Higher-Order Environments, Currying, and Function Scope Meta

Computer Science Mentors 61A

February 6–February 10, 2023

Recommended Timeline

- HOFs mini-lecture/review - 5 mins
- Inception OR ABDE - 10 mins (check in with your students to see how they feel about the general structure of higher order functions; do this if they feel a bit shaky)
- Compound OR Partial Summer - 15 mins
- General recursion mini-lecture - 10 mins (Since this week is a bit weird, this will likely be your students' first interaction with recursion! Take more time on this if needed.)
- Wrong factorial - 5 to 10 mins
- All true - 10 mins

As a reminder, there is no expectation that you get through all problems in a section. Pick the most pertinent problems for your section.

Created by Alyssa Smith, Esther Shen, Maya Romero, Natalie Wei, Sandhya Ganesan, Kaelyn Huang, Cecilia Aiko, Aurelia Wang

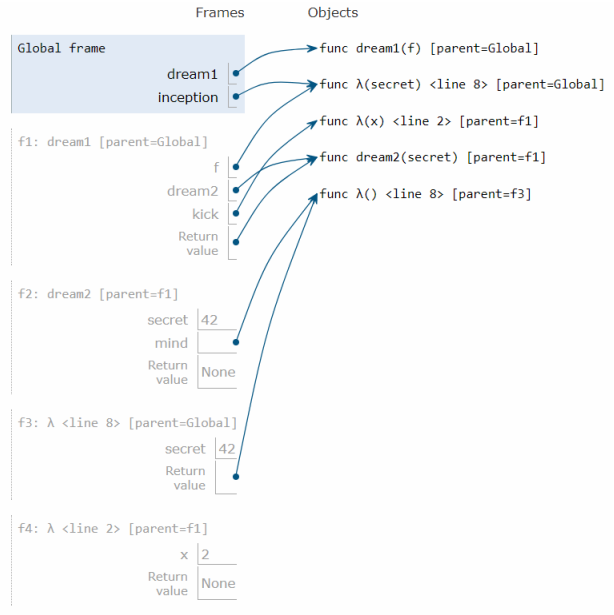
1 Higher-Order Functions cont.

1. 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)
```

Output: 3



https://goo.gl/q84uf4

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

```
def a(y):
    d = 1
    b = lambda x: y(x)
    e = lambda x: x(3)
    return e(b)
d = 5
a(lambda x: 4 - x + d)
```

https://goo.gl/9vxEwv

3. Implement `compound`, which takes in a single-argument function `base_func` and returns a two-argument compounder function `g`. The function `g` takes in an integer `x` and positive integer `n`.

   Each call to `g` will print the result of calling `f` repeatedly 0,1,.., n-1 times on `x`. That is, `g(x, 2)` prints `x`, then `f(x)`. Then, `g` will return the next two-argument compounder function.

```
def compound(base_func, prev_compound=lambda x: x):
    """
    >>> add_one = lambda x: x + 1
    >>> adder = compound(adder)
    >>> adder = adder(3, 2)
    3       # 3
    4       # f(3)
    >>> adder = adder(4, 4)
    6       # f(f(4))
    7       # f(f(f(4)))
    8       # f(f(f(f(4))))
    9       # f(f(f(f(f(4)))))
    """
    def g(x, n):
      new_comp = ____________________________________
      while n > 0:
        print(____________________________________)
        new_comp = (lambda save_comp: \
                    ____________________________)(_______________)
        ____________________________________________________________
      return ____________________________________________________
    return ____________________________________________
```

```
def compound(base_func, prev_compound=lambda x : x):
  def g(x, n):
    new_comp = prev_compound
    while n > 0:
      print(new_comp(x))
      new_comp = (lambda save_comp: \
                  lambda x: base_func(save_comp(x)))(new_comp)
      n -= 1
    return compound(base_func, new_comp)
  return g
```

4. Write a function `partial_summer`, which takes in a list of integers `lst` and returns a function. The returned function takes in a non-negative integer `n`. It prints a sum derived from the first `n` elements of `lst`: if element `X` is even, divide `X` by `2` before adding it to the sum, and if `X` is odd. add `1` to `X` before adding it to the sum. If `n > len(lst)`, then sum as many elements of `lst` as you can. After printing the sum, the returned function returns another function, that when called, will perform the same procedure on the remaining `len(lst) - n` elements of `lst`.

```
def partial_summer(lst):
    """
    >>> lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    >>> f = partial_summer(lst)(3)
    7 # 7 = (1+1) + (2/2) + (3+1)
    >>> g = f(4)
    19 # 19 = (4/2) + (5+1) + (6/2) + (7+1)
    >>> h = g(3)
    14 # 14 = (8/2) + (9+1)
    >>> i = h(1)
    0
    """
    def helper(n):

        total, i = _________, _________

        while ___________________ and ______________________:

            if ____________________________:

                total += _____________________________
            else:
                total += lst[i] + 1

            ____________________________________________
        print(total)

        return ____________________________________________
    return helper
```

```
def partial_summer(lst):
    def helper(n):
        total, i = 0, 0
        while i < n and i < len(lst):
            if lst[i] % 2 == 0:
                total += lst[i] // 2
            else:
                total += lst[i] + 1
            i += 1
        print(total)
        return partial_summer(lst[n:])
    return helper
```

Inception and ABDE are very similar in terms of the skills they test. If your students struggle on visualizing how higher-order functions work, focus on those for section, the consider moving onto the HOF challenge problems. Generally only choose either Compound or Partial Summer for your challenge problem, since they cover similar ground in terms of HOFs. These are on the harder side so also consider doing the rest of the worksheet and coming back.

There are three steps to writing a recursive function:

1. Create base case(s)
2. Reduce your problem to a smaller subproblem and call your function recursively to solve the smaller subproblem(s)
3. Use the subproblems' solutions as pieces to construct a larger problem's solution (This can happen in many layers!)

Real World Analogy for Recursion

Imagine that you're in line for boba, but the line is really long, so you want to know what position you're in. You decide to ask the person in front of you how many people are in front of them. That way, you can take their response and add 1 to it to find your place. Now, the person in front of you is faced with the same problem that you were trying to solve, with one less person in front of them than you. They decide to take the same approach that you did by asking the person in front of them. This continues until the very first person in line is asked. At this point, the person at the front knows that there are 0 people in front of them, so they can tell the person behind them that there are 0 people in front. Now, the second person can figure out that there is 1 person in front of them, and can relay that back to the person behind them, and so on, until the answer reaches you.

Looking at this example, we see that we have broken down the problem of "how many people are there in front of me?" to 1 + "how many people are there in front of the person in front of me"? This problem will terminate with the person at the front of the line (with 0 people in front of them). Putting this into more formal terms, we are breaking down the problem into a **recurrence relationship**, and the termination case (when the question gets to the very first person in line) is called the **base case**.

Teaching Tips

1. Base Case - What is the simplest case? Or in what case do you want your recursion to stop? It's helpful to use edge cases to nudge students if they get stuck.
2. Break the problem down into smaller problems (Try to address this in terms of each specific problem, then extrapolate for general understanding)
   - What do you need to do to reach your base case?
   - For example: in factorial (usually seen in lecture), we have to subtract by one each time we do a recursive call
3. Solve the smaller problem recursively

- How would you use the solution to the smaller problem to write a solution to the original problem?
- "Recursive Leap of Faith"—When writing the recursive statement, assume the function works as intended for the smaller problems. Trust. (Abstraction! woohoo!)
- If you don't know what the recursive call needs to be, you can take an educated guess and see what happens.
- It's often extremely helpful to run line by line through a doctest to test both a tentative solution and your understanding to a problem.
- When running through a problem, it's often helpful to find a way to visualize the recursion in action! For shorter problems, one way you can do this is through drawing a stack of "boxes," each containing the result of a recursive call inside them, going all the way until the base case. Other linear visualizations work also!

We tend to throw around the term "recursive leap of faith" a lot, and I think that it confuses students. The "recursive leap of faith" is not synonymous with "the recursive call is correct". Rather it's a specific assumption we make while writing a recursive function that the recursive calls we make produce the correct output, even if we're not done writing our function. That is, the function we're writing works even if we're not done writing it. The fact that recursive calls return the correct value in the completed function is a mathematical fact that does not require any faith, so you should not conflate the two. Recursion is not magic; it is math.

The recursive leap of faith is essentially the scaffolding we need to help us build the recursive function. We pour the concrete for the base case and then layer our recursive logic on top of that until we have a sturdy structure, using the leap of faith's scaffolding to help us, as fallible human builders, to figure out the right way for the pieces to fit together. Once we're done, we can remove the scaffolding, but our tower still

5. What is wrong with the following function? How can we fix it?

```
def factorial(n):
    return n * factorial(n)
```

There is no base case and the recursive call is made on the same `n`.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

stands strong and sturdy.

Great time to remind your students that the return type of the base c
same type as the function (i.e. if the function returns an int, the base cas
return an int)

6. Complete the definition for `all_true`, which takes in a list `lst` and returns `True` if there are no False-y values in the list and `False` otherwise. Make sure that your implementation is recursive.

```
def all_true(lst):
    """
    >>> all_true([True, 1, "True"])
    True
    >>> all_true([1, 0, 1])
    False
    >>> all_true([])
    True
    """

    if not lst:
        return True
    elif not lst[0]:
        return False
    else:
        return all_true(lst[1:])
```