CS 61A Final Review Session

What would Python/Scheme Output?

1. (8 points) Silence of the Lambdas

For each of the expressions in the table below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. If an error occurs, write "`Error`". If an expression yields (or prints) a function, write "`<Function>`". The first two rows have been provided as examples.

Important: The statements in the table are cumulative—assume that all preceding statements in the table have been executed before each entry.

Assume that `python3` has executed the statements on the left initially:

```
foster = 1
def f(foster):
    hopkins = foster+1
    def g(glenn):
        nonlocal foster
        foster = glenn
        hopkins = 2*glenn
    return (g, lambda: [foster, hopkins])
```

Expression	Interactive Output
`pow(2, 3)`	`8`
`print(4, 5) + 1`	`4 5` `Error`
`levine, demme = f(5)` `foster`	
`demme`	
`tally = demme()` `tally[0]`	
`tally[1]`	
`print(levine(9))`	
`foster`	
`tally = demme()` `tally[0]`	
`tally[1]`	

```
blue = Pair(3, Pair(4, nil))
gold = Pair(Pair(6, 7), Pair(8, 9))

def process(s):
    cal = s
    while isinstance(cal, Pair):
        cal.bear = s
        cal = cal.second
    if cal is s:
        return cal
    else:
        return Pair(cal, Pair(s.first, process(s.second)))

def display(f, s):
    if isinstance(s, Pair):
        print(s.first, f(f, s.second))

y = lambda f: lambda x: f(f, x)
```

Expression	Output
`Pair(1, nil)`	`Pair(1, nil)`
`print(Pair(1, nil))`	`(1)`
`1/0`	Error
`print(print(3), 1/0)`	
`print(Pair(2, blue))`	
`print(gold)`	

Expression	Output
`process(blue.second)`	
`print(process(gold))`	
`gold.second.bear.first`	
`y(display)(gold)`	

1. (18 points) What will Python output?

Include all lines that the interpreter would display. If it would display a function, then write Function. If it would cause an error, write Error. Assume that you have started Python 3 and executed the following. **These are entered into Python exactly as written.**

```
class Cat:
    name = "meow"
    def __init__(self, fish):
        self.fish = fish

    def __iter__(self):
        while len(self.fish) > 0:
            yield lambda: self.fish[0].getname()
            self.fish = self.fish[1:]

class SuperCat(Cat):
    def __init__(self):
        self.lives = 9

    def __next__(self):
        if len(self.fish) == 0:
            raise StopIteration
        return self.fish

class Fish:
    def __init__(self, name):
        self.name = name

    def __len__(self):
        fish = ['fish1', 'fish2', 'fish3']
        return len(self.fish) + 2

    def getname(self):
        print(self.name)
```

Expression	Interactive Output
`print('Cats are cool!')`	Cats are cool!

`dory = Fish('Dory')`
`marlene = Fish('Marlene')`
`dory.name`	

`dari = Cat([dory, marlene, Fish('Nemo')])` `print(dari.fish[2].getname())`	
`fishes = iter(dari)` `next(fishes)`	
`pusheen = SuperCat()` `pusheen.fish`	
`Cat.__init__(pusheen, dari.fish)` `next(pusheen).getname()`	
`for a in pusheen:` `    a()`	
`dari.getname = Fish.getname` `dari.getname(dory)`	
`print(len(pusheen.fish))`	
`print(Fish.__len__(pusheen))`	

Environment Diagrams

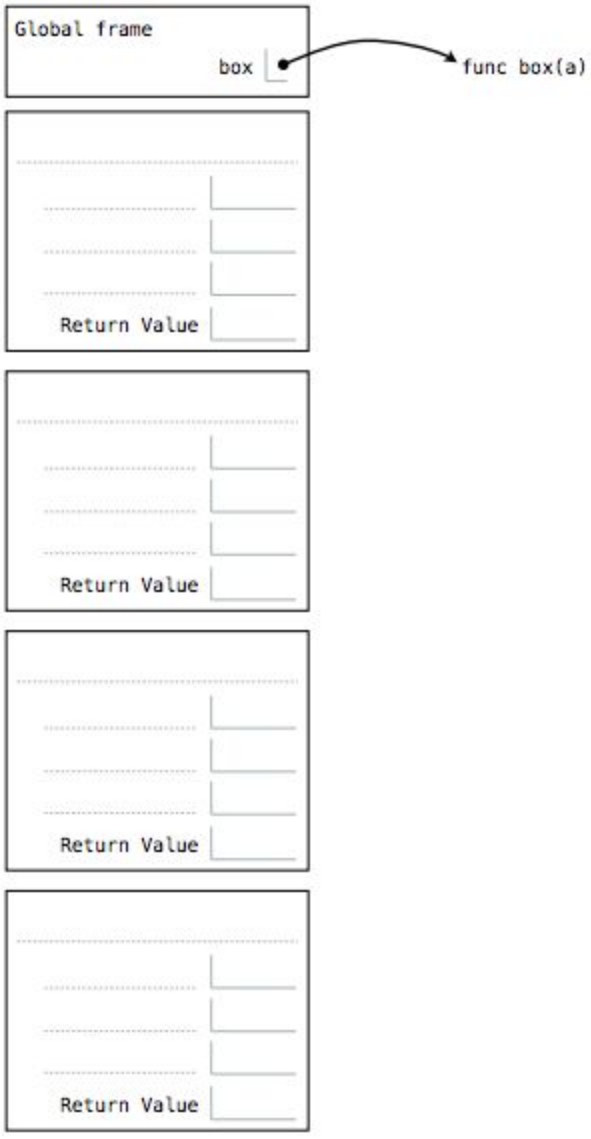
1)

(b) **(8 pt)** Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.*

A complete answer will:

- Add all missing names, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.

```
def box(a):
    def box(b):
        def box(c):
            nonlocal a
            a = a + c
            return (a, b)
        return box
    gift = box(1)
    return (gift(2), gift(3))
box(4)
```



2. (12 points) Environment Diagrams

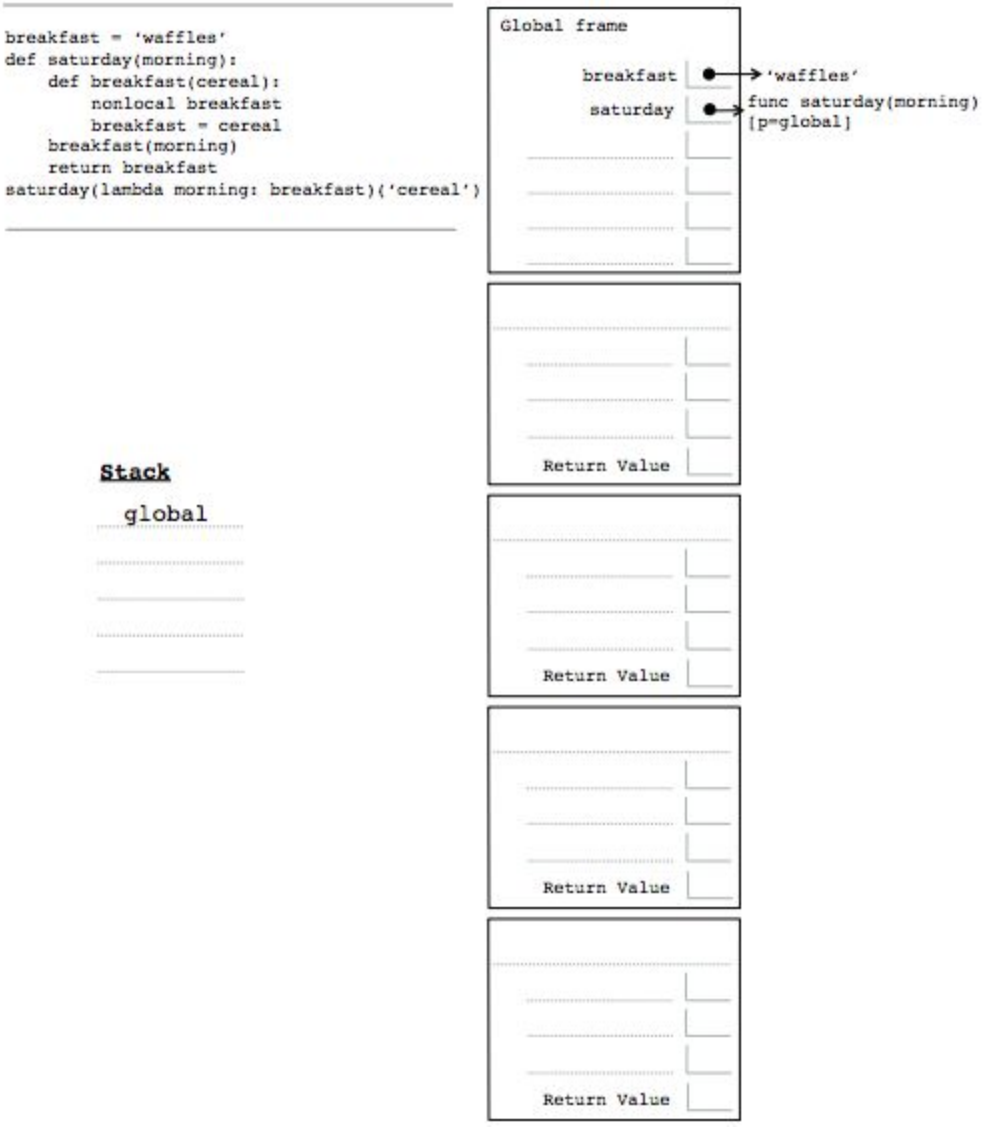
(a) (6 pt) Saturday Morning

Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.* You may want to keep track of the stack on the left, but this is not required.

A complete answer will:

- Add all missing names, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.
- The first function created by `lambda` should be labeled λ_1, the next one should be λ_2, and so on.

```
breakfast = 'waffles'
def saturday(morning):
    def breakfast(cereal):
        nonlocal breakfast
        breakfast = cereal
    breakfast(morning)
    return breakfast
saturday(lambda morning: breakfast)('cereal')
```



Functions / Higher-Order Functions

1)

(c) (4 pt) Fill in the blanks in the implementation of `pathfinder`, a higher-order function that takes an increasing function `f` and a positive integer `y`. It returns a function that takes a positive integer `x` and returns whether it is possible to reach `y` by applying `f` to `x` zero or more times. For example, 8 can be reached from 2 by applying `double` twice. A function f is *increasing* if $f(x) > x$ for all positive integers x.

```
def pathfinder(f, y):
    """Return a function find_from that takes x and returns whether
    repeatedly applying the increasing function f to x can reach y.

    >>> f = pathfinder(double, 8)
    >>> {k: f(k) for k in (1, 2, 3, 4, 5)}
    {1: True, 2: True, 3: False, 4: True, 5: False}
    >>> g = pathfinder(inc, 3)
    >>> {k: g(k) for k in (1, 2, 3, 4, 5)}
    {1: True, 2: True, 3: True, 4: False, 5: False}
    """

    def find_from(x):

        while ____________________________________________________________:

            ____________________________________________________________

        return ____________________________________________________________

    ____________________________________________________________________________
```

3. (6 points) A Higher Order of Protection

Louis Reasoner is making a web application, and he wants to secure it. (Good for him!) One of the ways he wants to secure it is through checking to make sure that the user is an admin when it tries to visit certain confidential pages. So, being a silly programmer, he does the following.

```
def delete_everything(is_admin, request):
    if not is_admin:
        print('ERROR: not admin')
        return
    confirmation = do_bad_stuff(request) # BAD STUFF HAPPENS HERE
    return confirmation

def steal_credit_card_info(is_admin, request):
    if not is_admin:
        print('ERROR: not admin')
        return
    cc_info = hack_a_shaq(request) # DO SOME 1337 HAXORING
    return cc_info
```

However, Alyssa P. Hacker comes across this code, and realizes that there is a better way to do this using higher-order functions! She modifies the above as follows.

```
def delete_everything(request):
    confirmation = do_bad_stuff(request) # BAD STUFF HAPPENS HERE
    return confirmation
delete_everything = protect_me(delete_everything)

def steal_credit_card_info(request):
    cc_info = hack_a_shaq(request) # DO SOME 1337 H4XORING
    return cc_info
steal_credit_card_info = protect_me(steal_credit_card_info)
```

Help her to complete the code by filling in the function below. The new code should provide the same functionality as the original code. For example, calling `delete_everything(True, my_request)` should have the same effect in both versions of the code.

You may leave lines blank if you do not need them.

Recursion

1)

(b) (6 pt) The D33P language includes three types of tokens: open parentheses, close parentheses, and integers. An expression is well-formed if it contains balanced parentheses, and each integer correctly indicates its depth: the number of nested sets of parentheses that surround that integer.

Implement `correct_depth`, which takes a list of tokens as input and returns True if and only if a prefix of the input is a well-formed D33P expression. Assume that the input contains a balanced set of nested parentheses with single-digit positive integers surrounded by parentheses. You only need to check that the integers indicate the correct depths.

Do not change any of the code that is provided. You may not use any `def` *statements or* `lambda` *expressions.*

```
def correct_depth(s, depth=0):
    """Return whether a prefix of list s is a well-formed D33P expression.

    >>> list('(1)')
    ['(', '1', ')']
    >>> correct_depth(list('(1)'))
    True
    >>> correct_depth(list('(2)'))
    False
    >>> correct_depth(list('((2)((3)))'))
    True
    >>> correct_depth(list('((2)(3))'))
    False
    >>> correct_depth(list('((3)(2))'))
    False
    >>> correct_depth(list('(((3)((4))(3))(2)((3)))'))
    True
    """
    first = s.pop(0)
    if first != '(':
        return depth==int(first)
```

2)

Question 1

In game theory, a *subtraction game* is a simple game with two players, player 0 and player 1. At the beginning, there is a pile of *n* cookies. The players alternate turns; each turn, a player can take anywhere from 1 to 3 cookies. The player who takes the last cookie wins. Fill in the function `can_win`, which returns `True` if it is possible to win starting at the given number of cookies. It uses the following ideas:

- if the number of cookies is negative, it is impossible to win.
- otherwise, the current player can choose to take either 1, 2, or 3 cookies.
- evaluate each action: if that action forces the opponent to lose, then return True (since we can win)
- if none of the actions can force a win, then we can't guarantee a win.

```
def can_win(number):
    """Returns True if the current player is guaranteed a win
    starting from the given state. It is impossible to win a game
    from an invalid game state.

    >>> can_win (-1) # invalid game state
    False
    >>> can_win (3) # take all three !
    True
    >>> can_win (4)
    False
    """
    "*** YOUR CODE HERE ***"
```

Orders of Growth

1)

(e) (2 pt) Define a simple mathematical function $f(n)$ such that calling `m(n)` on **positive integer** `n` prints $\Theta(f(n))$ **lines of output.**

```
def m(n):
    g(n)
    if n <= 2:
        print('The')
    else:
        m(n//3)

def g(n):
    if n == 42:
        print('Last')
    if n <= 0:
        print('Question')
    else:
        g(n-1)
```

$f(n) =$

2)

Question 3

Find the time complexity of `funny` in big-Theta (θ) notation.

```
def joke(n):
    for i in range(n**2):
        print(i)

def funny(n):
    for i in range(n**2):
        print(joke(100))
    return 'haha'
```

Toggle Solution

Linked Lists

1. Implement `combine_linked_val` that takes in two linked lists, an index, and a combiner function, and returns the combined value of the two values in the linked list at that index. If one of the lists is shorter than the given index, return Link.empty.

```
def combine_linked_val(link1, link2, index, combiner):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> b = Link(5, Link(6, Link(7, Link(8))))
    >>> combine_linked_val(a, b, 2, mul)
    21
    """
```

1. Write a function `link_range` that creates a linked list with all integers between `low` (inclusive) and `high` (exclusive), similar to Python's `range`.

```
def link_range(low, high):
    """
    >>> link_range(1, 5)
    Link(1, Link(2, Link(3, Link(4))))
    """
```

2. Write a function `stretch` that takes a linked list and creates a new linked list with each element duplicated `factor` times.

```
def stretch(link, factor):
    """
    >>> stretch(link_range(1, 4), 2)
    Link(1, Link(1, Link(2, Link(2, Link(3, Link(3))))))
    """
```

3. Write a function `extend` that takes two linked lists and appends the second list to the first. You should not create any new links (you cannot call the `Link` constructor). You may assume that the first list is nonempty.

```
def extend(left, right):
    """
    >>> left = link_range(1, 4)
    >>> right = link_range(4, 7)
    >>> extend(left, right)
    >>> left
    Link(1, Link(2, Link(3, Link(4, Link(5, Link(6))))))
    """
```

(a) **(4 pt)** Implement `group`, which takes a one-argument function `f` and a list `s`. It returns a list of groups. Each group is a list that contains all the elements `x` in `s` that return equal values for `f(x)`. The elements in a group appear in the same order that they appeared in `s`. The groups are ordered by the order in which their first elements appeared in `s`.

```
def group(f, s):
    """Return a list of groups that contain all x with equal f(x).

    >>> five = [3, 4, 5, 2, 1]
    >>> group(lambda x: x % 2, five)
    [[3, 5, 1], [4, 2]]
    >>> group(lambda x: x % 3, five)
    [[3], [4, 1], [5, 2]]
    """
    a = []
    for b in map(f, s):

        if ____________________________________________________________:

            a.append(b)

    return [[_____________________________________________] for b in a]
```

(b) **(4 pt)** Implement `group_link`, which takes a one-argument function `f` and a `Link` instance `s`. It returns a linked list of groups. Each group is a `Link` instance containing all the elements `x` in `s` that return equal values for `f(x)`. The order of groups and elements is the same as for `group`. The `Link` class appears on your midterm 2 study guide. The `filter_link` function appears in the appendix on the last page of this exam.

```
def group_link(f, s):
    """Return a linked list of groups that contain all x with equal f(x).

    >>> five = Link(3, Link(4, Link(5, Link(2, Link(1)))))
    >>> group_link(lambda x: x % 2, five)
    Link(Link(3, Link(5, Link(1))), Link(Link(4, Link(2))))
    >>> group_link(lambda x: x % 3, five)
    Link(Link(3), Link(Link(4, Link(1)), Link(Link(5, Link(2)))))
    """
    if s is Link.empty:
        return s
    else:
        a = filter_link(lambda x: ________________________, ___________________)

        b = filter_link(lambda x: ________________________, ___________________)

        return Link(a, group_link(f, b))
```

Definition. The *multi-grouping* by function `f` of list `s` is formed by the following iterative process with `k` starting at 0 and increasing by 1 each iteration.

- Group together all elements that yield equal values when applying `f` repeatedly, `k` times.
- If all elements are in a single group, the process is complete. Otherwise, place each new group in a (possibly nested) list and repeat.

For example, if `f` is `lambda x: max(x-3, 0)` and `s` is `[2, 4, 3, 4, 2]`, then

- In the `k=0` iteration, the 2's are grouped, the 4's are grouped, and the 3 is alone: `[[2, 2], [3], [4, 4]]`
- In the `k=1` iteration, `f(2)==f(3)`, and so the 2's group and 3's group are grouped: `[[[2, 2], [3]], [[4, 4]]]`
- In the `k=2` iteration, `f(f(2))==f(f(3))==f(f(4))`. All elements are in a single group, so we're done.

(c) (4 pt) Implement `multigroup`, which returns the multi-grouping by `f` of a list `s`. Assume that the process terminates and that `group` (part a) is implemented correctly.

```
def multigroup(f, s):
    """Return a multi-grouping by f of the elements in s.

    >>> multigroup(lambda x: max(x-3, 0), [2, 4, 3, 4, 2])
    [[[2, 2], [3]], [[4, 4]]]
    >>> multigroup(abs, [5])
    5
    >>> multigroup(abs, [5, 5])
    [5, 5]
    >>> multigroup(abs, [5, 5, -5])
    [[5, 5], [-5]]
    >>> multigroup(lambda x: x // 10, [123, 145, 126, 149])
    [[[123], [126]], [[145], [149]]]
    >>> multigroup(lambda x: x[1:], ['tin', 'man', 'can'])
    [[['tin']], [['man'], ['can']]]
    >>> multigroup(lambda x: max(x-1, 0), [2, 4, 3, 4, 2])
    [[[[2, 2]]], [[[3]]]], [[[[4, 4]]]]]
    """
    def using(g, s):

        if len(s) == 1:

            return ____________________________________________________________

        else:

            grouped = group(g, s)

            return using(lambda x: _____________________________________, grouped)

    return using(lambda x: ________________________________________________, s)
```

(d) (2 pt) How many square brackets are in the return value of `multigroup(hail, [3, 20, 128])`? Assume that `multigroup` is implemented correctly.

```
def hail(x):
    if x == 1:
        return 1
    elif x % 2 == 0:
        return x // 2
    else:
        return 3 * x + 1
```

Trees

0. Write a function `map_mutate` that takes in a Tree `t` and a function `f` and applies `f` to every entry in `t`. This `map_mutate` should mutate its argument and not return a copy. For example, if `t` is tree on the left, then after `map_mutate(lambda x: 10 - x, t)`, `t` should be the tree on the right:

```
def map_mutate(fn, t):
```

1. Write a function `reflect` that takes in a Tree and reflects it over its center. This function should mutate its argument and not return a copy. For example, if `t` refers to the following tree

```
def reflect(t):
```

2. Write a function `make_factor_tree` that takes in a natural number `num` and returns a tree with `num` at the root and all of `num`'s factors (excluding 1 and `num`) as branches. Each branch should also contain all of it's entry's factors. For example, `make_factor_tree(12)` is

```
def make_factor_tree(num):
```

Write a function `permutations` that takes in a list and returns a list of permutations possible (of the same length as the original list.)

```
def permutations(lst):
```

Object Oriented

1. (12 points) Class Hierarchy

For each row below, write the output displayed by the interactive Python interpreter when the expression is evaluated. Expressions are evaluated in order, and **expressions may affect later expressions.**

Whenever the interpreter would report an error, write ERROR. You *should* include any lines displayed before an error. *Reminder*: The interactive interpreter displays the `repr` string of the value of a successfully evaluated expression, unless it is `None`. Assume that you have started Python 3 and executed the following:

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting
class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'My job is to gather wealth'
class Proletariat(Worker):
    greeting = 'Comrade'
    def work(self, other):
        other.greeting = self.greeting + ' ' + other.greeting
        other.work() # for revolution
        return other
jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

Expression	Interactive Output
`5*5`	25
`1/0`	ERROR
`Worker().work()`	
`jack`	
`jack.work()`	

Expression	Interactive Output
`john.work()[10:]`	
`Proletariat().work(john)`	
`john.elf.work(john)`	

3. (30 points) Twin Breaker

(a) (8 pt) Run-length encoding (RLE) is a technique used to compress sequences that contain repeated elements. For example, the sequence $1, 1, 1, 4, 2, 2, 2, 2$ would be encoded as three 1's, one 4, and four 2's. Fill in the blanks in the `RLE` class below, so that all doctests pass.

```
class RLE(object):
    """A run-length encoding of a sequence.

    >>> RLE([2, 2, 2, 2, 2, 7]).runs
    [(5, 2), (1, 7)]
    >>> s = RLE([1, 1, 1, 4, 2, 2, 2, 2])
    >>> s.runs
    [(3, 1), (1, 4), (4, 2)]
    >>> len(s)
    8
    >>> s[2], s[3], s[4], s[5]
    (1, 4, 2, 2)
    """
    def __init__(self, elements):

        last, count = None, 0

        self.runs = []

        for elem in elements:

            if ____________________________________________________:

                self.runs.append(_____________________________________)

            if ____________________________________________________:

                last, count = _____________________________________


        else:

            count += 1

    ____________________________________________________________________

def __len__(self):

    return sum(_____________________________________________________)

def __getitem__(self, k):

    run = 0

    while ____________________________________________________________:

        k, run = ____________________________________________________

    return self.runs[run][1]
```

5. (8 points) We are Objectively Lazy

Suppose we wish to define a new lazily evaluated list type called `LazyList`. A `LazyList` does not hold elements directly; instead, it holds 0-argument functions to compute each element. The first time an element is accessed, the `LazyList` calls the stored function to compute that element. Subsequent accesses to the same element do not call the stored function. See the docstring for `LazyList` for examples of how to use it.

(a) Fill in the class definition of `LazyList` below to match its docstring description, so that all doctests pass.

```
class LazyList(object):
    """A lazy list that stores functions to compute an element. Calls
    the appropriate function on first access to an element; never
    calls an element's function more than once.

    >>> def compute_number(num):
    ...     print('computing', num)
    ...     return num
    ...
    >>> s = LazyList()
    >>> s.append(lambda: compute_number(1))
    >>> s.append(lambda: compute_number(2))
    >>> s.append(lambda: compute_number(3))
    >>> s[1]
    computing 2
    2
    >>> s[1]
    2
    >>> s[0]
    computing 1
    1
    >>> for item in s: print(item)
    1
    2
    computing 3
    3
    """
    def __init__(self):
        self._list = []
        self._computed_indices = set()


    def append(self, item):

        ----------------------------------------------------------------


    def __getitem__(self, index):

        if ----------------------------------------------------------:

            self._computed_indices.add(index)

            ----------------------------------------------------------

        return self._list[index]
```

Scheme

Implement a function `count-serpinski`, which takes a numbero `depth` as an argument. The function calculates how many triangles are contained in a Serpinski's triangle with the given depth. See the tests for examples.

How can you use recursive calls on *n-1* to build your answer for *n*?

Write **waldo-tail**, a Scheme procedure that takes in a Scheme list and outputs the index of the symbol waldo if the symbol waldo exists in the list. Otherwise, it outputs the symbol **nowhere** if waldo is not in the list. Write the procedure using TAIL RECURSION.

```
; Question 5
; Combines the elements of two streams. Assume same length or infinite.
(define (add-streams s1 s2)









)




; Question 6
; An infinite stream containing the fibonnaci sequence, computed
recursively. Use add-streams (from above) in your solution!

(define (fib-stream)









)
```