6.101 Quiz 1

Fall 2022

Name: **Answers**

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)

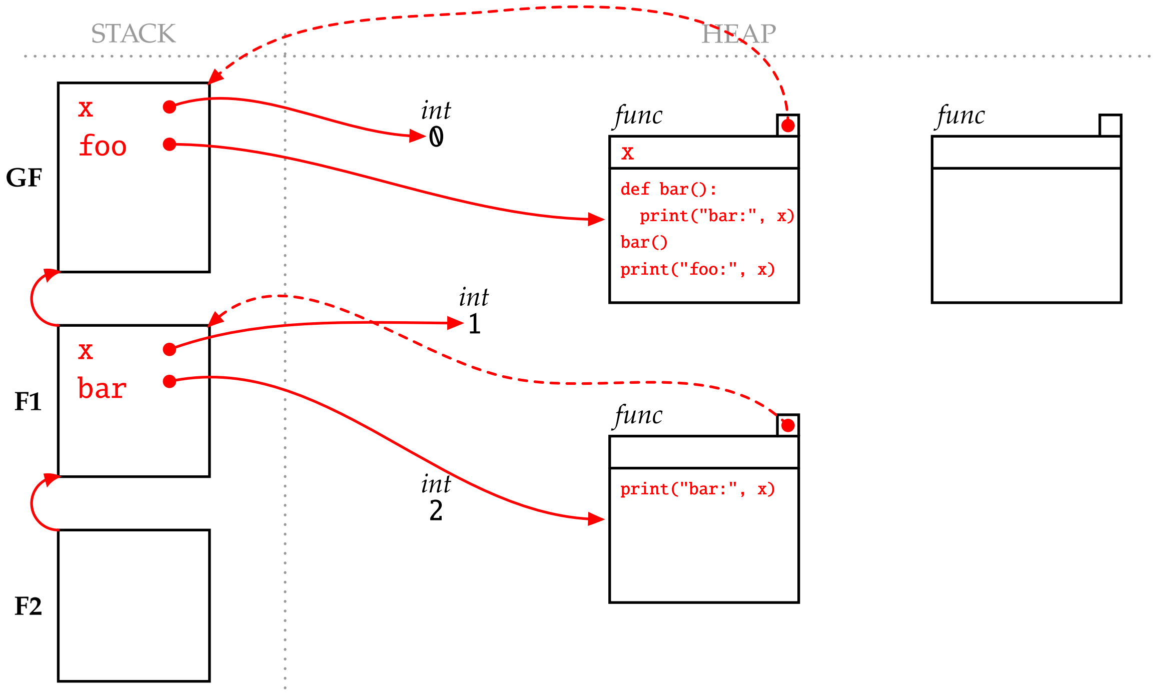
1 Environment Model: Function Calls

For each of the two pieces of code on the following pages, fill in the associated environment diagram and indicate what will be printed to the screen when the code is run. Each partial environment diagram contains all of the frames and objects necessary but is missing several pieces, which you should fill in. You need not write the full body of each function inside the associated object, but your diagram should otherwise be complete and represent the state of the program when all code is run but before garbage collection removes any objects. If a piece of code would raise an exception, instead write a brief description of the error and its cause.

Note also that `print` does not create new frames, so there is no need to include those in your environment diagram.

1.1 Fragment 1

```
x = 0
def foo(x):
    def bar():
        print("bar:", x)

    bar()
    print("foo:", x)

foo(1)
print("outside:", x)
```

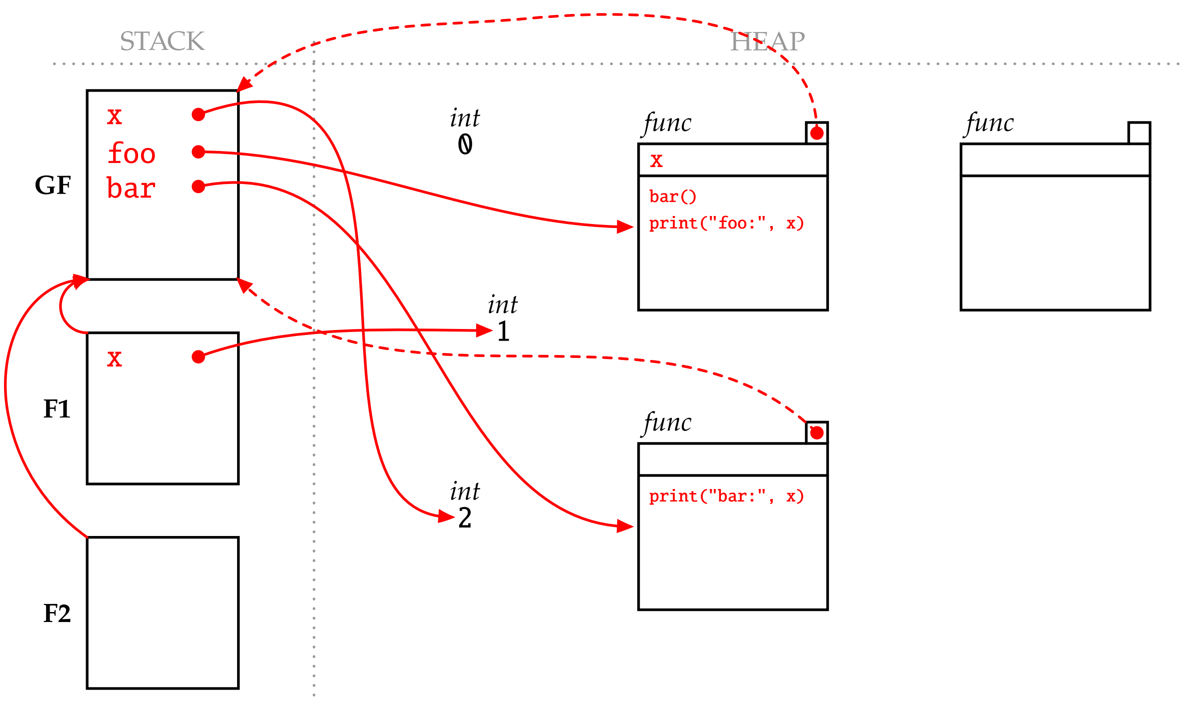


Output or brief description of error:

```
bar: 1
foo: 1
outside: 0
```

1.2 Fragment 2

```
x = 0
def foo(x):
    bar()
    print("foo:", x)

def bar():
    print("bar:", x)

x = 2
foo(1)
print("outside:", x)
```



Output or brief description of error:

```
bar: 2
foo: 1
outside: 2
```

2 Refactoring

Your roommate, Lem E. Tryit, is trying their hand at implementing code for dealing with color images like those we saw in 6.101 lab 2. They show you the following code for a helper function they have been working on. This code works exactly as they intend, but they ask if it could be written more concisely.

```
def get_color_value(image, color, x, y):
    red = {
        "height": image["height"],
        "width": image["width"],
        "pixels": [
            image["pixels"][j*image["width"] + i][0]
            for j in range(image["height"]) for i in range(image["width"])
        ],
    }
    green = {
        "height": image["height"],
        "width": image["width"],
        "pixels": [
            image["pixels"][j*image["width"] + i][1]
            for j in range(image["height"]) for i in range(image["width"])
        ],
    }
    blue = {
        "height": image["height"],
        "width": image["width"],
        "pixels": [
            image["pixels"][j*image["width"] + i][2]
            for j in range(image["height"]) for i in range(image["width"])
        ],
    }
    if color == "red":
        if y >= image["height"] or y < 0 or x >= image["width"] or x < 0:
            return None
        return red["pixels"][y*image["width"] + x]
    if color == "green":
        if y >= image["height"] or y < 0 or x >= image["width"] or x < 0:
            return None
        return green["pixels"][y*image["width"] + x]
    if color == "blue":
        if y >= image["height"] or y < 0 or x >= image["width"] or x < 0:
            return None
        return blue["pixels"][y*image["width"] + x]
    return None
```

In the box below, write a version of Lem's `get_color_value` function that avoids as much repetitious and/or redundant code as possible.

```
def get_color_value(image, color, x, y):
    index_map = {'red': 0, 'green': 1, 'blue': 2}
    if (0 <= x < image['width']
            and 0 <= y < image['height']
            and color in index_map):
        rgb_value = image['pixels'][y*image['width'] + x]
        return rgb_value[index_map[color]]
```

(many solutions are reasonable; this is just one example)

3 Word Chains

Inspired by the "word ladder" puzzles in the 6.101 week 4 readings, your friend Ward Smith decides to write code to solve another kind of linguistic puzzle. A "word chain" is a sequence of words where the last n characters of each word in the sequence are the first n characters of the next word in the sequence, for some value of n. For example, the following is a valid word chain for $n = 3$ since the last three letters of each word match the first three letters of the word that follows it:

atom $\rightarrow$ tomato $\rightarrow$ atonement $\rightarrow$ entrant $\rightarrow$ antifungal $\rightarrow$ galaxy

Ward has written the following code to solve puzzles of this form by finding the shortest word chain starting with a given word and ending with another (assuming that `ALL_WORDS` is a set containing a string for each word that is considered to be valid):

```
def setup_word_chain(start, end, n):
    assert len(start) >= n

    start_state = start

    def goal_test(state):
        return state == end

    def word_chain_neighbors(state):
        last_n = state[-n:]
        out = []
        for word in ALL_WORDS:
            if word[:n] == last_n:
                out.append(word)
        return out

    return word_chain_neighbors, start_state, goal_test
```

This code is intended for use in conjunction with the `find_path` function from the 6.101 week 4 readings. This code is reproduced on the last page of this handout (page 23), which you may remove. With that function, Ward's code can be used like so:

```
>>> graph, start, goal_test = setup_word_chain('cat', 'tomato', 2)
>>> print(find_path(graph, start, goal_test))
("cat", "atypical", "alto", "tomato")
>>> graph, start, goal_test = setup_word_chain('atom', 'galaxy', 3)
('atom', 'tomato', 'atonement', 'entrant', 'antifungal', 'galaxy')
```

This code works as written, but it takes a long time for some puzzles, especially when `ALL_WORDS` contains a lot of words. Given your experience with 6.101 labs 3 and 4, Ward comes to you for help with speeding up their code. On the facing page, fill in the boxes to produce a updated versions of `start_state`, `goal_test`, and `word_chain_neighbors`, as well as any other data structures needed, so as to solve this problem much more efficiently.

```
def setup_word_chain(start, end, n):
    assert len(start) >= n

    # additional code inside of setup_word_chain (if necessary)
    prefix_map = {}
    for word in ALL_WORDS:
        prefix_map.setdefault(word[:n], []).append(word)

    start_state = start

    def goal_test(state):
        return state == end

    def word_chain_neighbors(state):
        return prefix_map.get(state[-n:], [])

    # last line of setup_word_chain
    return word_chain_neighbors, start_state, goal_test
```

4 Undo

Wanting to go back in time to undo his promise to purchase the massive social-media company Fritter, powerful CEO Merlon Tusk has been experimenting with a new kind of programming-language feature that allows for "undo" functionality, which can be used to revert variables to older values. Merlon gave the following transcript to his engineering team, representing the desired behavior:

```
>>> x_set, x_get, x_undo = undoable("no i don't want to buy")
>>> x_get()
"no i don't want to buy"

>>> x_set("fifty cents")
>>> x_set("40 billion dollars")
>>> x_get()
"40 billion dollars"

>>> y_set, y_get, y_undo = undoable("self-driving cars may be possible at some point")
>>> y_set("self-driving cars by 2010")
>>> y_get()
"self-driving cars by 2010"

>>> x_undo()
>>> x_get()
"fifty cents"

>>> y_undo()
>>> y_get()
"self-driving cars may be possible at some point"

>>> x_undo()
>>> x_get()
"no i don't want to buy"

>>> x_undo()
>>> x_get()
"no i don't want to buy"

>>> # (continuing to 'undo' does not change the value at this point)
```

Consider the incomplete code on the facing page, which represents a start toward implementing the "undo" functionality described above. If the function can be completed so that it behaves as expected, fill in the boxes to complete the definition. Otherwise (if this structure cannot be made to work), instead briefly describe why in the box below the function.

```
def undoable(initial_value):
    values = [initial_value]

    def set_value(value):
        values.append(value)

    def undo():
        if len(values) > 1:
            values.pop()

    def get_value():
        return values[-1]

    return (set_value, get_value, undo)
```

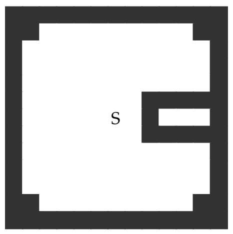
If the function cannot be completed correctly given this structure, briefly describe why below:
N/A (the answer above works)

5 Flood Fill Variants

In this problem, we will consider an implementation of the "flood fill" program from week 3's readings, 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 below. Note that this code is correct and *almost* complete except for a missing segment of code near the bottom.

```
def flood_fill(image, location, new_color):
    original_color = get_pixel(image, *location)

    # x is the horizontal component, increasing to the right
    # y is the vertical component, increasing downwards
    # x=0, y=0 corresponds to the upper-left corner of the image
    def get_neighbors(cell):
        x, y = cell
        potential_neighbors = [(x, y-1), (x, y+1), (x-1, y), (x+1, y)] # up, down, left, right
        return [
            (nx,ny)
            for nx,ny in potential_neighbors
            if 0 <= nx < get_width(image) and 0 <= ny < get_height(image)
        ]

    to_color = [location]
    visited = {location}

    while to_color:
        this_cell = to_color.pop(0)

        set_pixel(image, *this_cell, new_color)

        neighbors = [
            neighbor
            for neighbor in get_neighbors(this_cell)
            if (neighbor not in visited
                    and get_pixel(image, *neighbor) == original_color)
        ]

        ################
        # MISSING CODE #
        ################

        for neighbor in neighbors:
            visited.add(neighbor)

    return image
```

Here, we'll consider running the flood-fill process in the following image by clicking on the location labeled "S" and filling it with a grey color. The second-from-last page of this handout (page 21, which you may remove) contains several copies of the original image, which you may find helpful when thinking through the question below (though you do not need to use them if you don't want to).



Below are several snippets that could be used to fill in the missing part of the code on the facing page. For each, write a single letter in the box representing the resulting image **just before we remove the 26th item from the agenda**. The candidate images can be found on the third-from-last page of this handout (page 19), which you may remove. You may safely assume that the output for every code snippet is included on that sheet.

Code Segment 1

```
for neighbor in neighbors:
    to_color.append(neighbor)
```

Corresponding image: E

Code Segment 2

```
for neighbor in neighbors:
    to_color.insert(0, neighbor)
```

Corresponding image: Q

Code Segment 3

```
to_color += neighbors
```

Corresponding image: E

Code Segment 4

```
to_color = neighbors + to_color
```

Corresponding image: S

Worksheet (intentionally blank)

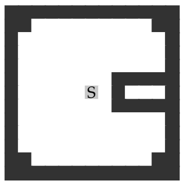
Worksheet (intentionally blank)

Worksheet (intentionally blank)

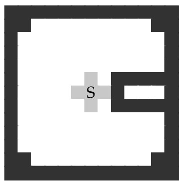
Worksheet (intentionally blank)

Worksheet (intentionally blank)

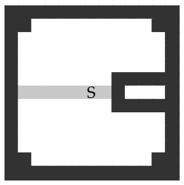
Options for Flood-Fill Question



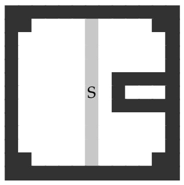
A
(1 grey pixel)



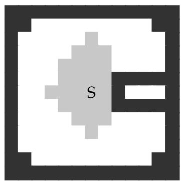
B
(5 grey pixels)



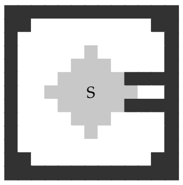
C
(7 grey pixels)



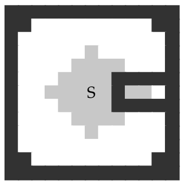
D
(11 grey pixels)



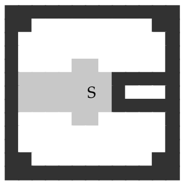
E
(25 grey pixels)



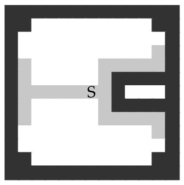
F
(25 grey pixels)



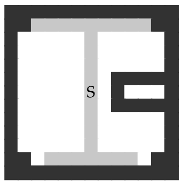
G
(25 grey pixels)



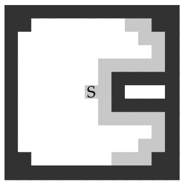
H
(25 grey pixels)



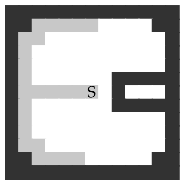
I
(25 grey pixels)



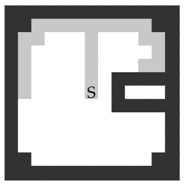
J
(25 grey pixels)



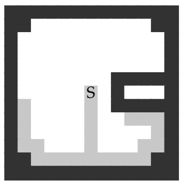
K
(25 grey pixels)



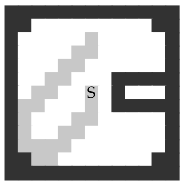
L
(25 grey pixels)



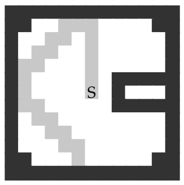
M
(25 grey pixels)



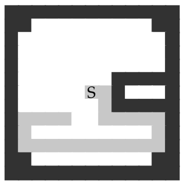
N
(25 grey pixels)



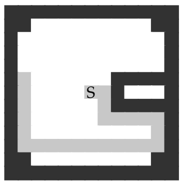
O
(25 grey pixels)



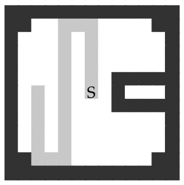
P
(25 grey pixels)



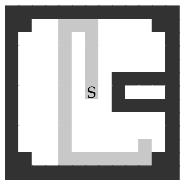
Q
(25 grey pixels)



R
(25 grey pixels)



S
(25 grey pixels)

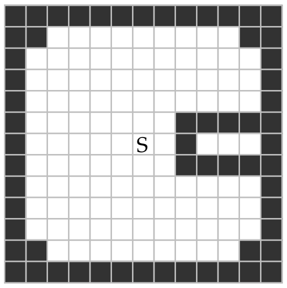


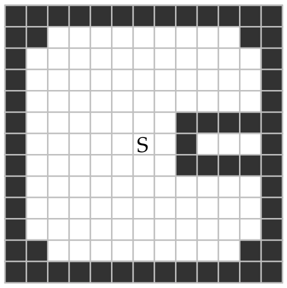
T
(25 grey pixels)

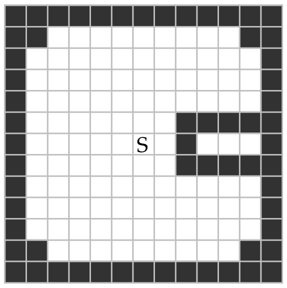
Worksheet (intentionally blank)

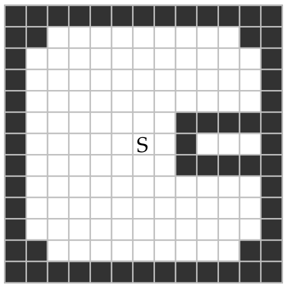
Optional Extra Workspace for Flood-Fill Question

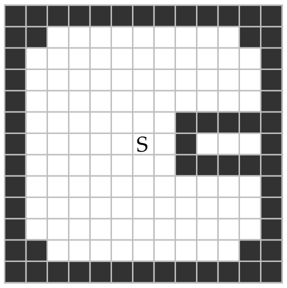
Below are 16 copies of the original image from the flood-fill question, which you are welcome to use as you are working through the problem. Note that **your work on this page will not be graded**.

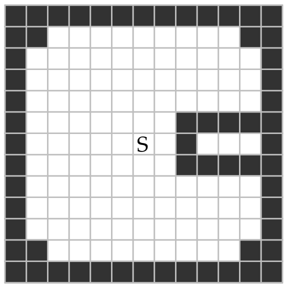


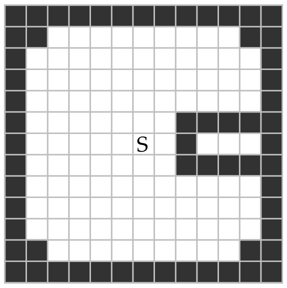


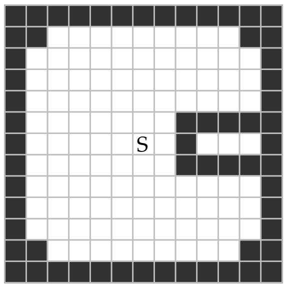


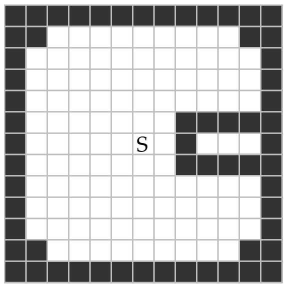


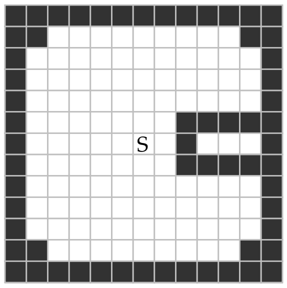


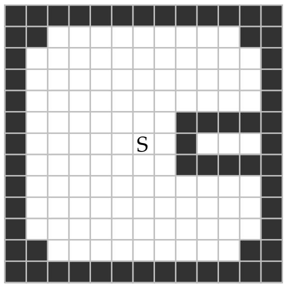


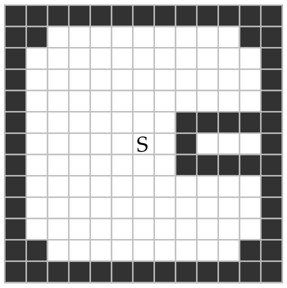


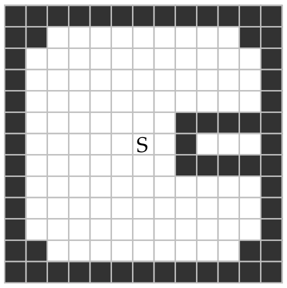


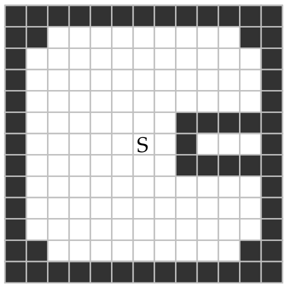


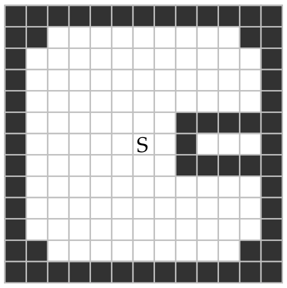


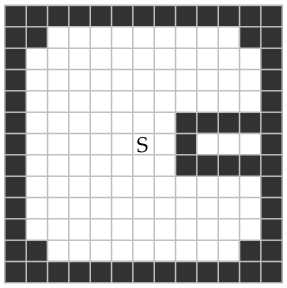












Worksheet (intentionally blank)

Code for `find_path` Function

```python
def find_path(neighbors_function, start, goal_test):
    if goal_test(start):
        return (start,)

    agenda = [(start,)]
    visited = {start}

    while agenda:
        this_path = agenda.pop(0)
        terminal_state = this_path[-1]

        for neighbor in neighbors_function(terminal_state):
            if neighbor not in visited:
                new_path = this_path + (neighbor,)

                if goal_test(neighbor):
                    return new_path

                agenda.append(new_path)
                visited.add(neighbor)
```

Worksheet (intentionally blank)