MIDTERM 1 REVIEW SHEET 1

COMPUTER SCIENCE 61A

September 15, 2012

All relevant midterm material can be found on the course website under Announcements, in the link to Midterm 1.

Disclaimer: This review session is not meant to be exhaustive. While the material covered during the session is important, there might be material that we can't get to that is equally important. For the exhaustive list of possible material for the exam, refer to the website.

0.1 Warmup

1. Figure out what this code is supposed to do and correct it:

```
>>> def nondescriptive_name(n):
...     k = 1
...     while k <= n:
...         if n % k == 0:
...             print(k, n//k)
...     k += 1
```

2. The factorial of a number n is the total product of all numbers before it, down to 1. That is, 7! is $7 * 6 * 5 * 4 * 3 * 2 * 1$. Write `prime_factorial` which only multiplies the prime numbers before n. That is, 7! is $7 * 5 * 3 * 2$. Assume you are given the function `is_prime`.

```
def prime_factorial(n):
```

1 What Would Python Do?

1. Recall `accumulate` from homework, which takes in `combiner, start n` and `term`.

```
>>> from operator import mul
>>> accumulate(mul, 1, 4, lambda x: pow(x, 2))
```

2. Recall the `compose` function from lecture, reproduced below. Assume that `square` and `cube` are given, and behave as expected.

```
>>> def compose(f, g):
...     def h(x):
...         return f(g(x))
...     return h

>>> compose(compose(print, square)), cube)(3)
```

3. It's a lambda free-for-all!

```
>>> tic = lambda num: lambda y: num + y
>>> tac, toe = 10, 20
>>> tic(tac)(toe)
```

2 Higher Order Functions

1. Try to figure out what Python would print at the interpreter:

```
>>> f = lambda x: lambda y: x + y
>>> f

>>> f(4)

>>> f(4)(5)

>>> f(4)(5)(6)
```

2. Define a function g such that the following returns 3:

```
>>> g(3)()
3

def g(n):
```

3. "Smoothing" a function is an important concept in signal processing. If `f` is a function and `dx` is some small number, then the smoothed version of `f` is the function whose value at a point `x` is the average of `f(x-dx)`, `f(x)` and `f(x+dx)`. Write a function `smooth` that takes as input a function `f` and a number `dx` and returns a function that computes the smoothed `f`.

```
def smooth(f, dx):
```

3 Environment Diagrams

1. Draw the environment diagram

```
>>> p, q = 3, 6
>>> def foo(f, p, q):
...     return f()
>>> foo(lambda: pow(p, q), 2, 4)
```

2. Draw the remaining environment diagram after the last call and fill in the blanks.

```
>>> from operator import add, sub
>>> def burrito(x, y):
...     return add(x, y)
...
>>> sandwich = (lambda f: lambda x, y: f(x, y))(add)
>>> add = sub
>>> burrito(3, 2)

>>> sandwich(3, 2)
```

4 Newton's Method and Iterative Improvement

1. In this problem, rather than finding zeros of functions, we will attempt to find the minima of functions. As we will see, both Newton's method and different iterative improvement method can be used to solve this problem. You can assume that your inputs are functions with exactly one minimum and no maximum (known as convex functions).

   Before we discuss a specific iterative improvement algorithm to find a local minimum, let's first consider the problem in general. What should be the terminating condition (i.e. the done function) be for this minimization problem?

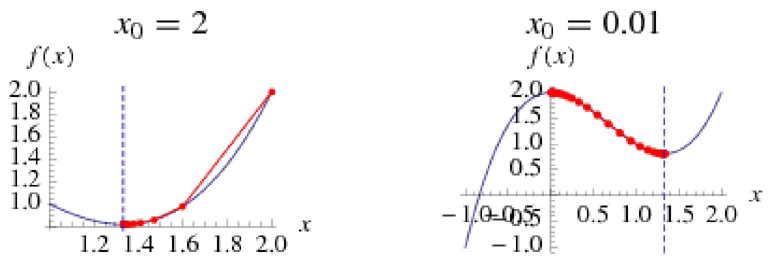
2. You learned in calculus that the minima of a function occur at zeros of the first derivative. Hey! We already know how to find derivatives and zeros! Here are the relevant functions:

```python
def iter_improve(update, done, guess=1, max_updates=1000):
    """Iteratively improve guess with update until
    done returns a true value.
    """
    k = 0
    while not done(guess) and k < max_updates:
        guess = update(guess)
        k = k + 1
    return guess

def approx_derivative(f, x, delta=1e-5):
    """Return an approximation to the derivative of f at x."""
    df = f(x + delta) - f(x)
    return df/delta

def find_root(f, guess=1):
    """Return a guess of a zero of the function f, near guess.
    """
    return iter_improve(newton_update(f), lambda x: f(x) == 0, guess)
```

   Use them to find the minimum of `f` by solving for the root of its derivative!

```python
def find_minimum(f, guess=1):
    """ Return a guess of a zero of the function f, near guess.
```

3. There is a very commonly used iterative improvement algorithm used to find the closest local minimum of a function, known as gradient descent. The idea behind gradient descent is to take a tiny step at each point towards the direction of the minimum. How the direction and size of this step is is a (small) constant factor of the derivative. Here is an illustration of what it does:



Since this is not a math class, we'll make your life easier by giving you the update equation, for some small α:

$$x' = x - \alpha f'(x)$$

Given all of this, implement the gradient descent algorithm by filling in the skeleton below. You may use the functions given in (2).

Hint: To check for termination, your current "guess" must remember the old value you were just at. Therefore, your guess should be a tuple of 2 numbers and you should update it like $(x, x') \to (x', x'')$.

```
def gradient_done(f, size = pow(10, 6)):
    """ Return a function to evaluate whether our answer is
        good enough, that is, when the change is less than size.
    """
```

```
def gradient_step(f, alpha=0.05):
    """ Return an update function for f using the
            gradient descent method.
    """
```

```
def gradient_descent(f, guess=1):
    """ Performs gradient descent on the function f,
        starting at guess"""
    guess = (None, guess)
```

5 Data Abstraction

1. Consider the following data abstraction for 2-element vectors:

```
def make_vect(xcor, ycor):
    return (xcor, ycor)
def xcor(vect):
    return vect[0]
def ycor(vect):
    return vect[1]
```

Change this definition of `vector_sum` to not have abstraction barrier violations.

```
def vector_sum(vect_tuple):
    xsum, ysum, i = 0, 0, 0
    while i < len(vect_tuple):  # len returns length of tuple
        xsum = vect_tuple[i][0] + xsum
        ysum = vect_tuple[i][1] + ysum
    return xsum, ysum
```

2. Suppose we redefine `make_vect` as follows:

```
def make_vect(xcor, ycor):
    vect = lambda f: f(xcor, ycor)
    return vect
```

What functions would we need to change? Go ahead and redefine them below.