CS 61A Higher-Order Functions, Self Reference

Fall 2021 Discussion 2: September 8, 2021 Solutions

##Walkthrough Videos

Feel free to try these problems on the worksheet in discussion or on your own, and then come back to reference these walkthrough videos as you study.

To see these videos, you should be logged into your berkeley.edu email.

Lambda Expressions

A lambda expression evaluates to a function, called a lambda function. For example, `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 function's 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. In the example below, `(lambda y: y + 5)` is the operator and `4` is the operand.

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

Higher Order Functions

A **higher order function** (HOF) is a function that manipulates other functions by taking in functions as arguments, returning a function, or both. For example, the function `compose` below takes in two functions as arguments and returns a function that is the composition of the two arguments.

```python
def composer(func1, func2):
    """Return a function f, such that f(x) = func1(func2(x))."""
    def f(x):
        return func1(func2(x))
    return f
```

HOFs are powerful abstraction tools that allow us to express certain general patterns as named concepts in our programs.

Q1: Make Keeper

Write a function that takes in a number `n` and returns a function that can take in a single parameter `cond`. When we pass in some condition function `cond` into this returned function, it will print out numbers from 1 to `n` where calling `cond` on that number returns `True`.

```python
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
```

HOFs in Environment Diagrams

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.

See the web version of this resource for the environment diagram.

Lambdas are represented similarly to functions in environment diagrams, but since they lack instrinsic names, the lambda symbol ( ) 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.

Currying

One important application of HOFs is converting a function that takes multiple arguments into a chain of functions that each take a single argument. This is known as **currying**. For example, the function below converts the `pow` function into its curried form:

```
>>> def curried_pow(x):
        def h(y):
            return pow(x, y)
        return h

>>> curried_pow(2)(3)
8
```

Q2: Curry2 Diagram

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

See the web version of this resource for the environment diagram.

Q3: Curry2 Lambda

Write `curry2` as a lambda function.

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

Self Reference

Self-reference refers to a particular design of HOF, where a function eventually returns itself. In particular, a self-referencing function will not return a function **call**, but rather the function object itself. As an example, take a look at the `print_all` function:

```python
def print_all(x):
    print(x)
    return print_all
```

Self-referencing functions will often employ helper functions that reference the outer function, such as the example below, `print_sums`.

```python
def print_sums(n):
    print(n)
    def next_sum(k):
        return print_sums(n + k)
    return next_sum
```

A call to `print_sums` returns `next_sum`. A call to `next_sum` will return the result of calling `print_sums` which will, in turn, return another function `next_sum`. This type of pattern is common in self-referencing functions.

Q4: Make Keeper Redux

In this question, we will build off of the `make_keeper` function from in Question 1.

The function `make_keeper_redux` is similar to `make_keeper`, but now the function returned by `make_keeper_redux` should be self-referential—i.e., the returned function should return a function with the same behavior as `make_keeper_redux`.

Feel free to paste and modify your code for `make_keeper` below.

> **Hint**: you only need to add one line to your `make_keeper` solution. What is currently missing from `make_keeper_redux`?

```python
def make_keeper_redux(n):
    """Returns a function. This function takes one parameter <cond>
    and prints out all integers 1..i..n where calling cond(i)
    returns True. The returned function returns another function
    with the exact same behavior.

    >>> def multiple_of_4(x):
    ...     return x % 4 == 0
    >>> def ends_with_1(x):
    ...     return x % 10 == 1
    >>> k = make_keeper_redux(11)(multiple_of_4)
    4
    8
    >>> k = k(ends_with_1)
    1
    11
    >>> k
    <function do_keep>
    """
    # Paste your code for make_keeper here!
    def do_keep(cond):
        i = 1
        while i <= n:
            if cond(i):
                print(i)
            i += 1
        return make_keeper_redux(n)
    return do_keep
```

Q5: Print N

Write a function `print_n` that can take in an integer `n` and returns a repeatable print function that can print the next `n` parameters. After the nth parameter, it just prints “done”.

```
def print_n(n):
    """
    >>> f = print_n(2)
    >>> f = f("hi")
    hi
    >>> f = f("hello")
    hello
    >>> f = f("bye")
    done
    >>> g = print_n(1)
    >>> g("first")("second")("third")
    first
    done
    done
    <function inner_print>
    """
    def inner_print(x):
        if n <= 0:
            print("done")
        else:
            print(x)
        return print_n(n - 1)
    return inner_print
```

Extra Practice

Feel free to reference this section as extra practice when studying for the exam in terms of tackling more involved or challenging problems.

Q6: HOF Diagram Practice

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

See the web version of this resource for the environment diagram.

Q7: YY Diagram

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

> Tip: 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"`.

See the web version of this resource for the environment diagram.

Q8: Match Maker

Implement `match_k`, which takes in an integer `k` and returns a function that takes in a variable `x` and returns True if all the digits in `x` that are `k` apart are the same.

For example, `match_k(2)` returns a one argument function that takes in `x` and checks if digits that are 2 away in `x` are the same.

`match_k(2)(1010)` has the value of `x = 1010` and digits 1, 0, 1, 0 going from left to right. `1 == 1` and `0 == 0`, so the `match_k(2)(1010)` results in `True`.

`match_k(2)(2010)` has the value of `x = 2010` and digits 2, 0, 1, 0 going from left to right. `2 != 1` and `0 == 0`, so the `match_k(2)(2010)` results in `False`.

Important: You may not use strings or indexing for this problem. You do not have to use all the lines, one staff solution does not use the line directly above the while loop.

Hint: Floor dividing by powers of 10 gets rid of the rightmost digits.

```
def match_k(k):
    """ Return a function that checks if digits k apart match

    >>> match_k(2)(1010)
    True
    >>> match_k(2)(2010)
    False
    >>> match_k(1)(1010)
    False
    >>> match_k(1)(1)
    True
    >>> match_k(1)(2111111111111111)
    False
    >>> match_k(3)(123123)
    True
    >>> match_k(2)(123123)
    False
    """
    def check(x):
        i = 0
        while 10 ** (i + k) < x:
            if (x // 10**i) % 10 != (x // 10 ** (i + k)) % 10:
                return False
            i = i + 1
        return True
    return check


# Alternate solution
def match_k_alt(k):
    """ Return a function that checks if digits k apart match

    >>> match_k_alt(2)(1010)
    True
    >>> match_k_alt(2)(2010)
    False
    >>> match_k_alt(1)(1010)
    False
    >>> match_k_alt(1)(1)
    True
    >>> match_k_alt(1)(2111111111111111)
    False
    >>> match_k_alt(3)(123123)
    True
    >>> match_k_alt(2)(123123)
    False
    """
    def check(x):
        while x // (10 ** k):
            if (x % 10) != (x // (10 ** k)) % 10:
                return False
```

Q9: Three Memory

A `k`-*memory function* takes in a single input, prints whether that input was seen exactly `k` function calls ago, and returns a new `k`-memory function. For example, a 2-memory function will display "Found" if its input was seen exactly two function calls ago, and otherwise will display "Not found".

Implement `three_memory`, which is a 3-memory function. You may assume that the value `None` is never given as an input to your function, and that in the first two function calls the function will display "Not found" for any valid inputs given.

```
def three_memory(n):
    """
    >>> f = three_memory('first')
    >>> f = f('first')
    Not found
    >>> f = f('second')
    Not found
    >>> f = f('third')
    Not found
    >>> f = f('second') # 'second' was not input three calls ago
    Not found
    >>> f = f('second') # 'second' was input three calls ago
    Found
    >>> f = f('third') # 'third' was input three calls ago
    Found
    >>> f = f('third') # 'third' was not input three calls ago
    Not found
    """
    def f(x, y, z):
        def g(i):
            if i == x:
                print('Found')
            else:
                print('Not found')
            return f(y, z, i)
        return g
    return f(None, None, n)
```

Q10: Natural Chain

For this problem, a `chain_function` is a higher order function that repeatedly accepts natural numbers (positive integers). The first number that is passed into the function that `chain_function` returns initializes a natural chain, which we define as a consecutive sequence of increasing natural numbers (i.e., 1, 2, 3). A natural chain breaks when the next input differs from the expected value of the sequence. For example, the sequence (1, 2, 3, 5) is broken because it is missing a 4.

Implement the `chain_function` so that it prints out the value of the expected number at each chain break as well as the number of chain breaks seen so far, including the current chain break. Each time the chain breaks, the chain restarts at the most recently input number.

For example, the sequence (1, 2, 3, 5, 6) would only print 4 and 1. We print 4 because there is a missing 4, and we print 1 because the 4 is the first number to break the chain. The 5 broke the chain and restarted the chain, so from here on out we expect to see numbers increasingly linearly from 5. See the doctests for more examples. You may assume that the higher-order function is never given numbers 0.

Important: For this problem, the starter code is a suggestion. You are welcome to add/delete/modify the starter code template, or even write your own solution that doesn't use the starter code at all.

```
def chain_function():
    """
    >>> tester = chain_function()
    >>> x = tester(1)(2)(4)(5) # Expected 3 but got 4, so print 3. 1
    st chain break, so print 1 too.
    3 1
    >>> x = x(2) # 6 should've followed 5 from above, so print 6. 2
    nd chain break, so print 2
    6 2
    >>> x = x(8) # The chain restarted at 2 from the previous line,
    but we got 8. 3rd chain break.
    3 3
    >>> x = x(3)(4)(5) # Chain restarted at 8 in the previous line,
    but we got 3 instead. 4th break
    9 4
    >>> x = x(9) # Similar logic to the above line
    6 5
    >>> x = x(10) # Nothing is printed because 10 follows 9.
    >>> y = tester(4)(5)(8) # New chain, starting at 4, break at 6,
    first chain break
    6 1
    >>> y = y(2)(3)(10) # Chain expected 9 next, and 4 after 10.
    Break 2 and 3.
    9 2
    4 3
    """
    def g(x, y):
        def h(n):
            if x == 0 or n == x:
                return g(n + 1, y)
            else:
                # BEGIN SOLUTION ALT="return
    ____________________________" NO PROMPT
                return print(x, y + 1) or g(n + 1, y + 1)
        return h
    return g(0, 0)
```

To better understand this solution, let's rename `y` to be `count` and `x` to be `expected_num`. Let `n` be `observed_num`. Then on each call to h, we first check whether `observed_num == expected_num`, which in our problem is a check for `n == x`. If this condition is satisfied, then we increment our `expected_num` by one and do not change our `count`. Hence, we have a call to g with an incremented `x` and an unchanged `y`, and we do not print anything. The only other case in which we enter this suite is when `(x, y) == (0, 0)`, which is when we pass in the very first number of the chain. We are told in the problem statement that we can assume there will never be a non-positive input to the chain. Hence, passing in

(0, 0) as arguments to g is a sure indicator to h that we are currently on the first number of a chain, and we should not print anything (otherwise, when else would you see a 0 passed in as `x`?).

The second case is trickier. First, we need to print a number if our expected number differs from the observed number, but we also need to update `expected_num` and `count` since the chain is going to restart when the chain breaks. To do this in one line, we appeal to lab01, in which we learned that the return value of print is None, which is false-y, and that or operators short-circuit only if a truth-y value is encountered before the last operand. We can print `x` which is our expected number along with the number of chain breaks `y + 1`. This whole procedure returns None, so if we stick it as the first operand of the or operator, we know that the or operator will not short-circuit. The program will then proceed to evaluate `g(n + 1, y + 1)`, which returns `h`. Although not necessary for this problem, it may be helpful to know that a function is truth-y. In this case, it doesn't matter because an or will return the last operand it is given if it doesn't short-circuit, regardless of the truth-y-ness of the last operand. Hence, the last line of `h` prints the appropriate numbers, updates the `expected_num` (restarting the chain), updates the count, and returns `h`, which will accept the next natural number into the chain.Why does `g` return `h`? Watch the indentations of each line very closely. The only thing the `g` function does is define a function h and then return `h`. All that other stuff where we check for equality and return print etc is all inside the `h` function. Hence, that code is not executed until we call `h`.

As a side note, if you are stuck on a problem like this one on an exam, it may be helpful to ignore the skeleton and first implement the code assuming you have unlimited lines and freedom to name your variables whatever you want. When I was writing this question, I named all my functions something meaningful, and you bet that my variables were not `x` and `y` but `expected_num` and `count`. I also had a separate line for the print. However, once I got into making a template skeleton for an exam level problem, I thought "how do I obfuscate the code to force people to think harder?" The variable name changes were easy, and condensing the print into the same line as the return was just a trick I've seen from previous exams in this course. I think that coming up with the whole solution from scratch by following the template is immensely harder than first coming up with your own working solution and then massaging your solution into the constraints of our blank lines.

- Derek Wan, Fall 2020