Environment Diagrams and Recursion 2

Computer Science 61A

September 10, 2015

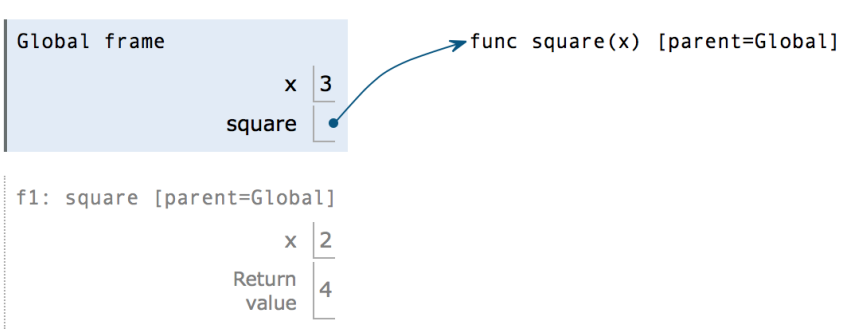
1 Environment Diagrams

An **environment diagram** keeps track of all the variables that have been defined and the values they are bound to.

```
x = 3

def square(x):
    return x ** 2

square(2)
```



When Python executes *assignment statements* (like `x = 3`), it records the variable name and the value:

1. Evaluate the expression on the right side of the = sign
2. Write the variable name and the expression's value in the current frame.

When Python executes *`def` statements*, it records the function name and binds it to a function object:

1. Write the function name (`square`) in the frame and point it to a function object (`func square(x) [parent=Global]`). The `[parent=Global]` denotes the frame in which the function was *defined*.

When Python executes a *call expression* (like `square(2)`), it creates a new frame to keep track of local variables.

1. Draw a new frame *. Label it with
   - a unique index (`f1`, `f2`, `f3` and so on)
   - the intrinsic name of the function (`square`)
   - the parent frame (`[parent=Global]`)
2. Bind the formal parameters to the arguments passed in (e.g. bind `x` to 3).
3. Evaluate the body of the function.

The **intrinsic name** is the name in the function object. For example, if the function object is `func square(x) [parent=Global]`, the intrinsic name is `square`.

If a function does not have a return value, it implicitly returns `None`. Thus, the "Return value" box should contain `None`.

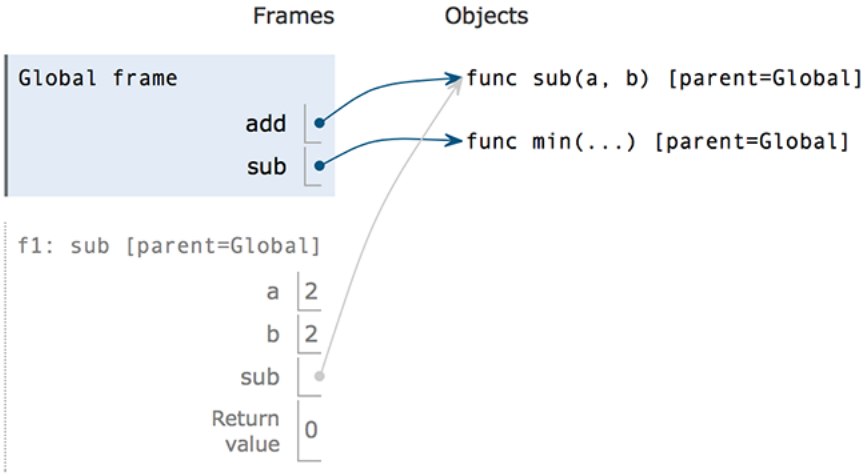
* Since we do not know how built-in functions like `add(...)` or `min(...)` are implemented, we do *not* draw a new frame when we call built-in functions.

1.1 Environment Diagram Questions

1. Draw the environment diagram so we can visualize exactly how Python evaluates the code. What is the output of running this code in the interpreter?

```
>>> from operator import add
>>> def sub(a, b):
...     sub = add
...     return a - b
>>> add = sub
>>> sub = min
>>> print(add(2, sub(2, 3)))
```

Solution:



Output:

```
0
```

2. What will Python print?

```
>>> from operator import add, mul
>>> def f(x, y):
...     print('h', y)
...     return add(add(x, y), 1)
>>> def g(x, y):
...     print('u', x)
...     return mul(mul(x, y), 2)
>>> f(2, g(2, f(500, 2)))
```

Solution:

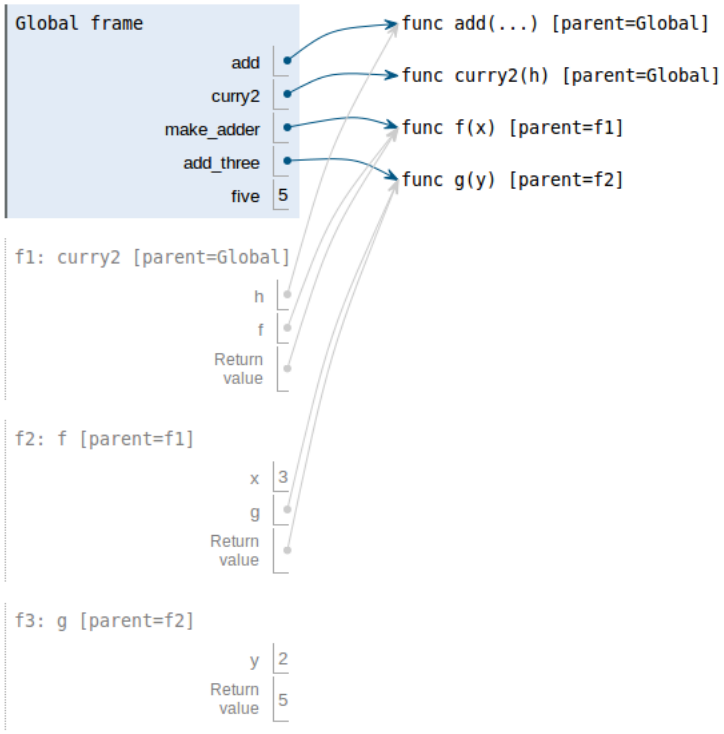
```
h 2
u 2
h 2012
2015
```

3. Draw the environment diagram for the following code:

```
from operator import add
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f

make_adder = curry2(add)
add_three = make_adder(3)
five = add_three(2)
```

Solution:



2 Recursion

A *recursive* function is a function that calls itself. Below is a recursive `factorial` 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 your 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 Cool recursion questions!

1. 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)
```

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

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

3. 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? Which one do we want to use?

Solution: 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)
```

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:

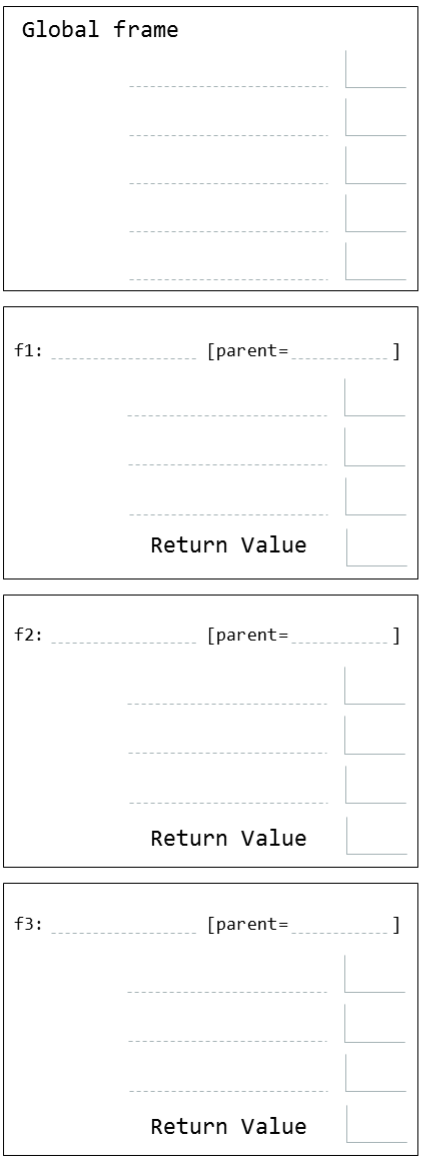
```
    if n < 10:
        return n
    else:
        return n % 10 + sum_digits(n // 10)
```

2.2 Recursive Environment Diagram!

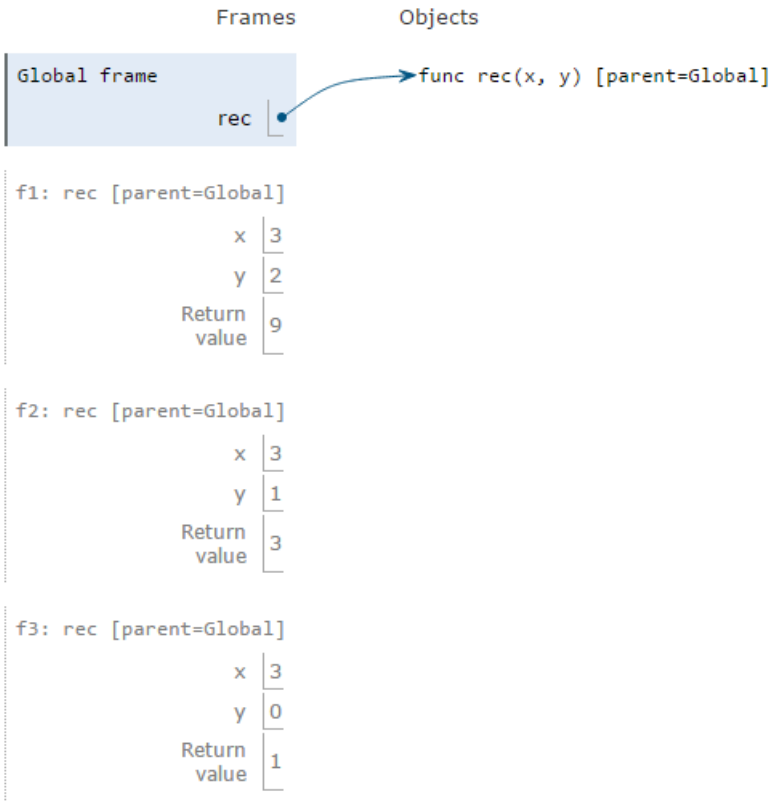
1. Draw an environment diagram for the following code:

```
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?



Solution:



This function computes `x ** math.ceil(y)`.

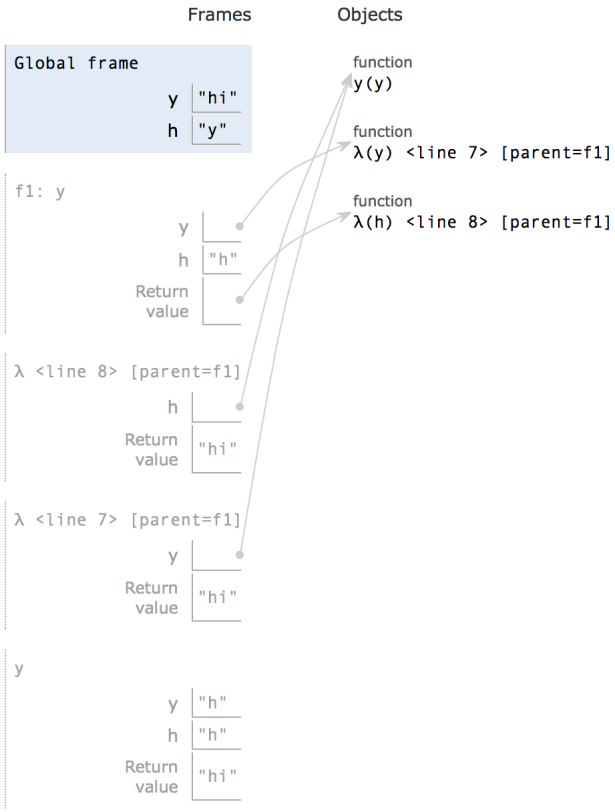
3 Challenge! (Midterm Practice)

1. 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:



4 Iteration vs. Recursion

We've written `factorial` recursively. Let's compare the iterative and recursive versions:

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

def factorial_iterative(n):
    total = 1
    while n > 1:
        total = total * n
        n = n - 1
    return total
```

Notice, while the recursive function "works" until n is less than or equal to 0, the iterative function "works" while n is greater than 0. They're essentially the same.

Let's also compare `fibonacci`.

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

def fib_iterative(n):
    current, next = 0, 1
    while n > 0:
        current, next = next, current + next
        n = n - 1
    return current
```

For the recursive version, we copied the definition of the Fibonacci sequence straight into code! The nth fibonacci number is simply the sum of the two before it. Iteratively, you need to keep track of more numbers and have a better understanding of the code.

Some code is easier to write iteratively and some recursively. Have fun experimenting with both!

5 Environment Diagram Rules

1. Creating a Function
   - Draw the func <name>(<arg1>, <arg2>, ...)
   - The parent of the function is wherever the function was defined (the frame we're currently in, since we're creating the function).
   - If we used def, make a binding of the name to the value in the current frame.
2. Calling User Defined Functions
   - Evaluate the operator and operands.
   - Create a new frame; the parent is whatever the operator's parent is. Now this is the current frame.
   - Bind the formal parameters to the argument values (the evaluated operands).
   - Evaluate the body of the operator in the context of this new frame.
   - After evaluating the body, go back to the frame that called the function.
3. Assignment
   - Evaluate the expression to the right of the assignment operator (=).
   - If nonlocal, find the frame that has the variable you're looking for, starting in the parent frame and ending just before the global frame (via Lookup rules). Otherwise, use the current frame. Note: If there are multiple frames that have the same variable, pick the frame closest to the current frame.
   - Bind the variable name to the value of the expression in the identified frame. Be sure you override the variable name if it had a previous binding.
4. Lookup
   - Start at the current frame. Is the variable in this frame? If yes, that's the answer.
   - If it isn't, go to the parent frame and repeat 1.
   - If you run out of frames (reach the Global frame and it's not there), complain.
5. Tips
   - You can only bind names to values. No expressions (like 3+4) allowed on environment diagrams!
   - Frames and Functions both have parents.