MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

**Course notes for Week 3**

# Basic Object-Oriented Programming and State Machines

## 1 Introduction

Here is our familiar framework for thinking about primitives and means of combination, abstraction, and capturing common patterns. In this lecture, we'll add ideas for abstracting and capturing common patterns in data structures, and ultimately achieving even greater abstraction and modularity by abstracting over data structures combined with the methods that operate on them.

| | Procedures | Data |
|---|---|---|
| Primitives | `+`, `*`, `==` | numbers, strings |
| Means of combination | `if`, `while`, `f(g(x))` | lists, dictionaries, objects |
| Means of abstraction | `def` | abstract data types, classes |
| Means of capturing common patterns | higher-order procedures | generic functions, classes |

Let's start with an example. We've looked at three different implementations of sets:

- as lists;
- as functions; and
- as whatever Python uses in its built-in data type.

What is in common between these implementations is that they included the same set of abstract operations, but with different details about how the data was stored and how the various operations were computed. The good thing about thinking about sets in the abstract is that we can build something more complex (such as non-deterministic behaviors) on top of an implementation of set operations, without caring too much about how those set operations are ultimately implemented. We'll call such a thing an *abstract data type*. It's abstract, in that you can use it without knowing the implementation details. We used Python sets by reading their documentation, but without having any idea how they're actually implemented in Python.

Using ADTs is good because it

1. allows us to concentrate on the high-level aspects of the job we're trying to do; and
2. allows us or someone else to change the underlying implementation of sets (perhaps to make it faster or more memory-efficient) without having to change any of the code that we've implemented that uses sets.

An *abstract data type* (ADT) is a collection of related operations that can be performed on a set of data. The documentation of an ADT specifies its *contract*: a promised set of relationships among the inputs and outputs of the functions involved. For instance, part of the contract of a `set` ADT would be that the `union` of two sets, `s1` and `s2`, would `contain` all and only elements

that are `contained` in `s1` or `s2`. ADTs are often used to describe generic structures like sets or dictionaries; they can also be used to describe more particular structures like bank accounts or telephone-directory entries.

A set ADT might include the following operations:

$$\begin{aligned}
\texttt{makeSet} &: \mathit{list} \rightarrow \mathit{set} \\
\texttt{contains} &: (\mathit{item}, \mathit{set}) \rightarrow \mathit{Boolean} \\
\texttt{isSubset} &: (\mathit{set}, \mathit{set}) \rightarrow \mathit{Boolean} \\
\texttt{intersection} &: (\mathit{set}, \mathit{set}) \rightarrow \mathit{set} \\
\texttt{union} &: (\mathit{set}, \mathit{set}) \rightarrow \mathit{set} \\
\texttt{len} &: \mathit{set} \rightarrow \mathit{int} \\
\texttt{contents} &: \mathit{set} \rightarrow \mathit{list}
\end{aligned}$$

A bank-account ADT might include the following operations:

$$\begin{aligned}
\texttt{makeBankAccount} &: (\mathit{float}, \mathit{float}, \mathit{string}, \mathit{string}) \rightarrow \mathit{account} \\
\texttt{balance} &: \mathit{account} \rightarrow \mathit{number} \\
\texttt{owner} &: \mathit{account} \rightarrow \mathit{string} \\
\texttt{creditLimit} &: \mathit{account} \rightarrow \mathit{number} \\
\texttt{deposit} &: \mathit{account} \rightarrow \texttt{None}
\end{aligned}$$

Operations like `makeSet` are often called *constructors*: they make a new instance of the ADT. Operatons like `balance` and `contents` are called *selectors*: they select out and return some piece of the data associated with the ADT. The `deposit` operation is special: it doesn't return a value, but it does *change* (sometimes we say *side-effect*) values stored inside the object (in this case, it will change the balance of the bank account, but we don't know exactly how it will do it, because we don't know how the balance is represented internally).

Different computer languages offer different degrees of support for using ADTs. Even in the most primitive language, you can write your code in a modular way, to try to preserve abstraction. But modern object-oriented languages offer built-in facilities to make this style easy to use.

# 2 Execution Model

*This is repeated from section 3 of week 1 notes as a review; we will rely on it in the next section.*

In order to really understand Python's object-oriented programming facilities, we have to start by understanding how it uses *environments* to store information, during and between procedure calls.

## 2.1 Environments

The first thing we have to understand is the idea of *binding environments* (we'll often just call them *environments*; they are also called *namespaces* and *scopes*). An environment is a stored mapping between names and entities in a program. The entities can be all kinds of things: numbers, strings,

lists, procedures, objects, etc. In Python, the names are strings and environments are actually dictionaries, which we've already experimented with.

Environments are used to determine values associated with the names in your program. There are two operations you can do to an environment: add a binding, and look up a name. You do these things implcitly all the time in programs you write. Consider a file containing

```
a = 5
print a
```

The first statement, `a = 5`, creates a binding of the name `a` to the value 5. The second statement prints something. First, to decide that it needs to print, it looks up `print` and finds an associated built-in procedure. Then, to decide what to print, it evaluates the associated expression. In this case, the expression is a name, and it is evaluated by looking up the name in the environment and returning the value it is bound to (or generating an error if the name is not bound).

In Python, there are environments associated with each module (file) and one called `__builtin__` that has all the procedures that are built into Python. If you do

```
>>> import __builtin__
>>> dir(__builtin__)
```

you'll see a long list of names of things (like `'sum'`), which are built into Python, and whose names are defined in the builtin module. You don't have to type `import __builtin__`; as we'll see below, you always get access to those bindings. You can try importing `math` and looking to see what names are bound there.

Another operation that creates a new environment is a function call. In this example,

```
def f(x):
    print x
>>> f(7)
```

when the function `f` is called with argument 7, a new *local* environment is constructed, in which the name `x` is bound to the value 7.

So, what happens when Python actually tries to evaluate `print x`? It takes the symbol `x` and has to try to figure out what it means. It starts by looking in the *local* environment, which is the one defined by the innermost function call. So, in the case above, it would look it up and find the value 7 and return it.

Now, consider this case:

```
def f(a):
    def g(x):
        print x, a
        return x + a
    return g(7)
>>> f(6)
```

What happens when it's time to evaluate `print x, a`? First, we have to think of the environments. The first call, `f(6)` establishes an environment in which `a` is bound to 6. Then the call `g(7)` establishes another environment in which `x` is bound to 7. So, when needs to print `x` it looks in the local environment and finds that it has value 7. Now, it looks for `a`, but doesn't find it in the local environment. So, it looks to see if it has it available in an *enclosing environment*; an environment

that was enclosing this procedure *when it was defined*. In this case, the environment associated with the call to `f` is enclosing, and it has a binding for `a`, so it prints `6` for `a`. So, what does `f(6)` return? 13.

You can think of every environment actually consisting of two things: (1) a dictionary that maps names to values and (2) an enclosing environment.

If you write a file that looks like this:

```
b = 3
def f(a):
    def g(x):
        print x, a, b
        return x + a + b
    return g(7)

>>> f(6)
7 6 3
16
```

When you evaluate `b`, it won't be able to find it in the local environment, or in an enclosing environment created by another procedure definition. So, it will look in the *global environment*. The name global is a little bit misleading; it means the environment associated with the file. So, it will find a binding for `b` there, and use the value 2.

One way to remember how Python looks up names is the LEGB rule: it looks, in order, in the *Local*, then the *Enclosing*, then the *Global*, then the *Builtin* environments, to find a value for a name. As soon as it succeeds in finding a binding, it returns that one.

Bindings are also created when you execute an `import` statement. If you execute

```
import math
```

Then the `math` module is loaded and the name `math` is bound, in the current environment, to the math module. No other names are added to the current environment, and if you want to refer to internal names in that module, you have to qualify them, as in `math.sqrt`. If you execute

```
from math import sqrt
```

then, the `math` module is loaded, and the name `sqrt` is bound, in the current environment, to whatever the the name `sqrt` is bound to in the `math` module. But note that if you do this, the name `math` isn't bound to anything, and you can't access any other procedures in the `math` module.

Another thing that creates a binding is the definition of a function: that creates a binding of the function's name, in the environment in which it was created, to the actual function.

Finally, bindings are created by `for` statements and list comprehensions; so, for example,

```
for element in listOfThings:
    print element
```

creates successive bindings for the name `element` to the elements in `listOfThings`.

Figure 2 shows the state of the environments when the print statement in the example code shown in figure 1 is executed. Names are first looked for in the local scope, then its enclosing scope (if there were more than one enclosing scope, it would continue looking up the chain of enclosing scopes), and then in the global scope.

```
a = 1
b = 2
c = 3
def d(a):
    c = 5
    from math import pi
    def e(x):
        for i in range(b):
            print a, b, c, x, i, pi, d, e
    e(100)

>>> d(1000)
1000 2 5 100 0 3.14159265359 <function d at 0x4e3f0> <function e at 0x4ecf0>
1000 2 5 100 1 3.14159265359 <function d at 0x4e3f0> <function e at 0x4ecf0>
```

Figure 1: Code and what it prints

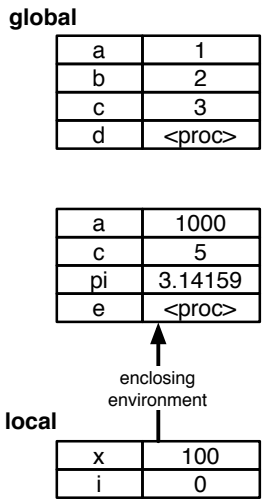


Figure 2: Binding environments that were in place when the first print statement in figure 1 was executed.

**Local versus global references** There is an important subtlety in the way names are handled in handled in the environment created by a procedure call. When a name that is not bound in the local environment is referred to, then it is looked up in the enclosing, global, and built-in environments. So, as we've seen, it is fine to have

```
a = 2
def b():
    print a
```

When a name is assigned in a local environment, a new binding is created for it. So, it is fine to have

```
a = 2
def b():
    a = 3
    c = 4
    print a, c
```

Both assignments cause new bindings to be made in the local environment, and it is those bindings that are used to supply values in the print statement.

But here is a code fragment that causes trouble:

```
a = 3
def b():
    a = a + 1
    print a
```

It seems completely reasonable, and you might expect it to print 4. But, instead, it generates an error if we try to call `b`. What's going on?? It all has to do with when Python decides to add a binding to the local environment. When it sees this procedure definition, it sees that the name `a` is assigned to, and so, at the very beginning, it puts an entry for `a` in the local environment. Now, when it's time to execute the statement

```
a = a + 1
```

it starts by evaluating the expression on the right hand side: `a + 1`. When it tries to look up the name `a` in the local environment, it finds that it has been added to the environment, but hasn't yet had a value specified. So it generates an error.

We can still write code to do what we intended (write a procedure that increments a number named in the global environment), by using the `global` declaration:

```
a = 3
def b():
    global a
    a = a + 1
    print a
>>> b()
4
>>> b()
5
```

The statement `global a` asks that a new binding for `a` *not* be made in the local environment. Now, all references to `a` are to the binding in the global environment, and it works as expected. In Python, we can only make assignments to names in the local scope or in the global scope, but not to names in an enclosing scope. So, for example,

```
def outer():
    def inner():
        a = a + 1
    a = 0
    inner()
```

In this example, we get an error, because Python has made a new binding for `a` in the environment for the call to `inner`. We'd really like for `inner` to be able to see and modify the `a` that belongs to the environment for `outer`, but there's no way to arrange this.

# 3 Object-oriented programming

Object-oriented programming (OOP, to its friends) helps us with a lot of aspects of abstraction in programming; this week, we'll just look at how it helps with constructing ADTs. The OOP facilities in Python are quite lightweight and flexible.

A *class* is a collection of procedures (and sometimes data) attached to names in an environment, which is meant to represent a generic *type* of object, like a set or a bank account. It typically contains the procedure definitions that allow it to fulfill the specifications of an ADT.

An *object* is also a collection of procedures and data, attached to names in an environment, but it is intended to represent a particular instance of a class, such as the set containing 1 and 2, or Leslie's bank account. When we want to make a particular bank account, we can create an object that is an *instance* of the bank account class. Instances (objects) are environments whose "enclosing" environment is the class of which they are instances. So, an object has access to all of the values defined in its class, but it can be specialized for the the particular instance it is intended to represent.

Here's a *very* simple class, and a little demonstration of how it can be used.

```
class SimpleThing:
    a = 6

>>> x = SimpleThing()
>>> x
<__main__.SimpleThing instance at 0x85468>
>>> x.a
6
>>> y = SimpleThing()
>>> y.a
6
>>> y.a = 10
>>> y.a
10
>>> x.a
6
>>> SimpleThing.a
6
```

To define a class, you start with a `class` statement, and then a set of indented assignments and definitions. Each assignment to a new name makes a new variable binding within the class. Whenever you define a class, you get a *constructor*, which will make a new instance of the class.

In our example above, the constructor is `SimpleThing()`.[1] When we make a new instance of `SimpleThing`, we get an object. We can look at the value of attribute `a` of the object `x` by writing `x.a`. An object is an environment, and this is the syntax for looking up name `a` in environment `x`. There is no binding for `a` in `x`, so it looks in its enclosing environment, which is the environment of the class `SimpleThing`, and finds a binding for `a` there, and returns 6.

If we make another instance, `y`, of the class, and assign a value to its attribute `a`, that makes a fresh binding for `a` in `y`, and doesn't change the original binding of `SimpleThing.a`.

Some of you may have experience with Java, which is much more rigid about what you can do with objects than Python is. In Python, you can add attributes to objects on the fly. So, we could continue the previous example with:

```
>>> x.newAttribute = "hi"
```

and there would be no problem.

Here's another example to illustrate the definition and use of *methods*, which are procedures whose first argument is the object, and that can be accessed via the object.

```
class Square:
    dim = 6

    def getArea (self):
        return self.dim * self.dim

    def setArea (self, area):
        self.dim = area**0.5
```

This class is meant to represent a square. Squares need to store, or remember, their dimension, so we make an attribute for it, and assign it initially to be 6 (we'll be smarter about this in the next example). Now, we define a method `getArea` that is intended to return the area of the square. There are a couple of interesting things going on here.

Like all methods, `getArea` has an argument, `self`, which will stand for the object that this method is supposed to operate on.[2] Now, remembering that objects are environments, the way we can find the dimension of the square is by looking up the name `dim` in this square's environment, which was passed into this method as the object `self`.

We define another method, `setArea`, which will set the area of the square to a given value. In order to change the square's area, we have to compute a new dimension and store it in the `dim` attribute of the square object.

Now, we can experiment with instances of class `Square`.

```
>>> s = Square()
>>> s.getArea()
36
>>> Square.getArea(s)
```

[1] A note on style. It is useful to adopt some conventions for naming things, just to help your programs be more readable. We've used the convention that variables and procedure names start with lower case letters and that class names start with upper case letters. And we try to be consistent about using something called "camel caps" for writing compound words, which is to write a compound name with the successiveWordsCapitalized. An alternative is_to_use_underscores.

[2] The argument doesn't have to be named `self`, but this is a standard convention.

```
36
>>> s.dim
6
>>> s.setArea(100)
>>> s.dim
10.0
```

We make a new instance using the constructor, and ask for its area by writing `s.getArea()`. This is the standard syntax for calling a method of an object, but it's a little bit confusing because its argument list doesn't really seem to match up with the method's definition (which had one argument). A style that is less convenient, but perhaps easier to understand, is this: `Square.getArea(s)`. Remembering that a class is also an environment, with a bunch of definitions in it, we can see that it starts with the class environment `Square` and looks up the name `getArea`. This gives us a procedure of one argument, as we defined it, and then we call that procedure on the object `s`. It is fine to use this syntax, if you prefer, but you'll probably find the `s.getArea()` version to be more convenient. One way to think of it is as asking the object `s` to perform its `getArea` method on itself.

Here's a version of the square class that has a special `initialization` method.

```
class Square1:
    def __init__(self, initialDim):
        self.dim = initialDim

    def getArea (self):
        return self.dim * self.dim

    def setArea (self, area):
        self.dim = area**0.5

    def __str__(self):
        return "Square of dim " + str(self.dim)
```

Whenever the constructor for a class is called, Python looks to see if there is a method called `__init__` and calls it, with the newly constructed object as the first argument and the rest of the arguments from the constructor added on. So, we could make two new squares by doing

```
>>> s1 = Square1(10)
>>> s1.dim
10
>>> s1.getArea()
100
>>> s2 = Square1(100)
>>> s2.getArea()
10000
>>> print s1
Square of dim 10
```

Now, instead of having an attribute `dim` defined at the class level, we create it inside the initialization method. The initialization method is a method like any other; it just has a special name. Note that it's crucial that we write `self.dim = initialDim` here, and not just `dim = initialDim`. All the usual rules about environments apply here. If we wrote `dim = initialDim`, it would make a variable in the method's local environment, called `dim`, but that variable would only exist during

the execution of the `__init__` procedure. To make a new attribute of the object, it needs to be stored in the environment associated with the object, which we access through `self`.

Our class `Square1` has another special method, `__str__`. This is the method that Python calls on an object whenever it needs to find a printable name for it. By default, it prints something like `<__main__.Square1 instance at 0x830a8>`, but for debugging, that can be pretty uninformative. By defining a special version of that method for our class of objects, we can make it so when we try to print an instance of our class we get something like `Square of dim 10` instead. We've used the Python procedure `str` to get a string representation of the value of `self.dim`. You can call `str` on any entity in Python, and it will give you a more or less useful string representation of it. Of course, now, for `s1`, it would return `'Square of dim 10'`. Pretty cool.

Okay. Now we can go back to running the bank.

```
class Account:
    def __init__(self, initialBalance):
        self.currentBalance = initialBalance
    def balance(self):
        return self.currentBalance
    def deposit(self, amount):
        self.currentBalance = self.currentBalance + amount
    def creditLimit(self):
        return min(self.currentBalance * 0.5, 10000000)

>>> a = Account(100)
>>> b = Account(1000000)

>>> Account.balance(a)
100
>>> a.balance()
100
>>> Account.deposit(a, 100)
>>> a.deposit(100)
>>> a.balance()
300
>>> b.balance()
1000000
```

We've made an `Account` class that maintains a balance as state. There are methods for returning the balance, for making a deposit, and for returning the credit limit. These methods hide the details of the internal representation of the object entirely, and each object encapsulates the state it needs.

Here's the definition of a class representing the `set` ADT. We've called it `Sset` so it won't clash with Python's `set`.

```
class Sset:
    def __init__(self, items):
        self.contents = items

    def contains(self, x):
        return x in self.contents

    def add(self, x):
        self.contents.append(x)
```

```
    def intersection(self, otherSet):
        return [x for x in self.contents if otherSet.contains(x)]

    def union(self, otherSet):
        return self.contents + otherSet.elements()

    def elements(self):
        return self.contents

    def __str__(self):
        return str(self.contents)
    def __repr__(self):
        return str(self.contents)
```

Note that we've included both a `__str__` and a `__repr__` method. The Python documentation has this to say on the subject:

> "The str() function is meant to return representations of values which are fairly human-readable, while repr() is meant to generate representations which can be read by the interpreter (or will force a SyntaxError if there is not equivalent syntax). For objects which don't have a particular representation for human consumption, str() will return the same value as repr()."

When you do `print x`, then `str(x)` will be printed. When you just cause the Python shell to evaluate `x` and print the result, then it will print `repr(x)`. When in doubt, it's usually handy to define both.

Here are some examples of using `Sset`:

```
>>> s1 = Sset([1, 2, 3])
>>> s2 = Sset([3, 4, 5, 6, 7])
>>> s3 = Sset([4, 6, 8, 10])
>>> print s1.intersection(s2)
[3]
>>> print Sset.intersection(s1, s2)
[3]
>>> print s1.elements()
[1, 2, 3]
>>> s1.add(57)
>>> print s1.elements()
[1, 2, 3, 57]
```

# 4 State Machines

Both because they are a very important idea in electrical engineering, computer science, and a variety of other fields, and because they're a good domain for exercising OOP, we'll turn our attention to *state machines*.

A state machine (SM) is characterized by:

- 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 *transition function*, $t$, that indicates, for every state and input pair, what the next state will be,
- an *output function*, $o$, that indicates, for every state, what output to produce in that state, and
- an *initial state*, $s_0$, that is the element of $S$ that the machine starts in.

Together, these functions describe a discrete-time transition system, that can be thought of as performing a *transduction*: it takes in a (potentially infinite) sequence of inputs and generates a (potentially infinite) sequence of outputs.

Let's start by considering a simple example, where:

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

This machine can count up and down. 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. In this case, the output is always the same as the state (because the output function $o$ is the identity). If we were to feed it an input sequence of $u, u, u, d, d, u, u,$, it would generate the output sequence $0, 1, 2, 3, 2, 1, 2, 3$.

**Delay** An even simpler machine just takes the input and passes it through to the output. No state machine can be an instant pass-through, though, so the $k$th element of the input sequence will be the $k+1$st element of the output sequence. Here's the machine definition, formally:

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

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.

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

$$\begin{aligned} S &= \textit{numbers} \\ I &= \textit{numbers} \end{aligned}$$

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

Given input sequence $1, 5, 3$, it will generate an output sequence $0, 1, 6, 9$.

**Language acceptor** Here is a machine whose output is 1 if the input string adheres to a simple pattern, and 0 otherwise. In this case, the pattern has to be $a, b, c, a, b, c, a, b, c, \ldots$.

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

**Elevator** As a final example, we'll make a state-machine model of a crippled elevator, which never actually changes floors. All we can ask the elevator to do is open or close its doors, or do nothing. So, the possible inputs to this machine are `commandOpen`, `commandClose`, and `noCommand`. The elevator doors don't open and close instantaneously, so we model the elevator as having four possible states: `opened, closing, closed,` and `opening`. These correspond to the doors being fully open, starting to close, being fully closed, and starting to open. Finally, the machine can generate three possible outputs, which give some useful information about the state of the elevator. If the doors are closed, the output is `sensorClosed`; if they are open, the output is `sensorOpened`; and otherwise the output is `noSensor`.

State machines with a finite (small) number of states are often diagrammed using *state diagrams*; figure 3 shows the transition and output functions for our elevator model. The circles represent states. The bold label in the circle is the state name; the other entry is the output from that state. The arcs indicate transitions. The labels on the arcs are one or more inputs that lead to those state transitions.

In the `closed` state, if the elevator is commanded to open, it goes into the `opening` state and the output is `noSensor`. In the `opening` state, the `commandOpen` or `noCommand` input causes a transition to the `opened` state and the output of `sensorOpened`. In the `opening` state, the `commandClose` input causes a transition to the `closing` state. The remaining transitions and outputs can be read from the state diagram.

The transition function for the machine should indicate what state the machine transitions to as a result of each of the inputs and so it captures the arcs of the machine. In general, we have to specify the effect of *every* input in *every* state. The output function is simpler; it just indicates the output for every state.

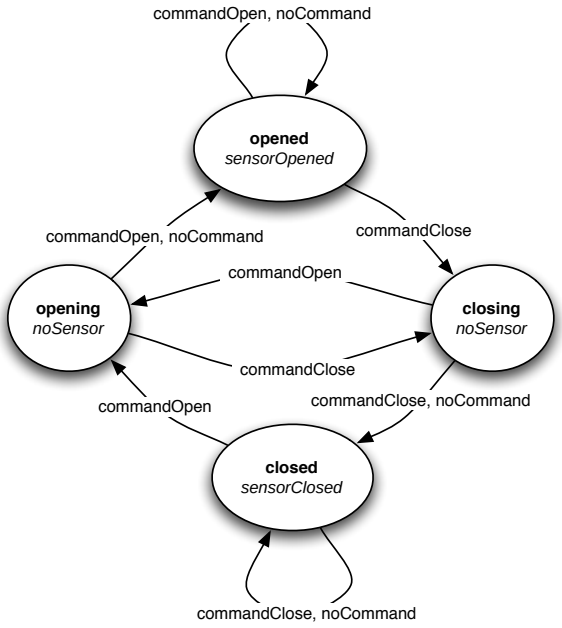


Figure 3: State diagram for a very simple elevator. Inspired by a figure from the Wikipedia article: Finite state machine

Here is the formal description of the elevator machine:

$$
\begin{aligned}
\mathsf{S} &= \{opened, closing, closed, opening\} \\
\mathsf{I} &= \{commandOpen, commandClose, noCommand\} \\
\mathsf{O} &= \{sensorOpened, sensorClosed, noSensor\} \\
\mathsf{s}_0 &= closed \\
\mathsf{t}(\mathsf{s},\mathsf{i}) &= table1 \\
\mathsf{o}(\mathsf{s}) &= table2
\end{aligned}
$$

The transition and output functions are most conveniently described using tables. The transition function is:

| s | i | t(s,i) |
|---|---|---|
| *opened* | *commandOpen* | *opened* |
| *opened* | *noCommand* | *opened* |
| *opened* | *commandClose* | *closing* |
| *closing* | *commandOpen* | *opening* |
| *closing* | *noCommand* | *closed* |
| *closing* | *commandClose* | *closed* |
| *closed* | *commandOpen* | *opening* |
| *closed* | *noCommand* | *closed* |
| *closed* | *commandClose* | *closed* |
| *opening* | *commandOpen* | *opened* |
| *opening* | *noCommand* | *opened* |
| *opening* | *commandClose* | *closing* |

The output function is:

| s | o(s) |
|---|---|
| *opened* | *sensorOpened* |
| *closing* | *noSensor* |
| *closed* | *sensorClosed* |
| *opening* | *noSensor* |

To help think about this machine, assume the elevator starts in the `closed` state, and try to predict the sequence of states and outputs that would result from the following sequence of inputs: *commandOpen*, *commandClose*, *noCommand*, *commandOpen*.

**Composition** Now we know how to define primitive state machines. Next week, we'll see how to apply our PCAP ideas to state machines, by developing a set of state-machine combinators, that will allow us to put primitive state machines together to make more complex machines.

**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*