6.009 CONFLICT Quiz 2

Spring 2022

Name: **Answers**

Kerberos/Athena Username:

4 questions 1 hour and 50 minutes

- Please **WAIT** until we tell you to begin.
- This quiz is closed-book, but you may use one 8.5 $\times$ 11 sheet of paper (both sides) as a reference.
- You may **NOT** use any electronic devices (including computers, calculators, phones, etc.).
- If you have questions, please **come to us at the front** to ask them.
- Enter all answers in the boxes provided. Work on other pages with QR codes may be taken into account when assigning partial credit. **Please do not write on the QR codes.**
- If you finish the exam more than 10 minutes before the end time, please quietly bring your exam to us at the front of the room. If you finish within 10 minutes of the end time, please remain seated so as not to disturb those who are still finishing their quizzes.
- You may not discuss the details of the quiz with anyone other than course staff until final quiz grades have been assigned and released.

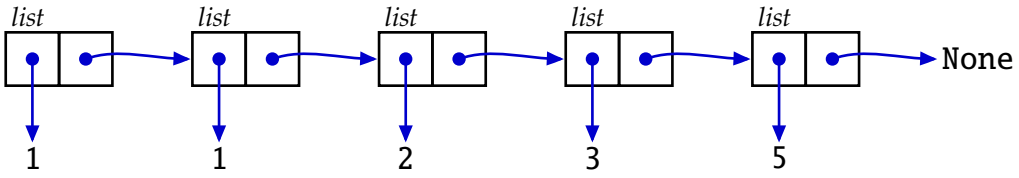
Worksheet (intentionally blank)

1 Cycles

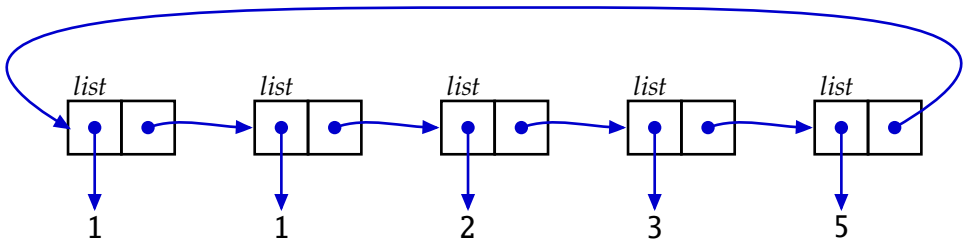
In this problem, we'll explore an interesting kind of data structure called a *cycle*. Cycles are like lists in that they represent an ordered collection of elements. Unlike lists, though, cycles have no beginning or end; the elements in a cycle repeat endlessly, with the first element following the last.

For this problem, we will represent cycles as a particular kind of linked list. Similarly to our representation from lab 9, we can represent regular linked lists as two-element Python lists, where the first element represents the first value in the linked list, and where the second element represents the remainder of the linked list (either as `None`, representing the empty list, or as another linked list). For example, we would represent a linked list containing 1, 1, 2, 3, and 5 (in that order) as: `[1, [1, [2, [3, [5, None]]]]]`.

In our environment diagram notation, this linked list would look like the following:



A *cycle* containing those same elements looks very similar, but with one key difference: the first node in the list follows from the last node, leading to a circular structure:



Answer the questions on the following pages about various operations on cycles.

1.1 Code Correctness

Consider the following set of functions designed to operate on linked lists and/or cycles.

```
def value(node):
    # Return the value associated with the given node
    return node[0]

def next_node(node):
    # Return the node that follows from the given node in the cycle containing
    # that node
    return node[-1]

def set_value(node, val):
    # Mutate the given node such that its value
    node[0] = val

def set_next(node, target):
    # Mutate the first argument (a node) such that it is now followed by the
    # second argument (another node)
    node[-1] = target

def last(inp):
    # Return the last node in a non-cyclic linked list (not expected to work
    # for cycles), without mutating the given input.
    n = next_node(inp)
    if n is None:
        return inp
    return last(n)

def make_cycle(inp):
    # Create a cycle containing all of the elements from the given input (a
    # non-cyclic linked list), without mutating the given input.
    out = inp
    set_next(last(out), out)
    return out
```

Note that the `is` keyword can be used to perform an identity check. For example, `x is y` will evaluate to `True` if `x` and `y` refer to exactly the same object in memory (i.e., if they are aliases of the each other).

Each of the functions on the facing page includes a comment at the top describing its intended behavior. Are all of these functions implemented in ways that are consistent with their stated behavior (from the comments)?

Yes/No: No

If not, please specify which functions are inconsistent with their comments, and briefly describe the issue with each:

Most of these functions are indeed completely correct.

However, there is a slight issue with `make_cycle`. It produces the correct output, but it also mutates its input, which is inconsistent with the documentation.

1.2 Backing Up

We have already defined a helper function for moving through a cycle in one direction (`next_node` can be used for this purpose). In this section, we ask you to write code for a function to help us move through a cycle in the *opposite* direction.

To this end, we will define a function `prev_node`, which should return the *previous* node in the cycle (including the behavior of wrapping around from the start of the list to the end). This is the opposite behavior from `next_node`, so for any node `n` that is part of a cycle, we expect `prev_node(next_node(n))` to return `n`.

For example, consider the following code to create a cycle:

```
test_list = [4, [8, [15, [16, [23, [42, None]]]]]]
test_cycle = make_cycle(test_list)
```

Using this example:

- `value(test_cycle)` should return `4`.
- `value(prev_node(test_cycle))` should return `42`.
- `value(prev_node(prev_node(test_cycle)))` should return `23`.

Fill in your definition of `prev_node` in the box below. You may assume that the given argument is a well-formed, nonempty cycle of the form described earlier in the problem, and you may assume that you have access to working versions of all helper functions from the previous page.

```
def prev_node(node):
    curr = node
    while next_node(curr) is not node:
        curr = next_node(curr)
    return curr
```

1.3 Node Replacement

We would also like to be able to modify cycles. To this end, we will implement a function `replace_node(node, lst)`, which should mutate the cycle that the given node belongs to such that the second argument (a regular nonempty linked list) replaces the given node in the cycle. After this replacement, the replaced node should no longer be part of the cycle and it should no longer point back into the cycle. It is OK to mutate either or both of the arguments. For example, using the test cycle from the previous page, the following REPL transcript shows the desired behavior:

```
>>> x = next_node(test_cycle)
>>> y = next_node(x)
>>> replace_node(x, [1, [2, [3, None]]])
>>> value(test_cycle)
4
>>> value(next_node(test_cycle))
1
>>> value(next_node(next_node(test_cycle)))
2
>>> next_node(next_node(next_node(next_node(test_cycle)))) is y
True
```

Fill in the definition of `replace_node` below. You may assume that the first argument is part of a well-formed cycle with at least 2 nodes and that the second argument is a well-formed, nonempty linked list. For full credit, your code should not use any built-in list manipulations, but rather should *only* use the helper functions defined throughout this problem (you may assume working versions of all helper functions from part 1, and a working version of `prev_node`).

```
def replace_node(node, lst):
    set_next(prev_node(node), lst)
    set_next(last(lst), next_node(node))
    set_next(node, None)
```

1.4 Flattening

Consider the following piece of code. The function `one_cycle` takes a node from a cycle as input, and it is intended to return a list containing the elements from the cycle, but containing the element from each node exactly once. So, for example, using the cycle from the last page as an example, `one_cycle(test_cycle)` should return `[4, 8, 15, 16, 23, 42]`, and `one_cycle(next_node(test_cycle))` should return `[8, 15, 16, 23, 42, 4]`.

```
01 |  def one_cycle(inp, end=None, sofar=[]):
02 |      if end is None:
03 |          end = prev_node(inp)
04 |
05 |      sofar = sofar + [value(inp)]
06 |      if inp is end:
07 |          return sofar
08 |      else:
09 |          one_cycle(next_node(inp), end, sofar)
```

However, there is at least one issue with the code as written, such that it does return the correct result in some cases but it is also possible for it to fail to return the correct result in some cases, even for well-formed inputs.

On the facing page, briefly describe the issue(s) with the code, inputs (or sequences of inputs) that cause the issue to manifest, and a brief description of how to fix the issue(s).

Briefly describe the nature of inputs (or sequences of inputs) for which this function fails to return the correct result, as well as the nature of the failure (what happens, and why?). How can you fix the code so that it returns the correct result for all well-formed inputs?

As written, this function only produces the correct answer for cycles of length 1. For everything else, the function returns `None`.

The issue is line 9, which makes a recursive call but does not return the result of that call. Adding `return` to the front of that line would fix the issue.

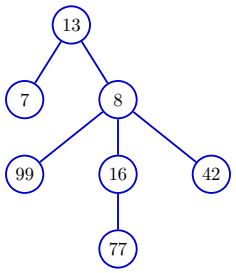
2 Dendrology

Throughout 6.009, we have seen and implemented several kinds of operations on trees. The function below is an attempt to generalize these operations:

```
def tree_op(tree, transform, combine):
    value = tree[0]
    children = tree[1:]
    return combine(transform(value),
                   [tree_op(child, transform, combine) for child in children])
```

This function assumes that its first input is a tree, where each node in the tree is represented as a list with the first element being the node value, and the rest of the list being the node's children. For example, the following structure (`tree1`) and picture represent the same tree.

```
tree1 = [13, [7], [8, [99], [16, [77]], [42]]]
```



Consider also the following list of possible behaviors that we could try to implement using the `tree_op` function:

A) Returns a reference to the original tree

B) Returns a "shallow copy": a new outer list but with aliases to the original children

C) Returns a "deep copy": an identical tree with no aliasing

D) Sums the values of all of the nodes in the tree

E) Sums the values of all of the "leaf" nodes (only those nodes that have no children)

F) Counts the number of nodes in the tree

G) Counts the number of leaf nodes in the tree

H) Flattens the input tree (returns a single list of all the tree's elements with no nesting)

I) Performs a "deep reversal" where each node's children are reversed

J) Reverses the top-level children but leaves *their* children in the original order

K) Enters an infinite loop or infinite recursion

L) Raises an exception (other than infinite recursion)

M) Something else

For each of the code examples below, indicate which of the behaviors from the facing page (A-M) describes the behavior of that example. Enter your answers as a single letter in each box.

Example 1

```
tree_op(tree,
        lambda x: 1,
        lambda x, y: sum([x, *y]))
```

Behavior: F

Example 2

```
tree_op(tree,
        lambda x: x,
        lambda x, y: x + sum(y))
```

Behavior: D

Example 3

```
tree_op(tree,
        lambda x: x,
        lambda x, y: [x] + [a for b in y for a in b])
```

Behavior: H

Example 4

```
tree_op(tree,
        lambda x: x,
        lambda x, y: [x]+y[::-1])
```

Behavior: I

Example 5

```
tree_op(tree,
        lambda x: x,
        lambda x, y: (0 if y else 1) + sum(y))
```

Behavior: G

3 Reordering

In this problem, we will write code to reorder data from a (potentially infinite) stream of out-of-order chunks.

We will assume that we have access to a generator that produces tuples of the form `(index, data)`, where `index` is an integer and `data` is a list of bytestrings. The data are intended to be used in order of increasing `index` values, but the generator may produce these tuples in the incorrect order.

Your task for this problem is to write a generator function called `reorder(stream)`, which takes as input a generator of the form described above. `reorder` should yield the given data elements in order of increasing `index`.

For example, consider the following generator:

```
def test_stream():
    yield (0, [b"i", b"think"])
    yield (3, [])
    yield (1, [b"that"])
    yield (4, [b"really", b"quite", b"cool!"])
    yield (2, [b"generators", b"are"])
```

With `test_stream` defined like this, `reorder(test_stream())` should be a generator that yields the values `b"i"`, `b"think"`, `b"that"`, `b"generators"`, `b"are"`, `b"really"`, `b"quite"`, and `b"cool!"`, in that order. We would expect this same result regardless of the order in which `test_stream` yields its elements.

You may assume that the smallest index represented in the input is always `0`, and that for any index n represented in the input (where n is greater than 0), there will always be an element with index $n-1$ as well. Aside from that, you should make no assumptions about the number of elements in the generator (which could even be infinite!), nor the order in which those elements are yielded from the generator, nor the length of the data lists.

For full credit, your generator should have the following properties:

- Your generator should yield results as soon as it is able to, rather than creating a structure to store all of the elements and then yielding from that structure.
- Your generator may store data in local variables, but it should not continue to store any values after they have been yielded.
- If the elements from the input generator are already in order, they should never be stored inside of any data structure within your generator.

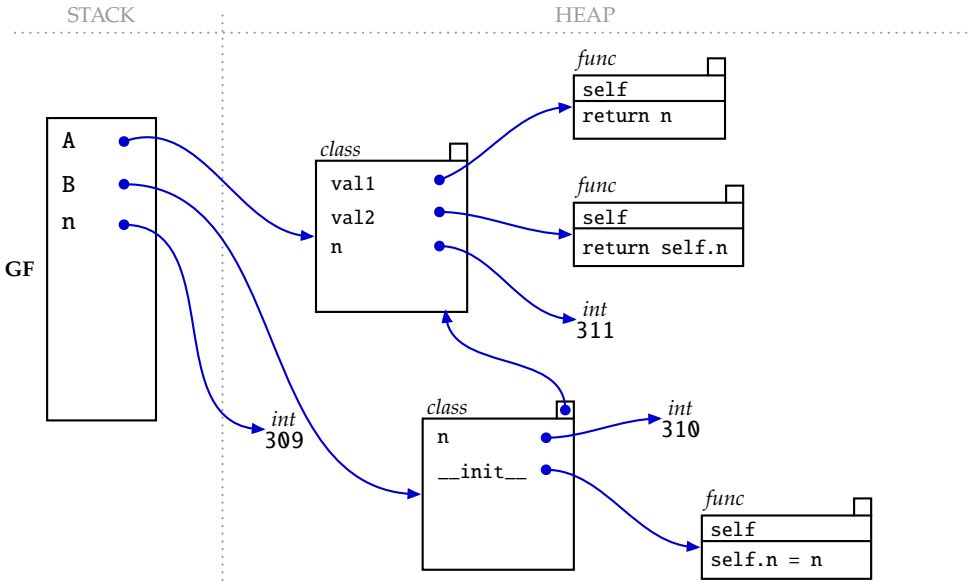
Fill in the definition of `reorder` on the facing page.

```
def reorder(stream):
    nextix = 0
    cache = {}

    for (ix, elt) in stream:
        if ix == nextix:
            yield from elt
            while (ix := ix + 1) in cache:
                yield from cache.pop(ix)
            nextix = ix
        else:
            cache[ix] = elt
```

4 Environment Diagrams

The following diagram represents the state of a program after some code has been executed, purely in the global frame (no additional frames have been created as of this point in the program's execution). This diagram is almost complete, except that the enclosing frames of the function objects are intentionally not included in the diagram.



Starting from this point, assume that the following code is run in the global frame:

```
n = 312

class C(A):
    def __init__(self):
        self.n = self.n

    n = 313
```

After this code is run, consider running the code on the facing page in the order it is specified. Each print statement in the code is followed by a box. For each, consider the value that would be printed by that print statement, and:

- If nothing would be printed because of infinite loop (or infinite recursion), write `infinite` in the box.
- If nothing would be printed because of an exception other than infinite recursion, write `exception` in the box.
- Otherwise, write the printed value in the box.

```
a = A()
print(a.val1())
```

312

```
print(a.val2())
```

311

```
b = B()
print(b.val1())
```

312

```
print(b.val2())
```

312

```
c = C()
print(c.val1())
```

312

```
print(c.val2())
```

313

```
def foo():
    n = 314
    out = B().val1()
    n = 315
    return out
print(foo())
```

312

```
def bar():
    n = 316
    out = B().val2()
    n = 317
    return out
print(bar())
```

312

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)