the diagram), we do not explicitly specify the next state in the code: instead, we cover it in the `else` clause, saying that unless otherwise specified, the state stays the same. So, for example, if the state is `raising` but the `gatePosition` is not yet `top`, then the state simply stays `raising` until the top is reached.

```
>>> spg = SimpleParkingGate()
>>> spg.transduce(testInput, verbose = True)
Start state: waiting
In: ('bottom', False, False) Out: nop Next State: waiting
In: ('bottom', True, False) Out: raise Next State: raising
In: ('bottom', True, False) Out: raise Next State: raising
In: ('middle', True, False) Out: raise Next State: raising
In: ('middle', True, False) Out: raise Next State: raising
In: ('middle', True, False) Out: raise Next State: raising
In: ('top', True, False) Out: nop Next State: raised
In: ('top', True, False) Out: nop Next State: raised
In: ('top', True, False) Out: nop Next State: raised
In: ('top', True, True) Out: lower Next State: lowering
In: ('top', True, True) Out: lower Next State: lowering
In: ('top', True, False) Out: lower Next State: lowering
In: ('middle', True, False) Out: lower Next State: lowering
In: ('middle', True, False) Out: lower Next State: lowering
In: ('middle', True, False) Out: lower Next State: lowering
```

```
In: ('bottom', True, False) Out: nop Next State: waiting
In: ('bottom', True, False) Out: raise Next State: raising
['nop', 'raise', 'raise', 'raise', 'raise', 'raise', 'nop', 'nop', 'nop', 'lower', 'lower',
'lower', 'lower', 'lower', 'lower', 'nop', 'raise']
```

Exercise 4.1. What would the code for this machine look like if it were written without using the `generateOutput` method?

4.2 Basic combination and abstraction of state machines

In the previous section, we studied the definition of a primitive state machine, and saw a number of examples. State machines are useful for a wide variety of problems, but specifying complex machines by explicitly writing out their state transition functions can be quite tedious. Ultimately, we will want to build large state-machine descriptions compositionally, by specifying primitive machines and then combining them into more complex systems. We will start here by looking at ways of combining state machines.

We can apply our PCAP (primitive, combination, abstraction, pattern) methodology, to build more complex SMs out of simpler ones. In the rest of this section we consider "dataflow" compositions, where inputs and outputs of primitive machines are connected together; after that, we consider "conditional" compositions that make use of different sub-machines depending on the input to the machine, and finally "sequential" compositions that run one machine after another.

4.2.1 Cascade composition

In cascade composition, we take two machines and use the output of the first one as the input to the second, as shown in **figure 4.4**. The result is a new composite machine, whose input vocabulary is the input vocabulary of the first machine and whose output vocabulary is the output vocabulary of the second machine. It is, of course, crucial that the output vocabulary of the first machine be the same as the input vocabulary of the second machine.

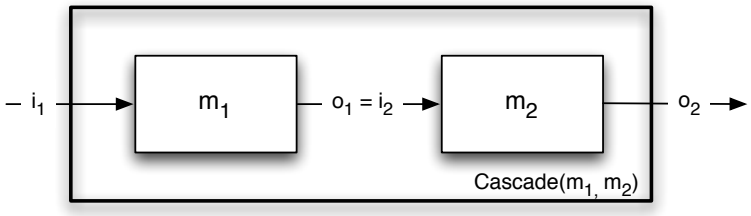


Figure 4.4 Cascade composition of state machines

Recalling the `Delay` machine from the previous section, let's see what happens if we make the cascade composition of two delay machines. Let m_1 be a delay machine with initial value 99 and m_2 be a delay machine with initial value 22. Then $Cascade(m_1, m_2)$ is a new state machine, constructed by making the output of m_1 be the input of m_2. Now, imagine we feed a sequence of values, $3, 8, 2, 4, 6, 5$, into the composite machine, m. What will come out? Let's try to understand this by making a table of the states and values at different times:

time	0	1	2	3	4	5	6
m_1 input	3	8	2	4	6	5	
m_1 state	99	3	8	2	4	6	5
m_1 output	99	3	8	2	4	6	
m_2 input	99	3	8	2	4	6	
m_2 state	22	99	3	8	2	4	6
m_2 output	22	99	3	8	2	4	

The output sequence is $22, 99, 3, 8, 2, 4$, which is the input sequence, delayed by two time steps.

Another way to think about cascade composition is as follows. Let the input to m_1 at time t be called $i_1[t]$ and the output of m_1 at time t be called $o_1[t]$. Then, we can describe the workings of the delay machine in terms of an equation:

$$o_1[t] = i_1[t-1] \text{ for all values of } t > 0;$$
$$o_1[0] = init_1$$

that is, that the output value at every time t is equal to the input value at the previous time step. You can see that in the table above. The same relation holds for the input and output of m_2:

$$o_2[t] = i_2[t-1] \text{ for all values of } t > 0.$$
$$o_2[0] = init_2$$

Now, since we have connected the output of m_1 to the input of m_2, we also have that $i_2[t] = o_1[t]$ for all values of t. This lets us make the following derivation:

$$\begin{aligned} o_2[t] &= i_2[t-1] \\ &= o_1[t-1] \\ &= i_1[t-2] \end{aligned}$$

This makes it clear that we have built a "delay by two" machine, by cascading two single delay machines.

As with all of our systems of combination, we will be able to form the cascade composition not only of two primitive machines, but of any two machines that we can make, through any set of compositions of primitive machines.

Here is the Python code for an `Increment` machine. It is a pure function whose output at time `t` is just the input at time `t` plus the constant `incr`. The `safeAdd` function is the same as addition, if the inputs are numbers. We will see, later, why it is important.

```
class Increment(SM):
    def __init__(self, incr):
        self.incr = incr
    def getNextState(self, state, inp):
        return safeAdd(inp, self.incr)
```

Exercise 4.2. Derive what happens when you cascade two delay-by-two machines?

Exercise 4.3. What is the difference between these two machines?

```
>>> foo1 = sm.Cascade(sm.Delay(100), Increment(1))
>>> foo2 = sm.Cascade(Increment(1), sm.Delay(100))
```

Demonstrate by drawing a table of their inputs, states, and outputs, over time.

4.2.2 Parallel composition

In parallel composition, we take two machines and run them "side by side". They both take the same input, and the output of the composite machine is the pair of outputs of the individual machines. The result is a new composite machine, whose input vocabulary is the same as the input vocabulary of the component machines (which is the same for both machines) and whose output vocabulary is pairs of elements, the first from the output vocabulary of the first machine and the second from the output vocabulary of the second machine. **Figure 4.5** shows two types of parallel composition; in this section we are talking about the first type.

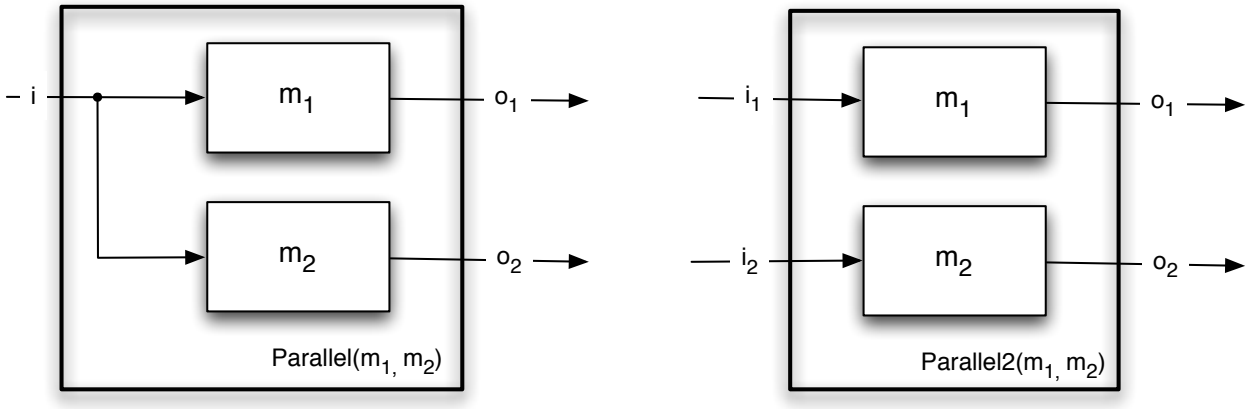


Figure 4.5 Parallel and Parallel2 compositions of state machines.

In Python, we can define a new class of state machines, called `Parallel`, which is a subclass of `SM`. To make an instance of `Parallel`, we pass two `SM`s of any type into the initializer. The state of the parallel machine is a pair consisting of the states of the constituent machines. So, the starting state is the pair of the starting states of the constituents.

```
class Parallel (SM):
    def __init__(self, sm1, sm2):
        self.m1 = sm1
        self.m2 = sm2
        self.startState = (sm1.startState, sm2.startState)
```

To get a new state of the composite machine, we just have to get new states for each of the constituents, and return the pair of them; similarly for the outputs.

```
    def getNextValues(self, state, inp):
        (s1, s2) = state
        (newS1, o1) = self.m1.getNextValues(s1, inp)
        (newS2, o2) = self.m2.getNextValues(s2, inp)
        return ((newS1, newS2), (o1, o2))
```

Parallel2

Sometimes we will want a variant on parallel combination, in which rather than having the input be a single item which is fed to both machines, the input is a pair of items, the first of which is fed to the first machine and the second to the second machine. This composition is shown in the second part of **figure 4.5**.

Here is a Python class that implements this two-input parallel composition. It can inherit the `__init__` method from `Parallel`, so use `Parallel` as the superclass, and we only have to define two methods.

```
class Parallel2 (Parallel):
    def getNextValues(self, state, inp):
        (s1, s2) = state
        (i1, i2) = splitValue(inp)
        (newS1, o1) = self.m1.getNextValues(s1, i1)
        (newS2, o2) = self.m2.getNextValues(s2, i2)
        return ((newS1, newS2), (o1, o2))
```

Later, when dealing with feedback systems (section **section 4.2.3**), we will need to be able to deal with `'undefined'` as an input. If the `Parallel2` machine gets an input of `'undefined'`, then we want to pass `'undefined'` into the constituent machines. We make our code more beautiful by defining the helper function below, which is guaranteed to return a pair, if its argument is either a pair or `'undefined'`.[33]

```
def splitValue(v):
    if v == 'undefined':
        return ('undefined', 'undefined')
    else:
        return v
```

ParallelAdd

The `ParallelAdd` state machine combination is just like `Parallel`, except that it has a single output whose value is the sum of the outputs of the constituent machines. It is straightforward to define:

[33] We are trying to make the code examples we show here as simple and clear as possible; if we were writing code for actual deployment, we would check and generate error messages for all sorts of potential problems (in this case, for instance, if `v` is neither `None` nor a two-element list or tuple.)

```
class ParallelAdd (Parallel):
    def getNextValues(self, state, inp):
        (s1, s2) = state
        (newS1, o1) = self.m1.getNextValues(s1, inp)
        (newS2, o2) = self.m2.getNextValues(s2, inp)
        return ((newS1, newS2), o1 + o2)
```

4.2.3 Feedback composition

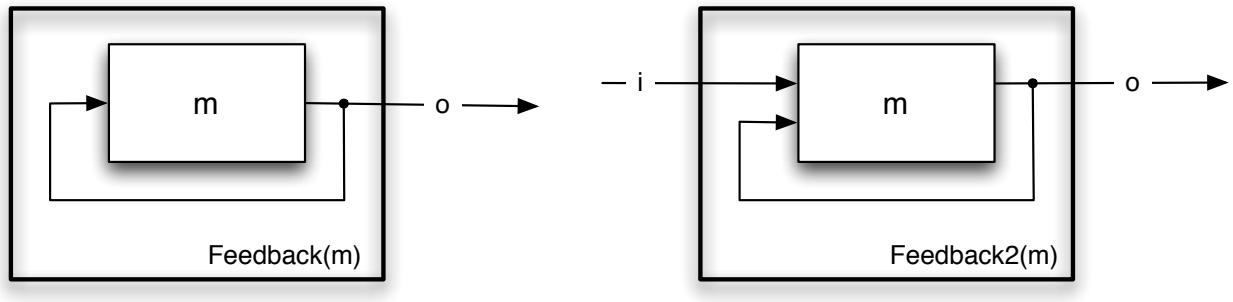


Figure 4.6 Two forms of feedback composition.

Another important means of combination that we will use frequently is the feedback combinator, in which the output of a machine is fed back to be the input of the same machine at the next step, as shown in **figure 4.6**. The first value that is fed back is the output associated with the initial state of the machine on which we are operating. It is crucial that the input and output vocabularies of the machine are the same (because the output at step t will be the input at step $t+1$). Because we have fed the output back to the input, this machine does not consume any inputs; but we will treat the feedback value as an output of this machine.

Here is an example of using feedback to make a machine that counts. We can start with a simple machine, an incrementer, that takes a number as input and returns that same number plus 1 as the output. By itself, it has no memory. Here is its formal description:

$$\begin{aligned} S &= \textit{numbers} \\ I &= \textit{numbers} \\ O &= \textit{numbers} \\ n(s,i) &= i+1 \\ o(s,i) &= n(s,i) \\ s_0 &= 0 \end{aligned}$$

What would happen if we performed the feedback operation on this machine? We can try to understand this in terms of the input/output equations. From the definition of the increment machine, we have

$$o[t] = i[t] + 1 \quad .$$

And if we connect the input to the output, then we will have

$$i[t] = o[t] \quad .$$

And so, we have a problem; these equations cannot be satisfied.

A crucial requirement for applying feedback to a machine is: *that machine must not have a direct dependence of its output on its input.*

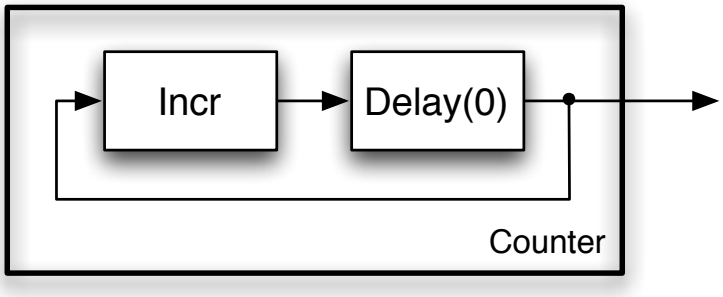


Figure 4.7 Counter made with feedback and serial combination of an incrementer and a delay.

We have already explored a `Delay` machine, which delays its output by one step. We can delay the result of our incrementer, by cascading it with a `Delay` machine, as shown in **figure 4.7**. Now, we have the following equations describing the system:

$$o_i[t] = i_i[t] + 1$$
$$o_d[t] = i_d[t-1]$$
$$i_i[t] = o_d[t]$$
$$i_d[t] = o_i[t]$$

The first two equations describe the operations of the increment and delay boxes; the second two describe the wiring between the modules. Now we can see that, in general,

$$o_i[t] = i_i[t] + 1$$
$$o_i[t] = o_d[t] + 1$$
$$o_i[t] = i_d[t-1] + 1$$
$$o_i[t] = o_i[t-1] + 1$$

that is, that the output of the incrementer is going to be one greater on each time step.

Exercise 4.4. How could you use feedback and a *negation* primitive machine (which is a pure function that takes a Boolean as input and returns the negation of that Boolean) to make a machine whose output alternates between *true* and *false*.

4.2.3.1 Python Implementation

Following is a Python implementation of the feedback combinator, as a new subclass of `SM` that takes, at initialization time, a state machine.

```
class Feedback (SM):
    def __init__(self, sm):
        self.m = sm
        self.startState = self.m.startState
```

The starting state of the feedback machine is just the state of the constituent machine.

Generating an output for the feedback machine is interesting: by our hypothesis that the output of the constituent machine cannot depend directly on the current input, it means that, for the purposes of generating the output, we can actually feed an explicitly undefined value into the machine as input. Why would we do this? The answer is that we do not know what the input value should be (in fact, it is defined to be the output that we are trying to compute).

We must, at this point, add an extra condition on our `getNextValues` methods. They have to be prepared to accept `'undefined'` as an input. If they get an undefined input, they should return `'undefined'` as an output. For convenience, in our files, we have defined the procedures `safeAdd` and `safeMul` to do addition and multiplication, but passing through `'undefined'` if it occurs in either argument.

So: if we pass `'undefined'` into the constituent machine's `getNextValues` method, we must not get `'undefined'` back as output; if we do, it means that there is an immediate dependence of the output on the input. Now we know the output `o` of the machine.

To get the next state of the machine, we get the next state of the constituent machine, by taking the feedback value, `o`, that we just computed and using it as input for `getNextValues`. This will generate the next state of the feedback machine. (Note that throughout this process `inp` is ignored—a feedback machine has no input.)

```
    def getNextValues(self, state, inp):
        (ignore, o) = self.m.getNextValues(state, 'undefined')
        (newS, ignore) = self.m.getNextValues(state, o)
        return (newS, o)
```

Now, we can construct the counter we designed. The `Increment` machine, as we saw in its definition, uses a `safeAdd` procedure, which has the following property: if either argument is `'undefined'`, then the answer is `'undefined'`; otherwise, it is the sum of the inputs.

```
def makeCounter(init, step):
    return sm.Feedback(sm.Cascade(Increment(step), sm.Delay(init)))

>>> c = makeCounter(3, 2)
>>> c.run(verbose = True)
Start state: (None, 3)
Step: 0
 Feedback_96
     Cascade_97
         Increment_98 In: 3 Out: 5 Next State: 5
```

```
        Delay_99 In: 5 Out: 3 Next State: 5
Step: 1
 Feedback_96
     Cascade_97
         Increment_98 In: 5 Out: 7 Next State: 7
         Delay_99 In: 7 Out: 5 Next State: 7
Step: 2
 Feedback_96
     Cascade_97
         Increment_98 In: 7 Out: 9 Next State: 9
         Delay_99 In: 9 Out: 7 Next State: 9
Step: 3
 Feedback_96
     Cascade_97
         Increment_98 In: 9 Out: 11 Next State: 11
         Delay_99 In: 11 Out: 9 Next State: 11
Step: 4
 Feedback_96
     Cascade_97
         Increment_98 In: 11 Out: 13 Next State: 13
         Delay_99 In: 13 Out: 11 Next State: 13
...

[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

(The numbers, like 96 in `Feedback_96` are not important; they are just tags generated internally to indicate different instances of a class.)

Exercise 4.5. Draw state tables illustrating whether the following machines are different, and if so, how:

```
m1 = sm.Feedback(sm.Cascade(sm.Delay(1),Increment(1)))
m2 = sm.Feedback(sm.Cascade(Increment(1), sm.Delay(1)))
```

4.2.3.2 Fibonacci

Now, we can get very fancy. We can generate the Fibonacci sequence (1, 1, 2, 3, 5, 8, 13, 21, etc), in which the first two outputs are 1, and each subsequent output is the sum of the two previous outputs, using a combination of very simple machines. Basically, we have to arrange for the output of the machine to be fed back into a parallel combination of elements, one of which delays the value by one step, and one of which delays by two steps. Then, those values are added, to compute the next output. **Figure 4.8** shows a diagram of one way to construct this system.

The corresponding Python code is shown below. First, we have to define a new component machine. An `Adder` takes pairs of numbers (appearing simultaneously) as input, and immediately generates their sum as output.

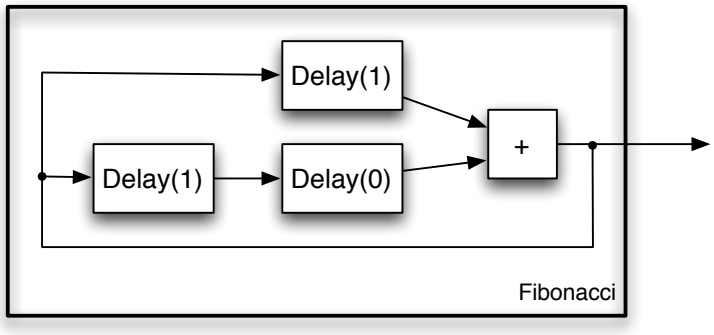


Figure 4.8 Machine to generate the Fibonacci sequence.

```
class Adder(SM):
    def getNextState(self, state, inp):
        (i1, i2) = splitValue(inp)
        return safeAdd(i1, i2)
```

Now, we can define our `fib` machine. It is a great example of building a complex machine out of very nearly trivial components. In fact, we will see in the next module that there is an interesting and important class of machines that can be constructed with cascade and parallel compositions of delay, adder, and gain machines. It is crucial for the delay machines to have the right values (as shown in the figure) in order for the sequence to start off correctly.

```
>>> fib = sm.Feedback(sm.Cascade(sm.Parallel(sm.Delay(1),
                                             sm.Cascade(sm.Delay(1), sm.Delay(0))),
                                  Adder()))

>>> fib.run(verbose = True)
Start state: ((1, (1, 0)), None)
Step: 0
 Feedback_100
     Cascade_101
         Parallel_102
             Delay_103 In: 1 Out: 1 Next State: 1
             Cascade_104
                 Delay_105 In: 1 Out: 1 Next State: 1
                 Delay_106 In: 1 Out: 0 Next State: 1
         Adder_107 In: (1, 0) Out: 1 Next State: 1
Step: 1
 Feedback_100
     Cascade_101
         Parallel_102
             Delay_103 In: 2 Out: 1 Next State: 2
             Cascade_104
                 Delay_105 In: 2 Out: 1 Next State: 2
                 Delay_106 In: 1 Out: 1 Next State: 1
         Adder_107 In: (1, 1) Out: 2 Next State: 2
Step: 2
 Feedback_100
     Cascade_101
         Parallel_102
             Delay_103 In: 3 Out: 2 Next State: 3
```

```
            Cascade_104
                Delay_105 In: 3 Out: 2 Next State: 3
                Delay_106 In: 2 Out: 1 Next State: 2
        Adder_107 In: (2, 1) Out: 3 Next State: 3
Step: 3
 Feedback_100
     Cascade_101
         Parallel_102
             Delay_103 In: 5 Out: 3 Next State: 5
             Cascade_104
                 Delay_105 In: 5 Out: 3 Next State: 5
                 Delay_106 In: 3 Out: 2 Next State: 3
         Adder_107 In: (3, 2) Out: 5 Next State: 5
...

[1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Exercise 4.6. What would we have to do to this machine to get the sequence `[1, 1, 2, 3, 5, ...]`?

Exercise 4.7. Define fib as a composition involving only two delay components and an adder. You might want to use an instance of the `Wire` class.
A `Wire` is the completely passive machine, whose output is always instantaneously equal to its input. It is not very interesting by itself, but sometimes handy when building things.

```
class Wire(SM):
    def getNextState(self, state, inp):
        return inp
```

Exercise 4.8. Use feedback and a multiplier (analogous to `Adder`) to make a machine whose output doubles on every step.

Exercise 4.9. Use feedback and a multiplier (analogous to `Adder`) to make a machine whose output squares on every step.

4.2.3.3 Feedback2

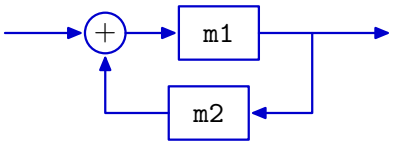
The second part of **figure 4.6** shows a combination we call *feedback2*: it assumes that it takes a machine with two inputs and one output, and connects the output of the machine to the second input, resulting in a machine with one input and one output.

Feedback2 is very similar to the basic feedback combinator, but it gives, as input to the constituent machine, the pair of the input to the machine and the feedback value.

```
class Feedback2 (Feedback):
    def getNextValues(self, state, inp):
        (ignore, o) = self.m.getNextValues(state, (inp, 'undefined'))
        (newS, ignore) = self.m.getNextValues(state, (inp, o))
        return (newS, o)
```

4.2.3.4 FeedbackSubtract and FeedbackAdd

In *feedback addition* composition, we take two machines and connect them as shown below:



If `m1` and `m2` are state machines, then you can create their feedback addition composition with

```
newM = sm.FeedbackAdd(m1, m2)
```

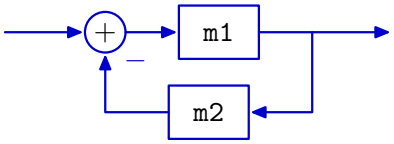
Now `newM` is itself a state machine. So, for example,

```
newM = sm.FeedbackAdd(sm.R(0), sm.Wire())
```

makes a machine whose output is the sum of all the inputs it has ever had (remember that `sm.R` is shorthand for `sm.Delay`). You can test it by feeding it a sequence of inputs; in the example below, it is the numbers 0 through 9:

```
>>> newM.transduce(range(10))
[0, 0, 1, 3, 6, 10, 15, 21, 28, 36]
```

Feedback subtraction composition is the same, except the output of `m2` is *subtracted* from the input, to get the input to `m1`.



Note that if you want to apply one of the feedback operators in a situation where there is only one machine, you can use the `sm.Gain(1.0)` machine (defined **section 4.1.2.2.1**), which is essentially a wire, as the other argument.

4.2.3.5 Factorial

We will do one more tricky example, and illustrate the use of `Feedback2`. What if we wanted to generate the sequence of numbers $\{1!, 2!, 3!, 4!, \ldots\}$ (where $k! = 1 \cdot 2 \cdot 3 \ldots \cdot k$)? We can do so by multiplying the previous value of the sequence by a number equal to the

"index" of the sequence. **Figure 4.9** shows the structure of a machine for solving this problem. It uses a counter (which is, as we saw before, made with feedback around a delay and increment) as the input to a machine that takes a single input, and multiplies it by the output value of the machine, fed back through a delay.

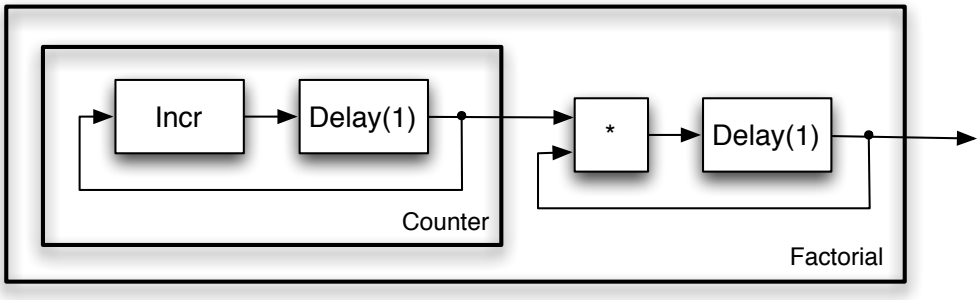


Figure 4.9 Machine to generate the Factorial sequence.

Here is how to do it in Python; we take advantage of having defined counter machines to abstract away from them and use that definition here without thinking about its internal structure. The initial values in the delays get the series started off in the right place. What would happen if we started at 0?

```
fact = sm.Cascade(makeCounter(1, 1),
                  sm.Feedback2(sm.Cascade(Multiplier(), sm.Delay(1))))
```

```
>>> fact.run(verbose = True)
Start state: ((None, 1), (None, 1))
Step: 0
 Cascade_1
     Feedback_2
         Cascade_3
             Increment_4 In: 1 Out: 2 Next State: 2
             Delay_5 In: 2 Out: 1 Next State: 2
     Feedback2_6
         Cascade_7
             Multiplier_8 In: (1, 1) Out: 1 Next State: 1
             Delay_9 In: 1 Out: 1 Next State: 1
Step: 1
 Cascade_1
     Feedback_2
         Cascade_3
             Increment_4 In: 2 Out: 3 Next State: 3
             Delay_5 In: 3 Out: 2 Next State: 3
     Feedback2_6
         Cascade_7
             Multiplier_8 In: (2, 1) Out: 2 Next State: 2
             Delay_9 In: 2 Out: 1 Next State: 2
Step: 2
 Cascade_1
     Feedback_2
         Cascade_3
             Increment_4 In: 3 Out: 4 Next State: 4
             Delay_5 In: 4 Out: 3 Next State: 4
```

```
    Feedback2_6
        Cascade_7
            Multiplier_8 In: (3, 2) Out: 6 Next State: 6
            Delay_9 In: 6 Out: 2 Next State: 6
Step: 3
 Cascade_1
    Feedback_2
        Cascade_3
            Increment_4 In: 4 Out: 5 Next State: 5
            Delay_5 In: 5 Out: 4 Next State: 5
    Feedback2_6
        Cascade_7
            Multiplier_8 In: (4, 6) Out: 24 Next State: 24
            Delay_9 In: 24 Out: 6 Next State: 24
...

[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

It might bother you that we get a 1 as the zeroth element of the sequence, but it is reasonable as a definition of 0!, because 1 is the multiplicative identity (and is often defined that way by mathematicians).

4.2.4 Plants and controllers

One common situation in which we combine machines is to simulate the effects of coupling a controller and a so-called "plant". A plant is a factory or other external environment that we might wish to control. In this case, we connect two state machines so that the output of the plant (typically thought of as sensory observations) is input to the controller, and the output of the controller (typically thought of as actions) is input to the plant. This is shown schematically in figure 4.10. For example, when you build a Soar brain that interacts with the robot, the robot (and the world in which it is operating) is the "plant" and the brain is the controller. We can build a coupled machine by first connecting the machines in a cascade and then using feedback on that combination.

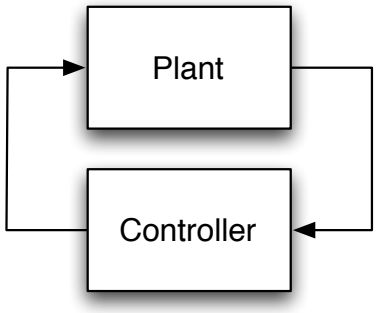


Figure 4.10 Two coupled machines.

As a concrete example, let's think about a robot driving straight toward a wall. It has a distance sensor that allows it to observe the distance to the wall at time t, d[t], and it desires to stop at some distance $d_{desired}$. The robot can execute velocity commands, and we program it to use the following rule to set its velocity at time t, based on its most recent sensor reading:

$$v[t] = K(d_{desired} - d[t-1]) \quad .$$

This controller can also be described as a state machine, whose input sequence is the observed values of d and whose output sequence is the values of v.

$$\begin{aligned} S &= \textit{numbers} \\ I &= \textit{numbers} \\ O &= \textit{numbers} \\ n(s,i) &= K(d_{desired} - i) \\ o(s) &= s \\ s_0 &= d_{init} \end{aligned}$$

Now, we can think about the "plant"; that is, the relationship between the robot and the world. The distance of the robot to the wall changes at each time step depending on the robot's forward velocity and the length of the time steps. Let δT be the length of time between velocity commands issued by the robot. Then we can describe the world with the equation:

$$d[t] = d[t-1] - \delta T v[t-1] \quad .$$

which assumes that a positive velocity moves the robot *toward* the wall (and therefore decreases the distance). This system can be described as a state machine, whose input sequence is the values of the robot's velocity, v, and whose output sequence is the values of its distance to the wall, d.

Finally, we can couple these two systems, as for a simulator, to get a single state machine with no inputs. We can observe the sequence of internal values of d and v to understand how the system is behaving.

In Python, we start by defining the controller machine; the values `k` and `dDesired` are constants of the whole system.

```
k = -1.5
dDesired = 1.0
class WallController(SM):
    def getNextState(self, state, inp):
        return safeMul(k, safeAdd(dDesired, safeMul(-1, inp)))
```

The output being generated is actually `k * (dDesired - inp)`, but because this method is going to be used in a feedback machine, it might have to deal with `'undefined'` as an input. It has no delay built into it.

Think about why we want `k` to be negative. What happens when the robot is closer to the wall than desired? What happens when it is farther from the wall than desired?

Now, we can define a class that describes the behavior of the "plant":

```
deltaT = 0.1
class WallWorld(SM):
    startState = 5
    def getNextValues(self, state, inp):
        return (state - deltaT * inp, state)
```

Setting `startState = 5` means that the robot starts 5 meters from the wall. Note that the output of this machine does not depend instantaneously on the input; so there *is* a delay in it.

Now, we can defined a general combinator for coupling two machines, as in a plant and controller:

```
def coupledMachine(m1, m2):
    return sm.Feedback(sm.Cascade(m1, m2))
```

We can use it to connect our controller to the world, and run it:

```
>>> wallSim = coupledMachine(WallController(), WallWorld())
>>> wallSim.run(30)
[5, 4.4000000000000004, 3.8900000000000001, 3.4565000000000001,
 3.088025, 2.77482125, 2.5085980624999999, 2.2823083531249999,
 2.0899621001562498, 1.9264677851328122, 1.7874976173628905,
 1.6693729747584569, 1.5689670285446884, 1.483621974262985,
 1.4110786781235374, 1.3494168764050067, 1.2970043449442556,
 1.2524536932026173, 1.2145856392222247, 1.1823977933388909,
 1.1550381243380574, 1.1317824056873489, 1.1120150448342465,
 1.0952127881091096, 1.0809308698927431, 1.0687912394088317,
 1.058472553497507, 1.049701670472881, 1.0422464199019488,
 1.0359094569166565]
```

Because `WallWorld` is the second machine in the cascade, its output is the output of the whole machine; so, we can see that the distance from the robot to the wall is converging monotonically to `dDesired` (which is 1).

> *Exercise 4.10.* What kind of behavior do you get with different values of `k`?

4.2.5 Conditionals

We might want to use different machines depending on something that is happening in the outside world. Here we describe three different conditional combinators, that make choices, at the run-time of the machine, about what to do.

4.2.6 Switch

We will start by considering a conditional combinator that runs two machines in parallel, but decides *on every input* whether to send the input into one machine or the other. So, only one of the parallel machines has its state updated on each step. We will call this *switch*, to emphasize the fact that the decision about which machine to execute is being re-made on every step.

Implementing this requires us to maintain the states of both machines, just as we did for parallel combination. The `getNextValues` method tests the condition and then gets a new state and output from the appropriate constituent machine; it also has to be sure to pass through the old state for the constituent machine that was not updated this time.

```
class Switch (SM):
    def __init__(self, condition, sm1, sm2):
        self.m1 = sm1
        self.m2 = sm2
        self.condition = condition
        self.startState = (self.m1.startState, self.m2.startState)

    def getNextValues(self, state, inp):
        (s1, s2) = state
        if self.condition(inp):
            (ns1, o) = self.m1.getNextValues(s1, inp)
            return ((ns1, s2), o)
        else:
            (ns2, o) = self.m2.getNextValues(s2, inp)
            return ((s1, ns2), o)
```

Multiplex

The `switch` combinator takes care to only update one of the component machines; in some other cases, we want to update both machines on every step and simply use the condition to select the output of one machine or the other to be the current output of the combined machine.

This is a very small variation on `Switch`, so we will just implement it as a subclass.

```
class Mux (Switch):
    def getNextValues(self, state, inp):
        (s1, s2) = state
        (ns1, o1) = self.m1.getNextValues(s1, inp)
        (ns2, o2) = self.m2.getNextValues(s2, inp)
        if self.condition(inp):
            return ((ns1, ns2), o1)
        else:
            return ((ns1, ns2), o2)
```

Exercise 4.11. What is the result of running these two machines

```
m1 = Switch(lambda inp: inp > 100,
            Accumulator(),
            Accumulator())
m2 = Mux(lambda inp: inp > 100,
         Accumulator(),
         Accumulator())
```

on the input

```
[2, 3, 4, 200, 300, 400, 1, 2, 3]
```

Explain why they are the same or are different.

If

Feel free to skip this example; it is only useful in fairly complicated contexts.

The *If* combinator. It takes a *condition*, which is a function from the input to *true* or *false*, and two machines. It evaluates the condition on the first input it receives. If the value is *true* then it executes the first machine forever more; if it is *false*, then it executes the second machine.

This can be straightforwardly implemented in Python; we will work through a slightly simplified version of our code below. We start by defining an initializer that remembers the conditions and the two constituent state machines.

```
class If (SM):
    startState = ('start', None)
    def __init__(self, condition, sm1, sm2):
        self.sm1 = sm1
        self.sm2 = sm2
        self.condition = condition
```

Because this machine does not have an input available at start time, it can not decide whether it is going to execute `sm1` or `sm2`. Ultimately, the state of the `If` machine will be a pair of values: the first will indicate which constituent machine we are running and the second will be the state of that machine. But, to start, we will be in the state `('start', None)`, which indicates that the decision about which machine to execute has not yet been made.

Now, when it is time to do a state update, we have an input. We destructure the state into its two parts, and check to see if the first component is `'start'`. If so, we have to make the decision about which machine to execute. The method `getFirstRealState` first calls the condition on the current input, to decide which machine to run; then it returns the pair of a symbol indicating which machine has been selected and the starting state of that machine. Once the first real state is determined, then that is used to compute a transition into an appropriate next state, based on the input.

If the machine was already in a non-start state, then it just updates the constituent state, using the already-selected constituent machine. Similarly, to generate an output, we have use the output function of the appropriate machine, with special handling of the `start` state.

```
    startState = ('start', None)

    def __init__(self, condition, sm1, sm2):
        self.sm1 = sm1
        self.sm2 = sm2
        self.condition = condition

    def getFirstRealState(self, inp):
        if self.condition(inp):
            return ('runningM1', self.sm1.startState)
        else:
            return ('runningM2', self.sm2.startState)

    def getNextValues(self, state, inp):
        (ifState, smState) = state
        if ifState == 'start':
            (ifState, smState) = self.getFirstRealState(inp)
```

```
        if ifState == 'runningM1':
            (newS, o) = self.sm1.getNextValues(smState, inp)
            return (('runningM1', newS), o)
        else:
            (newS, o) = self.sm2.getNextValues(smState, inp)
            return (('runningM2', newS), o)
```

4.3 Terminating state machines and sequential compositions

So far, all the machines we have discussed run forever; or, at least, until we quit giving them inputs. But in some cases, it is particularly useful to think of a process as consisting of a sequence of processes, one executing until termination, and then another one starting. For example, you might want to robot to clean first room A, **and then** clean room B; or, for it to search in an area **until** it finds a person **and then** sound an alarm.

Temporal combinations of machines form a new, different PCAP system for state machines. Our primitives will be state machines, as described above, but with one additional property: they will have a termination or *done* function, `d(s)`, which takes a state and returns *true* if the machine has finished execution and *false* otherwise.

Rather than defining a whole new class of state machines (though we could do that), we will just augment the `SM` class with a default method, which says that, by default, machines do not terminate.

```
    def done(self, state):
        return False
```

Then, in the definition of any subclass of `SM`, you are free to implement your own `done` method that will override this base one. The `done` method is used by state machine combinators that, for example, run one machine until it is done, and then switch to running another one.

Here is an example *terminating state machine* (TSM) that consumes a stream of numbers; its output is `None` on the first four steps and then on the fifth step, it generates the sum of the numbers it has seen as inputs, and then terminates. It looks just like the state machines we have seen before, with the addition of a `done` method. Its state consists of two numbers: the first is the number of times the machine has been updated and the second is the total input it has accumulated so far.

```
class ConsumeFiveValues(SM):
    startState = (0, 0)       # count, total

    def getNextValues(self, state, inp):
        (count, total) = state
        if count == 4:
            return ((count + 1, total + inp), total + inp)
        else:
            return ((count + 1, total + inp), None)

    def done(self, state):
        (count, total) = state
        return count == 5
```

Here is the result of running a simple example. We have modified the `transduce` method of `SM` to stop when the machine is done.

```
>>> c5 = ConsumeFiveValues()
>>> c5.transduce([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], verbose = True)
Start state: (0, 0)
In: 1 Out: None Next State: (1, 1)
In: 2 Out: None Next State: (2, 3)
In: 3 Out: None Next State: (3, 6)
In: 4 Out: None Next State: (4, 10)
In: 5 Out: 15 Next State: (5, 15)
[None, None, None, None, 15]
```

Now we can define a new set of combinators that operate on TSMs. Each of these combinators assumes that its constituent machines are terminating state machines, and are, themselves, terminating state machines. We have to respect certain rules about TSMs when we do this. In particular, it is not legal to call the `getNextValues` method on a TSM that says it is done. This may or may not cause an actual Python error, but it is never a sensible thing to do, and may result in meaningless answers.

4.3.1 Repeat

The simplest of the TSM combinators is one that takes a terminating state machine `sm` and repeats it `n` times. In the Python method below, we give a default value of `None` for `n`, so that if no value is passed in for `n` it will repeat forever.

```
class Repeat (SM):
    def __init__(self, sm, n = None):
        self.sm = sm
        self.startState = (0, self.sm.startState)
        self.n = n
```

The state of this machine will be the number of times the constituent machine has been executed to completion, together with the current state of the constituent machine. So, the starting state is a pair consisting of `0` and the starting state of the constituent machine.

Because we are going to, later, ask the constituent machine to generate an output, we are going to adopt a convention that the constituent machine is never left in a state that is done, unless the whole `Repeat` is itself done. If the constituent machine is done, then we will increment the counter for the number of times we have repeated it, and restart it. Just in case the constituent machine "wakes up" in a state that is done, we use a `while` loop here, instead of an `if`: we will keep restarting this machine until the count runs out. Why? Because we promised not to leave our constituent machine in a done state (so, for example, nobody asks for its output, when its done), unless the whole `repeat` machine is done as well.

```
    def advanceIfDone(self, counter, smState):
        while self.sm.done(smState) and not self.done((counter, smState)):
            counter = counter + 1
            smState = self.sm.startState
        return (counter, smState)
```

To get the next state, we start by getting the next state of the constituent machine; then, we check to see if the counter needs to be advanced and the machine restarted; the `advanceIfDone` method handles this situation and returns the appropriate next state. The output of the `Repeat` machine is just the output of the constituent machine. We just have to be sure to destructure the state of the overall machine and pass the right part of it into the constituent.

```
    def getNextValues(self, state, inp):
        (counter, smState) = state
        (smState, o) = self.sm.getNextValues(smState, inp)
        (counter, smState) = self.advanceIfDone(counter, smState)
        return ((counter, smState), o)
```

We know the whole `Repeat` is done if the counter is equal to `n`.

```
    def done(self, state):
        (counter, smState) = state
        return counter == self.n
```

Now, we can see some examples of `Repeat`. As a primitive, here is a silly little example TSM. It takes a character at initialization time. Its state is a Boolean, indicating whether it is done. It starts up in state `False` (not done). Then it makes its first transition into state `True` and stays there. Its output is always the character it was initialized with; it completely ignores its input.

```
class CharTSM (SM):
    startState = False
    def __init__(self, c):
        self.c = c
    def getNextValues(self, state, inp):
        return (True, self.c)
    def done(self, state):
        return state

>>> a = CharTSM('a')
>>> a.run(verbose = True)
Start state: False
In: None Out: a Next State: True
['a']
```

See that it terminates after one output. But, now, we can repeat it several times.

```
>>> a4 = sm.Repeat(a, 4)
>>> a4.run()
['a', 'a', 'a', 'a']
```

Exercise 4.12. Would it have made a difference if we had executed:

```
>>> sm.Repeat(CharTSM('a'), 4).run()
```

Exercise 4.13. Monty P. thinks that the following call

```
>>> sm.Repeat(ConsumeFiveValues(), 3).transduce(range(100))
```

will generate a sequence of 14 `None`s followed by the sum of the first 15 integers (starting at 0). R. Reticulatis disagrees. Who is right and why?

4.3.2 Sequence

Another useful thing to do with TSMs is to execute several different machines sequentially. That is, take a list of TSMs, run the first one until it is done, start the next one and run it until it is done, and so on. This machine is similar in style and structure to a `Repeat` TSM. Its state is a pair of values: an index that says which of the constituent machines is currently being executed, and the state of the current constituent.

Here is a Python class for creating a `Sequence` TSM. It takes as input a list of state machines; it remembers the machines and number of machines in the list.

```
class Sequence (SM):
    def __init__(self, smList):
        self.smList = smList
        self.startState = (0, self.smList[0].startState)
        self.n = len(smList)
```

The initial state of this machine is the value 0 (because we start by executing the 0th constituent machine on the list) and the initial state of that constituent machine.

The method for advancing is also similar that for `Repeat`. The only difference is that each time, we start the next machine in the list of machines, until we have finished executing the last one.

```
    def advanceIfDone(self, counter, smState):
        while self.smList[counter].done(smState) and counter + 1 < self.n:
            counter = counter + 1
            smState = self.smList[counter].startState
        return (counter, smState)
```

To get the next state, we ask the current constituent machine for its next state, and then, if it is done, advance the state to the next machine in the list that is not done when it wakes up. The output of the composite machine is just the output of the current constituent.

```
    def getNextValues(self, state, inp):
        (counter, smState) = state
        (smState, o) = self.smList[counter].getNextValues(smState, inp)
        (counter, smState) = self.advanceIfDone(counter, smState)
        return ((counter, smState), o)
```

We have constructed this machine so that it always advances past any constituent machine that is done; if, in fact, the current constituent machine is done, then the whole machine is also done.

```
    def done(self, state):
        (counter, smState) = state
        return self.smList[counter].done(smState)
```

We can make good use of the `CharTSM` to test our sequential combinator. First, we will try something simple:

```
>>> m = sm.Sequence([CharTSM('a'), CharTSM('b'), CharTSM('c')])
>>> m.run()
Start state: (0, False)
In: None Out: a Next State: (1, False)
In: None Out: b Next State: (2, False)
In: None Out: c Next State: (2, True)
['a', 'b', 'c']
```

Even in a test case, there is something unsatisfying about all that repetitive typing required to make each individual `CharTSM`. If we are repeating, we should abstract. So, we can write a function that takes a string as input, and returns a sequential TSM that will output that string. It uses a list comprehension to turn each character into a `CharTSM` that generates that character, and then uses that sequence to make a `Sequence`.

```
def makeTextSequenceTSM(str):
    return sm.Sequence([CharTSM(c) for c in str])
```

```
>>> m = makeTextSequenceTSM('Hello World')
>>> m.run(20, verbose = True)
Start state: (0, False)
In: None Out: H Next State: (1, False)
In: None Out: e Next State: (2, False)
In: None Out: l Next State: (3, False)
In: None Out: l Next State: (4, False)
In: None Out: o Next State: (5, False)
In: None Out:   Next State: (6, False)
In: None Out: W Next State: (7, False)
In: None Out: o Next State: (8, False)
In: None Out: r Next State: (9, False)
In: None Out: l Next State: (10, False)
In: None Out: d Next State: (10, True)
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
```

We can also see that sequencing interacts well with the `Repeat` combinator.

```
>>> m = sm.Repeat(makeTextSequenceTSM('abc'), 3)
>>> m.run(verbose = True)
Start state: (0, (0, False))
In: None Out: a Next State: (0, (1, False))
In: None Out: b Next State: (0, (2, False))
In: None Out: c Next State: (1, (0, False))
In: None Out: a Next State: (1, (1, False))
In: None Out: b Next State: (1, (2, False))
In: None Out: c Next State: (2, (0, False))
In: None Out: a Next State: (2, (1, False))
In: None Out: b Next State: (2, (2, False))
```

```
In: None Out: c Next State: (3, (0, False))
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

It is interesting to understand the state here. The first value is the number of times the constituent machine of the `Repeat` machine has finished executing; the second value is the index of the sequential machine into its list of machines, and the last Boolean is the state of the `CharTSM` that is being executed, which is an indicator for whether it is done or not.

4.3.3 RepeatUntil and Until

In order to use `Repeat`, we need to know in advance how many times we want to execute the constituent TSM. Just as in ordinary programming, we often want to terminate when a particular condition is met in the world. For this purpose, we can construct a new TSM combinator, called `RepeatUntil`. It takes, at initialization time, a condition, which is a function from an input to a Boolean, and a TSM. It runs the TSM to completion, then tests the condition on the input; if the condition is true, then the `RepeatUntil` terminates; if it is false, then it runs the TSM to completion again, tests the condition, etc.

Here is the Python code for implementing `RepeatUntil`. The state of this machine has two parts: a Boolean indicating whether the condition is true, and the state of the constituent machine.

```
class RepeatUntil (SM):
    def __init__(self, condition, sm):
        self.sm = sm
        self.condition = condition
        self.startState = (False, self.sm.startState)

    def getNextValues(self, state, inp):
        (condTrue, smState) = state
        (smState, o) = self.sm.getNextValues(smState, inp)
        condTrue = self.condition(inp)
        if self.sm.done(smState) and not condTrue:
            smState = self.sm.getStartState()
        return ((condTrue, smState), o)

    def done(self, state):
        (condTrue, smState) = state
        return self.sm.done(smState) and condTrue
```

One important thing to note is that, in the `RepeatUntil` TSM the condition is only evaluated when the constituent TSM is done. This is appropriate in some situations; but in other cases, we would like to terminate the execution of a TSM if a condition becomes true *at any single step of the machine*. We could easily implement something like this in any particular case, by defining a special-purpose TSM class that has a `done` method that tests the termination condition. But, because this structure is generally useful, we can define a general-purpose combinator, called `Until`. This combinator also takes a condition and a constituent machine. It simply executes the constituent machine, and terminates either when the condition becomes true, or when the constituent machine terminates. As before, the state includes the value of the condition on the last input and the state of the constituent machine.

Note that this machine will never execute the constituent machine more than once; it will either run it once to completion (if the condition never becomes true), or terminate it early.

Here are some examples of using `RepeatUntil` and `Until`. First, we run the `ConsumeFiveValues` machine until the input is greater than 10. Because it only tests the condition when `ConsumeFiveValues` is done, and the condition only becomes true on the 11th step, the `ConsumeFiveValues` machine is run to completion three times.

```
def greaterThan10 (x):
    return x > 10
>>> m = sm.RepeatUntil(greaterThan10, ConsumeFiveValues())
>>> m.transduce(range(20), verbose = True)
Start state: (0, 0)
In: 0 Out: None Next State: (1, 0)
In: 1 Out: None Next State: (2, 1)
In: 2 Out: None Next State: (3, 3)
In: 3 Out: None Next State: (4, 6)
In: 4 Out: 10 Next State: (0, 0)
In: 5 Out: None Next State: (1, 5)
In: 6 Out: None Next State: (2, 11)
In: 7 Out: None Next State: (3, 18)
In: 8 Out: None Next State: (4, 26)
In: 9 Out: 35 Next State: (0, 0)
In: 10 Out: None Next State: (1, 10)
In: 11 Out: None Next State: (2, 21)
In: 12 Out: None Next State: (3, 33)
In: 13 Out: None Next State: (4, 46)
In: 14 Out: 60 Next State: (5, 60)
[None, None, None, None, 10, None, None, None, None, 35, None, None, None, None, 60]
```

If we do `Until` on the basic `ConsumeFiveValues` machine, then it just runs `ConsumeFiveValues` until it terminates normally, because the condition never becomes true during this time.

```
>>> m = sm.Until(greaterThan10, ConsumeFiveValues())
>>> m.transduce(range(20), verbose = True)
Start state: (False, (0, 0))
In: 0 Out: None Next State: (False, (1, 0))
In: 1 Out: None Next State: (False, (2, 1))
In: 2 Out: None Next State: (False, (3, 3))
In: 3 Out: None Next State: (False, (4, 6))
In: 4 Out: 10 Next State: (False, (5, 10))
[None, None, None, None, 10]
```

However, if we change the termination condition, the execution will be terminated early. Note that we can use a `lambda` expression directly as an argument; sometimes this is actually clearer than defining a function with `def`, but it is fine to do it either way.

```
>>> m = sm.Until(lambda x: x == 2, ConsumeFiveValues())
>>> m.transduce(range(20), verbose = True)
Start state: (False, (0, 0))
In: 0 Out: None Next State: (False, (1, 0))
In: 1 Out: None Next State: (False, (2, 1))
In: 2 Out: None Next State: (True, (3, 3))
[None, None, None]
```

If we actually want to keep repeating `ConsumeFiveValues()` until the condition becomes true, we can combine `Until` with `Repeat`. Now, we see that it executes the constituent machine multiple times, but terminates as soon as the condition is satisfied.

```
>>> m = sm.Until(greaterThan10, sm.Repeat(ConsumeFiveValues()))
>>> m.transduce(range(20), verbose = True)
Start state: (False, (0, (0, 0)))
In: 0 Out: None Next State: (False, (0, (1, 0)))
In: 1 Out: None Next State: (False, (0, (2, 1)))
In: 2 Out: None Next State: (False, (0, (3, 3)))
In: 3 Out: None Next State: (False, (0, (4, 6)))
In: 4 Out: 10 Next State: (False, (1, (0, 0)))
In: 5 Out: None Next State: (False, (1, (1, 5)))
In: 6 Out: None Next State: (False, (1, (2, 11)))
In: 7 Out: None Next State: (False, (1, (3, 18)))
In: 8 Out: None Next State: (False, (1, (4, 26)))
In: 9 Out: 35 Next State: (False, (2, (0, 0)))
In: 10 Out: None Next State: (False, (2, (1, 10)))
In: 11 Out: None Next State: (True, (2, (2, 21)))
[None, None, None, None, 10, None, None, None, None, 35, None, None]
```

4.4 Using a state machine to control the robot

This section gives an overview of how to control the robot with a state machine. For a much more detailed description, see the *Infrastructure Guide*, which documents the `io` and `util` modules in detail. The `io` module provides procedures and methods for interacting with the robot; the `util` module provides procedures and methods for doing computations that are generally useful (manipulating angles, dealing with coordinate frames, etc.)

We can implement a robot controller as a state machine whose inputs are instances of class `io.SensorInput`, and whose outputs are instances of class `io.Action`.

Here is Python code for a brain that is controlled by the most basic of state machines. This machine always emits the default action, `io.Action()`, which sets all of the output values to zero. When the brain is set up, we create a "behavior", which is a name we will use for a state machine that transduces a stream of `io.SensorInputs` to a stream of `io.Actions`. Finally, we ask the behavior to start.

Then, all we do in the `step` method of the robot is:

- Read the sensors, by calling `io.SensorInput()` to get an instance that contains sonar and odometry readings;
- Feed that sensor input to the brain state machine, by calling its `step` method with that as input; and
- Take the `io.Action` that is generated by the brain as output, and call its `execute` method, which causes it to actually send motor commands to the robot.

You can set the `verbose` flag to `True` if you want to see a lot of output on each step for debugging.

Inside a Soar brain, we have access to an object `robot`, which persists during the entire execution of the brain, and gives us a place to store important objects (like the state machine that will be doing all the work).

```
import sm
import io

class StopSM(sm.SM):
    def getNextValues(self, state, inp):
        return (None, io.Action())

def setup():
    robot.behavior = StopSM()
    robot.behavior.start()

def step():
    robot.behavior.step(io.SensorInput(), verbose = False).execute()
```

In the following sections we will develop two simple machines for controlling a robot to move a fixed distance or turn through a fixed angle. Then we will put them together and explore why it can be useful to have the starting state of a machine depend on the input.

4.4.1 Rotate

Imagine that we want the robot to rotate a fixed angle, say 90 degrees, to the left of where it is when it starts to run a behavior. We can use the robot's odometry to measure approximately where it is, in an arbitrary coordinate frame; but to know how much it has moved since we started, we have to store some information in the state.

Here is a class that defines a `Rotate` state machine. It takes, at initialization time, a desired change in heading.

```
class RotateTSM (SM):
    rotationalGain = 3.0
    angleEpsilon = 0.01
    startState = 'start'

    def __init__(self, headingDelta):
        self.headingDelta = headingDelta
```

When it is time to start this machine, we would like to look at the robot's current heading (`theta`), add the desired change in heading, and store the result in our state as the desired heading. Then, in order to test whether the behavior is done, we want to see whether the current heading is close enough to the desired heading. Because the `done` method does not have access to the input of the machine (it is a property only of states), we need to include the current `theta` in the state. So, the state of the machine is `(thetaDesired, thetaLast)`.

Thus, the `getNextValues` method looks at the state; if it is the special symbol `'start'`, it means that the machine has not previously had a chance to observe the input and see what its current heading is, so it computes the desired heading (by adding the desired change to the current heading, and then calling a utility procedure to be sure the resulting angle is between plus and minus π), and returns it and the current heading. Otherwise, we keep the `thetaDesired` component of the state, and just get a new value of `theta` out of the input. We generate an action with a

rotational velocity that will rotate toward the desired heading with velocity proportional to the magnitude of the angular error.

```
    def getNextValues(self, state, inp):
        currentTheta = inp.odometry.theta
        if state == 'start':
            thetaDesired = \
               util.fixAnglePlusMinusPi(currentTheta + self.headingDelta)
        else:
            (thetaDesired, thetaLast) = state
        newState = (thetaDesired, currentTheta)
        action = io.Action(rvel = self.rotationalGain * \
                    util.fixAnglePlusMinusPi(thetaDesired - currentTheta))

        return (newState, action)
```

Finally, we have to say which states are done. Clearly, the `'start'` state is not done; but we are done if the most recent `theta` from the odometry is within some tolerance, `self.angleEpsilon`, of the desired heading.

```
    def done(self, state):
        if state == 'start':
            return False
        else:
            (thetaDesired, thetaLast) = state
            return util.nearAngle(thetaDesired, thetaLast, self.angleEpsilon)
```

Exercise 4.14. Change this machine so that it rotates *through* an angle, so you could give it 2 pi or minus 2 pi to have it rotate all the way around.

4.4.2 Forward

Moving the robot forward a fixed distance is similar. In this case, we remember the robot's `x` and y coordinates when it starts, and drive straight forward until the distance between the initial position and the current position is close to the desired distance. The state of the machine is the robot's starting position and its current position.

```
class ForwardTSM (SM):
    forwardGain = 1.0
    distTargetEpsilon = 0.01
    startState = 'start'

    def __init__(self, delta):
        self.deltaDesired = delta

    def getNextValues(self, state, inp):
        currentPos = inp.odometry.point()
        if state == 'start':
            print "Starting forward", self.deltaDesired
```

```
            startPos = currentPos
        else:
            (startPos, lastPos) = state
        newState = (startPos, currentPos)
        error = self.deltaDesired - startPos.distance(currentPos)
        action = io.Action(fvel = self.forwardGain * error)
        return (newState, action)

    def done(self, state):
        if state == 'start':
            return False
        else:
            (startPos, lastPos) = state
            return util.within(startPos.distance(lastPos),
                               self.deltaDesired,
                               self.distTargetEpsilon)
```

4.4.3 Square Spiral

Imagine we would like to have the robot drive in a square spiral, similar to the one shown in **figure 4.11**. One way to approach this problem is to make a "low-level" machine that can consume a goal point and the sensor input and drive (in the absence of obstacles) to the goal point; and then make a "high-level" machine that will keep track of where we are in the figure and feed goal points to the low-level machine.

4.4.3.1 XYDriver

Here is a class that describes a machine that takes as input a series of pairs of goal points (expressed in the robot's odometry frame) and sensor input structures. It generates as output a series of actions. This machine is very nearly a pure function machine, which has the following basic control structure:

- If the robot is headed toward the goal point, move forward.
- If it is not headed toward the goal point, rotate toward the goal point.

This decision is made on every step, and results in a robust ability to drive toward a point in two-dimensional space.

For many uses, this machine does not need any state. But the modularity is nicer, in some cases, if it has a meaningful `done` method, which depends only on the state. So, we will let the state of this machine be whether it is done or not. It needs several constants to govern rotational and forward speeds, and tolerances for deciding whether it is pointed close enough toward the target and whether it has arrived close enough to the target.

```
class XYDriver(SM):
    forwardGain = 2.0
    rotationGain = 2.0
    angleEps = 0.05
    distEps = 0.02
    startState = False
```

The `getNextValues` method embodies the control structure described above.

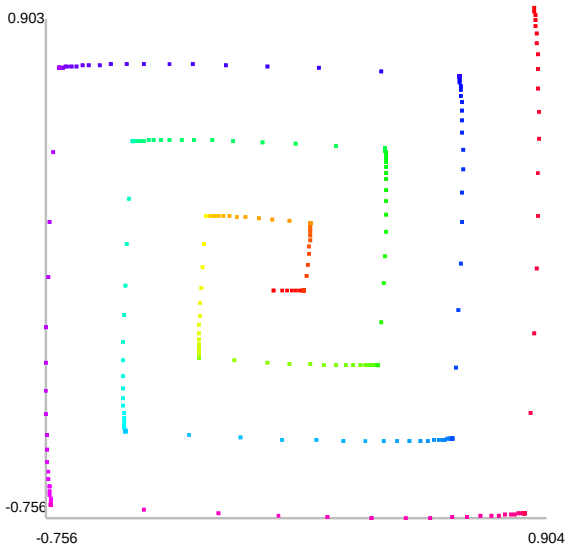


Figure 4.11 Square spiral path of the robot using the methods in this section.

```
    def getNextValues(self, state, inp):
        (goalPoint, sensors) = inp
        robotPose = sensors.odometry
        robotPoint = robotPose.point()
        robotTheta = robotPose.theta

        if goalPoint == None:
            return (True, io.Action())

        headingTheta = robotPoint.angleTo(goalPoint)
        if util.nearAngle(robotTheta, headingTheta, self.angleEps):
            # Pointing in the right direction, so move forward
            r = robotPoint.distance(goalPoint)
            if r < self.distEps:
                # We're there
                return (True, io.Action())
            else:
                return (False, io.Action(fvel = r * self.forwardGain))
        else:
            # Rotate to point toward goal
            headingError = util.fixAnglePlusMinusPi(\
```

```
                                                headingTheta - robotTheta)
            return (False, io.Action(rvel = headingError * self.rotationGain))
```

The state of the machine is just a boolean indicating whether we are done.

```
    def done(self, state):
        return state
```

4.4.3.2 Cascade approach

We make a spiral by building a machine that takes `SensorInput` objects as input and generates pairs of subgoals and sensorinputs; such a machine can be cascaded with `XYDriver` to generate a spiral.

Our implementation of this is a class called `SpyroGyra`. It takes the increment (amount that each new side is larger than the previous) at initialization time. Its state consists of three components:

- `direction`: one of 'north', 'south', 'east', or 'west', indicating which way the robot is traveling
- `length`: length in meters of the current line segment being followed
- `subGoal`: the point in the robot's odometry frame that defines the end of the current line segment

It requires a tolerance to decide when the current subgoal point has been reached.

```
class SpyroGyra(SM):
    distEps = 0.02
    def __init__(self, incr):
        self.incr = incr
        self.startState = ('south', 0, None)
```

If the robot is close enough to the subgoal point, then it is time to change the state. We increment the side length, pick the next direction (counter clockwise around the cardinal compass directions), and compute the next subgoal point. The output is just the subgoal and the sensor input, which is what the driver needs.

```
    def getNextValues(self, state, inp):
        (direction, length, subGoal) = state
        robotPose = inp.odometry
        robotPoint = robotPose.point()
        if subGoal == None:
            subGoal = robotPoint
        if robotPoint.isNear(subGoal, self.distEps):
            # Time to change state
            length = length + self.incr
            if direction == 'east':
                direction = 'north'
                subGoal.y += length
            elif direction == 'north':
                direction = 'west'
                subGoal.x -= length
            elif direction == 'west':
                direction = 'south'
                subGoal.y -= length
```

```
            else:  # south
                direction = 'east'
                subGoal.x += length
            print 'new:', direction, length, subGoal
        return ((direction, length, subGoal),
                (subGoal, inp))
```

Finally, to make the spiral, we just cascade these two machines together.

```
def spiroFlow(incr):
    return sm.Cascade(SpyroGyra(incr), XYDriver())
```

Exercise 4.15. What explains the rounded sides of the path in **figure 4.11**?

4.5 Conclusion

State machines

State machines are such a general formalism, that a huge class of discrete-time systems can be described as state machines. The system of defining primitive machines and combinations gives us one discipline for describing complex systems. It will turn out that there are some systems that are conveniently defined using this discipline, but that for other kinds of systems, other disciplines would be more natural. As you encounter complex engineering problems, your job is to find the PCAP system that is appropriate for them, and if one does not exist already, invent one.

State machines are such a general class of systems that although it is a useful framework for implementing systems, we cannot generally analyze the behavior of state machines. That is, we can't make much in the way of generic predictions about their future behavior, except by running them to see what will happen.

In the next module, we will look at a restricted class of state machines, whose state is representable as a bounded history of their previous states and previous inputs, and whose output is a linear function of those states and inputs. This is a *much* smaller class of systems than all state machines, but it is nonetheless very powerful. The important lesson will be that restricting the form of the models we are using will allow us to make stronger claims about their behavior.

Knuth on Elevator Controllers

Donald E. Knuth is a computer scientist who is famous for, among other things, his series of textbooks (as well as for TEX, the typesetting system we use to make all of our handouts), and a variety of other contributions to theoretical computer science.

"It is perhaps significant to note that although the author had used the elevator system for years and thought he knew it well, it wasn't until he attempted to write this section that he realized there were quite a few facts about the elevator's system of choosing directions that he did not know. He went back to experiment with the elevator six separate times, each time believing he

had finally achieved a complete understanding of its *modus operandi*. (Now he is reluctant to ride it for fear some new facet of its operation will appear, contradicting the algorithms given.) We often fail to realize how little we know about a thing until we attempt to simulate it on a computer."

The Art of Computer Programming, Donald E., Knuth, Vol 1. page 295. On the elevator system in the Mathematics Building at Cal Tech. First published in 1968

4.6 Examples

4.6.1 Practice problem: Things

Consider the following program

```
def thing(inputList):
    output = []
    i = 0
    for x in range(3):
        y = 0
        while y < 100 and i < len(inputList):
            y = y + inputList[i]
            output.append(y)
            i = i + 1
    return output
```

A. What is the value of

```
thing([1, 2, 3, 100, 4, 9, 500, 51, -2, 57, 103, 1, 1, 1, 1, -10, 207, 3, 1])
```

```
[1, 3, 6, 106, 4, 13, 513, 51, 49, 106]
```

It's important to understand the loop structure of the Python program: It goes through (at most) three times, and adds up the elements of the input list, generating a partial sum as output on each step, and terminating the inner loop when the sum becomes greater than 100.

B. Write a single state machine class `MySM` such that `MySM().transduce(inputList)` gives the same result as `thing(inputList)`, if `inputList` is a list of numbers. Remember to include a `done` method, that will cause it to terminate at the same time as `thing`.

```
class MySM(sm.SM):
    startState = (0,0)
    def getNextValues(self, state, inp):
        (x, y) = state
        y += inp
        if y >= 100:
            return ((x + 1, 0), y)
        return ((x, y), y)
    def done(self, state):
        (x, y) = state
        return x >= 3
```

The most important step, conceptually, is deciding what the state of the machine will be. Looking at the original Python program, we can see that we had to keep track of how many times we had completed the outer loop, and then what the current partial sum was of the inner loop.
The `getNextValues` method first increments the partial sum by the input value, and then checks to see whether it's time to reset. If so, it increments the 'loop counter' (`x`) component of the state and resets the partial sum to 0. It's important to remember that the output of the `getNextValues` method is a pair, containing the next state and the output.
The `done` method just checks to see whether we have finished three whole iterations.

C. Recall the definition of `sm.Repeat(m, n)`: Given a terminating state machine `m`, it returns a new terminating state machine that will execute the machine `m` to completion `n` times, and then terminate.

Use `sm.Repeat` and a very simple state machine that you define to create a new state machine `MyNewSM`, such that `MyNewSM` is equivalent to an instance of `MySM`.

```
class Sum(sm.SM):
    startState = 0
    def getNextValues(self, state, inp):
        return (state + inp, state + inp)
    def done(self, state):
        return state > 100

myNewSM = sm.Repeat(Sum(), 3)
```

4.6.2 Practice problem: Inheritance and State Machines

Recall that we have defined a Python class `sm.SM` to represent state machines. Here we consider a special type of state machine, whose states are always integers that start at 0 and increment by 1 on each transition. We can represent this new type of state machines as a Python subclass of `sm.SM` called `CountingStateMachine`.

We wish to use the `CountingStateMachine` class to define new subclasses that each provide a single new method `getOutput(self, state, inp)` which returns *just the output* for that state

and input; the `CountingStateMachine` will take care of managing and incrementing the state, so its subclasses don't have to worry about it.

Here is an example of a subclass of `CountingStateMachine`.

```
class CountMod5(CountingStateMachine):
    def getOutput(self, state, inp):
        return state % 5
```

Instances of `CountMod5` generate output sequences of the form `0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, ...`.

Part a. Define the `CountingStateMachine` class. Since `CountingStateMachine` is a subclass of `sm.SM`, you will have to provide definitions of the `startState` instance variable and `getNextValues` method, just as we have done for other state machines. You can assume that every subclass of `CountingStateMachine` will provide an appropriate `getOutput` method.

```
class CountingStateMachine(sm.SM):
    def __init__(self):
        self.startState = 0
    def getNextState(self, state, inp):
        return(state + 1, self.getOutput(state, inp))
```

Part b. Define a subclass of `CountingStateMachine` called `AlternateZeros`. Instances of `AlternateZeros` should be state machines for which, on even steps, the output is the same as the input, and on odd steps, the output is 0. That is, given inputs, $i_0, i_1, i_2, i_3, \ldots$, they generate outputs, $i_0, 0, i_2, 0, \ldots$.

```
class AlternateZeros(CountingStateMachine):
    def getOutput(self, state, inp):
        if not state % 2:
            return inp
        return 0
```

MIT OpenCourseWare
http://ocw.mit.edu

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.