CS 61A Spring 2021

Recursion and Midterm Review

Lost 3: February 12, 2021

1 Learning Goals

- Understand the general idea behind recursion
- Understand how to structure recursive functions
- Understand the general structure of counting problems and how to solve them
- Understand how to approach exam-level problems for various topics

2 Recursion Overview

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

2.2 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?

2.3 Below is a Python function that computes the nth Fibonacci number. 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)
```

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

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

2.5 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)
```

3 Exam-Level Recursion + Lambda

3.1 **Fall 2016 Midterm 1, Question 5** An order 1 numeric function is a function that takes a number and returns a number. An order 2 numeric function is a function that takes a number and returns an order 1 numeric function. Likewise, an order n numeric function is a function that takes a number and returns an order $n-1$ numeric function. The argument sequence of a nested call expression is the sequence of all arguments in all subexpressions, in the order they appear. For example, the expressionf(3)(4)(5)(6)(7)has the argument sequence 3, 4, 5, 6, 7.

Implement multiadder, which takes a positive integer n and returns an order n numeric function that sums an argument sequence of length n.

```
def multiadder(n):
    """Return a function that takes N arguments, one at a time, and adds them.
    >>> f = multiadder(3)
    >>> f(5)(6)(7)                # 5 + 6 + 7
    18
    >>> multiadder(1)(5)
    5
    >>> multiadder(2)(5)(6)       # 5 + 6
    11
    >>> multiadder(4)(5)(6)(7)(8) # 5 + 6 + 7 + 8
    26
    """

    assert n > 0


    if ________________________________________________________________________:


        return _________________________________________________________________


    else:


        return _________________________________________________________________
```

Complete the expression below by writing one integer in each blank so that the whole expression evaluates to 2016. Assume multiadder is implemented correctly.

```
def compose1(f, g):
    """Return the composition function which given x, computes f(g(x)).

    >>> add_one = lambda x: x + 1        # adds one to x
    >>> square = lambda x: x**2
    >>> a1 = compose1(square, add_one)   # (x + 1)^2
    >>> a1(4)
    25
    >>> mul_three = lambda x: x * 3      # multiplies 3 to x
    >>> a2 = compose1(mul_three, a1)    # ((x + 1)^2) * 3
    >>> a2(4)
    75
    >>> a2(5)
    108
    """
    return lambda x: f(g(x))
```

```
compose1(multiadder(_______)(1000), multiadder(_______)(10)(_______))(1)(2)(3)
```

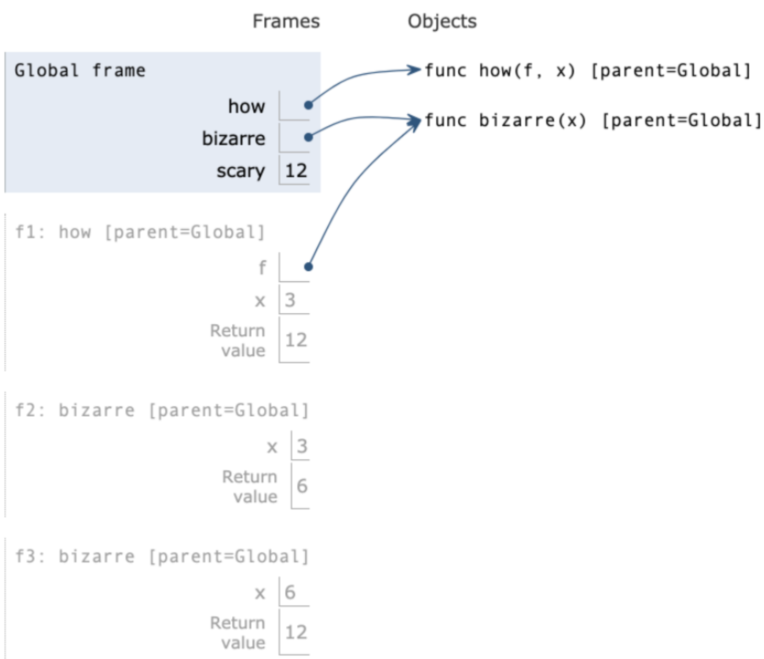
4 Reverse Environment Diagram Practice

4.1 Fill in the lines below so that the execution of the program would lead to the environment diagram below. You may not use any numbers in any blanks.

```
def how(f, x):
    return _________________

def bizarre(___):
    return 2 * _______

scary = ________(________, 3)
```



4.2 Fill in the lines below so that the execution of the program would lead to the environment diagram below. You may not use any numbers in any blanks.

```
def what(_______):
    def _______(x):
        return ________
    return __________

def who(n):
    def __________(k):
        return 2 * k + n
    return ___________

y = 3
_____(____(____))(4)
```