HIGHER-ORDER FUNCTIONS & ENVIRONMENT DIAGRAMS Solutions

COMPUTER SCIENCE MENTORS 61A

February 3 – February 7, 2025
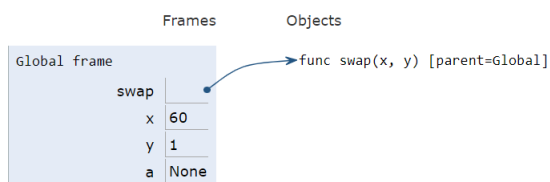
1 Environment Diagrams

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

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

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

Print output (drag lower right corner to resize)
```
Swapped! 1 60
Swapped! 1 None
```

```
Frames              Objects

Global frame                        → func swap(x, y) [parent=Global]
        swap  •
          x  60
          y  1
          a  None
```

```
f1: swap [parent=Global]
                     x  1
                     y  60
              Return
              value   None

f2: swap [parent=Global]
                     x  1
                     y  None
              Return
              value   None
```
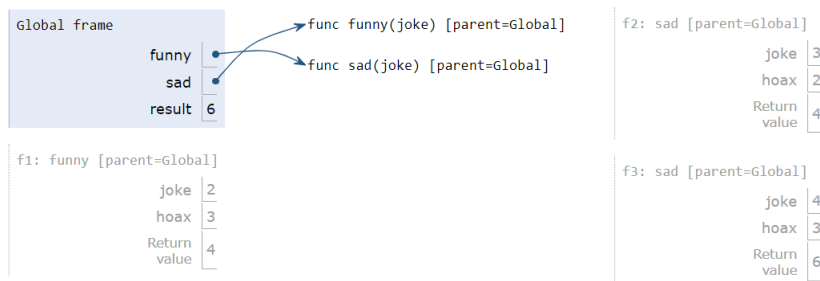
https://tinyurl.com/y68m6qdj

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

```python
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))
```


https://tinyurl.com/y5lc4fez

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?

Higher-order functions are functions that does at least one of the following: take at least one or more functions as arguments and returns a function. In practice, we use lambda functions to pass code as data in a concise manner. Lambda functions can be passed as arguments to higher-order functions. Higher order functions serve as a tool of abstraction, allowing us to simplify repeated actions into one function that we can use over and over again. Students can have varying answers for practice uses of HOFs, though here are some suggestions for the average student coming across this worksheet:

- Our method signature is composed of one parameter, but we wish to use a higher order function with more parameters to abstract extra steps.

- When our function is long and complex; easier to read code when it's organized into several different higher order functions.

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)
```


https://tinyurl.com/4dkbpnyc
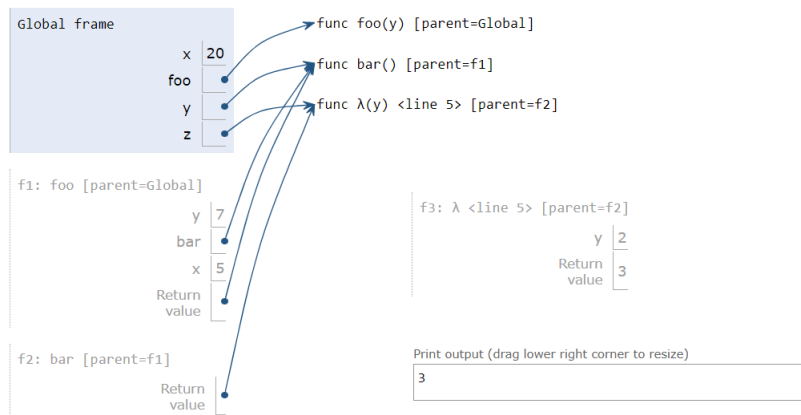
3. Implement `compose`, a function which takes in two functions `f` and `g`, both of which take in one argument each. `compose` returns a function which can take one argument as well. When this returned function is called with an argument, `f(g(x))` is returned.

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

    _____ lambda _____


    return lambda x: f(g(x))
```

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 _____
```

```
def whole_sum(n):
    def check(x):
        total = 0
        while x > 0:
            last = x % 10
            x = x // 10
            total += last
        return total == n
    return check
```

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, ... 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
    """

    ____ alternator(x):

        _____

        _____ i _____:

            if _____:

                print(____)

            _____:

                _____

        _____

    _____
```

```
def alternator(x):
    i = 1
    while i <= x:
        if i % 2 == 1:
            print(f(i))
        else:
            print(g(i))
        i += 1
return alternator
```

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.

```python
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 _____

    _____
```

```python
def curry_forever(f, arg_num, base=0):
    def helper(arg_num, amt):
        if arg_num == 0:
            return amt
        return lambda x: helper(arg_num - 1, f(amt, x))
    return helper(arg_num, base)
```