CS 61A Higher Order Functions and Recursion Fall 2018

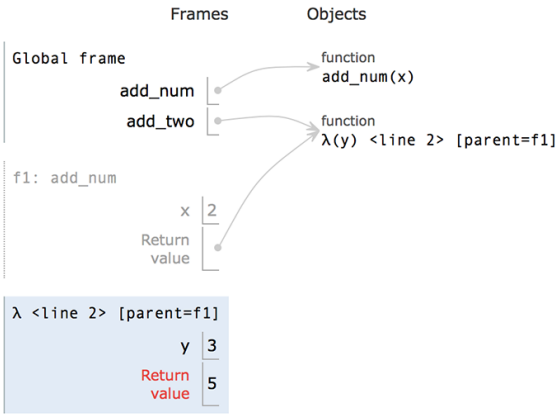
Discussion 2: June 26, 2018 Solutions

1 Higher Order Functions

HOFs in Environment Diagrams

Recall that an **environment diagram** keeps track of all the variables that have been defined and the values they are bound to. However, values are not necessarily only integers and strings. Environment diagrams can model more complex programs that utilize higher order functions.

```
def add_num(x):
    return lambda y: x + y

add_two = add_num(2)
add_two(3)
```



Lambdas are represented similiarly to functions in environment diagrams, but since they lack instrinsic names, the lambda symbol (λ) is used instead. The parent of any function (including lambdas) is always the frame in which the function is defined. It is useful to include the parent in environment diagrams in order to find variables that are not defined in the current frame. In the previous example, when we call `add_two` (which is really the lambda function), we need to know what `x` is in order to compute `x + y`. Since `x` is not in the frame `f2`, we look at the frame's parent, which is `f1`. There, we find `x` is bound to `2`.

As illustrated above, higher order functions that return a function have their return value represented with a pointer to the function object.

A Note on Lambda Expressions

A lambda expression evaluates to a function, called a lambda function. In the code above, `lambda y: x + y` is a lambda expression, and can be read as a function that takes in one parameter `y` and returns `x + y`.

A lambda expression by itself evaluates to a function but does not bind it to a name. Also note that the return expression of this function is not evaluated until the lambda is called. This is similar to how defining a new function using a def statement does not execute the functions body until it is later called.

```
>>> what = lambda x : x + 5
>>> what
<function <lambda> at 0xf3f490>
```

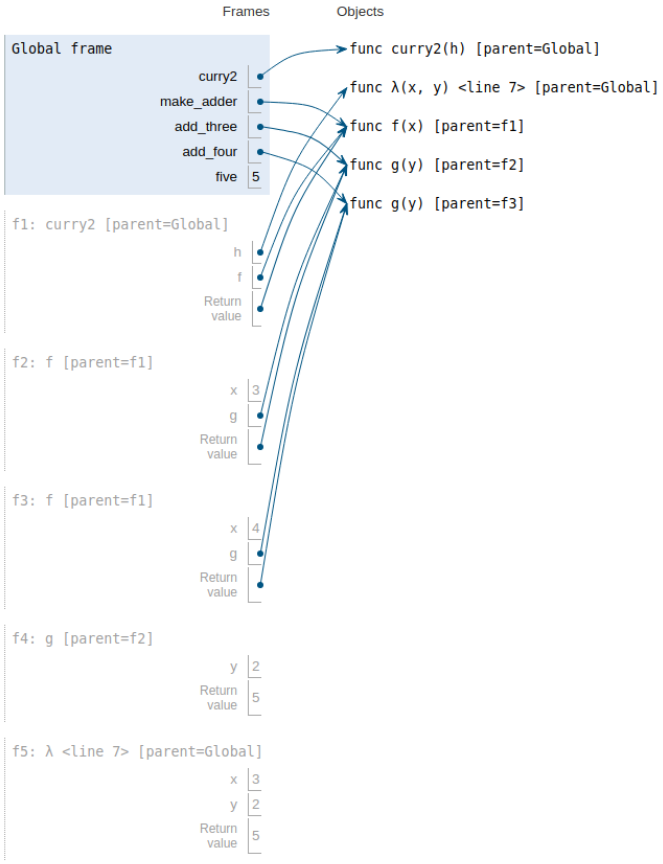
Unlike `def` statements, lambda expressions can be used as an operator or an operand to a call expression. This is because they are simply one-line expressions that evaluate to functions.

```
>>> (lambda y: y + 5)(4)
9
>>> (lambda f, x: f(x))(lambda y: y + 1, 10)
11
```

Questions

1.1 Draw the environment diagram that results from executing the code below.

```
1  def curry2(h):
2          def f(x):
3                  def g(y):
4                          return h(x, y)
5                  return g
6          return f
7  make_adder = curry2(lambda x, y: x + y)
8  add_three = make_adder(3)
9  add_four = make_adder(4)
10 five = add_three(2)
```

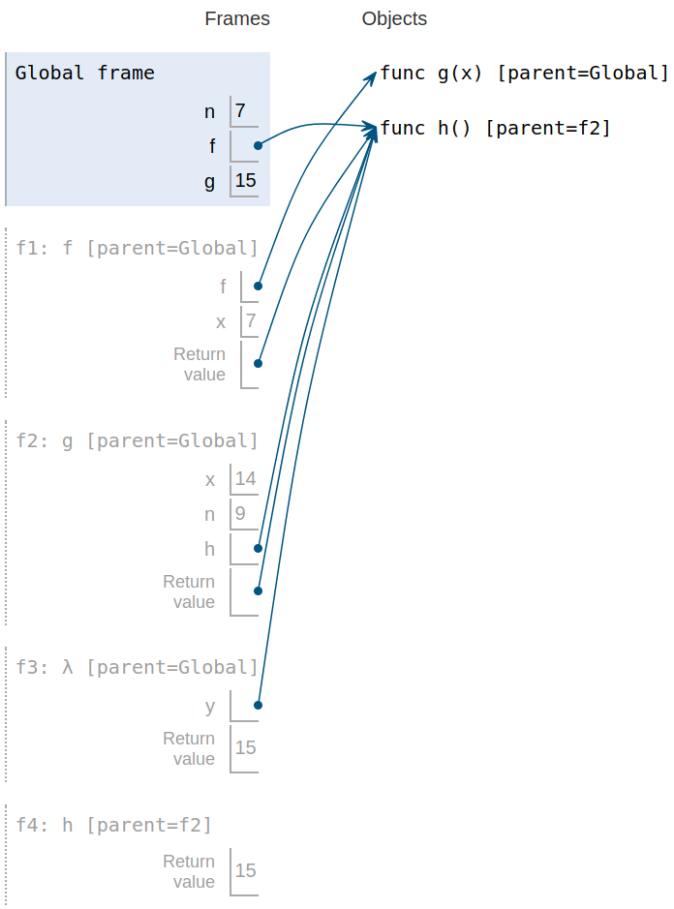


1.2 Write `curry2` as a lambda function

```
curry2 = lambda h: lambda x: lambda y: h(x, y)
```

1.3 Draw the environment diagram that results from executing the code below.

```
n = 7

def f(x):
    n = 8
    return x + 1

def g(x):
    n = 9
    def h():
        return x + 1
    return h

def f(f, x):
    return f(x + n)

f = f(g, n)
g = (lambda y: y())(f)
```



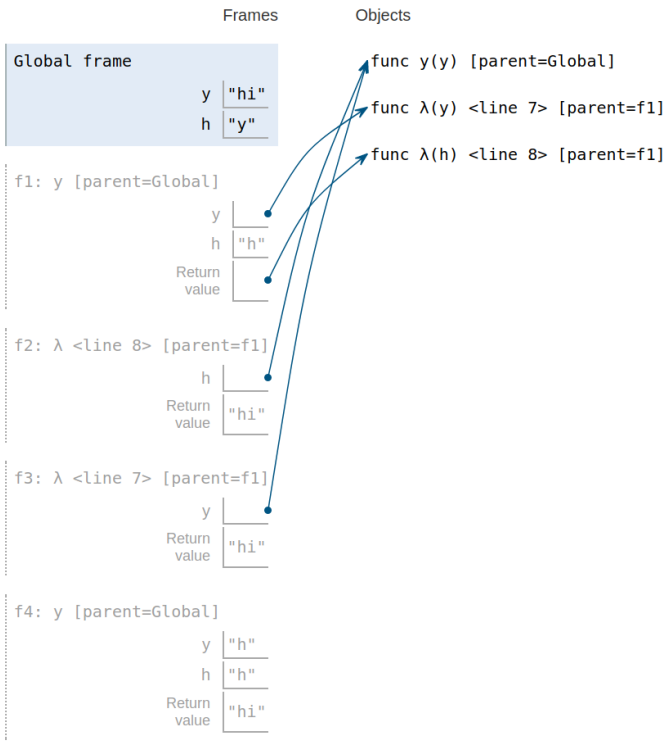
Video Walkthrough

1.4 *The following question is extremely difficult. Something like this would not appear on the exam. Nonetheless, it's a fun problem to try.*

Draw the environment diagram that results from executing the code below.

Note that using the `+` operator with two strings results in the second string being appended to the first. For example `"C" + "S"` concatenates the two strings into one string `"CS"`

```
y = "y"
h = y
def y(y):
    h = "h"
    if y == h:
        return y + "i"
    y = lambda y: y(h)
    return lambda h: y(h)
y = y(y)(y)
```



Video walkthrough

Writing Higher Order Functions

1.5 Write a function that takes in a function `cond` and a number `n` and prints numbers from `1` to `n` where calling `cond` on that number returns `True`.

```
def keep_ints(cond, n):
    """Print out all integers 1..i..n where cond(i) is true

    >>> def is_even(x):
    ...     # Even numbers have remainder 0 when divided by 2.
    ...     return x % 2 == 0
    >>> keep_ints(is_even, 5)
    2
    4
    """


    i = 1
    while i <= n:
        if cond(i):
            print(i)
        i += 1
```

Video walkthrough

1.6 Write a function similar to `keep_ints` like before, but now it takes in a number `n` and returns a function that has one parameter `cond`. The returned function prints out numbers from 1 to `n` where calling `cond` on that number returns `True`.

```
def keep_ints(n):
    """Returns a function which takes one parameter cond and prints out
    all integers 1..i..n where calling cond(i) returns True.

    >>> def is_even(x):
    ...     # Even numbers have remainder 0 when divided by 2.
    ...     return x % 2 == 0
    >>> keep_ints(5)(is_even)
    2
    4
    """


    def do_keep(cond):
        i = 1
        while i <= n:
            if cond(i):
                print(i)
            i += 1
    return do_keep
```

Video Walkthrough

2 Recursion

A *recursive* function is a function that is defined in terms of itself. A good example is the `factorial` function. Although we haven't finished defining `factorial`, we are still able to call it since the function body is not evaluated until the function is called. Note that when `n` is `0` or `1`, we just return `1`. This is known as the *base case*, and it prevents the function from infinitely recursing. Now we can compute `factorial(2)` in terms of `factorial(1)`, and `factorial(3)` in terms of `factorial(2)`, and `factorial(4)` – well, you get the idea.

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

There are **three** common steps in a recursive definition:

1. **Figure out your base case**: The base case is usually the simplest input possible to the function. For example, `factorial(0)` is 1 by definition. You can also think of a base case as a stopping condition

for the recursion. If you can't figure this out right away, move on to the recursive case and try to figure out the point at which we can't reduce the problem any further.

2. **Make a recursive call with a simpler argument**: Simplify your problem, and assume that a recursive call for this new problem will simply work. This is called the "leap of faith". For `factorial`, we reduce the problem by calling `factorial(n-1)`.

3. **Use your recursive call to solve the full problem**: Remember that we are assuming the recursive call works. With the result of the recursive call, how can you solve the original problem you were asked? For `factorial`, we just multiply $(n-1)!$ by n.

Note: One way to go understand recursion is to separate out two things: "internal correctness" and not running forever (known as "halting").

A recursive function is internally correct if it is always does the right thing assuming that every recursive call does the right thing. For example, the same factorial function from above but with no base case is internally correct, but does not halt.

A recursive function is correct if and only if it is both internally correct and halts; but you can check each property separately. The "recursive leap of faith" is temporarily placing yourself in a mindset where you only check internal correctness.

Questions

2.1 Write a function that takes two numbers `m` and `n` and returns their product. Assume `m` and `n` are positive integers. **Use recursion**, not `mul` or `*`!

Hint: `5*3 = 5 + 5*2 = 5 + 5 + 5*1`.

For the base case, what is the simplest possible input for `multiply`?

If one of the inputs is one, you simply return the other input.

For the recursive case, what does calling `multiply(m - 1, n)` do? What does calling `multiply(m, n - 1)` do? Do we prefer one over the other?

The first call will calculate a value that is `m` less than the total, while the second will calculate a value that is `n` less.

Either recursive call will work, but only `multiply(m, n - 1)` is needed.

```
def multiply(m, n):
    """
    >>> multiply(5, 3)
    15
    """


    if n == 1:
        return m
    else:
        return m + multiply(m, n - 1)
```

Video walkthrough

2.2 Write a recursive function that takes in an integer `n` and prints out a countdown from `n` to 1.

First, think about a base case for the `countdown` function. What is the simplest input the problem could be given?

When `n` equals `0`

After you've thought of a base case, think about a recursive call with a smaller argument that approches the base case. What happens if you call `countdown(n - 1)`?

A countdown starting from `n - 1` is printed.

Then, put the base case and the recursive call together, and think about where a print statement would be needed.

```
def countdown(n):
    """
    >>> countdown(3)
    3
    2
    1
    """


    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

Video walkthrough

2.3 How can we change `countdown` to count up instead without modifying a lot of the code?

Move the `print` statement to after the recursive call.

Video walkthrough

2.4 Write a recursive function that takes a number `n` and returns the sum of every other digit, starting from the rightmost digit. Assume `n` is non-negative.

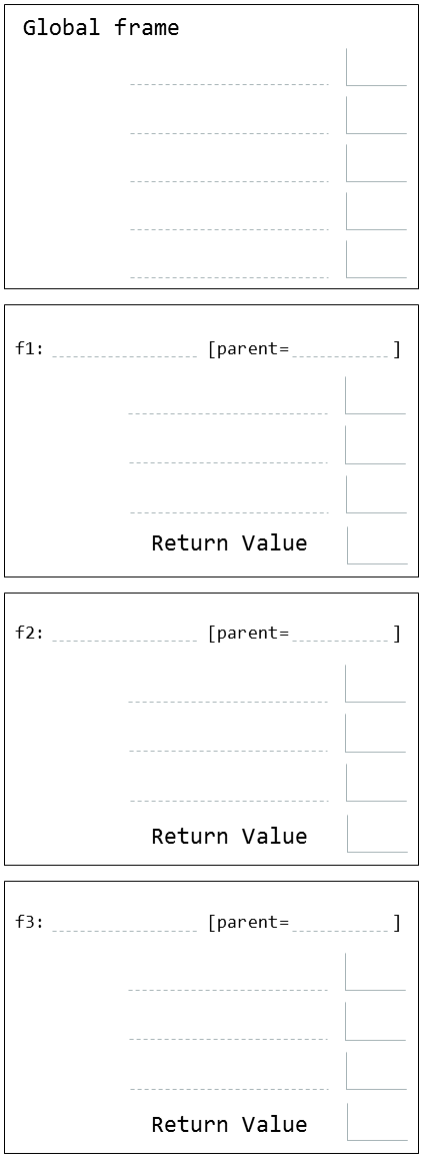
You might find the operators `//` and `%` useful.

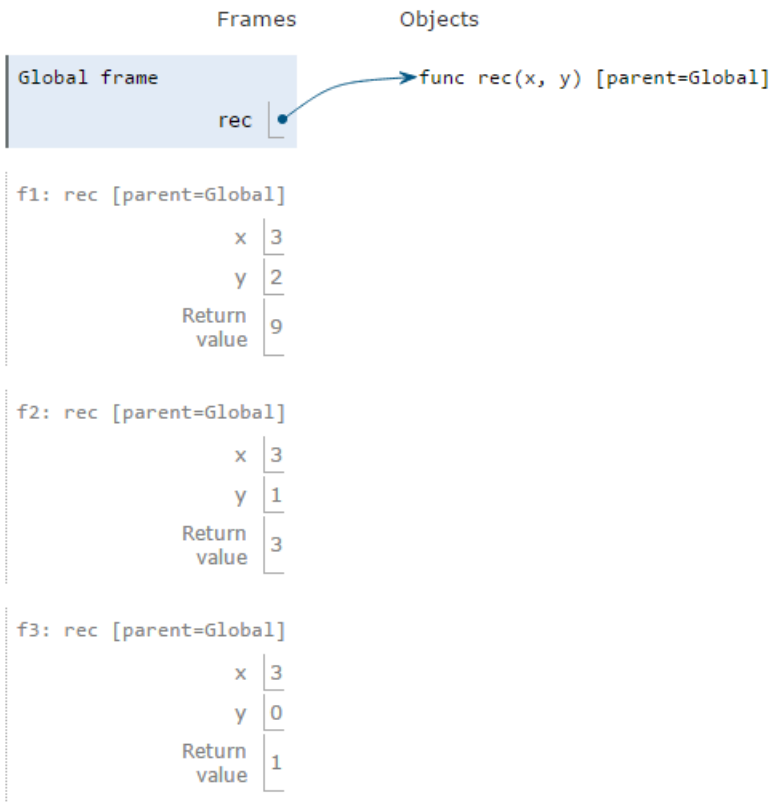
```
def sum_every_other_digit(n):
    """
    >>> sum_every_other_digit(7)
    7
    >>> sum_every_other_digit(30)
    0
    >>> sum_every_other_digit(228)
    10
    >>> sum_every_other_digit(123456)
    12
    >>> sum_every_other_digit(1234567) # 1 + 3 + 5 + 7
    16
    """


    if n == 0:
        return 0
    else:
        return n % 10 + sum_every_other_digit(n // 100)
```

2.5 Draw an environment diagram for the following code:

```
def rec(x, y):
    if y > 0:
        return x * rec(x, y - 1)
    return 1
rec(3, 2)
```

Bonus question: what does this function do?





This function computes `x ** y`.

Video Walkthrough