CS 61A Recursion and Tree Recursion, HOFs
Spring 2019 Guerrilla Section 1: February 23, 2019

1 Recursion and Tree Recursion

Questions

1.1 What are three things you find in every recursive function?

1.2 When you write a Recursive function, you seem to call it before it has been fully defined. Why doesn't this break the Python interpreter?

1.3 The **domain** is the type of data that a function takes in as argument. The **range** is the type of data that a function returns. For example, the domain of the function **square** is numbers. The range is numbers.
Below is a Python function that computes the nth Fibonacci number. What's its domain and range? Also identify the three things it contains as a recursive function (from 1.1).

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

1.4 With the definition of the Fibonacci function above, draw out a diagram of the recursive calls made when **fib(4)** is called.

1.5 What does the following function **cascade2** do? What is its domain and range?

```
def cascade2(n):
    print(n)
    if n >= 10:
        cascade2(n//10)
        print(n)
```

1.6 What does each of the the following functions do?

```
def mystery(n):
    if n == 0:
        return 0
    else:
        return n + mystery(n - 1)
```

```
def foo(n):
    if n <= 1:
        return n
    return foo(n - 2) + foo(n - 1)
```

```
def fooply(n):
    if n < 0:
        return 0
    return foo(n) + fooply(n - 1)
```

2 Higher Order Functions

Questions

2.1 What do lambda expressions do? Can we write all functions as lambda expressions? In what cases are lambda expressions useful?

2.2 Determine if each of the following will error:

```
>>> 1/0
```

```
>>> boom = lambda: 1/0
```

```
>>> boom()
```

2.3 Express the following lambda expression using a **def** statement, and the **def** statement using a lambda expression.

```
pow = lambda x, y: x**y
```

```
def foo(x):
    def f(y):
        def g(z):
            return x + y * z
        return g
    return f
```

2.4 Draw Environment Diagrams for the following lines of code

```
square = lambda x: x * x
higher = lambda f: lambda y: f(f(y))
higher(square)(5)

a = (lambda f, a: f(a))(lambda b: b * b, 2)
```

2.5 Write **make_skipper**, which takes in a number n and outputs a function. When this function takes in a number x, it prints out all the numbers between 0 and x, skipping every nth number (meaning skip any value that is a multiple of n).

```
def make_skipper(n):
    """
    >>> a = make_skipper(2)
    >>> a(5)
    1
    3
    5
    """
```

2.6 Write **make_alternator** which takes in two functions, f and g, and outputs a function. When this function takes in a number x, it prints out all the numbers between 1 and x, applying the function f to every odd-indexed number and g to every even-indexed number before printing.

```
def make_alternator(f, g):
    """
    >>> a = make_alternator(lambda x: x * x, lambda x: x + 4)
    >>> a(5)
    1
    6
    9
    8
    25
    >>> b = make_alternator(lambda x: x * 2, lambda x: x + 2)
    >>> b(4)
    2
    4
    6
    6
    """
```