

# HIGHER-ORDER FUNCTIONS & ENVIRONMENT DIAGRAMS

---

## COMPUTER SCIENCE MENTORS 61A

February 3 – February 7, 2025

### 1 Environment Diagrams

---

1. Give the environment diagram and console output that result from running the following code.

```
def swap(x, y):  
    x, y = y, x  
    return print("Swapped!", x, y)
```

```
x, y = 60, 1  
a = swap(x, y)  
swap(a, y)
```

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

```
def funny(joke):  
    hoax = joke + 1  
    return funny(hoax)  
  
def sad(joke):  
    hoax = joke - 1  
    return hoax + hoax  
  
funny, sad = sad, funny  
result = funny(sad(2))
```

## 2 Higher-Order Functions

---

1. What are higher-order functions? Why and where do we use lambda and higher-order functions? Can you give a practical example of where we would use a HOF?

2. Give the environment diagram and console output that result from running the following code.

```
x = 20
def foo(y):
    x = 5
    if y == 5:
        return lambda y: x + y
    else:
        print('hello!')

y = foo(5)
x = y(7)
z = foo(7)
```

3. Implement `compose`, a function which takes in two functions `f` and `g`, both of which take in one argument each. `compose` returns a function which can take one argument as well. When this returned function is called with an argument, `f(g(x))` is returned.

```
def compose(f, g):
    """
    >>> a = compose(lambda x: x * x, lambda x: x + 4)
    >>> a(2)
    36
    """
    _____ lambda _____
```

4. Write a function, `whole_sum`, which takes in an integer, `n`. It returns another function which takes in an integer, and returns `True` if the digits of that integer sum to `n` and `False` otherwise.

```
def whole_sum(n):  
    """  
    >>> whole_sum(21) (777)  
    True  
    >>> whole_sum(142) (10010101010)  
    False  
    """  
    def check(x):  
  
        _____  
  
        while _____:  
  
            last = _____  
  
            _____  
  
            _____  
  
        return _____  
  
return _____
```

5. Implement `make_alternator` which takes in two functions `f` and `g` and outputs a function. The returned function takes in a number `x` and the function goes through the numbers in the sequence  $\{1, 2, 3, \dots, n\}$  in ascending order; for each number in the sequence the function applies `f` if the number is odd and `g` if the number is even and then prints the result of applying `f` or `g` and moves on to the next number in the sequence.

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

    def alternator(x):
        _____
        _____ i _____:
            if _____:
                print(_____)
            _____:
                _____
            _____
    _____
```

6. Write a function, `curry_forever`, which takes in a two-argument function, `f`, and an integer, `arg_num`. It returns another function that helps in calling `f` `arg_num` number of times on input provided to this returned function.

```
def curry_forever(f, arg_num, base=0):  
    """  
    >>> g = curry_forever(lambda x, y: x + y, 4)  
    >>> g(1)(2)(3)(4) # 1 + 2 + 3 + 4  
    10  
    """
```

```
    def helper(arg_num, amt):
```

```
        if arg_num == 0:
```

```
            _____  
  
            return _____  
  
            _____
```