6.009 Quiz 1

Fall 2021

Name:

Kerberos/Athena Username:

5 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 Snowfall

In this problem, we will look at two small programs that are designed with the same purpose in mind: computing the average amount of snowfall per day in a particular area. The data are passed in as a list of numbers representing daily snowfall amounts, and the representation uses the following two conventions:

- The list may (but need not) contain the number `-999`. This value is not a snowfall amount but rather indicates the end of the snowfall data of interest. Values after the first `-999`, if it occurs, should be ignored.
- The list may contain other negative numbers. These numbers are taken to be data-entry errors and should be ignored in computing the average.

The function should return the average of the nonnegative snowfalls of interest. It should return `None` if this average is undefined (i.e., if the average of numbers of interest cannot be computed).

What follows are two attempts to solve this problem. For each, you are asked to provide four separate examples of **nonempty** inputs to the function that produce a variety of results:

- one example of an input for which the function returns the correct result,
- one example of an input for which the function runs to completion but returns an incorrect result,
- one example of an input that causes an infinite loop, and
- one example of an input that causes an exception to be raised.

If any of these outcomes is not possible for a given implementation, write an `X` in the associated box instead of providing an input.

Snowfall: Attempt 1

```
def average_snowfall(snowfalls):
    total = 0
    count = 0
    for i in snowfalls:
        while i != -999:
            if i >= 0:
                total += i
                count += 1
    return total / count
```

Nonempty input that produces correct result:

X *(no valid input)*

Nonempty input that produces incorrect result:

X *(no valid input)*

Nonempty input that results in an infinite loop:

```
[1]
```

Nonempty input that results in an exception:

```
[-999]
```

Snowfall: Attempt 2

```
def average_snowfall(snowfalls):
    total = 0
    count = 0
    ix = 0
    while ix < len(snowfalls):
        if snowfalls[ix] != -999:
            if snowfalls[ix] < 0:
                continue

            total += snowfalls[ix]
            count += 1
        ix += 1
    return total / count
```

Nonempty input that produces correct result:

```
[1, 2, 3]
```

Nonempty input that produces incorrect result:

```
[1, 2, 3, -999, 10000]
```

Nonempty input that results in an infinite loop:

```
[1, 2, 3, -7]
```

Nonempty input that results in an exception:

```
[-999]
```

2 Predicting Program Behavior

Part a

Consider the following small program:

```
y = 2

def add(x):
    return x + y

y = 7

print(add(8))
```

Will this program run to completion (Yes/No)? Yes

If yes, what will be printed to the screen when this program is run?
If no, briefly describe the nature of the issue that prevents the program from running to completion.

15

Worksheet (intentionally blank)

Part b

Consider the following small program:

```
def square(x):
    return square(x * x)

first = square

def square(y):
    return y*y

print(first(3))
```

Will this program run to completion (Yes/No)? Yes

If yes, what will be printed to the screen when this program is run?
If no, briefly describe the nature of the issue that prevents the program from running to completion.

81

Worksheet (intentionally blank)

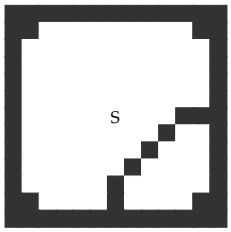
Worksheet (intentionally blank)

3 Flood Fill

In this problem, we will consider an implementation of the "flood fill" program from Lecture 2, which recolors all the cells of a particular color in an enclosed region in an image. The code for a nearly complete implementation of flood fill is included on the last page of this exam, which you may remove from the exam packet.

This code has been implemented almost completely, but there are two segments of code missing. For each of these segments, there are several possibilities for what we could put in those locations. All of the possibilities are listed on the reverse of this page.

Consider running the flood-fill process in the following image by clicking on the location labeled "S" and filling it with a grey color:



For each of the images on the following page, write one combination of code segments that would result in that image. For example, if an image would have resulted from option A for segment 1 and option i for segment 2, write "`A, i`" in the box below that image.

If there are multiple possible answers for an image, you need only write one such answer. If an image could not have resulted from any of these combinations, write `None` in its box instead.

Code Segment 1: Option A

```
set_pixel(image, *location, fill_color)
```

Code Segment 1: Option B

```
if get_pixel(original_image, *location) == clicked_color:
    set_pixel(image, *location, fill_color)
```

Code Segment 1: Option C

```
pass  # do nothing
```

Code Segment 2: Option i

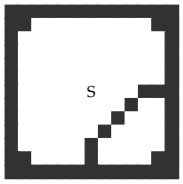
```
pass  # do nothing
```

Code Segment 2: Option ii

```
continue
```

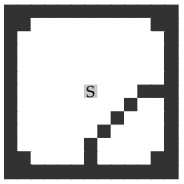
Code Segment 2: Option iii

```
if get_pixel(original_image, *child) != clicked_color:
    continue
```

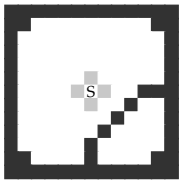


C, *any*

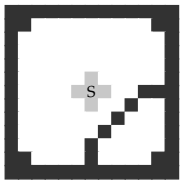
None



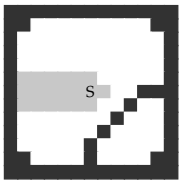
A or B, ii



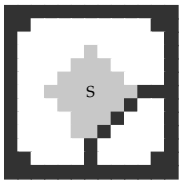
None



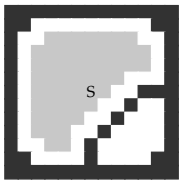
None



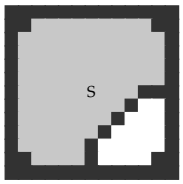
None



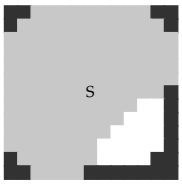
None



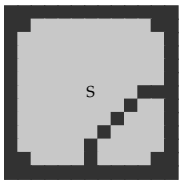
None



A or B, iii



None



B, i

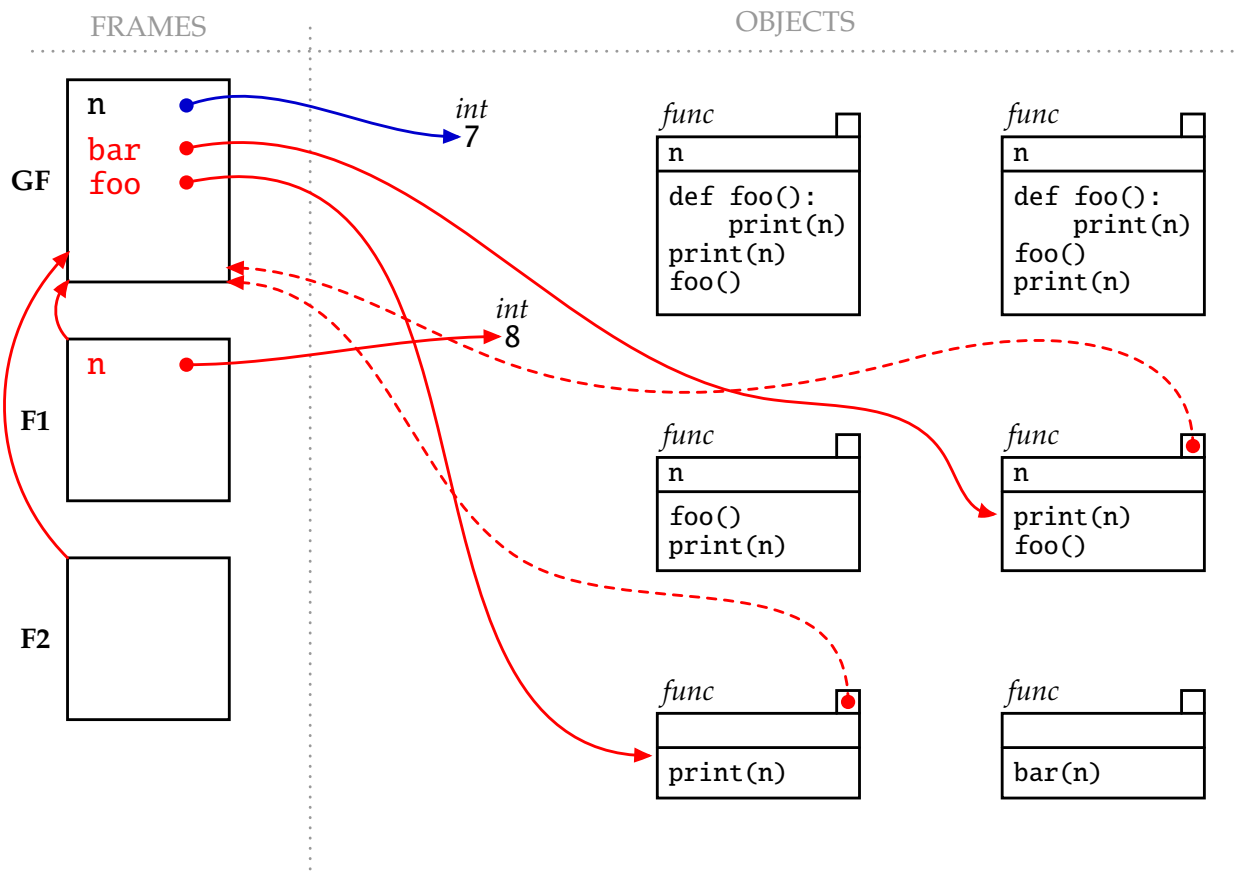
A, i

4 Environments

Part a.

Consider the execution of a program, after which evaluating the expression `bar(8)` prints an `8` followed by a `7`. The following shows a partially complete environment diagram representing the state of this evaluation just before the `7` is printed.

The diagram has all of the necessary frames and objects, but it is missing some pieces (specifically, variable bindings, enclosing frames for function objects, and parent pointers for frames).



Is it possible for this description to be consistent with the environment diagram on the facing page? (Yes/No)

Yes

If yes, add the missing variable bindings, enclosing environment pointers, and parent pointers (but **no new objects**) to the diagram on the facing page; and write the corresponding code that generates that diagram in the box below. Note that you do not have to use all of the objects present in the diagram.

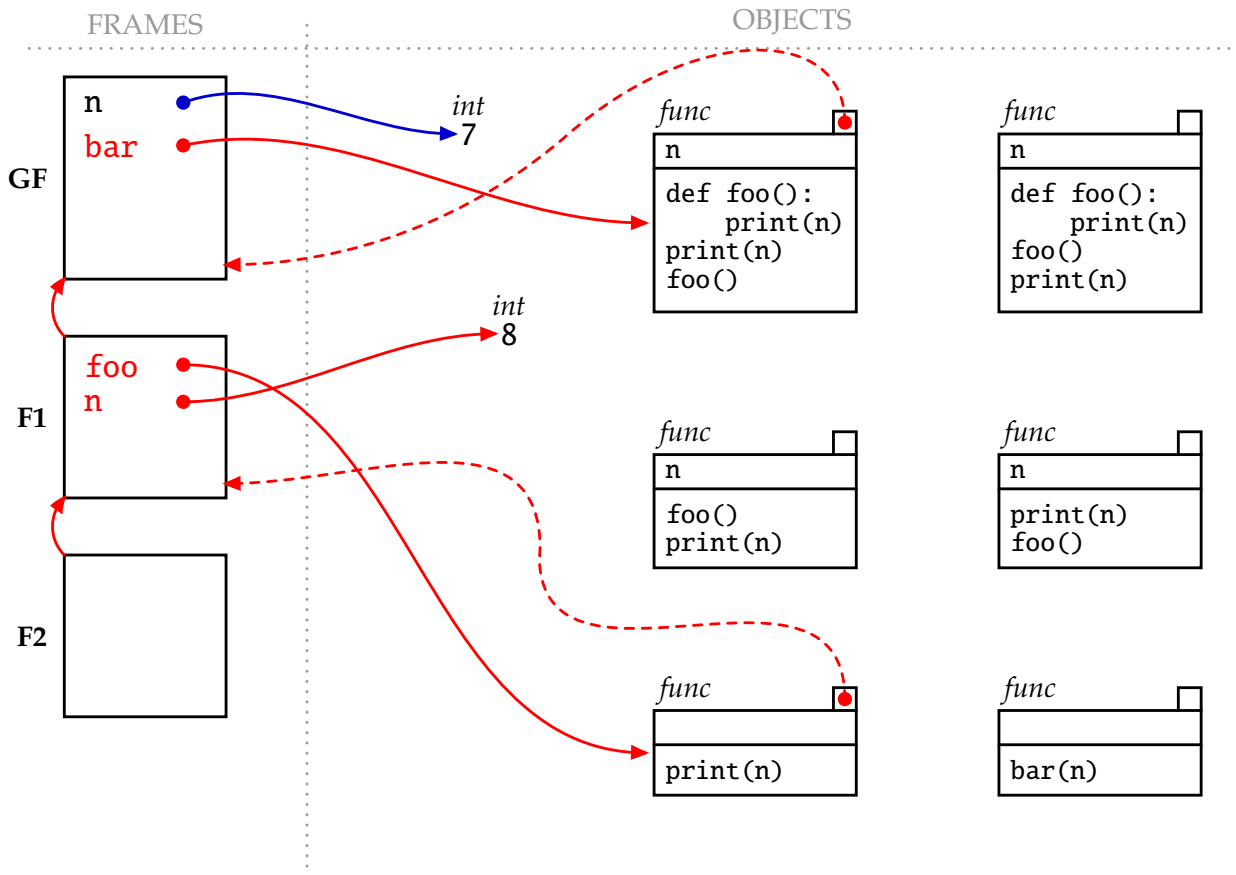
If no, briefly describe why this is not possible.

```
n = 7

def foo():
    print(n)

def bar(n):
    print(n)
    foo()

bar(8)
```

Part b.

Consider the execution of a different program, after which evaluating the expression `bar(8)` prints an `8` followed by another `8`. The following shows a partially complete environment diagram representing the state of this evaluation just before the second `8` is printed.

The diagram has all of the necessary frames and objects, but it is missing some pieces (specifically, variable bindings, enclosing frames for function objects, and parent pointers for frames).



Is it possible for this description to be consistent with the environment diagram on the facing page? (Yes/No)

Yes

If yes, add the missing variable bindings, enclosing environment pointers, and parent pointers (but **no new objects**) to the diagram on the facing page; and write the corresponding code that generates that diagram in the box below. Note that you do not have to use all of the objects present in the diagram.

If no, briefly describe why this is not possible.

```
n = 7

def bar(n):
    def foo():
        print(n)
    print(n)
    foo()

bar(8)
```

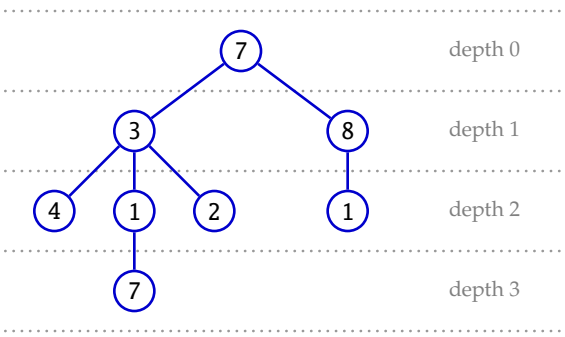
5 Trees

In this problem, we want to find all of the elements in a tree that are at a given depth in the tree. Each node in the tree is represented as a dictionary with two keys:

- `'v'` maps to the node's value (an integer).
- `'c'` maps to a list containing the node's children (each of which is itself a tree of this same form).

For example, the following code (`tree1`) and picture (with depth levels labeled) represent the same tree.

```
tree1 = {'v': 7,
         'c': [{'v': 3, 'c': [{'v': 4, 'c': []},
                              {'v': 1, 'c': [{'v': 7, 'c': []}]},
                              {'v': 2, 'c': []}]},
               {'v': 8, 'c': [{'v': 1, 'c': []}]}]}
```



On the facing page, complete the definition of the function `vals_at_depth(tree, depth)`, which should return a set of all the distinct values at a specified `depth` in the tree (where depth 0 corresponds to the root of the tree).

Examples:

- `vals_at_depth(tree1, 0)` returns `{7}`
- `vals_at_depth(tree1, 1)` returns `{3, 8}`
- `vals_at_depth(tree1, 2)` returns `{4, 1, 2}`
- `vals_at_depth(tree1, 3)` returns `{7}`
- `vals_at_depth(tree1, 4)` returns `set()`

```
def vals_at_depth(tree, depth):
    assert depth >= 0  # negative depth doesn't make sense

    if depth == 0:
        return {tree['v']}

    out = set()
    for child in tree['c']:
        out |= vals_at_depth(child, depth-1)
    return out
```

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Code For Flood Fill

```
def flood_fill(image, clicked_location, fill_color):
    original_image = dict(image)
    original_image['pixels'] = image['pixels'][:]

    clicked_color = get_pixel(original_image, *clicked_location)


    def neighbors(location):
        x, y = location
        neighbors = []
        for (dx, dy) in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            new_x = x + dx
            new_y = y + dy
            if 0 <= new_x < get_width(image) and 0 <= new_y < get_height(image):
                neighbors.append((new_x, new_y))
        return neighbors


    agenda = [clicked_location]
    visited = {clicked_location}

    while agenda:
        location = agenda.pop(0)

        ##################
        # Code Segment 1 #
        ##################

        for child in neighbors(location):

            ##################
            # Code Segment 2 #
            ##################

            if child not in visited:
                agenda.append(child)
                visited.add(child)

    return image
```

Worksheet (intentionally blank)