MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Fall Semester, 2007

**Assignment 3, Issued: Tuesday, Sept. 18**

## To do this week

### ...in Tuesday software lab

1. Start writing code and test cases for the numbered questions in the software lab. Paste all your code, including your test cases, into the box provided in the "Software Lab" (Part 3.1) problem on the on-line Tutor. This will not be graded.

### ...before the start of lab on Thursday

1. Read the lecture notes.
2. Do the on-line Tutor problems for week 3 that are due on Thursday (Part 3.2).
3. Read through the entire description of Thursday's lab.

### ...in Thursday robot lab

1. Answer the numbered questions in the robot lab and demonstrate them to your LA.
2. Do the nanoquiz; it will be based on the material in the lecture notes and the on-line Tutor problems due on Thursday.

### ...before the start of lecture next Tuesday

1. Do the on-line Tutor problems for week 3 that are due on Tuesday (Part 3.3).
2. Do the lab writeup, providing written answers (including code and test cases) for **every** numbered question in this handout.

On Athena machines make sure you do:
`athrun 6.01 update`
so that you can get the `Desktop/6.01/lab3` directory which has the files mentioned in this handout.

- You need the file `fsm.py` for the software lab.
- You need the files `sequence.py` and `UtilityBrainTB.py` for the robot lab.

During software lab, if you are using your own laptop, download the files from the course Web site (on the Calendar page).

# Object-Oriented Programming; Combining Sequential Behaviors

Tuesday's lecture, the Tuesday software lab, and the on-line problems cover Python's support for *object-oriented programming*. Thursday's lab applies the tools of object-oriented programming to develop a new system for defining and combining robot behaviors sequentially, rather than the utility-based combination we looked at last week.

## Tuesday Software Lab: Practice

At the end of this lab, go to the on-line Tutor at `http://sicp.csail.mit.edu/6.01/fall07`, choose PS3, and paste all your code, including your test cases, into the box provided in the "Software Lab" problem. This will not be graded, it is simply a checkpoint for us to see how people are doing. Complete the assignment later and hand it in as part of your lab writeup. **Make sure that you include the test cases and results that you used to conclude that your code was working.**

This week, we'll get practice with object-oriented programming, and build some tools that will be useful in future labs.

### Polynomial Arithmetic

We can represent a polynomial as a list of coefficients starting with the highest-order term. For example, the polynomial $x^4-7x^3+10x^2-4x+6$ would be represented as the list `[1,-7,10,-4,6]`. Let's say that the *length* of a polynomial is the length of its list of coefficients.

Write a class called `Polynomial` that supports evaluation and addition on polynomials. You can structure it any way you'd like, but it should be correct and be beautiful. It should support, at least, an interaction like this:

```
>>> p1 = Polynomial([3, 2, 1])
>>> print p1
poly[3.0, 2.0, 1.0]
```

This represents the polynomial $3x^2+2x+1$.

```
>>> p2 = Polynomial([100, 200])
>>> print p1.add(p2)
poly[3.0, 102.0, 201.0]
>>> print p1 + p2
poly[3.0, 102.0, 201.0]
>>> p1.val(1)
6.0
>>> p1.val(10)
321.0
```

The constructor accepts a list of coefficients, highest-order first.

**Question** 1. Define the basic parts of your class, with an `__init__` method and a `__str__` method (see the lecture notes for more information about this), so that if you do

```
>>> print Polynomial([3, 2, 1])
poly[3.0, 2.0, 1.0]
```

something nice and informative is printed out.

**Question** 2. Implement the `add` method for your class.

A cool thing about Python is that you can *overload* the arithmetic operators. So, for example, if you add the following method to your `Polynomial` class

```
def __add__(self, v):
    return self.add(v)
```

or, if you prefer,

```
def __add__(self, v):
    return Polynomial.add(self, v)
```

then you can do

```
>>> print Polynomial([3, 2, 1]) + Polynomial([100, 200])
poly[3.0, 102.0, 201.0]
```

Exactly what gets printed as a result of this statement depends on how you've defined your `__str__` procedure; this is just an example.

**Question** 3. Add the `__add__` method to your class.

A straightforward way to evaluate polynomials is to explicitly add up the terms $a_i x^i$. We can do this with list comprehension and `sum`. Hint: note that in a polynomial with $k$ coefficients, the highest power of the variable is $k-1$, for example, our example polynomial of length 3 has the highest power of $x^2$.

**Question** 4. Add the `val` method to your class, which evaluates the polynomial for the specified value.

Try to do things as simply as possible. Don't do anything twice.

### Finite State Machines

This section builds on the FSM discussion in lecture. All the code from lecture can be found in the `fsm.py` file. There is additional written material covering the lecture presentation at the end of this handout.

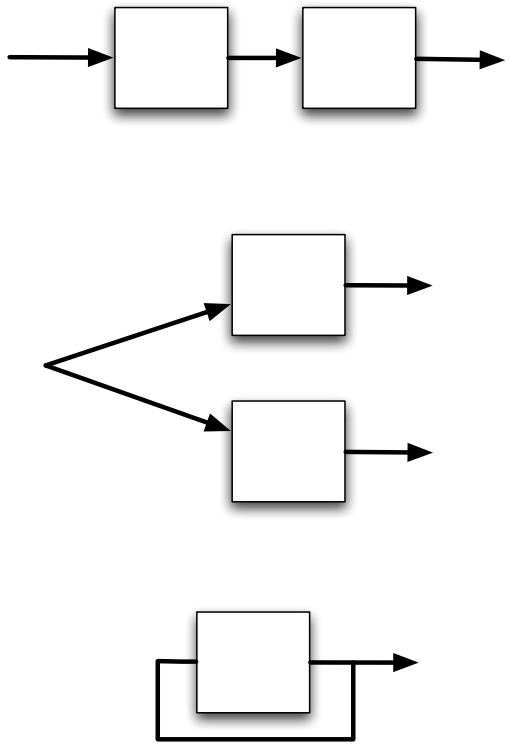

Figure 1: Serial, parallel, and feedback compositions of state machines.

**Question** 5. Assuming the elevator starts in the `closed` state, what is the sequence of states and outputs for the following sequence of inputs: `commandOpen, commandClose, noCommand, commandOpen`?

If you have a very complicated system to model or implement, writing a single monolithic state transition function would be very difficult. Happily, we can apply our PCA (primitive, combination, abstraction) methodology here, to build more complex FSMs out of simpler ones.

We wil treat the `PrimitiveFSM` class as a primitive and consider some possible means of combination. Figure 1 sketches three types of FSM composition: serial, parallel, and feedback. In this lab, we'll focus on serial and feedback composition.

### Serial composition

In serial composition, we take two machines and use the output of the first one as the input to the second. 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.

Here is the skeleton of a `SerialFSM` class. It takes two FSMs as input at initialization time, and then implements the `step` function of the combined machine.

```
class SerialFSM:
    def __init__(self, fsm1, fsm2):
        <your code here>
    def step(self, input):
        <your code here>
```

**Question** 6. Implement the `SerialFSM` class.

**Question** 7. Make a new FSM that is the serial composition of the `clockButtons` FSM (in the `fsm.py` file) and the `elev` FSM and step it through some steps.

Go to the on-line Tutor at `http://sicp.csail.mit.edu/6.01/fall07`, choose PS3, and paste the code that you wrote during lab, including your test cases, into the box provided in the "Software Lab" problem. Do this even if you have not finished everything. Your completed answers to these questions are to be handed in with the rest of your writeup for the on Tuesday, September 25. **Make sure that you include the test cases and results that you used to conclude that your code was working.**

If you have extra time, you can start on the homework below which asks for extensions to the `Polynomial` class and implementation of additional means of combining FSMs.

# Homework: Due Tuesday Sept. 25

## Polynomial Arithmetic

Extend your `Polynomial` class to support, at least, an interaction like this:

```
>>> p1 = Polynomial([3, 2, 1])
>>> print p1
poly[3.0, 2.0, 1.0]
```

This represents the polynomial $3x^2 + 2x + 1$.

```
>>> p2 = Polynomial([100, 200])
>>> print p1 + p2
poly[3.0, 102.0, 201.0]
>>> print p1 * p1
poly[9.0, 12.0, 10.0, 4.0, 1.0]
>>> print p1 * p2
Polynomial[300.0, 800.0, 500.0, 200.0]
>>> print (p1 * p1) + p1
poly[9.0, 12.0, 13.0, 6.0, 2.0]
>>> p1.val(1)
6.0
>>> p1.val(10)
321.0
>>> p1.roots()
[(-0.33333333333333331+0.47140452079103173j),
 (-0.33333333333333331-0.47140452079103173j)]
>>> p3 = Polynomial([3, 2, -1])
>>> p3.roots()
[(0.33333333333333331+0j), (-1+0j)]
>>> (p1 * p2).roots()
Order too high to solve for roots.  Try Newton.
```

The constructor accepts a list of coefficients, highest-order first. You only have to compute the roots if it is quadratic or less, otherwise `roots` can generate an error message.[1]

The tricky part of the implementation is how to implement the multiplication of polynomials, including polynomials of different numbers of coefficients. Here are a few hints about polynomial multiplication. The length of the resulting polynomial is the sum of the lengths of the input polynomials minus 1. So, the product of two polynomials of length 3 is a polynomial of length 5. The key observation is that the $k^{th}$ coefficient is the result of adding up all the coefficients whose indices add up to $k$. Be sure you understand the results in the example above.

As for beauty, try to do things as simply as possible. Don't do anything twice. If you need some extra procedures to help you do your work, you can put them in the same file as your class definition,

[1] But we'd be delighted (and give you extra credit) if you wanted to implement higher-order root-finding as an exploration. See the discussion of Newton's method in Lecture 2.

but outside the class (so, put them at the end of the file, with no indentation). For instance, we found it useful to write helper functions to: add two lists of numbers of the same length, to extend a list of numbers to a particular length by adding zeros at the front, and to compute quadratic roots (you should already have this one!).

**Question** 8. Add the `roots`, `mul` and mul `__mul__` methods to your `Polynomial` class.

**Question** 9. Explain, in comments, your strategy for representing a polynomial internally, as well as for evaluation, addition and multiplication.

Also, submit this code as your answer to the Polynomial Tutor question for PS 3. This will perform tests of the basic capabilities of the class.

## Finite State Machines

### Feedback composition

Another important means of combination that we will make much use of later is the 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. We initialize this by specifying the initial input for the machine (since we don't have a previous output to feed back). Note that once the feedback is started, no additional input is required. It is crucial that the input and output vocabularies of the machine that is being operated on are the same (because the output at step $t$ will be the input at step $t+1$.

```
class FeedbackFSM (FSM):
    def __init__(self, fsm, initInput):
        <your code here>
    def step(self):
        <your code here>
```

Once we have fed the output back to the input, this machine doesn't consume any inputs, which is why this `step` method has no input argument.

Here is an example of using feedback to make a machine that counts.

We start with a simple machine, called `incrementer`, that takes an integer as input and returns that same integer plus 1 as the output; it has no real internal state, in the sense of having memory. However, the output depends *only* on the state and not the input. Therefore, to achieve an output that depends only on the input, we set the state to be the input plus 1 and the output to be just the state. Here is an example of this machine in operation:

Remember that the transition function takes as its argument a list `[state, input]`; and the output function takes as argument a list whose single element is the state.

```
>>> incrementer = PrimitiveFSM('incr', lambda s_i: s_i[1]+1, lambda s: s[0], 0)
>>> incrementer.step(0)
incr  oldState:  0  input:  0
incr  newState:  1  output:  1
1
>>> incrementer.step(99)
incr  oldState:  1  input:  99
incr  newState:  100  output:  100
100
```

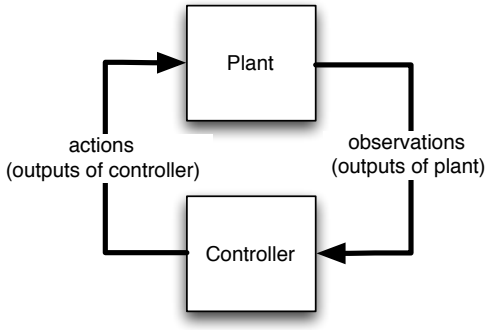


Figure 2: Two machines connected together, as for a simulator.

To make a counter, which *does* need memory, we do a feedback operation on the `incrementer`, connecting its output up to its input, and saying that the machine starts with a 0 on its input. Now, as we do step operations, we can see that we have built a machine that counts by ones.

```
>>> counter = FeedbackFSM(incrementer, 0)
>>> counter.step()
incr  oldState:  0  input:  0
incr  newState:  1  output:  1
>>> counter.step()
incr  oldState:  1  input:  1
incr  newState:  2  output:  2
>>> counter.step()
incr  oldState:  2  input:  2
incr  newState:  3  output:  3
>>> counter.step()
incr  oldState:  3  input:  3
incr  newState:  4  output:  4
```

**Question** 10. Implement the internals of the FeedbackFSM class and test it on the counter example (we hope it's not a counterexample!).

## 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 FSMs so that the output of the plant (typically thought of as some sensory observations) is input to the controller, and the output of the controller (typically thought of as some actions) is input to the plant. This is shown schematically in figure 2. For example, that's what happens when you build a Soar brain that interacts with the robot: the robot (and the world it is operating in) is the "plant" and the brain is the controller. We can build a coupled machine by first connecting the machines serially and then using feedback on that combination.

**Question** 11. Define a procedure `simulatorFSM` that takes two FSMs as input which uses serial and feedback combination to create and return a "simulator" (another FSM) that couples the input of one machine to the output of the other, and vice versa. It will also need to take as input the first input to the first machine.

**Question** 12. You are the engineer in charge of testing the basic elevator mechanism. You need to write a controller that will make sure the elevator works reliably, without you having to give the elevator individual commands. Write a controller that, whenever it senses the elevator is closed, issues an open command, and that whenever it senses the elevator is open, issues a closed command. Use your `simulatorFSM` procedure to couple your controller with the `elev` FSM, and run some simulations to see what happens.

## Non-Determininstic machines

The machines we have been considering thus far are *deterministic*: given a particular state and an input, the resulting next state is always the same. Similarly, the output for any particular state is always the same. In many situations, we want to model non-deterministic machines, which can generate different state transitions or outputs from the same state. For example, the elevator may occasionally make an unexpected transition and close the door instead of opening it. Or, a robot may unexpectedly rotate a bit when it tries to move forward. We will study non-deterministic systems in technical detail later in the term; but we can start now by making a relatively simple extension to our `FSM` and `TableMapping` implementation to handle non-deterministic machines.

The key idea is that we can specify a probabilistic *distribution*, instead of a single value, as the output of a mapping. So, for example, instead of indicating that in the `closed` state, the `commandOpen` input always causes a transition to the `opening` state, we could say that with 0.9 probability the machine moves to the `opening` state, but with probability 0.1, it goes to the `closing` state.

We would like to be able to use probability distributions in the specification of transition and output tables anywhere we previously used a state or output value. The distribution specifies the probability that we get each particular value of the state or output. That is, we specify a probability space such as the ones we dealt with in last week's software lab.

A distribution can be represented as a list of pairs `[prob, value]` specifying those values with non-zero probability.

```
Dist([[0.9, 'opening'],[0.1, 'closing']])
```

This distribution encodes a 0.9 probability of the value `opening` and a 0.1 probability of the value `closing`.

Note that in many situations, there is a big space of states, but only a few that have a non-zero transition probability from the current (the robot may have some noise in its motion, but it can't randomly teleport just anywhere!). In this representation, we only need to specify the states with non-zero probability.

Now, we can specify a stochastic elevator:

```
stransTable = [['opened', 'closing', 'opened'],
               ['opening', 'closed', 'closed'],
               [Dist([[0.9, 'opening'],[0.1, 'closing']]),
                'closed', 'closed'],
```

```
                  ['opened', 'closing', 'opened']]
selevTrans = StochasticTableMapping([elevStates, elevActions], stransTable)
selev = FSM('noisyElevator', selevTrans, 'closed')
```

We've used a new class here, called `StochasticTableMapping`. Below, we give a skeleton for it. It is a subclass of `TableMapping`, which redefines the `lookup` method.

```
class StochasticTableMapping (TableMapping):
    def lookup(self, args):
        pass
```

You should make this be a "wrapper" method, in the sense that it calls the `lookup` method for the parent class to do some of the work. You can do this by `TableMapping.lookup(self,args)`. You may then want to do something different based on what you get from the `lookup` operation. You will probably want to know whether it is a distribution or not. `isinstance(x, Classname)` returns `True` if object x is an instance of class `Classname`, and `False` otherwise.

**Question** 13. Provide an implementation for the `StochasticTableMapping` class.

**Question** 14. Define a stochastic version of the elevator and test it.

### Buttons

Now that we have an elevator, we need a user interface for it. Imagine that there are three buttons that people can push: a `call` button on the outside of the elevator, and `open` and `close` buttons on the inside of the elevator. Design a FSM that takes, as input, the values of all of those buttons, and generates, as output, commands to the elevator. We would like it to behave so that:

- Whenever someone pushes the `call` button, the elevator opens as soon as possible;
- Whenever someone pushes the `open` button from inside the elevator, it opens as soon as possible;
- Whenever someone pushes the `close` button from inside the elevator, it waits 4 steps (in case someone is running to get in!), and then closes.
- If there have been no buttons pushed for 6 steps, the elevator should close.

Of course, the constraints above are not a complete specification of how the elevator should behave. Your machine will have to react to every possible combination of button-presses. What should it do when all three buttons are pressed simultaneously? [2]

**Question** 15. If there are three buttons, how big is the input vocabulary?

**Question** 16. Implement a `buttonHandler` primitive FSM. Demonstrate that it behaves as specified above. Its outputs should be elements of `elevInputs`.

**Question** 17. In your writeup show a test for a serial combination of the `buttonHandler` FSM that you build and the `elev` FSM. Put it through a sequence of steps to show that it works.

[2] The topic of error checking and handling in programs of all sorts is difficult but important. In general, when programming one should guard against violated expectations (due to bugs, user mistakes, or whatever) by doing error checking. However, in any sufficiently complex program, the set of expectations is so large that it may be impossible to check for every case. The issue of what to do when an error is detected is also difficult; in many cases it is not acceptable simply to exit the program; like shows, programs must go on...

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*

## Robot Lab: Combinations of terminating behaviors

In lab 2, we explored a system for making primitive behaviors and combining them. In that case, our behaviors were all operating at once, and their outputs were expressed as utility functions. It's possible to do a lot of different things this way, but many people are frustrated by the inability to ask their robot to do things sequentially, like drive up to the wall and then turn left. It would be possible to use a finite-state machine to control the modes of the robot, but that ends up being an awkward way of specifying behavior that is mostly sequential.

In this lab, we'll develop a new system of behavior definition and combination that allows sequential combination.

In order for the system to remember its place in a sequence of actions, it needs to have *state* that persists between invocations of the `step` function. We will use Python objects to store and encapsulate this state. We'll represent each behavior in our new system as a type of object called a *terminating behavior*. A terminating behavior must have three methods: `start`, `step`, and `done`. These methods have no arguments except `self`.

The `start` method will be called once when the behavior is supposed to begin running; it's an opportunity to remember some initial condition, like the current odometry values. The `step` method will be called repeatedly until the behavior is done. Although it has no arguments, it can call `pose()` or `sonarDistances()` to get sensory information, and can call `motorOutput` to cause the robot to move.

In order to define a new primitive terminating behavior object, say, `GoForwardUntilXLimit`, which moves the robot forward until its `x` coordinate is greater than some limit value, you might write code like this:

```
class GoForwardUntilXLimit:
    def __init__ (self, xLimit):
        self.xLimit = xLimit
    def start(self):
        pass
    def step(self):
        motorOutput(0.1, 0.0)
    def done(self):
        return pose()[0] > self.xLimit
```

The initializer for the class just remembers the limit value; and there's no special work to be done when the behavior is started. On each step, the behavior sets its motor output values to move straight forward. The `done` method returns the value `True` when the first component of the list returned by the `pose` function (which represents the robot's `x` coordinate) is greater than the limit.

Here is the code for a brain that creates a new instance of the `GoForwardUntilXLimit` behavior class, and then uses that behavior to select actions until its `done` method returns true, at which point the robot stops moving (though the brain continues to run).

```
def setup():
    robot.behavior = GoForwardUntilXLimit(1.0)
    robot.behavior.start()

def step():
    if robot.behavior.done():
        motorOutput(0.0,0.0)
```

```
else:
    robot.behavior.step()
```

If you defined a new terminating behavior class, like `DoTheMacarena`, then you'd have `DoTheMacarena()` in the code above in place of `GoForwardUntilXLimit()`.

Notice that the brain and the behavior each have their own `step` method, and that these are different. This is an example of generic functions, implemented with the aid of object-oriented programming. In this implementation, the brain's `step` method calls the behavior's `step` method as long as the behavior is not `done`.[3]

Something to watch out for is this: the `setup` method of the brain, and therefore the `start` method of your behavior are called when the brain is *loaded*. Then, when you click `start`, the `step` method of the brain is called repeatedly. If you're using the simulator, and want to reposition the robot by dragging it, do this *before* you reload the brain. If you do it between reloading and hitting `start`, the pose readings will be all wrong.

### Primitive terminating behaviors

Implement the following primitive terminating behaviors and test them by themselves using the brain code above, found in `sequence.py`. That file also contains some helper functions that you might find useful.

When you're testing terminating behaviors, you'll need to reload the brain inside SoaR once a behavior has terminated. Why? Because the program has to be run again from the beginning; if it just continues from where it was, the behavior might still think it's done.

**Question** 18. `ForwardTB(d)` Define a class `ForwardTB`, with an `__init__` method that takes as an argument a distance `d` to move. Move forward a fixed distance, `d`, from the robot's position at the time the `start` method is called, along the current heading. *Caveat:* Remember that the robot might be moving at an arbitrary angle with respect to its global coordinate system.

**Question** 19. `TurnTB(a)` Class to turn a fixed angle `a` (in radians) from the robot's heading at the time the `start` method is called. *Caveat:* Remember that `pose()` returns an angle between 0 and $2\pi$, so finding the difference between two angles is not trivial. We have provided a helper in `sequence.py` function you may find useful.

**Question** 20. `ForwardUntilBlockedTB()` Class to move forward until the robot is blocked in front.

Think carefully about an appropriate termination condition to use in each case. **Don't start implementing these until you've talked with your LA about your plan!**

[3] Are you wondering why we refer to the brain's `step` as a *method*, when you don't see any class definition here? There really is a `Brain` class with `step` and `setup` methods, but the class definition is buried inside the implementation of SoaR. There's also a `robot` object that the brain uses in order to store attributes. That's what lets the `setup` procedure store something as `robot.behavior`, which can be referenced by the brain's `step`. You can use `robot` for storing anything that you like, and it will be local to the particular instance of the brain that SoaR is running.

**Checkpoint: 11AM**

- Explain your strategy for implementing each of these classes to your LA.
- Demonstrate each behavior to your LA.

## Means of combination

Now, we'd like to implement some means of combining these behaviors so we can make more complex overall behavioral structures. How can we make this happen? We'd like to define a new class of terminating behaviors, called `TBSequence`, that takes as an initialization argument a list of terminating behavior objects, and executes them in order.

To do this, we have to keep track of which behavior we are currently executing, and keep doing its associated step function until it is done, and then start executing the next behavior. The code below contains all but the `step` and `done` methods for the class, which you should complete.

```
class TBSequence:
    def __init__(self, actions):
        self.actions = actions

    def start(self):
        self.counter = 0
        self.actions[0].start()
```

**Question** 21. Add the `step` and `done` methods for `TBSequence` to the `sequence.py` file.

**Question** 22. Now, use sequencing and your primitives to make the robot drive in a one meter square. Explain how your program works. Draw a diagram of the different class instances involved.

**Question** 23. Execute your program for driving in a square repeatedly. Think about a simple way of measuring and describing the accuracy of your program running on the robot.

**Checkpoint: 11:30 PM**

- Explain your strategy for implementing the `TBSequence` class to your LA.
- Demonstrate your program driving in a square to your LA.

## Safety

We have now constructed a useful framework for building sequential behaviors on top of a substrate that still affords frequent reading of and reaction to sensor readings. This means that, while carrying out a sequence of behaviors, the robot can react to surprises. For now, we'll show how it can change its low-level behavior in reaction to sensor values, without changing the actual sequence of high-level steps it is taking. Later in the class, we'll develop a stream-based method of sequential programming that affords a great deal of flexibility in the high-level sequencing, as well.

**Question** 24. Write a class `SafeForwardTB` that has the basic `ForwardTB` terminating behavior as a superclass, and that overrides one of the superclass methods so that the robot will sit and wait rather than run into an obstacle. Note that it shouldn't terminate when it's blocked.

**Checkpoint: 12:00 PM**

- Demonstrate your safe driving to your LA by asking your robot to drive in a one meter square and then putting an obstacle in its way as it goes along.

## More means of combination

Implement two more means of combination for terminating behaviors. Here are some that we have thought of:

- **TBRepeat**: take a terminating behavior and a number $n$, and execute the behavior $n$ times in a row before terminating.
- **TBIf**: take a condition and two terminating behaviors. Test the condition when this behavior is started (not created!), and then execute the first behavior if the condition was true and the second if it was false.
- **TBWhile**: take a condition and a terminating behavior. When the behavior is started, evaluate the condition. If it's false, terminate. If not, execute the terminating behavior, then test the condition again, etc.
- **TBParallelFun**: take a terminating behavior and a function, and call the function on every step of the terminating behavior.

**Question** 25. Use your new means of combination to make an interesting robot behavior (the robot macarena, for example).

**Checkpoint: 12:30 PM**

- Demonstrate your new means of combination to your LA.

## Concepts covered in this lab

- Encapsulation of state into objects can provide useful abstraction.
- There are many different frameworks for abstraction and combination, and it's important to choose or design one suited to your problem.
- It can be hard to get repeatable behavior from a physical device.

## Post-Lab Writeup and Exercises: Due before lecture on Sept. 25

All post-lab hand-ins should be written in clear English sentences and paragraphs. We expect at least a couple of sentences or a paragraph in answer to **all** of the numbered questions in this lab handout (including the software lab).

We also want the code you wrote. When you write up answers to programming problems in this course, *don't* just simply print out the code file and turn it in. Especially, don't turn in long sections of code that we've given you. Turn in your own code, with examples showing how it runs, and explanations of what you wrote and why.

We also expect at least a couple of sentences or a paragraph in answer to the reflection questions below. We're in interested in an explanation of your thinking, as well as the answer.

**Question** 26. Suggest how you could arrange it so that an FSM could actually produce outputs in the world, for example, send commands to a robot, or actually run the motor on an elevator door.

**Question** 27. Here's the definition of a class of terminating behaviors that take a string at initialization time, print it when they step, and then terminate.

```
class T:
    def __init__(self, text):
        self.text = text
    def start(self):
        self.isDone = False
    def step(self):
        print self.text, " "
        self.isDone = True
    def done(self):
        return self.isDone
```

Here's a procedure that takes a terminating behavior and executes it until it's done

```
def execute(tb):
    tb.start()
    while not tb.done():
        tb.step()
```

Here is the definition of a somewhat complex behavior

```
s1 = TBSequence([T('a'), T('b'), T('c')])
r4 = TBSequence([TBRepeat(s1, 4), T('hi')])
b = TBRepeat(r4, 3)
```

What happens when you do `execute(b)` ?

**Question** 28. Describe (you don't need to write any code) how you could build a finite state machine that would output the same sequence of words as the example above. How many states would it need to have?

**Question** 29. How reliable was your program to drive in a square? What do you think was the main contributing source to the errors? What are some strategies for reducing the error?

**Question** 30. If we had asked you to write a program to drive in a square before you had read and done this lab, how would you have approached it? What are the advantages or disadvantages of this approach relative to the ones you have seen in this and the previous lab?

**Question** 31. We have now seen two different frameworks for making primitive robot behaviors and combining them: the parallel (utility-based) and sequential (terminating behaviors) approaches. Consider a robot for operating in a household, doing chores. Give examples of situations in which each type of behavior combination would be appropriate.

The following section is not required. Explorations are things that you can work on if you're interested and/or have extra time in lab. If you hand in a page or two describing some work you did on this, you are eligible for up to an extra half a point on the lab.

### Exploration: Obstacle avoidance

We can do better than sitting and waiting! We can build a terminating behavior that moves to the desired location, even if there is a permanent obstacle in the way. In the previous lab, we explored utility functions as a method for creating complex primitive behaviors. We can easily wrap up those behaviors to use as components in a sequential behavior program.

In the file `UtilityBrainTB.py` we have given you the code from lab 2, augmented by a `go-to-goal` behavior combined with the `avoid` utility function to build a terminating behavior that drives to a goal location while avoiding obstacles. Use this and `TBSequence` to have the robot drive to a goal location and then drive back to where it started. If you can't get to do this on the robot, do it on the simulator.

Demonstrate that your robot can drive to a location and then return from it, even when there are (simple) obstacles in the way.

How effective is the collision-avoidance strategy? Can you improve it? Could memory be useful?

## Background on Finite State Machines

This is background on FSMs; this was covered in lecture

Our goal will be to build models of *finite-state machines* (FSMs), which can be used to model (and implement) the behavior of machines and processes. They are widely used in digital circuit design, linguistics, computer science, and many other disciplines.

A state-machine 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* that indicates, for every state and action pair, what the next state will be, and
- an *output function* that indicates, for every state, what output to produce in that state.

As a simple example, we'll make an FSM 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`.

FSMs 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.

### Mappings

To specify the elevator state machine, we need to specify the transition function, which is a mapping from states and inputs into the next state (often denoted as: $S \times I \rightarrow S$), and the output function, which is a mapping from states to outputs (denoted $S \rightarrow O$).

There are many ways of implementing mappings in Python and we shouldn't have to care when we're building an FSM how that mapping is represented. For many FSMs, such as our elevator model,

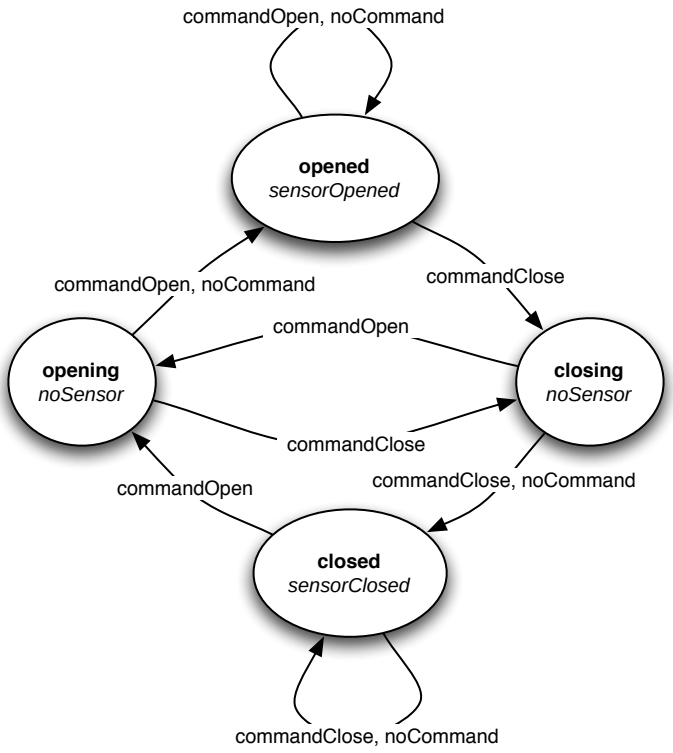


Figure 3: State diagram for a very simple elevator. Inspired by a figure from the Wikipedia article: Finite state machine

these mapping are best described in terms of tables. So, we will define a class called `TableMapping` that will conveniently implement a mapping. This class will provide a `lookup` method that will take as an argument a list of values for the inputs to the mapping and will generate a single output value.

Although we could use numbers for everything, we'd rather use symbolic names for states, inputs, and outputs. Here is a set of names for the elevator domain.

```
>>> elevStates = ['opened', 'closing', 'closed', 'opening']
>>> elevInputs = ['commandOpen', 'commandClose', 'noCommand']
```

Now, we can specify the state transitions in a table, represented as a list of lists, so that if we index first by a state index (the index of `'closed'`, for example, is 2), and next by an input index (the index of `'commandOpen'` is 0), we'll get the index of the next state (`transTable[2][0]`, which is `'opening'`, which is the `opening` state). The output table is specified in a similar way, but depends only on the current state.

```
>>> transTable = [['opened', 'closing', 'opened'],
                  ['opening', 'closed', 'closed'],
                  ['opening', 'closed', 'closed'],
                  ['opened', 'closing', 'opened']]
>>> outTable = ['sensorOpened', 'noSensor', 'sensorClosed', 'noSensor']
```

Now, we can make instances of our `TableMapping` class, by specifying two arguments: a list of lists of names (one list of names for each input argument), and a table.

```
>>> elevTrans = TableMapping([elevStates, elevInputs], transTable)
>>> elevOut = TableMapping([elevStates], outTable)
```

Here's how we will use the class.

```
>>> elevTrans.lookup(['opened', 'commandClose'])
'closing'
>>> elevOut.lookup(['closing'])
'noSensor'
```

We have given you an implementation of the `TableMapping` class in `fsm.py`.

In many other situations, we'll want to use transition functions that have some internal regularities, which makes it easier to write code to compute the function rather than to specify the entire transition table. An example of this type of transition function describes the behavior of a robot moving on a grid of squares in two dimensions. Its location is described by a state `[x, y]`, specifying its `x` and `y` coordinates. If the inputs to the robot are commands to move North, South, East, or West, we might have a transition function like the following:

```
def robotMoveNSEW(args):
  [state, input] = args
  [x, y] = state
  if input == 'N':
      return [x, y+1]
  elif input == 'S':
      return [x, y-1]
  elif input == 'E':
      return [x+1, y]
  elif input == 'W':
```

```
        return [x-1, y]
    else:
        print "Unknown input:", input
        return state
```

Using functions, we could also build a mapping that has continuous state variables, for example, the $[x, y, \theta]$ pose of the robot and where the inputs are the commanded forward and rotational velocity. Of course, neither this nor the previous robot example is exactly a *finite* state machine, because the state spaces are infinite.

### FSM

Now, it's time to build an FSM class. The only operation we really need to do on an FSM is to ask it to take a step. Taking a step means that we give it an input; it makes a transition to a new state, depending on the input and the old state; and then it returns an output.

Here is the `PrimitiveFSM` class. We call it primitive not because its knuckles drag on the ground, but because we'll be exploring ways to make composite FSMs in the following sections. The `__init__` method takes a `name` argument, just because it might be useful to print out for debugging, and the transition and output functions. It also needs to know what state it should be in when it starts.

```
class PrimitiveFSM:
    def __init__(self, name, transitionfn, outputfn, initState):
        self.name = name
        self.transitionFunction = transitionfn
        self.outputFunction = outputfn
        self.state = initState

    def step(self, input):
        print self.name, " oldState: ", self.state, " input: ", input
        self.state = self.transitionFunction([self.state, input])
        output = self.outputFunction([self.state])
        print self.name, " newState: ", self.state, " output: ", output
        return output
```

Once we have this class definition, we can construct a new instance, representing our elevator. We have it start in the `closed` state, then ask it to open. It makes a transition to the state `opening`, and generates the output `'noSensor'`. Then we didn't send it another command, but we had to give it an input to make it take a step, so we gave it the input `'noCommand'`.

```
>>> elev = PrimitiveFSM('elevator', elevTrans.lookup, elevOut.lookup, 'closed')

>>> elev.step('commandOpen')
elevator  oldState:  closed  input:  commandOpen
elevator  newState:  opening  output:  noSensor
'noSensor'

elev.step('noCommand')
elevator  oldState:  opening  input:  noCommand
elevator  newState:  opened  output:  sensorOpened
'sensorOpened'
```

Note that we are using `elevTrans.lookup` as the transition function for the `elev` FSM and `elevOut.lookup` as the output function. You should realize that the methods of a class are just functions, like any other function, and can be passed as arguments. So, we are using the table mappings that we constructed to define the elevator class. In general, we can create `PrimitiveFSM` instances using any functions that implement the appropriate mappings $S \times I \rightarrow S$ and $S \rightarrow O$.