HIGHER ORDER FUNCTIONS & ENVIRONMENT DIAGRAMS 1b

COMPUTER SCIENCE 61A

June 27, 2013

1 Warmup Questions

1. Describe what the following function does and how it works.

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

This function works, but looping `k` all the way to `n` is inefficient. Can you think of a better point to stop?

2. Fill in the following function, 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.

   Hint: you can use the function from question 1.

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

   What is a simple way to modify `nth_prime` so that it prints a *sequence of primes* up to the n^{th} prime?

3. The Fibonacci sequence is a famous sequence in mathematics where each term is generated by adding the two previous terms: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$. 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 Environment Diagrams

Environment diagrams will feature prominently in CS 61A, so here is one to try for practice. Environment diagrams can help you understand difficult coding problems, and also give you an idea of what's happening inside the interpreter.

Write the environment diagram for the following code:

```
>>> from operator import add
>>> def curry2(f):
...     def g(x):
...         def h(y):
...             return f(x,y)
...         return h
...     return g
>>> make_adder  = curry2(add)
>>> add_three = make_adder(3)
>>> five = add_three(2)
```

CS61A Summer 2013: Steven Tang and Eric Tzeng, with
Rohan Chitnis, Brian Hou, Andrew Huang, Robert Huang, Jeffrey Lu, Leonard Truong, Mark Miyashita, and Albert Wu

3 Higher-Order Functions

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

4 Functions as Argument Values

Suppose we want 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 one-argument functions that do as their names imply, fill out the following:

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

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

Note that the only difference between `square_every_number` and `double_every_number` is simply which function we call on the numbers. It would be nice to generalize functions of this form. When we pass in the number, couldn't we also specify the function we apply to each number less than `n`?

To do that, we define a higher-order function called `every`. `every` takes in the function you want to apply to each number, and applies it to n natural numbers, starting from 1. We can rewrite `square_every_number` and `double_every_number`:

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

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.

4.1 Questions

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

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

2. Similarly, implement the function `keep`, which takes in a function predicate `cond` and a number `n`, and only prints numbers from 1 to n to the screen if they fulfill `cond`:

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

5 Functions as Return Values

This problem comes up often: write a function that, given something, **returns a function** that does something else. To create this sort of higher-order function, we define another function *inside* the original function, and then return it:

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

5.1 Moar Questions

1. Write `and_add_one`, which takes a one-argument function `f`. `and_add_one` should return yet another one-argument function that returns 1 plus the result of calling `f`.

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

2. Write a function `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. The following code has been loaded into the Python interpreter:

```
def skipped(f):
    def g():
        return f
    return g

def composed(f, g):
    def h(x):
        return f(g(x))
    return h

def added(f, g):
    def h(x):
        return f(x) + g(x)
    return h

def square(x):
        return x*x

def two(x):
    return 2
```

What will Python output when the following lines are evaluated?

```
>>> composed(square, two)(7)


>>> skipped(added(square, two))()(3)


>>> composed(two, square)(2)
```

4. 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 square(x):
    return x * x

def lazy_accumulate(f, start, n, term):
    """Returns a one-argument function, h(m). h, when called,
    returns the first n + m terms of the sequence defined by
    TERM.

    PARAMETERS:
    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 h(m), where m is the number of additional
    values to combine.

    >>> # The following does
    >>> # 12 + (1*1 + 2*2 + 3*3) + (4*4 + 5*5)
    >>> lazy_accumulate(add, 12, 3, square)(2)
    67
    """
```