CONTROL AND HIGHER ORDER FUNCTIONS 2

COMPUTER SCIENCE 61A

June 26, 2014

1 Warmup Question

1. Draw the environment diagram for this code

```
bob = lambda f: f(x + 1)
x = 5
bob(lambda x: x + 1)
```

2. Draw the environment diagram for this code

```
n = 7
def f(x):
    return x + 3
def g(f, x):
    return f(f(x) * 2)
m = g(f, n)
```

2 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.

2.1 Functions 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 functions 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 function called `every`. `every` takes in the function 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.

2.2 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):
   ```

2.3 Functions as Return Values

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

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

For simple functions, we can also do so using lambda.

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

2.4 Moar Questions

1. Write a function `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 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):
    return lambda x: f(g(x))

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. Draw the environment diagram for this.

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

3 Domain and Range

In lecture we saw how to create pairs using functions and an if-else statement. In fact, it is even possible to create pairs using just functions! The mechanism for doing this can be confusing, but it is easiest to understand through domain and range. This is a midterm-level question that tests understanding of higher order functions and demonstrates why domain and range is so useful.

Here is the definition of `cons`:

```
def cons(x, y):
    return lambda m: m(x, y)
```

1. What can be inferred about the value of the variable `m`?

2. Suppose we create a pair as follows:

   ```
   pair = cons(1, 2)
   ```

   What is the type of the value of `pair`?

3. Describe as completely as possible the domain and range of `cons`.

4. Write the functions `car` and `cdr`, which get the first and second item of the pair respectively. Make sure you think about the domain and range requirements that you found in the previous questions. Both function can be defined with one line of code.

   ```
   >>> car(cons(1, 2))
   1
   >>> cdr(cons(1, 2))
   2
   ```