CS 61A Structure and Interpretation of Computer Programs
Fall 2020

Quiz 2 Solutions

INSTRUCTIONS

- Please review this worksheet before the exam prep session. Coming prepared will help greatly, as the TA will be live solving without allocating much time for individual work.
- Either Sean or Derek will be on video live solving these questions. The other TA will be answering questions in the chat. It is in your best interest to come prepared with **specific** questions.
- This is not graded, and you do not need to turn this in to anyone.
- Fall 2020 students: the boxes below are an artifact from more typical semesters to simulate exam environments. Obviously this doesn't apply to this semester's exams, but we just kept the fields to keep our materials looking professional :) Feel free to ignore them.
- For multiple choice questions, fill in each option or choice completely.
  - ☐ means mark **all options** that apply
  - ◯ means mark a **single choice**

Last name	
First name	
Student ID number	
CalCentral email (_@berkeley.edu)	
Discussion Section	___ ___ ___
All the work on this exam is my own. **(please sign)**	

1. **Natural Chainz**

   (a) 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. 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.

```
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.
    """
    def g(x, y):

        def h(n):

            if x == 0 or n == x:

                return g(n + 1, y)

            else:

                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 a quiz, 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.

2. **Abusing the Call Stack Pt 1**

   (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!".
   Note: you are *not* allowed to use any lists / sets / dictionaries / iterators, or any other data structures or built-in functions we have not yet covered in the class.

```
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
```

3. Abusing the Call Stack Pt 2

(a) 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. **Note:** you are *not* allowed to use any lists / dictionaries / iterators, or any other data structures / functions we have not yet covered in the class. However, you may use ternary statements: `lambda x: 1 if x == 9 else 2` for example returns 1 if x is 9 and returns 2 otherwise.

```
def stacklist():
    """
    >>> append, get, insert = stacklist()
    >>> get, idx = append(2)
    >>> get, idx = append(13, get, idx)
    >>> get, idx = append(4, get, idx)
    >>> get, idx = insert(1, 19, get, idx)
    >>> get(0)
    2
    >>> get(1)
    19
    >>> get(2)
    13
    >>> get(3)
    4
    """
    # Assume f and g are defined correctly from the previous question

    def h(y, value, g, n):

        e = g

        def g(i):

            if i == y:

                return value

            return e(i)

        k = y

        while k < n:

            g, ret = f(e(k), (lambda a, b: lambda i: e(b) if i == b + 1 else a(i))(g, k), k + 1)

            k += 1

        return g, ret
        return f, g, h
```

It turns out that passing in `g` instead of `(lambda a, b: lambda i: e(b) if i == b + 1 else a(i))(g, k)` works as well. I can't think of any edge cases in which it fails...