HIGHER ORDER FUNCTIONS 2

COMPUTER SCIENCE 61A

June 21, 2012

1 Warmup Questions

1. You already saw in lecture a method to check if a number is prime:

```
def is_prime(n):
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
```

This is a decent way of testing if a number is prime, but looping `k` all the way to `n` might be a bit cumbersome. As a little bonus question, can you think of a better place to stop?

Using the `is_prime` function, fill in the following procedure, which generates the n^{th} prime number. For example, the 2^{nd} prime number is 3, the 5^{th} prime number is 11, and so on.

```
def nth_prime(n):
```

2. Now, what if we wanted to generate a sequence of primes up to the n^{th} prime. What would be a simple way to do this?

3. The Fibonacci sequence is a famous sequence in mathematics where each term is generated by adding the two previous terms: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...$ Using a `while` loop, write a function that would find the n^{th} Fibonacci number. For example, the 4^{th} number would be 2 and the 6^{th} number would be 5.

   ```
   def nth_fibo(n):
   ```

2 Procedures

A procedure that manipulates other procedures as data is called a *higher order function* (HOF). For instance, a HOF can be a procedure that takes procedures as arguments, returns a procedure as its value, or both.

3 Procedures as Argument Values

Suppose we would like to square or double every natural number from 1 to `n` and print the result as we go. Using the functions `square` and `double`, each of which are functions that take one argument that do as their name imply, fill out the following:

```
def square_every_number(n):
```

```
def double_every_number(n):
```

Note that the only thing different about `square_every_number` and `double_every_number` is just what function we call on `n` when we print it. Wouldn't it be nice to generalize procedures of this form into something more convenient? When we pass in the number, couldn't we specify, also, what we want to do to each number $<$ n.

To do that, we can define a higher order procedure called `every`. `every` takes in the procedure you want to apply to each element as an argument, and applies it to n natural numbers starting from 1. So to write `square_every_number`, we can simply do:

```
def square_every_number(n):
    every(square, n)
```

Equivalently, to write `double_every_number`, we can write:

```
def double_every_number(n):
    every(double, n)
```

Note: These functions are not pure — as defined below, `every` will actually print values to the screen.

4 Questions

1. Now implement the function `every` that takes in a function `func` and a number `n`, and applies that function to the first n numbers from 1 and prints the result along the way:

   ```
   def every(func, n):
   ```

2. Similarly, implement the function `keep`, which takes in a function condition `cond` and a number `n`, and only prints a number from 1 to n to the screen if it fulfills the condition:

   ```
   def keep(cond, n):
   ```

5 Procedures as Return Values

This problem comes up often: write a procedure that, given something, **returns a function** that does something else. The key message — conveniently emphasized — is that your procedure is supposed to return a procedure. For now, we can do so by defining an internal function within our function definition and then returning the internal function.

```
def my_wicked_procedure(blah):
    def my_wicked_helper(more_blah):
        ...
    return my_wicked_helper
```

That is the common form for such problems but we will learn another way to do this shortly.

6 Moar Questions

1. Write a procedure `and_add_one` that takes a function `f` as an argument (such that `f` is a function of one argument). It should return a function that takes one argument, and does the same thing as `f`, except adds one to the result.

```
def and_add_one(f):
```

2. Write a procedure `and_add` that takes a function `f` and a number `n` as arguments. It should return a function that takes one argument, and does the same thing as the function argument, except adds n to the result.

```
def and_add(f, n):
```

3. Python represents a programming community, and for things to run smoothly, there are some standards to keep things consistent. The following is the recommended style for documentation so that collaboration with other python programmers becomes standard and easy. Write your code at the very end, using `accumulate` from homework:

```
def identity(x):
    return x

def lazy_accumulate(f, start, n, term):
    """
    Takes the same arguments as accumulate from homework and
    returns a function that takes a second integer m and
    will return the result of accumulating the first n
    numbers starting at 1 using f and combining that with
    the next m integers.

    Arguments:
    f - the function for the first set of numbers.
```

```
    start - the value to combine with the first value in
            the sequence.
    n - the stopping point for the first set of numbers.
    term - function to be applied to each number before
           combining.

    Returns:
    A function (call it h) h(m) where m is the number of
    additional values to combine.

    >>> # The following does
    >>> # (1 + 2 + 3 + 4 + 5) + (6 + 7 + 8 + 9 + 10)
    >>> lazy_accumulate(add, 0, 5, identity)(5)
    55
    """
```

7 Lambda Expressions

One way of returning functions is by using nested inner functions. But, what if the function you need is very short and will only be used in one particular situation? The solution would be the `lambda` expression. A `lambda` expression has the following syntax:

```
lambda <args> :  <body>
```

With this simple expression, you can define functions on the fly, without having to use `def` statements and without having to give them names. In other words, lambda expressions allow you to reate anonymous functions. There is a catch though: The body must be a single expression, which is also the return value of the function.

One other difference between using the `def` keyword and `lambda` expressions we would like to point out is that `def` is a *statement*, while `lambda` is an *expression*. Evaluating a `def` statement will have a side effect, namely it creates a new function binding in the current environment. On the other hand, evaluating a `lambda` expression will not change the environment unless we do something with the function created by the lambda. For instance, we could assign it to a variable or pass it as a function argument.

8 One last question

1. What would Python do?

```
>>> square = lambda x: x * x
>>> def double(f):
...     def doubler(x):
...         return f(f(x))
...     return doubler
>>> foo = double(square)
>>> foo(4)
```