Exam Prep 2: Higher-Order Functions, Self Reference

Students from past semesters wanted more content and structured time to prepare for exams. Exam Prep sections are a way to solidify your understanding of the week's materials. The problems are typically designed to be a bridge between discussion/lab/homework difficulty and exam difficulty.

Reminder: There is nothing to turn in and there is no credit given for attending Exam Prep Sections.

We try to make these problems **exam level** , so you are not expected to be able to solve them coming straight from lecture without additional practice. To get the most out of Exam Prep, we recommend you **try these problems first on your own** before coming to the Exam Prep section, where we will explain how to solve these problems while giving tips and advice for the exam. Do not worry if you struggle with these problems, **it is okay to struggle while learning**.

You can work with anyone you want, including sharing solutions. We just ask you don't spoil the problems for anyone else in the class. Thanks!

You may only put code where there are underscores for the codewriting questions.

You can test your functions on their doctests by clicking the red test tube in the top right corner after clicking Run in 61A Code. Passing the doctests is not necessarily enough to get the problem fully correct. You must fully solve the stated problem.

We recommending reading sections 1.1-1.6 from the textbook for these problems. We also recommend reviewing Hog project `announce_lead_changes` and `announce_highest` for question 2 and 3.

Test your work! For example, for `match_k` , you can type `test(match_k)` in the python interpreter you get once you click Run in 61A Code to verify if you pass the doctests or not.

Q1: Match Maker

Difficulty: ☆

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`.

RESTRICTION: You may not use strings or indexing for this problem.

IMPORTANT: 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.

```
1   def match_k(k):
2       """Return a function that checks if digits k apart match
3
4       >>> match_k(2)(1010)
5       True
6       >>> match_k(2)(2010)
7       False
8       >>> match_k(1)(1010)
9       False
10      >>> match_k(1)(1)
11      True
12      >>> match_k(1)(2111111111111111)
13      False
14      >>> match_k(3)(123123)
```

```
        True
        >>> match_k(2)(123123)
        False
        """

        def f(x):
            # _____________________________
            while x // (10**k):
                if (x % 10) != (x // (10**k)) % 10:
                    return False
                x = x // 10
            return True

        return f
```

Q2: Natural Chainz

Difficulty: ⭐⭐

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 template is just 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. 1st chain break, so print 1 too.
    3 1
    >>> x = x(2) # 6 should've followed 5 from above, so print 6. 2nd 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
        """
    # g(期待得到第x个元素，第y条链子)
    # h(实际得到的元素)
        def g(x, y):
            def h(n):
                if x == 1 or n == x:
                    return g(n + 1, y)
                else:
                    return print(x, y) or g(n + 1, y + 1)

```

Q3: CS61 - NAY

Difficulty: ☆☆☆

Part A: Implement `cs61nay`, which takes a two argument function `combiner` and positive integer `n` and returns a function.

The returned function then takes `n` arguments, one at a time, and computes `combiner(... (combiner(combiner(arg1, arg2), arg3)...), arg_n)`. Notice combiner takes in two integers and returns one integer.

For example, the first doctest has the returned function `f = cs61nay(lambda x, y: x * y, 3)`. Now when `f` is applied to three arguments, like `f(2)(3)(4)`, it multiplies them together, `2*3*4` to get 24.

IMPORTANT: For this problem, the starter code template is just 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.

HINT: For the `n = 1` case, the returned function doesn't use `combiner`.

```
def cs61nay(combiner, n):
    """Return a function that takes n arguments,
    one at a time, and combines them using combiner.

    >>> f = cs61nay(lambda x, y: x * y, 3)
    >>> f(2)(3)(4) # 2 * 3 * 4
    24
    >>> f(-1)(2)(3) # -1 * 2 * 3
    -6
    >>> f = cs61nay(lambda x, y: x - y, 4)
    >>> f(1)(2)(-2)(-1) # 1 - 2 - -2 - -1
    2
    >>> f = cs61nay(lambda x, y: x + y, 1)
    >>> f(61)
    61
```

```
16        >>> f(2021)
17        2021
18        """
19        if n == 1:
20            return lambda x: x
21        else:
22            return lambda x: lambda y: cs61nay(combiner, n - 1)(combiner(x, y))
23
```

Difficulty: ☆☆☆

Part B: Somebody who writes very complicated code has given you a challenge! You would hopefully never see something so hard to comprehend in the real world.

Complete the expression below by writing one integer in each blank so that the whole expression evaluates to 2021. Assume `cs61nay` is implemented correctly.

HINTS:

- You can fill the blanks and test your code with a python interpreter
- Try to understand what all the subparts and smaller calls do
- You can split this up into multiple lines to make it more readable
- You can add spaces/indentation to make it easier to read

```
1   from operator import sub, add, mul
2
3   compose = lambda f, g: lambda x: f(g(x))
4
5   print(compose(cs61nay(sub, 2), compose(cs61nay(mul, 3)(2),
6         cs61nay(pow, 2)(10))(3))(1)(-21))
7
8
```

Difficulty: ⭐⭐

Part C: All those lines of code are unnecessary! Solve `cs61NAY` but only using one line.

RESTRICTION: You may not use the python ternary operator (the one line if/else statment).

HINT: Use short circuiting and boolean operators to your advantage

```
2
3   # This syntax adds a doctest to a lambda, which can be run using `test(cs61NAY)`
4   # after clicking Run in 61A Code or `python3 -m doctest -v examprep02.py`
5   # if you save the questions to a .py file
6   cs61NAY.__name__ = "cs61NAY"
7   cs61NAY.__doc__ = """ Return a function that takes n arguments,
8       one at a time, and combines them using combiner.
9
10      >>> f = cs61NAY(lambda x, y: x * y, 3)
11      >>> f(2)(3)(4) # 2 * 3 * 4
12      24
13      >>> f(-1)(2)(3) # -1 * 2 * 3
14      -6
15      >>> f = cs61NAY(lambda x, y: x - y, 4)
16      >>> f(1)(2)(-2)(-1) # 1 - 2 - -2 - -1
17      2
18      >>> f = cs61NAY(lambda x, y: x + y, 1)
19      >>> f(61)
20      61
21      >>> f(2021)
22      2021
23      """
24
```

Just for Fun

This is a challenge problem and not reflective of exam difficulty. We will not be going over this problem in examprep section, but we will be releasing solutions.

Q4: Abusing the Call Stack

(a) Implement the following higher order functions so that we can simulate `append` and `get` behavior. As the name suggests, the `get` function should get the `ith` element that was appended (the first element that was appended is element 0). For example, if I append 2, append 30, and then append 4, then `get(0)` is 2 (the first element appended), `get(1)` is 30 (the second element appended), and `get(2)` is 4 (the third element appended). If I append more items, `get(0)` through `get(2)` should not be affected. Assume all get calls ask for non-negative indices (i.e., you'd never do `get(-1)`). If the argument to get would go out of bounds otherwise, the call should return the string `"Error: out of bounds!"`.

RESTRICTION: you are not allowed to use any lists / sets / dictionaries / iterators, or any other data structures.

```
def stacklist():
    """
    >>> append, get = stacklist()
    >>> get, y = append(2)
    >>> get, y = append(3, get, y)
    >>> get, y = append(4, get, y)
    >>> get(0)
    2
    >>> get(1)
    3
    >>> get(2)
    4
    >>> get, y = append(8, get, y)
    >>> get(1)
    3
    >>> get(3)
    8
    """
    g = lambda i: "Error: out of bounds!"
    def f(value, g=g, y=0):
        f = g
```

```
        def g(i):
            if i == y:
                return value
            return f(i)
        return g,y+1
    return f, g

```

(b) Build on your solution to the previous question to implement `insert` functionality! As the name suggests, the `insert` function inserts a value into an existing sequence of numbers. The function takes an insertion index, the value to insert into that index, as well as two other arguments whose purpose is left for you to determine. When the value is inserted into the provided index, all numbers from that index and to the right are shifted one element right. For example, if my current sequence is 5, 9, 14, 3 and I specify an insertion index of 1 with value 100, then my updated sequence should be 5, 100, 9, 14, 3. The 100 is inserted at index 1, and all numbers from the original index 1 to the end are shifted to the right by one position. You can always assume that the provided insertion index will be within bounds.

You don't actually have to represent the sequence as a contiguous block of numbers that need to shift around though. As long as the `get(i)` call returns the correct value, that'll do.

RESTRICTION: You are not allowed to use any lists / dictionaries / iterators, or any other data structures.

```
10        >>> get(1)
11        19
12        >>> get(2)
13        13
14        >>> get(3)
15        4
16        """
17        # Assume f and g are defined correctly from the previous question
18        g = lambda i: "Error: out of bounds!"
19
20        def f(value, g=g, y=0):
21            f = g
22
23            def g(i):
24                if i == y:
25                    return value
26                return f(i)
27
28            return g, y + 1
29
30        def h(y, value, g, n):
31            e = g
```

```
            e = g
32
33          def g(i):
34              if i == y:
35                  return value
36              return e(i)
37
38          k = y
```