CS 61A
Spring 2019

Higher Order Functions

Discussion 2: February 6, 2019 Solutions

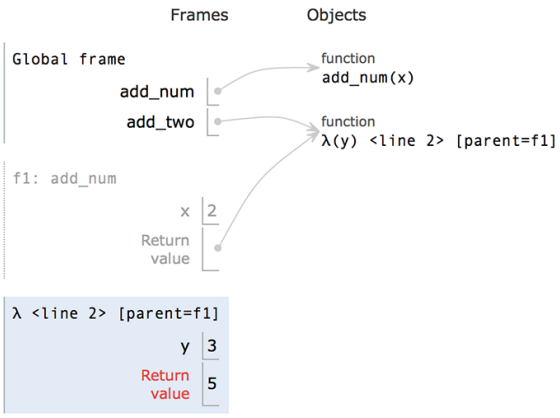
1 Higher Order Functions

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>
```

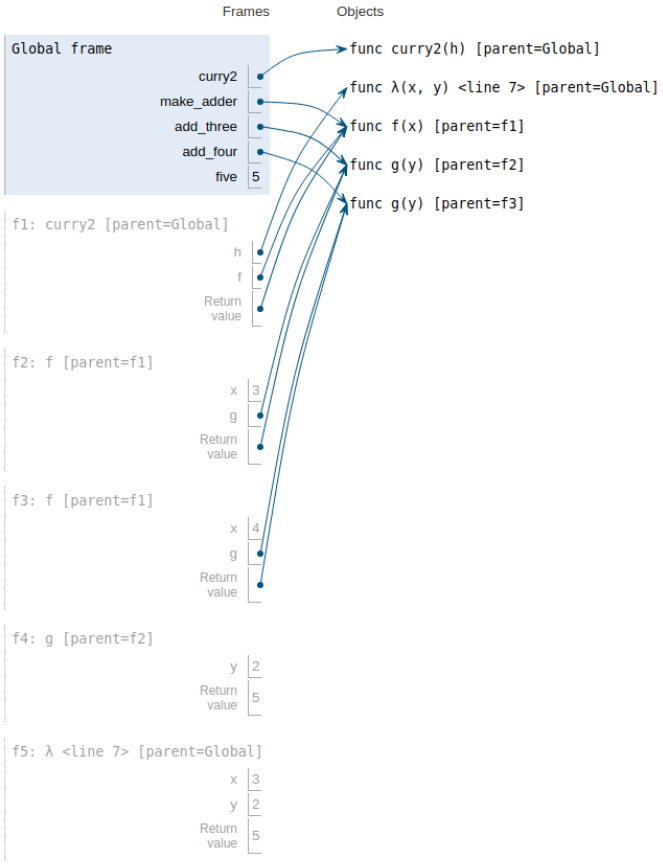
Unlike `def` statements, lambda expressions can be used as an operator or an operand to a call expression. This is because they are simply one-line expressions that evaluate to functions.

```
>>> (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.

```
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)
add_four = make_adder(4)
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.

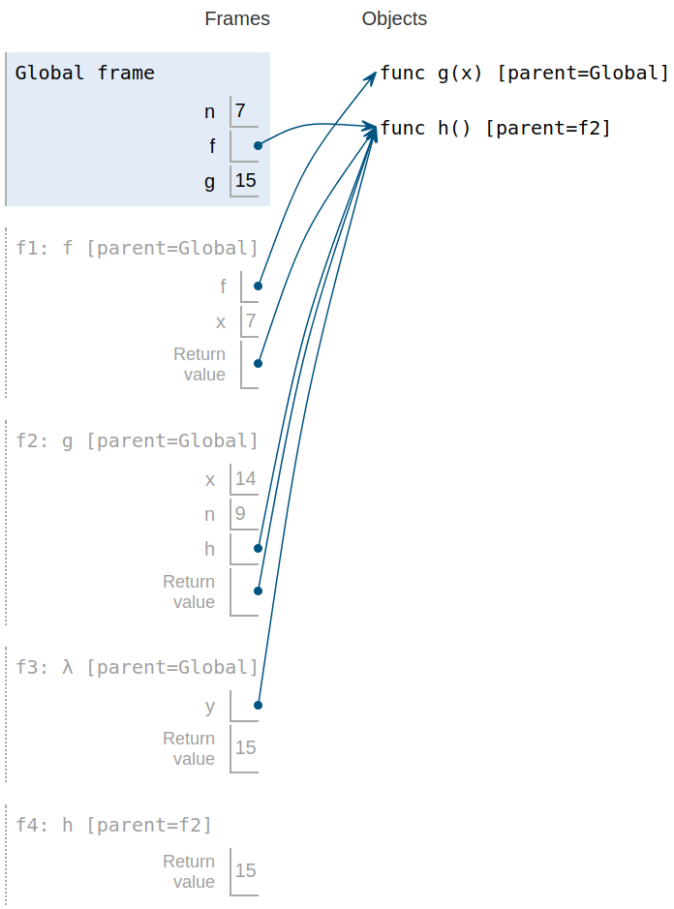
```
n = 7

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)
```



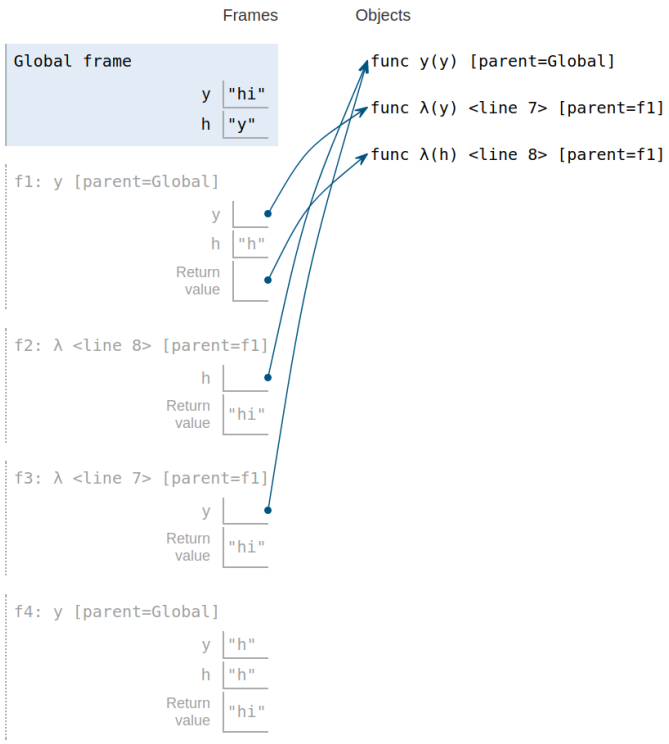
Video Walkthrough

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 make_keeper(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
    >>> make_keeper(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

1.7 Write a function `and_add` that takes a one-argument function `f` and a number `n` as arguments. It should return a function that takes one argument, and does the same thing as the function `f`, except also adds `n` to the result.

```
def and_add(f, n):
    """Return a new function. This new function takes an
    argument x and returns f(x) + n.

    >>> def square(x):
    ...     return x * x
    >>> new_square = and_add(square, 3)
    >>> new_square(4)   # 4 * 4 + 3
    19
    """


    def g(x):
        return f(x) + n
    return g
```