MIDTERM 1 REVIEW

COMPUTER SCIENCE MENTORS 61A

February 5, 2018 - February 8, 2018

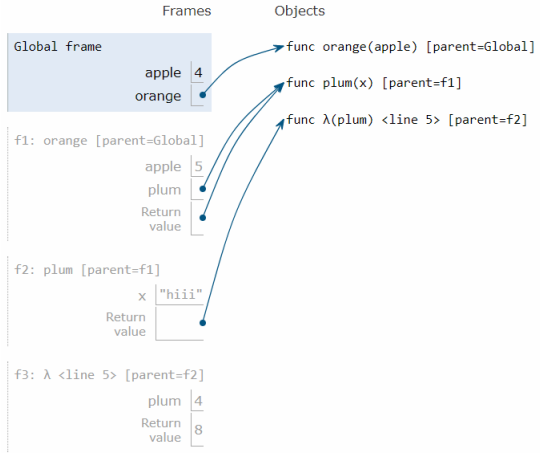
Environment Diagrams

1. Draw the environment diagram that results from running the code.

```
apple = 4
def orange(apple):
    apple = 5
    def plum(x):
        return lambda plum: plum * 2
    return plum

orange(apple)("hiii")(4)
```

Solution: Output: 3

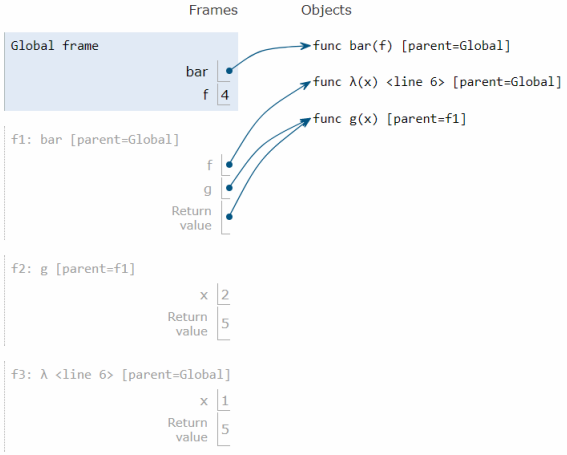


https://goo.gl/pZ0RLR

2. Draw the environment diagram that results from running the code.

```
def bar(f):
    def g(x):
        return f(x - 1)
    return g
f = 4
bar(lambda x: x + f)(2)
```

Solution: Output: 3



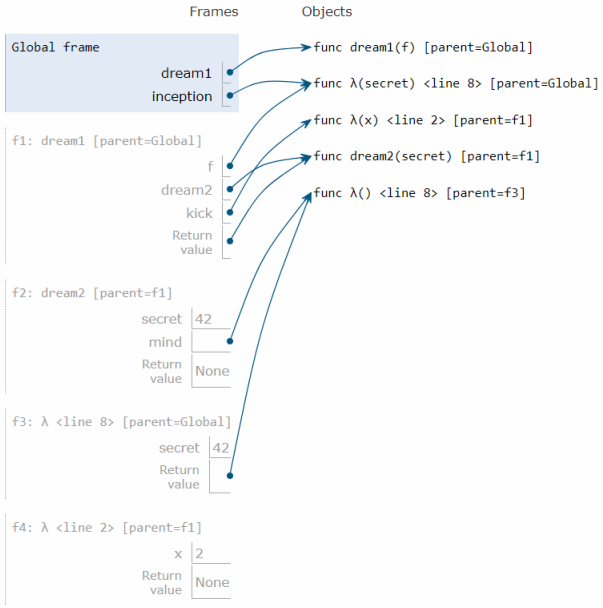
https://goo.gl/RtLZnV

3. Draw the environment diagram that results from running the code.

```
def dream1(f):
    kick = lambda x: mind()
    def dream2(secret):
        mind = f(secret)
        kick(2)
    return dream2

inception = lambda secret: lambda: secret
real = dream1(inception)(42)
```

Solution: Output: 3



https://goo.gl/q84uf4

Higher Order Functions

1. Write a higher-order function that passes the following doctests.

   Challenge: Write the function body in one line.

```
def mystery(f, x):
    """
    >>> from operator import add, mul
    >>> a = mystery(add, 3)
    >>> a(4) # add(3, 4)
    7
    >>> a(12)
    15
    >>> b = mystery(mul, 5)
    >>> b(7) # mul(5, 7)
    35
    >>> b(1)
    5
    >>> c = mystery(lambda x, y: x * x + y, 4)
    >>> c(5)
    21
    >>> c(7)
    23
    """
```

Solution:

```
    def helper(y):
        return f(x, y)
    return helper
```

Challenge solution:

```
    return lambda y : f(x, y)
```

2. What would Python display?

```
>>> foo = mystery(lambda a, b: a(b), lambda c: 5 + square(c))
>>> foo(-2)
```

Solution:
9

3. (Fall 2013 MT1 Q3D) The CS61A staff has developed a formula for determining what a fox might say. Given three strings, a start, a middle, and an end, a fox will say the start string, followed by the middle string repeated a number of times, followed by the end string. These parts are all separated by hyphens.

   Complete the definition of `fox_says`, which takes the three string parts of the fox's statement (`start`, `middle`, and `end`) and a positive integer `num` indicating how many times to repeat `middle`. It returns a string.

   You cannot use any **`for`** or **`while`** statements. Use recursion in `repeat`. Moreover, you cannot use string operations other than the + operator to concatenate strings together.

```
def fox_says(start, middle, end, num):
    """
    >>> fox_says('wa', 'pa', 'pow', 3)
    'wa-pa-pa-pa-pow'
    >>> fox_says('fraka', 'kaka', 'kow', 4)
    'fraka-kaka-kaka-kaka-kaka-kow'
    """
    def repeat(k):




    return start + '-' + repeat(num) + '-' + end
```

Solution:

```
def fox_says(start, middle, end, num):
    def repeat(k):
        if k == 1:
            return middle
        else:
            return middle + '-' + repeat(k - 1)
    return start + '-' + repeat(num) + '-' + end
```

4. Fill in the blanks (*without using any numbers in the first blank*) such that the entire expression evaluates to `9`.

Solution:

```
(lambda x: lambda y: lambda: y(x))(3)(lambda z: z*z)()
```

Recursion

1. (Spring 2015 MT1 Q3C) Implement the `combine` function, which takes a non-negative integer `n`, a two-argument function `f`, and a number `result`. It applies `f` to the first digit of `n` and the result of combining the rest of the digits of `n` by repeatedly applying `f` (see the doctests). If `n` has no digits (because it is zero), `combine` returns `result`.

```
def combine(n, f, result):
    """
    Combine the digits in non-negative integer n using f.

    >>> combine(3, mul, 2) # mul(3, 2)
    6
    >>> combine(43, mul, 2) # mul(4, mul(3, 2))
    24
    >>> combine(6502, add, 3) # add(6, add(5, add(0, add(2, 3)
       )))
    16
    >>> combine(239, pow, 0) # pow(2, pow(3, pow(9, 0))))
    8
    """
    if n == 0:
        return result
    else:
        return combine(_______ , _______ ,
           _____________________________)
```

Solution:

```
def combine(n, f, result):
    if n == 0:
        return result
    else:
        return combine(n // 10, f, f(n % 10, result))
```

2. James wants to print this week's discussion handouts for all the students in CS 61A. However, both printers are broken! The first printer only prints multiples of `n` pages, and the second printer only prints multiples of `m` pages. Help James figure out whether or not it's possible to print exactly `total` number of handouts!

```
def has_sum(total, n, m):
    """
    >>> has_sum(1, 3, 5)
    False
    >>> has_sum(5, 3, 5) # 0 * 3 + 1 * 5 = 5
    True
    >>> has_sum(11, 3, 5) # 2 * 3 + 1 * 5 = 11
    True
    """
    if _____________________________________________________:
        return _____________________________________
    elif _____________________________________________________:
        return _____________________________________
    return _____________________________________________________
```

Solution:

```
def has_sum(total, n, m):
    if total == 0: # (total == n or total == m) works too
       except when total equals 0
        return True
    elif total < 0: # (total < min(n1, n2)) works given
       alternate base case
        return False
    return has_sum(total - n, n, m) or has_sum(total - m, n
       , m)
```

3. The next day, the printers break down even more! Each time they are used, the first printer prints a random `x` copies $50 \leq x \leq 60$, and the second printer prints a random `y` copies $130 \leq y \leq 140$. James also relaxes his expectations: he's satisfied as long as there's at least `lower` copies so there are enough for everyone, but no more than `upper` copies to prevent waste.

```
def sum_range(lower, upper):
    """
    >>> sum_range(45, 60) # Printer 1 prints within this range
    True
    >>> sum_range(40, 55) # Printer 1 can print a number 56-60
    False
    >>> sum_range(170, 201) # Printer 1 + 2 will print between
        180 and 200 copies total
    True
    """
    def copies(pmin, pmax):
        if ____________________________________________________:
            return ____________________________________________
        elif ____________________________________________________:
            return ____________________________________________
        return ____________________________________________________
    return copies(0, 0)
```

Solution:

```
def sum_range(lower, upper):
    def copies(pmin, pmax):
        if lower <= pmin and pmax <= upper:
            return True
        elif upper < pmin:
            return False
        return copies(pmin + 50, pmax + 60) or copies(pmin
           + 130, pmax + 140)
    return copies(0, 0)
```