CS 88 Midterm Solutions [Sp 21]

[Blank Exam]

Question 2: Conceptual Questions

a) What are the two different ways a function can be a higher order function?

A higher order function is a function that takes in a function as an argument or returns a function.

b) What is a recursive function and how do you avoid infinite recursion?

A recursive function is a function that calls itself to solve a subproblem. To avoid infinite recursion, it is important to have a base case (the simplest case) that does not consist of a recursive call.

Question 3: What Made Python Print That

a) Create a list lst such that the following expression evaluates to 10.

```
>>> lst = [_______________]
>>> len(lst) == len(lst[1]) and (not lst[2]) and lst[0] * 2
10
```

Solution
Example: lst = [5, [1, 2, 3], 0]

Explanation:
For the following expression to evaluate to 10, the last subpart must evaluate to 10 an and expression evaluates to either the first falsy value or the last truthy value. Since 10 is a truthy value, the last sub expression must evaluate to 10, which means that lst[0] must be 5.

Additionally, the first two sub expressions must evaluate to Truthy values, because if either of them are Falsy values, the first Falsy value will be the value of the overall expression. So we make the len(lst) == len(lst[1]) meaning that lst[1] should be some iterable like a list so that we can find the length of it. And lst[2] must be a Falsy value so that not lst[2] is a Truthy value.

b)
```
>>> def f1(m):
     def f2(lst):
          c = 0
          for i in range(len(lst) - 1):
               if lst[i+1] - lst[i] == m:
                    c += 1
          return c
     return f2

>>> f1(____)(________________)
4
```

Write an expression that would evaluate to 4 by using the above function and identifying what goes in the blanks.

Solution
Example: f1(3)([1, 4, 7, 10, 13])

Explanation:
f1 takes in a number m and returns a function that takes in a list lst and returns the number of elements in the list that are m greater than the element to their left.

In the example solution, there are 4 numbers that are m = 3 greater than the number to their left: 4 (since its left neighbor is 1), 7 (since its left neighbor is 4) 10 (since its left neighbor is 7), and 13 (since its left neighbor is 10) so this expression would return 4.

c) Assuming `apple`, `orange`, and `pear` are variables with integer values and `salad` is a dictionary, answer the two following questions.

i. The statement below will Always/Sometimes/Never evaluate to a Truthy value if `orange - apple == 5`.

```
>>> (pear > orange) or (apple or orange)
```

Always
Sometimes
Never

Solution:
Always

Explanation:
If orange - apple = 5, then at least one of them must be a nonzero number so the second part of the expression is always Truthy. Thus, the expression will definitely evaluate to some Truthy value, regardless of the value of the pear > orange sub expression.

ii. The following expression will Always/Sometimes/Never evaluate to a Falsy value if the variable `pear` exists **at least once as a value** in the dictionary `salad`.

```
>>> len(list(salad.values())) == len(list(salad.keys())) and
pear in salad
```

Always
Sometimes
Never

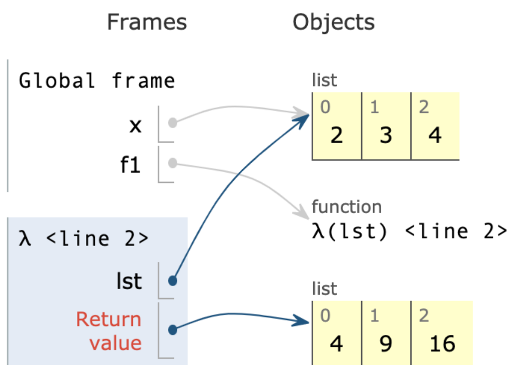
Solution:
Sometimes

Explanation:
the first subpart of the expression will always be True because a dictionary is a collection of key/value pairs, so there must be the same number of keys as there are values. For the second subpart of the expression, "pear in salad", this will look to see if "pear" exists as a **key** in the dictionary "salad" but we are only told that it exists as a **value**, so this subpart could either be True or False. As such, this overall expression will sometimes evaluate to False and sometimes evaluate to True.

Question 4: Reverse Environment Diagram

Copy and paste the skeleton code from above into the box below and fill in the blanks in the code so that when it is executed, it will result in the given environment diagram.

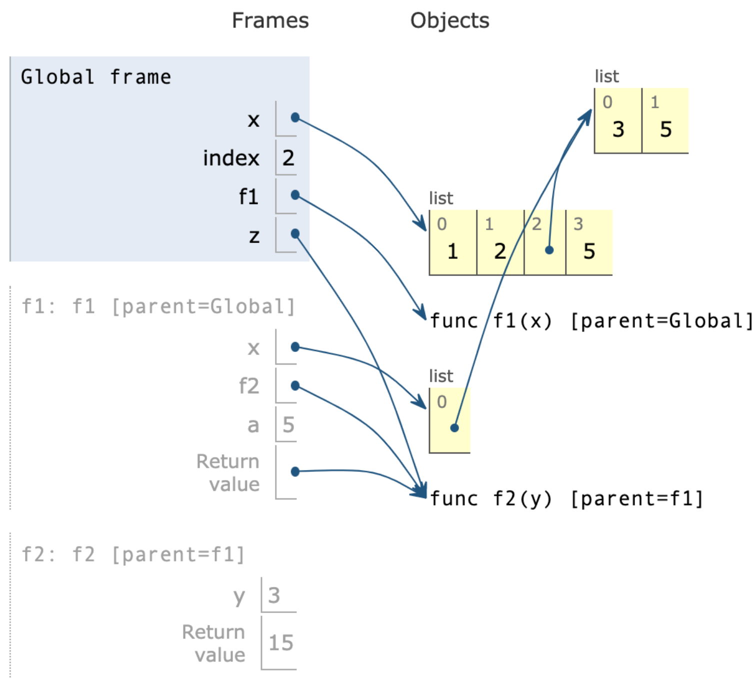
a)



Python Tutor Link Solution

```
x = [2, 3, 4]
f1 = lambda lst : [x**2 for x in lst]
f1(x)
```

b)



[Python Tutor Solution Link]

```
x, index = [1, 2, [3, 4], 5], 2

def f1(x):
    def f2(y):
        return  y * a
    a = x.pop()
    x[0][1] = a
    return f2

z = f1(x[index:])

z(3)
```

Question 5: The Ehrman Elevator

a) One of the Unit 2 elevators has broken down yet again and all the code was lost! Fill in the function `elevator` so that it returns a list order that represents the order of the floors from the list `requests` that the elevator should visit. Each floor in the `requests` list is represented as an integer. The `next_floor` function takes in the current floor `cur_floor` and a non-empty list requests that contains the floors that the elevator still must go to. `Next_floor` returns an integer representing the next floor that the elevator should go to next and removes this integer from requests.

Write the completed elevator function below. You may not add new lines.

```
def elevator(cur_floor, requests, next_floor):
        """
        >>> def smallest_to_largest(cur_floor, requests):
        ...     smallest = min(requests)
        ...     requests.remove(smallest)
        ...     return smallest
        >>> elevator(4, [55, 2, 1, 3], smallest_to_largest)
        [1, 2, 3, 55]
        >>> def descending_only(cur_floor, requests):
        ...     largest = max(requests)
        ...     while largest > cur_floor:
        ...     requests.remove(largest)
        ...     largest = max(requests)
        ...     requests.remove(largest)
        ...     return largest
        >>>  elevator(4, [55, 2, 1, 3], descending_only)
        [3, 2, 1]
"""

    order = []
    while requests:
        cur_floor = next_floor(cur_floor, requests)
        order.append(cur_floor)
    return order
```

b) Now implement the `closest_floor` function that can be passed into the elevator function for the `next_floor` parameter. `closest_floor` returns the floor from the

non-empty list requests that is closest to the current floor of the elevator `cur_floor`. The floor that is returned should be removed from `requests` within this function. Assume there are no ties for the closest floor. For more information about the `min` function, you may refer to the documentation from Problem 0.2 of the maps project.

```
def closest_floor(cur_floor, requests):
    """
    >>> requests = [1, 7]
    >>> closest_floor(6, requests)
    7
    >>> requests
    [1]
    >>> closest_floor(7, requests)
    1
    >>> elevator(4, [6, 10, 5, 1, 30], closest_floor)
    [5, 6, 10, 1, 30]
    """
    floor = min(requests, key=lambda x: abs(cur_floor - x))
    requests.remove(floor)
    return floor
```

Question 6: Exaggerateeee

Implement the `exaggerate` function that returns a string that is identical to the input string `phrase` except that every vowel in `phrase` is replaced by itself `repeat` times.

HINT: Recall that the operator * can be applied to strings (e.g. ‘ab’ * 2 = ‘abab’ and ‘c’ * 3 = ‘ccc’).

```
def exaggerate(words, vowels, repeat):
    """
    >>> vowels = ['a', 'e', 'i', 'o', 'u']
    >>> exaggerate("nice", vowels, 3)
    'niiiceee'
    >>> exaggerate("Woah so cool", vowels, 2)
    'Wooaah soo cooool'
    """
    if words == '':
        return ''
    else:
        cur = words[0]
        if cur in vowels:
            cur = cur * repeat
        rest = exaggerate(words[1:], vowels, repeat)
        return cur + rest
```

Question 7: Debugging

Your friend wrote a function that takes in a `start` and `stop` value. It returns another function that takes in a list and returns the product of all the numbers from `start`(inclusive) to `stop`(exclusive). However, it has exactly 3 distinct bugs.

1) Identify 3 unique bugs and explain how to fix the bug. After fixing all the bugs the code should work as intended
2) Write a working function for `productSlice`

```
def productSlice(start, stop):
    """
    >>> lst = [1,2,3,4,5]
    >>> productSlice(1,4)(lst) # 2 * 3 * 4
    24
    >>> productSlice(0,4)(lst) # 1 * 2 * 3 * 4
    24
    >>> productSlice(0,5)(lst) # 1 * 2 * 3 * 4 * 5
    120
    """
    def slicer(lst):
        output = 1
        while start <= stop:
            output = output * lst[start]
            start += 1
        return output
    return slicer(lst)

def productSlice(start, stop):
    """
    >>> lst = [1,2,3,4,5]
    >>> productSlice(1,4)(lst) # 2 * 3 * 4
    24
    >>> productSlice(0,4)(lst) # 1 * 2 * 3 * 4
    24
    >>> productSlice(0,5)(lst) # 1 * 2 * 3 * 4 * 5
    120
    """
```

```
def slicer(lst):
    n = start
    output = 1
    while n < stop:
        output = output * lst[n]
        n += 1
    return output
return slicer
```

Question 8: Shhh, I'm Booked

Your boss has asked you to implement an interface that allows users to donate and borrow books from local libraries. You've started some work, and have already built out methods for two abstract data types representing **libraries** and **users**.

A **user** can donate books to a **library** and borrow books from the **library**, but there are some constraints that need to be enforced.

For a **user** to borrow a book from a **library**, they must not already have the book (if not, print "The user already owns this book.") and the **library** they are borrowing from must have that book available (if not, print "The library does not have this book.").

For a **user** to donate a book to a **library**, they must currently have that book (if not, print "The user does not have this book.") and the **library** should not already be at max capacity (if it is at maximum capacity, print "The library is already at max capacity.").

The books are removed from the **user** and added to the **library** for a donation and added to the **user** and removed from the **library** for a borrow. Implement the borrow and donate functions below to implement the desired behavior.

You may find reading the doctests useful for understanding the behavior of these functions. Assume that the following functions of the ADTs are already defined for you (their implementation is hidden).

Note: not all the lines need to be used.

```
def borrow(user, lib, book):
    user_books = get_books(user)
    library_books = get_available_books(lib)
    if book not in library_books:
        print("This library doesn't have this book.)
    elif book in user_books:
        print("This user already has this book.)
    else:
```

```
        remove_book_for_lib(lib, book)
        add_book_for_user(user, book)

def donate(user, lib, book):
    user_books = get_books(user)
    library_books = get_available_books(lib)
    if get_capacity(lib) == len(library_books):
        print("This library is already at max capacity.)
    elif book not in get_books(user):
        print("This user does not have this book.)
    else:
        remove_book_for_user(user, book)
        add_book_for_lib(lib, book)
```

Note: For this question, the ordering of the first if/elif was graded leniently as the problem did not clearly state what to do in the case of

1) an attempted borrow - if the library didn't have the book **and** the user already had it, which error message should get printed first was unclear
2) an attempted donate - if the library is already at max capacity **and** the user does not have the book, which error message should get printed first was unclear

Question 9: Compress

Write a function `compress` that takes in a list of integers. It returns a compressed version of the integers in the form of a list that contains two element lists where the first element is a number and the second element is the amount of times that number appears consecutively.

Solution 1

```
def compress(lst):
    """
    >>> lst = [1,1,1,0,0,1]
    >>> compress(lst)
    [[1, 3], [0, 2], [1, 1]]
    # [1, 3] represents that there are 3 consecutive 1s
    # [0, 2] following that, there are 2 consecutive 0s
    # [1, 1] finally there is 1 consecutive 1s
    >>> lst2 = [1,0,1,0]
    >>> compress(lst2)
    [[1, 1], [0, 1], [1, 1], [0, 1]]
    >>> lst3 = [1,1,1,1,1]
    >>> compress(lst3)
    [[1, 5]]
    """

    #This is mostly just to make indexing easier in the for loop
    lst_copy = lst + ["#"]
    output = []
    curr = lst[0]
    count = 1
    for elem in lst_copy[1:]:
        if elem != curr:
            output.append([curr, count])
            curr = elem
            count = 0
        count += 1
    return output
```

Solution 2

```
def compress(lst):
    """
    >>> lst = [1,1,1,0,0,1]
```

```
>>> compress(lst)
[[1, 3], [0, 2], [1, 1]]
# [1, 3] represents that there are 3 consecutive 1s
# [0, 2] following that, there are 2 consecutive 0s
# [1, 1] finally there is 1 consecutive 1s
>>> lst2 = [1,0,1,0]
>>> compress(lst2)
[[1, 1], [0, 1], [1, 1], [0, 1]]
>>> lst3 = [1,1,1,1,1]
>>> compress(lst3)
[[1, 5]]
"""
if lst == []:
      return []
output = []
curr = lst[0]
count = 1
for elem in lst[1:]:
    print(elem)
    print(count)
    if elem != curr:
        output.append([curr, count])
        curr = elem
        count = 0
    count += 1
if count != 0: # to account for when we need to add the last "compressed" item
    output.append([curr, count])
return output
```