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

Assignment 3, Issued: Tuesday, February 19

To do this week

...before your lab on February 21 or 22

1. Read the notes on state machines (section 4 of the course notes).
2. Do the on-line tutor problems in section PS.4.2.
3. **Attend lecture: at one of the following places and times:**
   - **Thursday, Feb 21, 10AM - 11AM, in 32-123**
   - **Thursday, Feb 21, 2PM - 3PM, in 32-123**
   - **Friday, Feb 22, 10AM - 11AM, in 32-155**

...in your lab on February 21 or 22

1. **Hand in assignment 2 (software and design labs, stapled together) and the week 1 exploration, if you've done it (separately)**
2. Work through the numbered questions on the lab.

...before the start of your software lab on February 26 or 27

1. Do the on-line tutor problems in section PS.4.3.
2. Turn in written answers to all numbered questions in this lab.

Lab for Week 3

Do this lab on your own.

On the lab laptops, be sure you do:
`athrun 6.01 update`
so that you can get the `Desktop/6.01/lab3` directory which has the file `PrimitiveSM.py`.

The work for this week is a substantial problem set using simple object-oriented programming, with two parts: one on state machines and one on polynomials. During software lab, we want to get you started on each of these sections, so we'll ask you to spend an hour working on the first part of the state machine assignment, and then to switch and spend an hour working on the polynomial part. We don't expect you to finish either part during lab.

Simple State Machines

You have read about state machines in the course notes already. Now, it's time to build a state-machine class. The only operation we really need to do on a state machine 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 `PrimitiveSM` class. We call it primitive not because its knuckles drag on the ground, but because we'll be exploring ways to make composite SMs in the following week. The `__init__` method takes the transition and output functions as input. It also needs to know what state it should be in when it starts; for generality that we will take advantage of next week, we'll provide a function that can be called to generate an initial state. The types of the arguments to `__init__` are:

$$\begin{aligned} \texttt{transitionfn} &: (\mathit{state}, \mathit{input}) \rightarrow \mathit{state} \\ \texttt{outputfn} &: \mathit{state} \rightarrow \mathit{output} \\ \texttt{initfn} &: () \rightarrow \mathit{state} \end{aligned}$$

The return type is `PrimitiveSM`.

```
class PrimitiveSM:
    def __init__(self, transitionfn, outputfn, initfn):
        self.transitionFunction = transitionfn
        self.outputFunction = outputfn
        self.currentState = initfn()

    def step(self, input=None):
        self.currentState = self.transitionFunction(self.currentState, input)
        return self.currentOutput()

    def currentOutput(self):
        return self.outputFunction(self.currentState)
```

The `step` method takes an input for the state machine; but since we will sometimes be interested in state machines that don't have an input, we've supplied a *default* argument of `None`. This means that when we call it, `step` can either be given zero or one arguments; if it is given zero, then it will act as if it was given one argument, `None`.

Once we have this class definition, we can construct a simple state machine instance. We'll make `tickTock`, which alternates between outputs of `True` and `False`.

```
tickTock = PrimitiveSM(lambda s, i: not s,
                       lambda s: s,
                       lambda : True)
```

Now type

```
>>> tickTock.step()
```

a few times and see what happens.

To save typing, we've defined a procedure that will run a state machine with no input for some pre-specified number of steps:

```
def run(sm, n = 10):
    print sm.currentOutput()
    for i in range(n):
        print sm.step()
```

Try using this to run the `tickTock` machine. The argument `n = 10` means that you can specify a second argument to say how many steps the machine should be run, but if you don't it will default to 10.

Question 1: Make a new state machine that counts. Its first output should be 0, then 1, then 2, etc. It should ignore its input. Test it using `run`.

Question 2: Make a state machine that ignores its input and generates this sequence: $0, 1, 2, 3, 0, 1, 2, 3, \ldots$. Test it appropriately.

Question 3: Write a procedure that takes an input n and outputs a state machine that ignores its input and generates the sequence $0, 1, \ldots, n-1, 0, 1, \ldots, n-1, 0, \ldots$. Test it appropriately.

Question 4: Write a procedure `transduce` that takes as an argument a state machine and a list, and runs the state machine for a number of steps equal to the length of the list, feeding each element of the list in as input to the state machine, and returning the list of outputs. The first element of the list of outputs should be the output the machine generates based on the initial state (see how `run` calls `sm.currentOutput()` to get the first output). The length of the output list should be one more than the length of the input list (because we're interested in the initial output).

Question 5: Make a new state machine that simply delays its input by one time step. So, if you give it an input sequence of `[3, 4, 2, 5]`, its output sequence should be `[init, 3, 4, 2, 5]`, where `init = 0`. Test it with `transduce`.

Question 6: Write a procedure `makeDelayMachine` that takes a single argument `init`. It should return a state machine like the one you created above, but which outputs `init` as the first element.

Question 7: Write a procedure `makeDoubleDelayMachine` that takes two arguments, `init1` and `init2`. It should return a state machine that outputs `init1` as the first output, and then outputs `init2`, and then outputs its first input, and then its second, input, etc. So, for example, if you gave it an input sequence of `[3, 4, 2, 5]`, it should generate an output sequence of `[init1, init2, 3, 4, 2]`. (Hint: let the state be a list or tuple containing the two most recent inputs).

Question 8: Make a new state machine whose output is the average of all of the inputs it has seen so far. There are many ways to do this. Try to find a way that doesn't involve storing all the inputs you have ever seen in the state. Test it appropriately.

Question 9: A *set-reset* flip-flop is a simple finite-state component that is (or, at least was) used in the design and construction of computers. The machine has two states, 0 and 1. The output is the same as the state. It has two separate input variables, called S and R for *set* and *reset*, each of which can take on values 0 or 1. If the *set* input is equal to 1, then the flip-flop should change to state 1. If the *reset* input is equal to 1, then the flip-flop should change to state 0. If neither input is equal to 1, then it should stay in whatever state it's currently in. If both inputs are equal to 1, then you can do whatever you want to (it's supposed to be an illegal input condition). Let it start in state 0. Make and test an SR flip-flop.

Question 10: The Fibonacci numbers are the sequence $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$. Each number in the sequence is the sum of the previous two numbers. Make a state machine that takes no input, but generates the Fibonacci numbers as output.

After an hour working on state machines, please switch to the next section, even if you're not done, so you can get familiar with both ideas while we're all in lab.

Polynomial Class

We have already seen that 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]`. We have seen a couple of methods for evaluating a polynomial represented this way at a particular value for `x`. We are going to extend our ability to work with polynomials, by defining a polynomial class, and providing operations for performing algebraic operations on polynomials. It's good practice with object-oriented programming, and it will be useful to us in a couple of weeks.

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 the following. We suggest converting all of the coefficients to floats, to forestall any future problems with integer division.

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

The constructor accepts a list of coefficients, highest-order first, so 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]
```

Use a `__str__` method to make the polynomials print out nicely (see the course notes for more information about this). Limit the precision with which the coefficients are printed out to make it more readable. You can, for example, convert a number `n` to a string with 4 digits after the decimal place, using scientific notation if necessary, using this expression:

```
"%.5g" % n
```

Question 11: Define the basic parts of your class, with an `__init__` method and a `__str__` method.

Question 12: Implement the `add` method for your class. **Be sure that performing the operation `p1 + p2` *does not change* the original value of `p1` or `p2`.**

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. Now, you can also ask for the value of the polynomial at a particular x:

```
>>> p1.val(1)
6.0
>>> p1.val(10)
321.0
>>> (p1 + p2).val(10)
1521.0
```

Question 13: Add the `__add__` method to your class.

Question 14: Add the `val` method to your class, which evaluates the polynomial for the specified value. Use one of the rules we implemented in software lab 1.

Extend your `Polynomial` class to support multiplication.

```
>>> p1 = Polynomial([3, 2, 1])
>>> p2 = Polynomial([100, 200])
>>> print p1 * p1
poly[9.0, 12.0, 10.0, 4.0, 1.0]
>>> print p1 * p2
poly[300.0, 800.0, 500.0, 200.0]
>>> print (p1 * p1) + p1
poly[9.0, 12.0, 13.0, 6.0, 2.0]
```

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 the products of all the pairs of 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, 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, and to extend a list of numbers to a particular length by adding zeros at the front.

Question 15: Add the `mul` and `__mul__` methods to your `Polynomial` class. Show that they work, and that the results can be added and multiplied.

Now, add to your polynomial class the ability to find roots. Here's what we'd like you to do in the different cases:

- If the polynomial is linear, return the single real root.
- If it is quadratic, return both roots, whether or not they are complex.

- If it is higher order, then return one root, using Newton's method (with a complex guess, but returning a real number if the result is nearly real), as we did in software lab 2.

So, now, you might get something like:

```
>>> p1.roots()
[(-0.33333333333333331+0.47140452079103173j),
 (-0.33333333333333331-0.47140452079103173j)]
>>> p3 = Polynomial([3, 2, -1])
>>> p3.roots()
[0.33333333333333331, -1.0]
>>> (p1 * p2).roots()
Looking for a single root with Newton's method
-1.9999999999282596
```

Newton's method might fail to find a root; change `fixedPoint` so that it only runs for some maximum number of iterations, and then terminates even if it hasn't found a solution.

Question 16: Demonstrate your root finding for polynomials with 2, 3, and 4 coefficients. Show that, in the quadratic case, you can find both real and complex roots.

Writeup: Due at the beginning of your lab on February 26 or 27

Please hand in your solutions and test cases to all of the questions in this handout.