CS 61A Spring 2022

Structure and Interpretation of Computer Programs

Final Solutions

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address `<EMAILADDRESS>`. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

- ◯ You must choose either this option
- ◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

- ☐ You could select this choice.
- ☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

Preliminaries

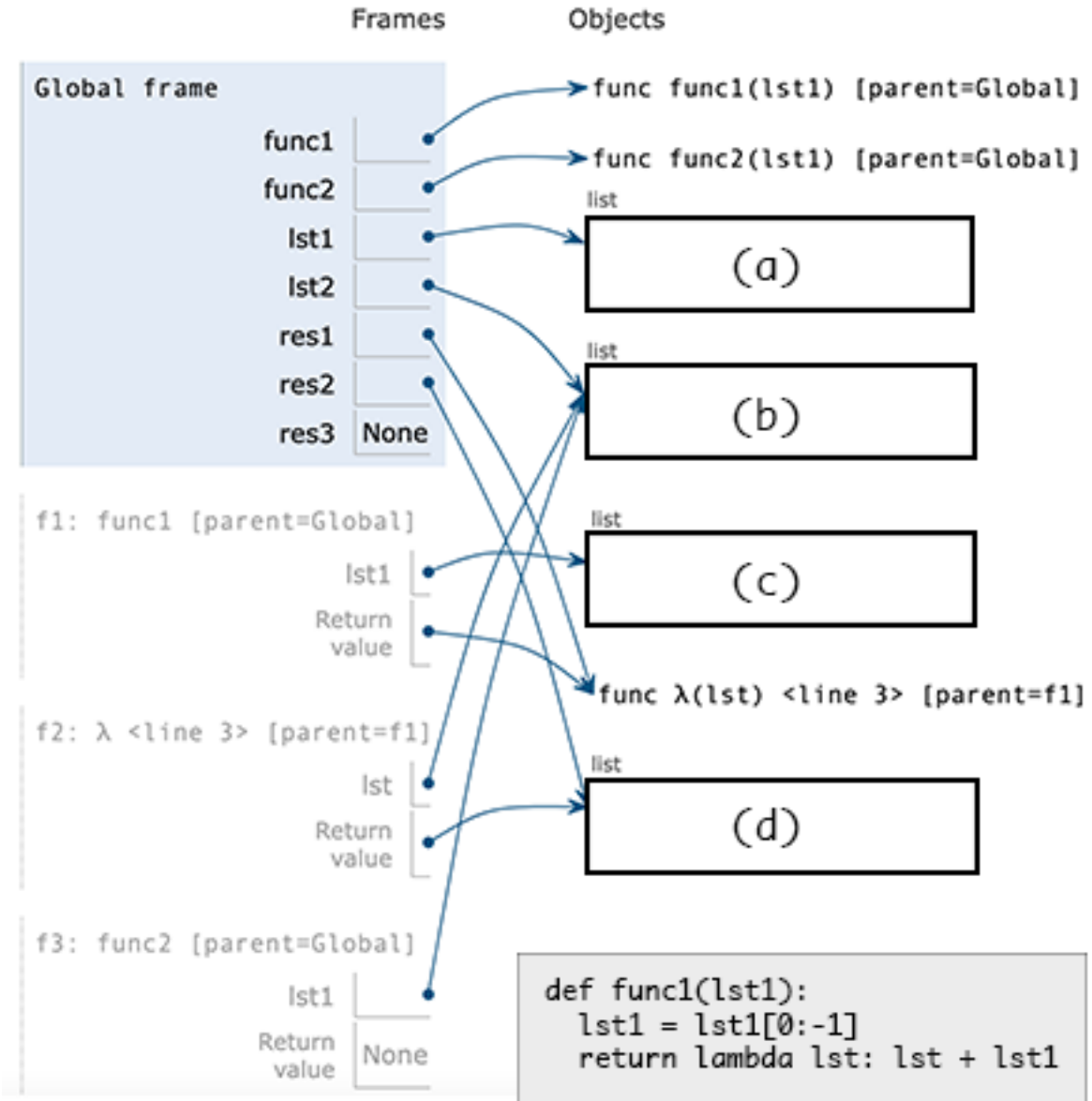
You can complete and submit these questions before the exam starts.

(a) What is your full name?

(b) What is your student ID number?

1. (5.0 points) Mutation Station

The environment diagram below was generated by code that is provided at the bottom right of the diagram. The diagram represents the full execution of the code.




```
def func1(lst1):
  lst1 = lst1[0:-1]
  return lambda lst: lst + lst1

def func2(lst1):
  lst1[0] = lst1.pop()

lst1 = [1, 19, 2022]
lst2 = [5, 10, 2022]

res1 = func1(lst1)
res2 = res1(lst2)
res3 = func2(lst2)
```

Each of the blanks represents a box and pointer diagram of a list. You can just use standard Python list notation for your answers, however (square brackets around comma separated numbers).

(a) **(1.0 pt)** Fill in blank (a).

```
[1, 19, 2022]
```

(b) **(1.0 pt)** Fill in blank (b).

```
[2022, 10]
```

(c) **(1.0 pt)** Fill in blank (c).

```
[1, 19]
```

(d) **(2.0 pt)** Fill in blank (d).

```
[5, 10, 2022, 1, 19]
```

2. (3.0 points) Filter Kilter

Consider the following code from a Python interpreter session:

```
>>> f = filter(lambda x: x % 3 == 0, range(1, 10))
>>> __________
3
>>> __________
6
>>> __________
9
>>> __________
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

(a) (1.0 pt) What line of code can fill in all of the blanks in that session? (The same line of code should work for all of them)

```
next(f)
```

(b) (2.0 pt) Now consider what would happen if the following code was run *immediately after* the code above. What would be displayed?

```
>>> for v in f:
...     print(v)
```

If there would be multiple lines in the output, use multiple lines in your answer. If the output would be an error, write "Error". If there would be no output, write "Nothing".

Nothing

3. (2.0 points) LinkMaker, LinkMaker, Make Me A Link

Consider the fully implemented `linkerator` function:

```
def linkerator(num):
    l = Link(num)
    whole_l = l

    while True:
        yield whole_l
        num += 1
        l.rest = Link(num)
        l = l.rest
```

A student tries using that function but encounters an error:

```
>>> g = linkerator(5)
>>> g[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not subscriptable
```

(a) (1.0 pt) What line of code could they write instead of `g[0]` that would run without error and yield the result `Link(5)`?

```
next(g)
```

(b) (1.0 pt) What is an expression that includes a call to `linkerator` and would cause an infinite loop? *(Note that it must be a single expression, not a statement or multiple statements).*

```
list(linkerator(5))
```

4. (4.0 points) Summer Reading List

The function `sort_books_by_option` takes two parameters, `books` (a list of tuples), and `sort_option` (a string), and returns a new list where the `books` are ordered according to the given `sort_option`.

The tuple that represents each book in the list of `books` starts with the title of the book and is followed by another tuple representing parts of the author's name.

```
('The Fifth Season', ('N.', 'K.', 'Jemisin'))
```

An author tuple will always have at least a first and last name, in that order (e.g. `('Frank', 'Herbert')`). If an author has a middle name, that'll be between the first and last name (e.g. `('Octavia', 'E.', 'Butler')`).

The `sort_option` can be either "title", "first_name", or "last_name", and is used to sort the books according to that part of the tuple (alphabetically, from A-Z). When two books share the same value for that option (e.g. the same author last name), it doesn't matter what order they show up in.

The skeleton code includes a call to the Python built-in function `sorted`, which takes an iterable and a `key` argument, and returns a new sorted list. The `key` must be a function that takes a single argument (an item in the iterable) and returns back the value to use when sorting that item. When given string values, it sorts the strings alphabetically. The `sorted` function is similar to `min` and `max` in how the `key` argument is used.

Complete the implementation of `sort_books_by_option` per the function description and doctests.

```
def sort_books_by_option(books, sort_option):
    """
    >>> scifi_books = [
    ... ('Dawn', ('Octavia', 'E.', 'Butler')),
    ... ('Dune', ('Frank', 'Herbert')),
    ... ('Wildseed', ('Octavia', 'E.', 'Butler')),
    ... ('The Fifth Season', ('N.', 'K.', 'Jemisin'))]
    >>> sort_books_by_option(scifi_books, 'title')
    [('Dawn', ('Octavia', 'E.', 'Butler')),
     ('Dune', ('Frank', 'Herbert')),
     ('The Fifth Season', ('N.', 'K.', 'Jemisin')),
     ('Wildseed', ('Octavia', 'E.', 'Butler'))]
    >>> sort_books_by_option(scifi_books, 'first_name')
    [('Dune', ('Frank', 'Herbert')), ('The Fifth Season',
     ('N.', 'K.', 'Jemisin')),
     ('Dawn', ('Octavia', 'E.', 'Butler')),
     ('Wildseed', ('Octavia', 'E.', 'Butler'))]
    >>> sort_books_by_option(scifi_books, 'last_name')
    [('Dawn', ('Octavia', 'E.', 'Butler')),
     ('Wildseed', ('Octavia', 'E.', 'Butler')),
     ('Dune', ('Frank', 'Herbert')),
     ('The Fifth Season', ('N.', 'K.', 'Jemisin'))]
    """
    sorter = None
    if sort_option == 'title':
        sorter = __________
                    (a)
    elif sort_option == 'first_name':
        sorter = __________
                    (b)
    elif sort_option == 'last_name':
        sorter = __________
                    (c)
    return sorted(books, key=__________)
                                (d)
```

(a) (1.0 pt) Fill in blank (a).

```
lambda book: book[0]
```

(b) (1.0 pt) Fill in blank (b).

```
lambda book: book[1][0]
```

(c) (1.0 pt) Fill in blank (c).

```
lambda book: book[1][-1]
```

(d) (1.0 pt) Fill in blank (d).

```
sorter
```

5. (5.0 points) Higher Order Constraints

The function `constrainer` has three parameters: `original_func` (a function), `min_val` (a number), and `max_val` (a number). It returns a function that accepts a single parameter (a number), constrains that number to be no less than `min_val` and no greater than `max_val`, calls the function `original_func` on the constrained number, and returns the result.

For example, this call returns a function that constrains the input of `calc_percent` to be between 0 and 300.

```
cp = constrainer(calc_percent, 0, 300)
```

If `cp` is called with a number less than 0, it will be turned into a 0 before being sent into the original `calc_percent` function. If `cp` is called with a number greater than 300, it will be turned into a 300 before being sent into the original `calc_percent` function.

Complete the `constrainer` function below per the description and doctests.

```
def constrainer(original_func, min_val, max_val):
    """
    Returns a function that constrains the inputs to the original function
    based on the given minimum and maximum.

    >>> # Calculates percentage of a score out of 300
    >>> calc_percent = lambda points: round(points / 300 * 100)
    >>> cp = constrainer(calc_percent, 0, 300)
    >>> cp(200)
    67
    >>> cp(0)
    0
    >>> cp(300)
    100
    >>> cp(-100) # Gets constrained to 0 when sent into calc_percent
    0
    >>> cp(360)  # Gets constrained to 300 when sent into calc_percent
    100
    """
    def new_func(input):
        constrained_input = __________
                                (a)

        ----------
            (b)
    ----------
        (c)
```

(a) **(2.0 pt)** Fill in blank (a). You may find it helpful to use `min`, `max`, or conditional expressions.

```
min(max(input, min_val), max_val)
```

(b) **(2.0 pt)** Fill in blank (b).

```
return original_func(constrained_input)
```

(c) **(1.0 pt)** Fill in blank (c).

```
return new_func
```

6. (1.0 points) Naming Is Hard

Consider this function that Pamela wrote to brainstorm baby names:

```
def name_options(names):
    """
    Returns possible combinations of first and middle names based on NAMES list,
    where a first name and middle name should not be the same.
    Any of the names in NAMES can be either a first or a middle name.

    >>> name_options(['Sequoia', 'Alexia'])
    ['Sequoia Alexia', 'Alexia Sequoia']
    >>> name_options(['Sierra', 'Elantris', 'Maritima', 'Armeria'])
    ['Sierra Elantris', 'Sierra Maritima', 'Sierra Armeria',
     'Elantris Sierra', 'Elantris Maritima', 'Elantris Armeria',
     'Maritima Sierra', 'Maritima Elantris', 'Maritima Armeria',
     'Armeria Sierra', 'Armeria Elantris', 'Armeria Maritima']
    """
    return [ f'{first} {middle}' for first in names
             for middle in names if first != middle]
```

(a) (1.0 pt) What is the order of growth of `name_options` in respect to the size of the input list `names`?

- ◯ Constant
- ◯ Logarithmic
- ◯ Linear
- ● Quadratic
- ◯ Exponential

7. (5.0 points) Cut the Deck

The function `subdeck` takes two parameters, `cards` (a linked list of integer values), and `max_value` (an integer).

The function returns a new linked list that consists only of the values from `cards` that add up to a total less than or equal to `max_value`. The new linked list contains values in the order seen in `cards`; it does not skip any values.

Complete the implementation of the `subdeck` function below per the function description and doctests.

```
def subdeck(cards, max_value):
    """
    Creates a new linked list only of the values from linked list CARDS that add
    up to a total less than or equal to MAX_VALUE.

    >>> cards = Link(7, Link(10, Link(11, Link(10))))
    >>> subdeck(cards, 21)
    Link(7, Link(10))
    >>> subdeck(cards, 28)
    Link(7, Link(10, Link(11)))
    >>> subdeck(cards, 37)
    Link(7, Link(10, Link(11)))
    >>> subdeck(cards, 38)
    Link(7, Link(10, Link(11, Link(10))))
    """
    if __________:
           (a)
        __________
           (b)
    __________
        (c)
```

(a) (2.0 pt) Fill in blank (a).

```
cards is Link.empty or cards.first > max_value
```

(b) (1.0 pt) Fill in blank (b).

```
return Link.empty
```

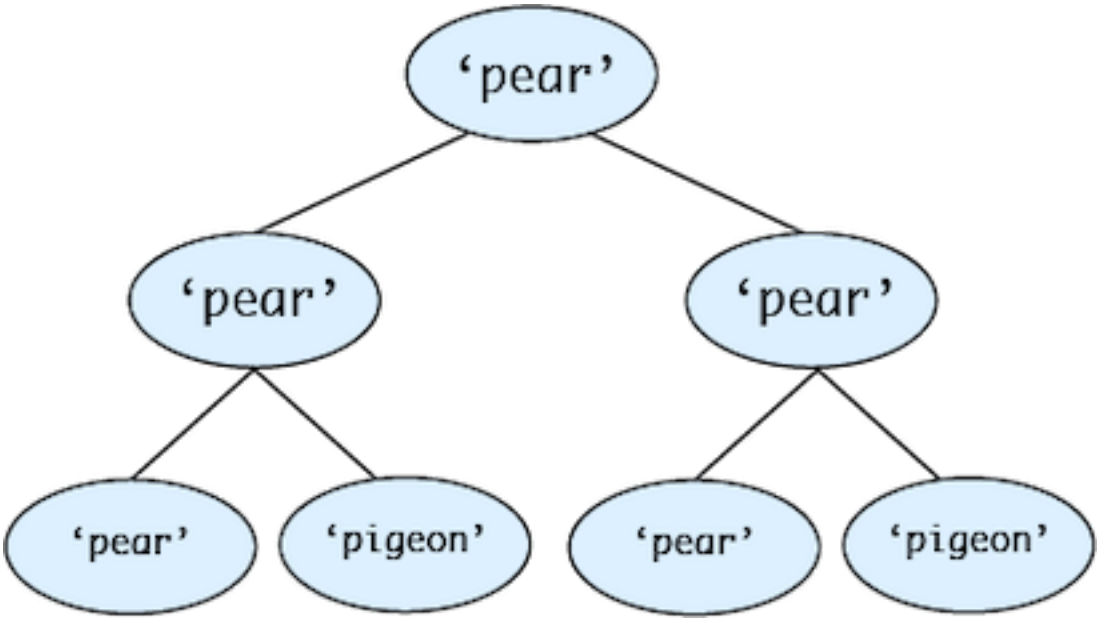
(c) (2.0 pt) Fill in blank (c).

```
return Link(cards.first, subdeck(cards.rest, max_value - cards.first))
```

8. (6.0 points) Pigeons in a Pear Tree

The function `pigeon_locations` accepts a single parameter `t`, an instance of the `Tree` class where all non-leaf nodes have two branches, the label of each node is either 'pear' or 'pigeon', and only leaf nodes can have the 'pigeon' label.

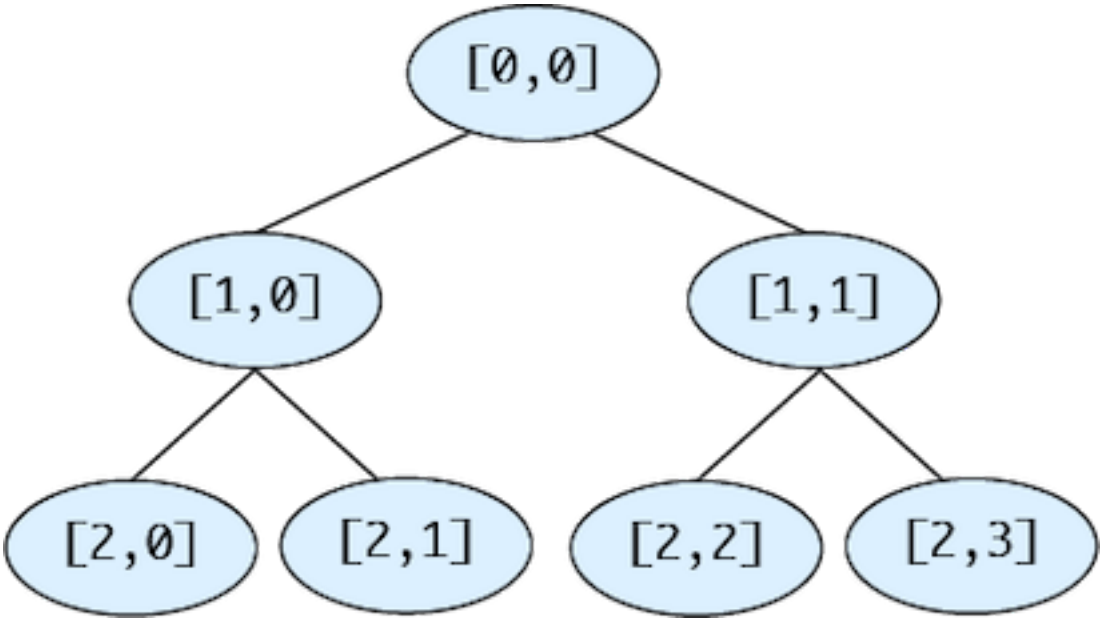
Here's a drawing of an example input tree:



The function returns a list of locations of pigeons in `t`, where each location is a two-element list with the "depth" as the first element and the "left" as the second element.

The "depth" of the root node is 0, and it increases by one at each level of the tree down from the root node. The left-most node of each level of the tree has a "left" of 0, and it increases from there.

Here's the same tree where each node is labeled with its location:



When `pigeon_locations` is called on the example tree, it returns `[[2, 1], [2, 3]]`, since the two pigeons are located at `[2, 1]` and `[2, 3]`.

Complete the `pigeon_locations` function below per the doctests and description.

```
def pigeon_locations(t):
    """
    Returns a list of location of pigeons in the tree T, where each location
    is a two-element list, with the depth as the first element
    and the left as the second element. The depth of the root node is 0 and
    increases from there. The left starts at 0 from the left-most branch of each tree.
    Every non-leaf node has two branches.

    >>> t1 = Tree('pear', [Tree('pigeon'), Tree('pear')])
    >>> print(t1)
    pear
      pigeon
      pear
    >>> pigeon_locations(t1)
    [[1, 0]]
    >>> t2 = Tree('pear', [Tree('pear', [Tree('pear'), Tree('pigeon')]),
    ...                    Tree('pear', [Tree('pigeon'), Tree('pear')])])
    >>> print(t2)
    pear
      pear
        pear
        pigeon
      pear
        pigeon
        pear
    >>> pigeon_locations(t2)
    [[2, 1], [2, 2]]
    >>> no_pigeons = Tree('pear', [Tree('pear'),
    ...                            Tree('pear', [Tree('pear'), Tree('pear')])])
    >>> pigeon_locations(no_pigeons)
    []
    """
    def helper(t, depth, left):
        if t.label == 'pigeon':
            return __________
                       (a)
        locations = [helper(_____, _____, _____)  for i in _____]
                             (b)    (c)    (d)              (e)
        return sum(locations, [])
    __________
       (f)
```

(a) **(1.0 pt)** Fill in blank (a).

```
[[depth, left]]
```

(b) **(1.0 pt)** Fill in blank (b).

```
t.branches[i]
```

(c) **(1.0 pt)** Fill in blank (c).

```
depth + 1
```

(d) **(1.0 pt)** Fill in blank (d).

```
(left * 2 ) + i
```

(e) **(1.0 pt)** Fill in blank (e).

```
range(len(t.branches))
```

(f) **(1.0 pt)** Fill in blank (f).

```
return helper(t, 0, 0)
```

9. (5.0 points) Santa Is Slacking

The function `wrap_it_up` takes a single argument `t`, an instance of the `Tree` class. The nodes in `t` have a label of either 'fir', 'wrapped', or 'gift'.

The function looks for any unwrapped gifts in the tree: nodes which have a label of 'gift' that are *not* the child of a node with the label 'wrapped'. When it finds an unwrapped gift, it mutates the tree to insert a new subtree with the label 'wrapped' as the parent of that 'gift' node.

All 'gift' nodes will be the children of either 'fir' or 'wrapped' nodes. The root node *cannot* have a 'gift' label.

Complete the implementation of `wrap_it_up` per the description and doctests.

```
def wrap_it_up(t):
    """
    >>> t = Tree('fir', [Tree('fir', [Tree('gift')]),
    ...                  Tree('fir', [Tree('wrapped', [Tree('gift')])])])
    >>> print(t)
    fir
      fir
        gift
      fir
        wrapped
          gift
    >>> wrap_it_up(t)
    >>> print(t)
    fir
      fir
        wrapped
          gift
      fir
        wrapped
          gift
    """
    if t.label == 'fir':
        for i in __________:
                     (a)
            b = t.branches[i]
            if __________:
                   (b)
                t.branches[i] = __________
                                    (c)

            __________
               (d)
```

(a) (1.0 pt) Fill in blank (a).

```
range(len(t.branches))
```

(b) (1.0 pt) Fill in blank (b).

```
b.label == 'gift'
```

(c) (2.0 pt) Fill in blank (c).

```
Tree('wrapped', [b])
```

(d) (1.0 pt) Fill in blank (d).

```
wrap_it_up(b)
```

10. (14.0 points) Going Shopping!

The next set of questions uses Python classes to represent data used in a grocery list application.

(a) Item

The first class needed is `Item`, which represents an item in the list. Each item has instance variables tracking the `name`, `quantity`, and `category`.

Complete the implementation of `Item` below so that the doctests pass.

```
class Item:
    """
    >>> broccoli = Item("broccoli", 1, "veggies")
    >>> broccoli.name
    'broccoli'
    >>> broccoli.quantity
    1
    >>> broccoli.category
    'veggies'
    >>> broccoli
    Item('broccoli', 1, 'veggies')
    """
    def __init__(self, name, quantity, category):
        self.name = name
        self.quantity = quantity
        self.category = category

    def __repr__(self):
        ----------
            (a)
```

i. (2.0 pt) Fill in blank (a).

```
return f'Item({repr(self.name)}, {repr(self.quantity)},
{repr(self.category)})'
```

(b) GroceryList

The next class is `GroceryList`, which represents a list for a particular store. Each instance of `GroceryList` has instance variables tracking the `store_name` and `items` (a list of `Item` instances).

The class has two methods:

- `add_new_item` which adds new items to `items` if they aren't already there.
- `all_for_category` which returns all the items for a given `category`.

Complete the implementation of `GroceryList` per the description and doctests.

```
class GroceryList:
    """
    >>> tjlist = GroceryList('Trader Joes')
    >>> tjlist.store_name
    'Trader Joes'
    >>> tjlist.items
    []
    >>> tjlist.add_new_item('Truffle Chips', 2, 'snacks')
    Item('Truffle Chips', 2, 'snacks')
    >>> tjlist.items
    [Item('Truffle Chips', 2, 'snacks')]
    >>> tjlist.add_new_item('Zesty Popcorn', 1, 'snacks')
    Item('Zesty Popcorn', 1, 'snacks')
    >>> tjlist.items
    [Item('Truffle Chips', 2, 'snacks'), Item('Zesty Popcorn', 1, 'snacks')]
    >>> tjlist.add_new_item('Truffle Chips', 3, 'snacks')
    >>> tjlist.add_new_item('Apple', 5, 'fruits')
    Item('Apple', 5, 'fruits')
    >>> tjlist.all_for_category('snacks')
    [Item('Truffle Chips', 2, 'snacks'), Item('Zesty Popcorn', 1, 'snacks')]
    >>> tjlist.all_for_category('fruits')
    [Item('Apple', 5, 'fruits')]
    """
    def __init__(self, store_name):
        self.store_name = store_name
        self.items = []

    def add_new_item(self, name, quantity, category):
        """Creates a new Item with the provided name, quantity, and category,
        adds the new item to the list's items, and returns it.
        If an item with that name already exists, it just returns None.
        """
        existing_item_names = __________
                                  (a)
        if __________:
              (b)
            new_item = __________
                           (c)

            __________
                (d)
            return new_item

    def all_for_category(self, category):
        """Returns a list of all the items for a given category."""
        __________
           (e)
```

i. **(1.0 pt)** Fill in blank (a).

```
[item.name for item in self.items]
```

ii. **(1.0 pt)** Fill in blank (b).

```
name not in existing_item_names
```

iii. **(1.0 pt)** Fill in blank (c).

```
Item(name, quantity, category)
```

iv. **(1.0 pt)** Fill in blank (d).

```
self.items.append(new_item)
```

v. **(2.0 pt)** Fill in blank (e).

```
return [item for item in self.items if item.category == category]
```

(c) SharableList

The final class is `SharableList`, a class that inherits from `GroceryList` and allows a shopping list to be shared by multiple users. Each instance of `SharableList` has the same instance variables as `GroceryList` (`store_name` and `items`) but also has two additional instance variables:

- `collaborators`: A list of email addresses, specified when constructing the instance.
- `items_by_adder`: A dictionary tracking which email address added which item. Starts off as a dictionary with a key for each email address mapped to an empty list.

To support tracking who added what, `SharableList` overrides the `add_new_item` method so that it calls the original method but then updates `items_by_adder` accordingly.

Complete the implementation of `SharableList` per the description and doctests.

```
class SharableList(GroceryList):
    """
    >>> roomie_list = SharableList('Trader Joes', ['don@key.com', 'star@burns.com'])
    >>> roomie_list.store_name
    'Trader Joes'
    >>> roomie_list.items
    []
    >>> roomie_list.collaborators
    ['don@key.com', 'star@burns.com']
    >>> roomie_list.items_by_adder
    {'don@key.com': [], 'star@burns.com': []}
    >>> roomie_list.add_new_item('Wasabi Peas', 100, 'snacks', 'don@key.com')
    Item('Wasabi Peas', 100, 'snacks')
    >>> roomie_list.items_by_adder
    {'don@key.com': [Item('Wasabi Peas', 100, 'snacks')], 'star@burns.com': []}
    """
    def __init__(self, store_name, collaborators):
        __________
           (a)
        self.collaborators = collaborators
        self.items_by_adder = {_____ for _____}
                                 (b)       (c)

    def add_new_item(self, name, quantity, category, adder):
        new_item = __________
                     (d)
        if new_item:
            __________
               (e)
        return new_item
```

i. (1.0 pt) Fill in blank (a).

```
super().__init__(store_name)
```

ii. (1.0 pt) Fill in blank (b).

```
collaborator:[]
```

iii. (1.0 pt) Fill in blank (c).

```
collaborator in self.collaborators
```

iv. (1.0 pt) Fill in blank (d).

```
super().add_new_item(name, quantity, category)
```

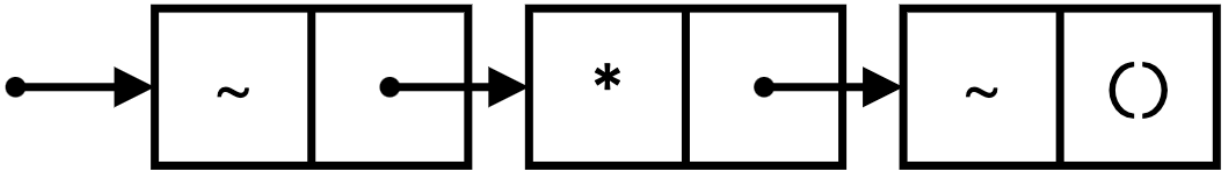
v. (2.0 pt) Fill in blank (e).

```
self.items_by_adder[adder].append(new_item)
```

11. (6.0 points) Beadazzled, The Scheme-quel

Implement `make-necklace`, a Scheme procedure that creates a Scheme list where each value comes from a given Scheme list of `beads`, and the beads are repeated in order to make a necklace of a given `length`.

For example, if `make-necklace` is called with `(~ *)` and a `length` of 3, then the linked list will contain `~`, then `*`, then `'~'`. Here's a diagram of that list:



See the docstring and doctests for further details on how the function should behave.

```
(define (make-necklace beads length)
  ; Returns a list where each value is taken from the BEADS list,
  ; repeating the values BEADS until the list has reached
  ; LENGTH. You can assume that LENGTH is greater than or equal to 1,
  ; and that there is at least one bead in BEADS.
  (if __________
         (a)
      __________
         (b)
      (cons __________
               (c)
            (make-necklace
             __________
               (d)
             __________
               (e)
             )
        )
    )
)

; Doctests
(expect (make-necklace '(~ *) 3)  (~ * ~))
(expect (make-necklace '(~ ^) 4) (~ ^ ~ ^))
(expect (make-necklace '(> 0 <) 9) (> 0 < > 0 < > 0 <))
```

(a) (1.0 pt) Fill in blank (a).

```
(= 0 length)
```

(b) (1.0 pt) Fill in blank (b).

```
nil
```

(c) (1.0 pt) Fill in blank (c).

```
(car beads)
```

(d) (2.0 pt) Fill in blank (d).

```
(append (cdr beads) (cons (car beads) nil))
```

(e) (1.0 pt) Fill in blank (e).

```
(- length 1)
```

12. (6.0 points) The Art of Abstraction

In this group of questions, you will use Scheme data abstractions to model pixels and icons.

A pixel represents a color. Computer monitors are made up of thousands of pixels, and images can also be stored as pixels.

Each pixel has three components: red, green blue. Each component has a value from 0 to 255. For example, a perfectly red pixel has a red component of 255, green component of 0, and blue component of 0. A purple pixel has a red component of 128, green component of 0, and blue component of 128.

The data abstraction for a pixel includes:

- The constructor `(pixel r g b)` which takes in three integers representing the red/green/blue components.
- The selectors `(red px)`, `(green px)`, and `(blue px)` which return the relevant component (as an integer value).

Here is how a pixel data abstraction can be constructed and used:

```
scm> (define purple-pixel (pixel 128 0 128))
purple-pixel
scm> (red purple-pixel)
128
scm> (green purple-pixel)
0
scm> (blue purple-pixel)
128
```

Here is one possible implementation for the data abstraction:

```
(define (pixel r g b)
    (cons r (cons g (cons b nil)))
)

(define (red px)
    (car px)
)

(define (green px)
    (car (cdr px))
)

(define (blue px)
    (car (cdr (cdr px)))
)
```

(a) (1.0 points) Red Removal

The `remove-red` procedure should take in a `pixel` data abstraction and return a new pixel where the red component is 0, but the green and blue components are the same as the original pixel.

Complete the `remove-red` procedure per the description and doctests. For full credit, your solution should use the data abstractions and *not* violate any abstraction barriers.

```
(define (remove-red px)
    ----------
)

; Doctests
(define changed-pixel (remove-red (pixel 255 125 50)))
(expect (red changed-pixel) 0)
(expect (green changed-pixel) 125)
(expect (blue changed-pixel) 50)
```

i. (1.0 pt) Fill in the blank.

```
(pixel 0 (green px) (blue px))
```

(b) (3.0 points) Grayscale

The `grayscale` procedure should take in a `pixel` data abstraction and return a new pixel that is a grayscale version of the original pixel. That is calculated by summing up all the red, blue, and green components, computing their average, and setting each component to the average. A grayscale pixel always has the same value for each component.

For example, if a pixel starts off with a red of 240, green of 120, and blue of 160, the grayscale version is 140, 140, 140.

Complete the `grayscale` procedure per the description and doctests. For full credit, your solution should use the data abstractions and *not* violate any abstraction barriers.

```
(define (grayscale px)
    (define sum-p __________)
                    (a)
    (define avg-p __________)
                    (b)

    __________
        (c)
)

; Doctests
(define grayed-pixel (grayscale (pixel 240 120 60)))
(expect (red grayed-pixel) 140)
(expect (green grayed-pixel) 140)
(expect (blue grayed-pixel) 140)
```

i. (1.0 pt) Fill in blank (a).

```
(+ (red px) (green px) (blue px))
```

ii. (1.0 pt) Fill in blank (b).

```
(quotient sum-p 3)
```

iii. (1.0 pt) Fill in blank (c).

```
(pixel avg-p avg-p avg-p)
```

(c) (2.0 points) Icons

The `icon` data abstraction represents a square icon with a size and pixel list.

The data abstraction for the `icon` data abstraction includes:

- The constructor `(icon size pixels)` which takes in an integer for the size and a list of `pixel` data abstractions for the pixels.
- The selector `(size ic)` for returning the icon's size (integer)
- The selector `(pixels ic)` for returning the icon's pixels (list of `pixel` abstractions).

For example, this code constructs a new 2x2 icon with the 4 pixels specified:

```
(define tiny (icon 2 (list (pixel 255 0 0)
                   (pixel 255 255 0)
                   (pixel 0 0 255)
                   (pixel 200 0 100)
                   )))
```

Here is a possible implementation for the `icon` data abstraction:

```
(define (icon size pixels)
    (cons size pixels)
)

(define (size ic)
    (car ic)
)

(define (pixels ic)
    (cdr ic)
)
```

The procedure `grayscale-icon` should take in an icon and return a new icon where every pixel is now grayscale (using the `grayscale` procedure defined earlier).

Complete the implementation of `grayscale-icon` per the description and doctests. For full credit, your solution should use the data abstractions and *not* violate any abstraction barriers.

```
(define (grayscale-icon i)
    --------------
)

; Doctests
(define tiny (icon 2 (list (pixel 255 0 0)
                   (pixel 255 255 0)
                   (pixel 0 0 255)
                   (pixel 200 0 100)
                   )))
(define grayscaled (grayscale-icon tiny))
(expect (size grayscaled) 2)
(expect (red (car (pixels grayscaled))) 85)
(expect (green (car (pixels grayscaled))) 85)
(expect (blue (car (pixels grayscaled))) 85)
```

i. (2.0 pt) Fill in the blank.

```
(icon (size i) (map grayscale (pixels i)))
```

13. (4.0 points) Comprehending Scheme Lists

The Scheme procedure `comp` returns a Scheme expression that behaves similarly to Python list comprehensions.

For example, consider this call to `comp`:

```
(comp '(+ x 3) 'x '(list 1 2 3))
```

That generates an expression that goes through each item in the Scheme list `(1 2 3)`, assigns each item to the symbol `x`, calls `(+ x 3)` on the item, and puts the resulting value in a new Scheme list.

The generated expression can be evaluated to return the new list:

```
scm> (eval (comp '(+ x 3) 'x '(list 1 2 3)))
(4 5 6)
```

That line of code above achieves the same result as using the built-in `map`:

```
scm> (map (lambda (x) (+ x 3)) '(1 2 3))
(4 5 6)
```

However, `comp` returns a Scheme expression that must be evaluated to get the new list, whereas `map` returns the new list immediately.

Complete the implementation of `comp` below per the description and the doctests:

```
(define (comp item-call-expr item items)
  `(begin (define (comp-helper old-lst)
            (if (null? old-lst)
                nil
                (begin (define ,item __________)
                                    (a)
                       (cons __________ (comp-helper __________)))))
                                 (b)                  (c)
          (comp-helper __________)))
                         (d)

(expect (eval (comp '(+ x 3) 'x '(list 1 2 3))) (4 5 6))
(expect (eval (comp '(* y 2) 'y '(list 1 2 3))) (2 4 6))
```

(a) (1.0 pt) Fill in blank (a).

```
(car old-lst)
```

(b) (1.0 pt) Fill in blank (b).

- ○ `item`
- ○ `,item`
- ○ `items`
- ○ `,items`
- ○ `item-call-expr`
- ● `,item-call-expr`

(c) (1.0 pt) Fill in blank (c).

```
(cdr old-lst)
```

(d) (1.0 pt) Fill in blank (d).

- ○ `item`
- ○ `,item`
- ○ `items`
- ● `,items`
- ○ `item-call-expr`
- ○ `,item-call-expr`

14. (2.0 points) Matchy Matchy

Consider this regular expression from the CS61 codebase:

```
([0-9]+)\.[jJ][pP][eE]?[gG]
```

(a) (2.0 pt) Which of the following strings would be fully matched by that regular expression? Select all that apply.

- [x] `0.jpg`
- [x] `123.jpeg`
- [ ] `a.jpeg`
- [ ] `7\.jpg`
- [x] `001.JPEG`
- [ ] `a1.jpg`
- [x] `0.jPg`

15. (2.0 points) Will it Hash?

A "hashtag" is a "word or phrase with the symbol # in front of it, used on social media websites and apps so that you can search for all messages with the same subject."

The following are all valid hashtags on Twitter: #python, #climate_change, #100DaysOfCode, #happy-earth-day, #8675309

A hash tag cannot contain any whitespace; only letters, numbers, hyphens, or underscores.

Complete the `has_hashtag` function below so that it returns `True` only for strings containing valid hash tags.

```
import re

def has_hashtag(text):
    """
    >>> has_hashtag("#climate_change")
    True
    >>> has_hashtag("#100DaysOfCode")
    True
    >>> has_hashtag("#happy-earth-day")
    True
    >>> has_hashtag("its #party time")
    True
    >>> has_hashtag("# party time")
    False
    >>> has_hashtag("#!&^$")
    False
    """
    return bool(re.search(r"__________", text))
                              (a)
```

(a) (2.0 pt) Fill in blank (a).

```
#[\w-]+
```

16. (2.0 points) Will it Float?

The Python documentation uses a BNF grammar to describe valid floating point numbers.

Here is the grammar (adapted to use the syntax we use in CS61A for BNF grammars):

```
?start: floatnumber
floatnumber: pointfloat | exponentfloat
pointfloat: digitpart? fraction | digitpart "."
exponentfloat: (digitpart | pointfloat) exponent
digitpart: /\d/ (["_"] /\d/)*
fraction: "." digitpart
exponent: ("e" | "E") ("+" | "-")? digitpart
```

(a) (2.0 pt) Which of the following strings can be parsed successfully by that BNF? Select all that apply.

- [x] `55.555`
- [x] `3.4_5_6`
- [x] `5E100`
- [x] `9e1_2`
- [ ] `3_400`
- [x] `7_7.7`
- [ ] `33.333_`
- [ ] `2e10.5`
- [x] `0.0`

17. (3.0 points) The Structure and Interpretation of Scheme

The official Scheme specification includes a BNF grammar for describing the syntax of the language.

Here is a subset of the Scheme grammar (adapted to use the syntax we use in CS61A for BNF grammars):

```
?start: expression
expression: constant | variable | if_expression
if_expression: "(if " expression expression expression? ")"
constant: BOOLEAN | NUMBER
variable: identifier

identifier: initial subsequent* | "+" | "-" | "..."
initial: LETTER | "!" | "$" | "&" | "*" | "/" | ":" | "<" | "=" | ">" | "?"
subsequent: initial | DIGIT | "." | "+" | "-"
LETTER: /[a-zA-z]/
DIGIT: /[0-9]/
BOOLEAN:  "#t" | "#f"

%import common.NUMBER
%ignore /\s+/
```

(a) (1.0 pt) The grammar currently permits an `if` form to have only a "consequent" (what gets evaluated when the predicate is true) and not need an "alternate" (what gets evaluated when the predicate is false). For example, both of these strings can be parsed:

```
(if #t 5 10)
(if #t 5)
```

If you wanted to create a stricter version of Scheme that required `if` forms to have an alternate, how would you modify the rule for the `if_expression` non-terminal?

In such a dialect, these strings should still be parseable:

```
(if #t 5 10)
(if #t a b)
```

But *not* these strings:

```
(if #t 5)
(if #t 10)
```

Fill in the blank with your rewrite of the `if_expression` non-terminal.

```
if_expression: ______________________
```

> `"(if " expression expression expression ")"`

(b) (2.0 pt) The grammar does not yet have support for comparison expressions, such as `(> 5 3)`. How can you modify the grammar so that the following strings can be parsed?

```
(> 5 2)
(< 5 2)
(= #t #t)
(= (> a b) (> c d))
(if (> 5 2) #t #f)
(if (> 5 2) #t (> 3 1))
(if (> a b) (= c d) (= e f))
(> 5 (if (> 5 2) 2 6))
```

To get you started, we have already modified the `expression` non-terminal rule and added a `COMPARATOR` terminal rule; you only need to fill in the blank for the `comparison` non-terminal rule.

```
expression: constant | variable | comparison | if_expression
comparison: ____________________________________
COMPARATOR: ">" | "<" | "="
```

```
"(" COMPARATOR expression expression ")"
```

18. (0.0 points) Just for Fun

Draw something that reflects your 61A experience.

No more questions.