Environment Diagrams and Recursion 2

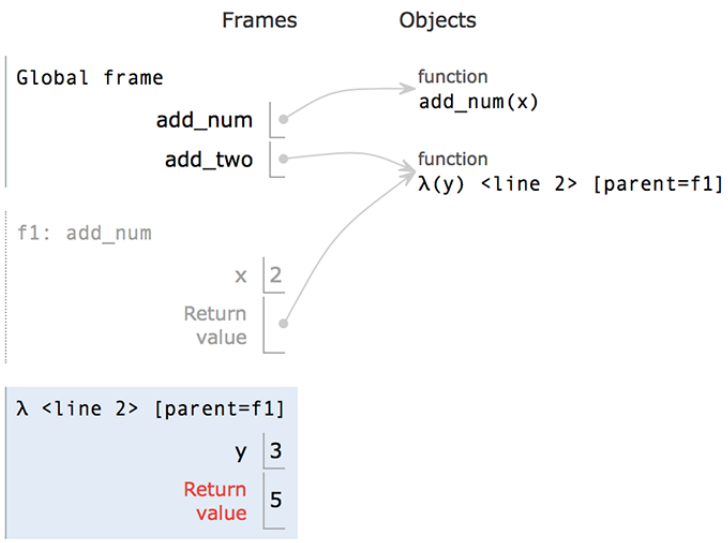
Computer Science 61A

September 8, 2016

1 More 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.

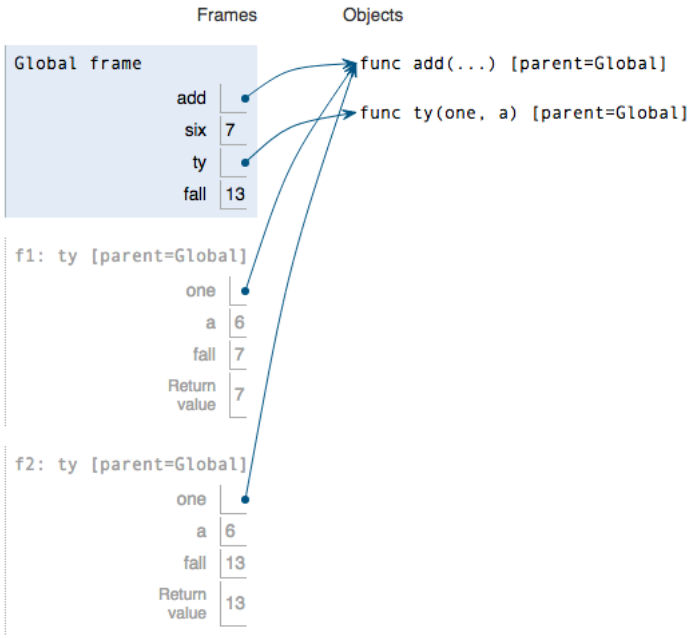
The parent of any function (including lambdas) is always the frame in which the function is defined. It is useful to include the parent in environment diagrams in order to find variables that are not defined in the current frame. In the previous example, when we call `add_two` (which is really the lambda 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.

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

```
from operator import add

six = 1

def ty(one, a):
    fall = one(a, six)
    return fall

six = ty(add, 6)
fall = ty(add, 6)
```

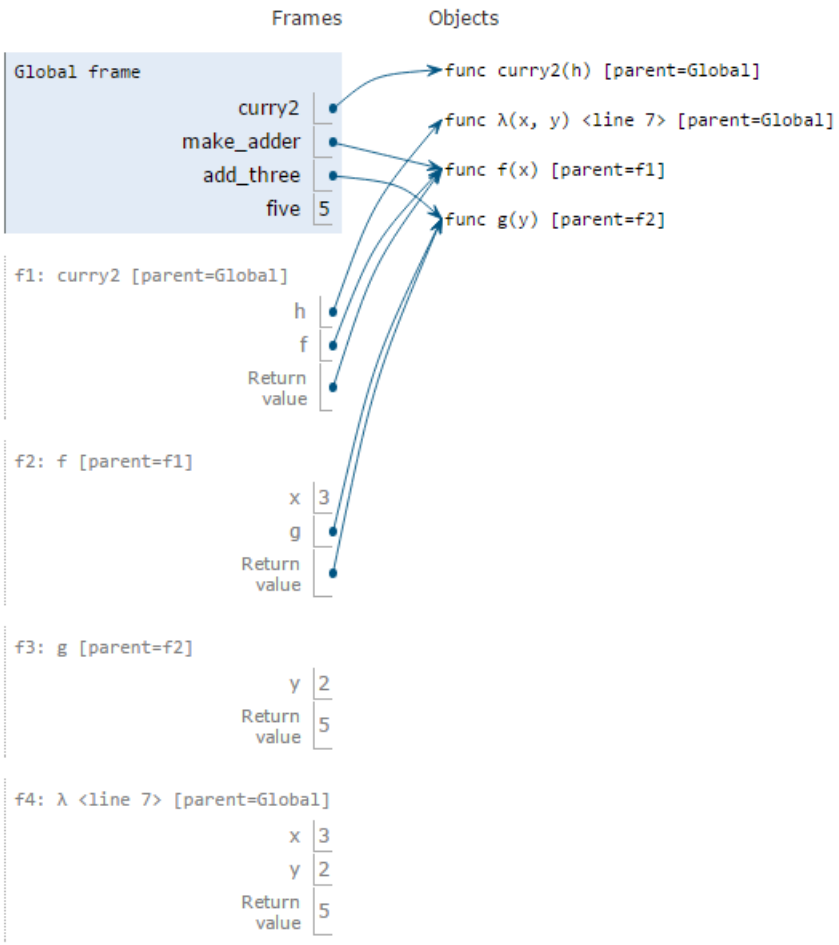
Solution:



2. Draw the environment diagram for the following code:

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f

make_adder = curry2(lambda x, y: x + y)
add_three = make_adder(3)
five = add_three(2)
```

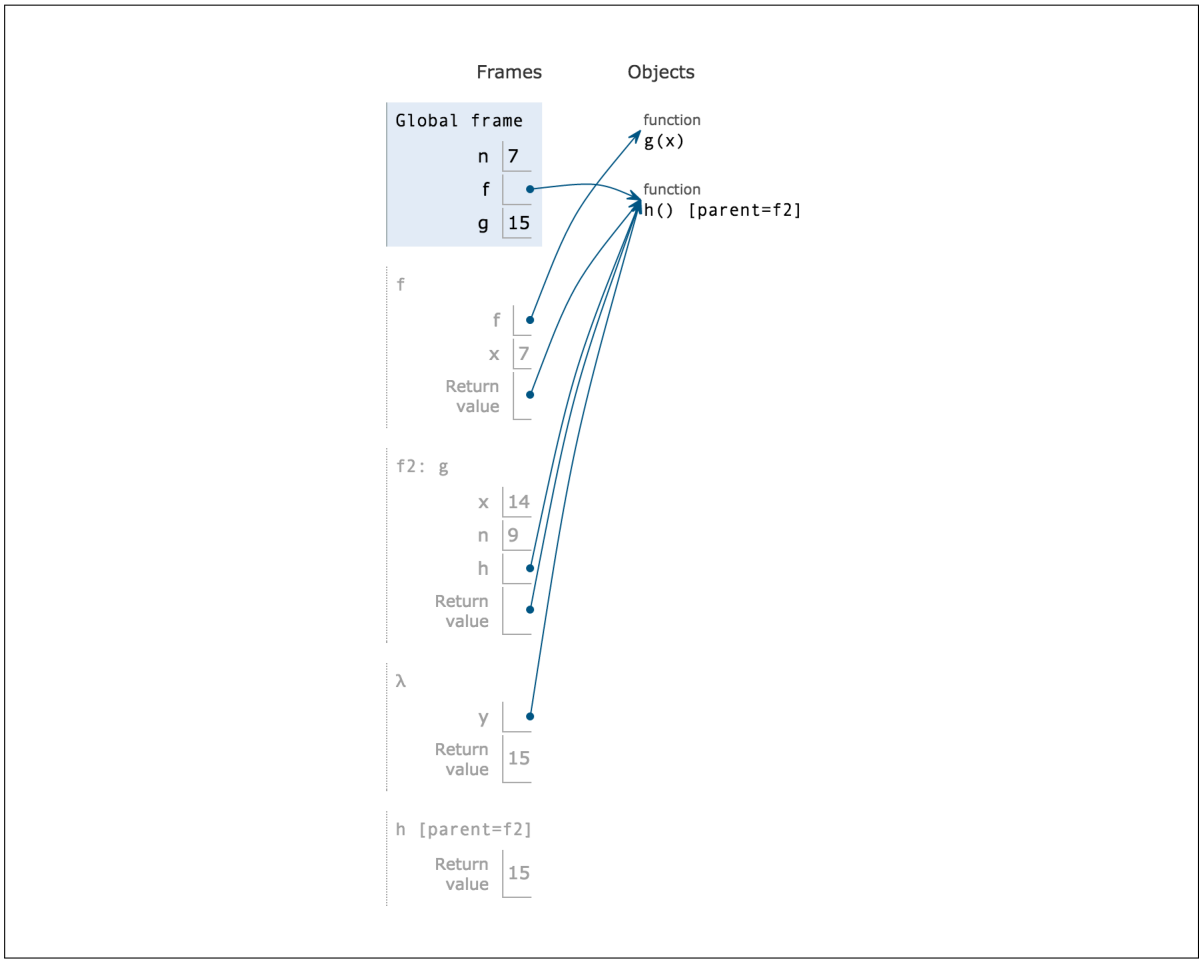
Solution:



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

```
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)
```

Solution:

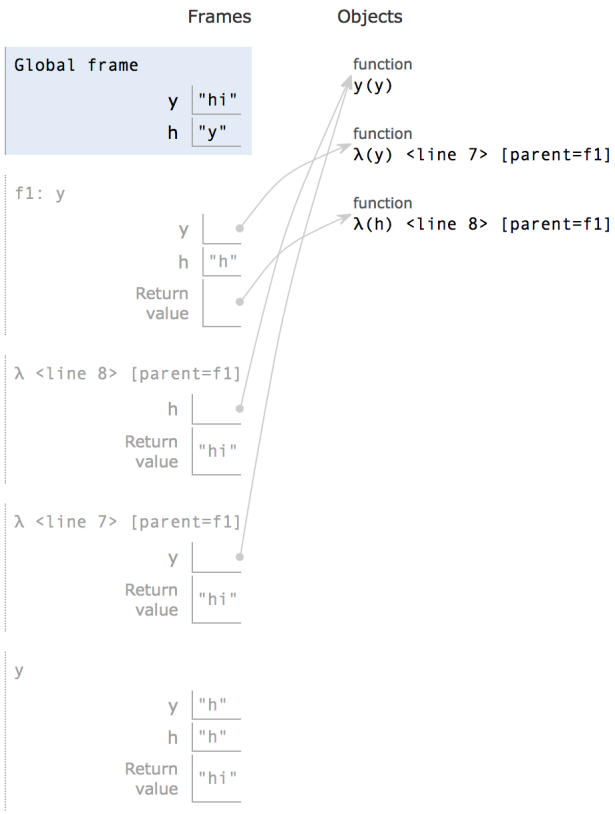

Frames
Objects
Global frame
n 7
f
g 15
function
g(x)
function
h() [parent=f2]
f
f
x 7
Return value
f2: g
x 14
n 9
h
Return value
λ
y
Return value 15
h [parent=f2]
Return value 15

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 for the following code: (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)
```

Solution:



2 Recursion

A *recursive* function is a function that calls itself. Here's a recursive function:

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

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. We do have one *base case*: when `n` is `0` or `1`. 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.

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

1. *Figure out your base case*: What is the simplest argument we could possibly get? For example, `factorial(0)` is 1 by definition.
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.

2.1 Questions

1. Write a function `multiply(m, n)` that multiplies two numbers `m` and `n`. 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`?

   > **Solution:** 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?

Solution: 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
    """
```

Solution:

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

2. Create a recursive countdown function that takes in an integer `n` and prints out a countdown from `n` to 1. The function is defined on the next page.

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

Solution: 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)`?

Solution: 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
    """
```

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

3. Is there an easy way to change `countdown` to count up instead?

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

4. Write a recursive function that sums the digits of a number `n`. Assume `n` is positive. You might find the operators `//` and `%` useful.

```
def sum_digits(n):
    """
    >>> sum_digits(7)
    7
    >>> sum_digits(30)
    3
    >>> sum_digits(228)
    12
    """
```

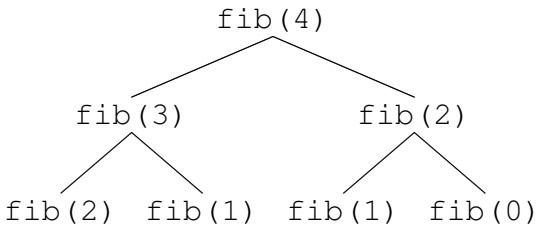
Solution:
```
if n < 10:
    return n
else:
    return n % 10 + sum_digits(n // 10)
```

3 Tree Recursion

Consider a function that requires more than one recursive call. A simple example is the previous recursive `fibonacci` function:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

This type of recursion is called *tree recursion*, because it makes more than one recursive call in its recursive case. If we draw out the recursive calls, we see the recursive calls in the shape of an upside-down tree:



We could, in theory, use loops to write the same procedure. However, problems that are naturally solved using tree recursive procedures are generally difficult to write iteratively. As a general rule of thumb, whenever you need to try multiple possibilities at the same time, you should consider using tree recursion.

1. I want to go up a flight of stairs that has `n` steps. I can either take 1 or 2 steps each time. How many different ways can I go up this flight of stairs? Write a function `count_stair_ways` that solves this problem for me. Assume `n` is positive.

   Before we start, what's the base case for this question? What is the simplest input?

   Solution: When there is only 1 step, there is only one way to go up the stair. When there are two steps, we can go up in two ways: take a two-step, or take 2 one-steps.

   What do `count_stair_ways(n - 1)` and `count_stair_ways(n - 2)` represent?

   Solution: `count_stair_ways(n - 1)` represents the number of different ways to go up the last $n-1$ stairs. `count_stair_ways(n - 2)` represents the number of different ways to go up the last $n-2$ stairs. Our base cases will take care of the remaining 1 or 2 steps.

   Use those two recursive calls to write the recursive case:
   `def count_stair_ways(n):`

   Solution:
   ```
   if n == 1:
       return 1
   elif n == 2:
       return 2
   return count_stair_ways(n-1) + count_stair_ways(n-2)
   ```

2. Consider a special version of the `count_stairways` problem, where instead of taking 1 or 2 steps, we are able to take **up to and including** `k` steps at a time.

   Write a function `count_k` that figures out the number of paths for this scenario.

```
def count_k(n, k):
    """
    >>> count_k(3, 3) # 3, 2 + 1, 1 + 2, 1 + 1 + 1
    4
    >>> count_k(4, 4)
    8
    >>> count_k(10, 3)
    274
    >>> count_k(300, 1) # Only one step at a time
    1
    """
```

Solution:

```
    if n == 0:
        return 1
    elif n < 0:
        return 0
    else:
        total = 0
        i = 1
        while i <= k:
            total += count_k(n - i, k)
            i += 1
        return total
```