Topic Based OH - Final Review - CS61A - Sp. 2013

Higher-order functions

1. Define a higher-order function piecewise that returns a piecewise-defined function. piecewise takes as argument `cutoff`, `lower`, and `upper` and returns a function that is equivalent to `lower` when its input is below `cutoff`, and equivalent to `upper` when its input is at or above `cutoff`.

```
def piecewise(cutoff, lower, upper):
      """YOUR CODE HERE"""
```

2. Define a higher-order function integrate that, given a function f, a lower bound a, and an upper bound b, returns an approximation of \int_a^b f dx. Hint: you can approximate integration as many small sums, ie f(a) + f(a + dx) + f(a + 2dx) + … + f(b)

```
def integrate(f, a, b):
    “***YOUR CODE HERE***”
    return 0
```

Now, use your integrate function to approximate pi. We will use this relation: \int_{-1}^1 sqrt{1-x^2} dx = \pi / 2.

Codes

pi = 0 # Codes

Lambda expressions

1. Fill in the blanks so each of the following expressions evaluate to 6:
   a. `(lambda ____: _____)([0, 1, 2, 3])`
   b. `(lambda ____: _____)(3)(2)`
   c. `(lambda ____: _____)(lambda ___: _____)(3)`
2. Draw environment diagrams for the following:
   a.

```
y = 5
def f(g, x):
    y = 4
    return g(x + y)
f(lambda x: x + y, 2)
```

   b.

```
g = lambda f: lambda x: lambda y: f(x, y)
g(lambda x, y: x*y)(3)(4)
```

Iterative improvement

1. Using Newton's method, write a function that calculates the fifth root of a number x:

```
def fifth_root(x):
    """Calculates the fifth root of x.

    >>> fifth_root(32)
    2
    """
```

Environment diagrams

1. Draw an environment diagram for the following code:

```
def julia(julia):
      julia = julia(julia)
      def julia():
            return julia
      return (lambda julia: julia)(julia())
julia(lambda julia:julia)
```

solution

Sequences, nested tuples

1. Write a method that reverses an rlist. (multiple ways to do this. Assume that the Rlist class definition is defined)

```
def reverse_recur(r):
      #reverses an rlist recursively in place.
      if r.rest != Rlist.empty:
            first, second = r, r.rest
            r = reverse(second)
            first.rest, second.rest = Rlist.empty, first
      return r
def reverse_iter(r):
      if r != Rlist.empty:
            rlist = Rlist(r.first)
            while r.rest!= Rlist.empty:
                  rlist = Rlist(r.rest.first, rlist)
                  r = r.rest
            return rlist
      return r
def reverse_rlist(r):
      def helper_rlist(rlist, sofar):
            if(len(rlist)==0):
            return sofar
            else:
            return helper_rlist(rlist.rest,
                              Rlist(rlist.first,sofar))
      return helper_rlist(r, Rlist.empty)
```

Lists and dictionaries

1. Draw the environment diagram for the following:
   x = [0, 1, 2, 3]
   x[1] = [0, 1, 2, 3]
   x[0] = x
   y = x[:]
2. What do each of the following list methods do?
   a. append
   b. extend
   c. pop
   d. remove
3. Define a function `remove_all` that removes all instances of a given element `x` from a list `lst`

```
def remove_all(lst, x):
    """Removes all instances of x from lst.

    >>> lst = [1, 1, 4, 2, 1]
    >>> remove_all(lst, 1)
    >>> lst
    """
```

Non-local assignment

1. Define a function make_every_other such that the doctests would pass.

```
def make_every_other():
    """Doctests:

    >>> e_o = make_every_other()
    >>> e_o(1)
    1
    >>> e_o(2000)
    'NO'
    >>> e_o('hello there!'
    'hello there!'
    >>> e_o(-10)
    'NO'
    """
```

2. Taken from Albert's Nonlocal exam questions: Implement a function sentence_buffer which returns another one-argument function. This function will take in a word at a time, and it saves all the words that it has seen so far. If takes in a word that ends in a period ("."), that denotes the end of a sentence, and the function returns all the words in the sentence. It will also clear its memory, so that it no longer remembers any words.

```
def sentence_buffer():
    """Returns a function that will return entire sentences when it
    receives a string that ends in a period.

    >>> buffer = sentence_buffer()
    >>> buffer("This")
    >>> buffer("is")
    >>> buffer("Spot.")
    'This is Spot.'
    >>> buffer("See")
    >>> buffer("Spot")
    >>> buffer("run.")
    'See Spot run.'
    """
```

Dispatch functions and dictionaries

1. Let's implement dictionaries using dispatch dictionaries! Dictionaries should support the set and get operations, and use lists to store the data. Do not worry about duplicate keys.

```
def dict():
    “””
    >>> d = dict()
    >>> d[‘set’](‘hi’, 5)
    >>> d[‘get’](‘hi’)
    5
    “””
    “””YOUR CODE HERE”””
```

2. Let's implement a multi-child tree data type! Each tree can have multiple children which can be added with the add_child method. Each tree will also support a get_child method which gets the child at a given index, in addition to having an entry attribute, which will be the entry for the given tree node. Use dispatch functions.

```
def tree(entry):
    “””
    >>> t = tree(5)
    >>> x = tree(4)
    >>> x(‘add_child’)(t)
    >>> x(‘entry’)
    4
    >>> x(‘get_child’)(0)(‘entry’)
    5
    “””
    “””YOUR CODE HERE”””
```

Identity, equality, and mutable values

1. Assume the following statements are executed in order. Fill in the blanks with True or False.

```
>>>a = [1, 2, [3], 4, 5]
>>>b = a[1:4]
>>>a = a[1:4]
>>>b is a
_______
>>>b[1] is a[1]
_______
>>>c = a
>>>c is a
_______
>>>c[1][0] = 10
>>>b[1] == a[1]
_______
>>>b[1] is a[1]
_______
>>>a[1] = [11]
>>>c[1] is a[1]
_______
>>>c[1] is b[1]
_______
```

```
>>>lst = [1, 2, 3, 4, 5]
>>>lst[2] = lst
>>>lst[2] == lst
_______
>>>lst[2] is lst
_______
>>>lst[2][2] is lst
_______
```

2. Suppose we have the following definitions of classes A and B:

```
class C:
    x = [1, 2]
    def y(self):
        return 5

class D(C):
   def z(self):
        return 6
```

Now we execute the following statements. Fill in the blanks:

```
>>>c = C()
>>>d = D()
>>>C.x is D.x
_______
>>>c.x is d.x
_______
>>>C.y is D.y
_______
>>>c.y is d.y
_______
```

Classes, instances, and inheritance

1. Suppose we have the following class definitions. What would Python print?

```
class S:
    def __init__(self):
        self.lst = []

    def m(self, t):
        self.f = t.x
    def n(self):
        self.f()
    def p(self):
        self.f(self)

class T:
    def __init__(self):
        self.lst = []

    def x(self):
        self.list.append(1)
```

```
>>>s = S()
>>>t = T()
>>>s.m(t)
>>>s.n()
>>>s.lst, t.lst
```

_____

```
>>>s.m(type(t))
>>>s.p()
>>>s.lst, t.lst
```

_____

2. Given below is an incomplete definition of the `Parrot` and `SuperParrot` class. `Parrot` should implement a repeat method, which causes the `Parrot` to `speak` the last `phrase` it said (if `repeat` is called without calling `speak` first, the `Parrot` should `speak` the `default` phrase, `'Squawk!'`).

```
class Parrot:
    def __init__(self, name):
        self.phrase = 'Squawk!'
        self.name = name
    def speak(self, phrase):
        print self.name + 'says: ' + phrase
        """YOUR CODE HERE"""
    def repeat(self):
        """YOUR CODE HERE"""
```

`SuperParrot` inherits from `Parrot` and implements one additional method - `repeat_all`, which should cause the `SuperParrot` to `speak` every *unique* `phrase` it has ever said. Use inheritance whenever possible!

```
class SuperParrot:
    """YOUR CODE HERE"""
```

Dot expressions and bound methods

```
class Owner(object):
    all = []

    def __init__(self, name):
        self.name = name
        self.pets = []
        Owner.all.append(self)

    def add_pet(self, pet):
        self.pets.append(pet)

    def __repr__(self):
        return 'Owner(' + self.name + ')'

class Pet(object):
    all = []

    def __init__(self, name, weight, height):
        self.name = name
        self.weight = weight
        self.height = height
        Pet.all.append(self)

    @property
    def bmi(self):
        return self.weight / (self.height * self.height)

    def __repr__(self):
        return 'Pet(' + self.name + ')'
```

QUESTIONS

```
>>> bob = Owner('bob')
>>> joe = Owner('joe')
>>> bob.all                    #1

>>> bob.all.append(bob)
>>> joe.all                    #2

>>> type(joe.add_pet)          #3

>>> type(Owner.add_pet)        #4

>>> harry = Pet('harry', 50, 50)
>>> type(harry.bmi)            #5
```

```
>>> joe.pets.append(harry)
>>> bob.add_pet(harry)
>>> bob.pets[0].all            #6

>>> bob.pets.append(Pet('jimmy', 40, 10))
>>> bob.pets[1].owner          #7

>>> Pet.all                    #8

>>> jimmy                      #9
```

ANSWERS

1) [Owner(bob), Owner(joe)]

2) [Owner(bob), Owner(joe), Owner(bob)]

3) <class 'method'>

4) <class 'function'>

5) <class 'float'>

6) [Pet(harry)]

7) Traceback (most recent call last):

   File "<stdin>", line 1, in <module>

   AttributeError: 'Pet' object has no attribute 'owner'

8) [Pet(harry), Pet(jimmy)]

9) Traceback (most recent call last):

   File "<stdin>", line 1, in <module>

   NameError: name 'jimmy' is not defined

Type dispatching and coercion

```
from types import GeneratorType

def make_natural_numbers_generator():
    i = 0
    while True:
        yield i
        i += 1

def make_natural_numbers_stream(first=0):
    def compute_rest():
        return make_natural_numbers_stream(first + 1)
    return Stream(first, compute_rest)

class NaturalNumbersIterator(object):
    def __init__(self):
        self.current = 0

    def __next__(self):
        next = self.current
        self.current += 1
        return next

    def __iter__(self):
        return self


def type_tag(x):
    return type_tag.tags[type(x)]

type_tag.tags = {GeneratorType: 'generator', NaturalNumbers: 'iterator', Stream: 'stream'}

def add(seq1, seq2):
    types = (type_tag(seq1), type_tag(seq2))
    return add.impl[types](seq1, seq2)

def add_generators(gen1, gen2):
    while True:
        yield next(gen1) + next(gen2)

def add_generator_and_iterator(gen, iterator):
    while True:
        yield next(gen) + next(iterator)

def add_generator_and_stream(gen, stream):
    while True:
```

```
        yield next(gen) + stream.first
        stream = stream.rest

def add_iterator_and_stream(iterator, stream):
    while True:
        yield next(iterator) + stream.first
        stream = stream.rest

def add_iterators(i1, i2):
    while True:
        yield next(i1) + next(i2)

def add_streams(s1, s2):
    while True:
        yield s1.first + s2.first
        s1, s2 = s1.rest, s2.rest

add.impl = {('generator', 'generator') : add_generators,
           ('generator', 'iterator') : add_generator_and_iterator,
           ('generator', 'stream') : add_generator_and_stream,
           ('iterator', 'stream') : add_iterator_and_stream,
           ('iterator', 'iterator') : add_iterators,
           ('stream', 'stream') : add_streams,
           ('stream', 'generator') : lambda s, g: add_generator_and_stream(g, s),
           ('iterator', 'generator') : lambda i, g: add_generator_and_iterator(g, i),
           ('stream', 'iterator') : lambda s, i: add_iterator_and_stream(i, s)
}
```

Recursion

```
def binary_search(item, lst):
    """
    >>> l = [1, 4, 5, 8, 10, 12]
    >>> binary_search(4, l)
    True
    >>> binary_search(9, l)
    False
    """
    def helper(item, lst, low, high):
      if low >= high:
          return False
      mid = (low + high) // 2
      if item == lst[mid]:
          return True
      if item < lst[mid]:
          return binary_search(item, lst, low, mid - 1)
      if item > lst[mid]:
          return binary_search(item, lst, mid + 1, high)
    return helper(item, lst, 0, len(lst))

def find_secret(secret, sentence):
    """
    If the individual letters of the secret are in order within the sentence,
    then the function reutrns the list of indices of those letters. If not, it returns False.

    >>> secret = "mark"
    >>> sentence = "hi my name is ridiculously karl."
    >>> find_secret(secret, sentence)
    [3, 7, 14, 27]
    >>> find_secret("hello", sentence)
    False
    """
    def helper(secret, sentence, indices, index):
        if len(secret) == 0:
            return indices
        if len(sentence) == 0:
            return False
        if secret[0] == sentence[0]:
            return helper(secret[1:], sentence[1:], indices + [index], index + 1)
        return helper(secret, sentence[1:], indices, index + 1)
    return helper(secret, sentence, [], 0)
```

```
def mutable_reverse(lst):
    """
    >>> l = [1, 4, 5, 1, 4]
    >>> mutable_reverse(l)
    >>> l
    [4, 1, 5, 4, 1]
    >>> l = [1, 4, 5, 1, 4, 5]
    >>> mutable_reverse(l)
    >>> l
    [5, 4, 1, 5, 4, 1]
    """
    def helper(index, lst):
        if len(lst) % 2 == 0 and index == len(lst) // 2 or len(lst) % 2 == 1 and index == len(lst)
// 2 + 1:
            return
        lst[index], lst[len(lst) - 1 - index] = lst[len(lst) - 1 - index], lst[index]
        helper(index + 1, lst)
    helper(0, lst)

def mutable_reverse(lst):
    """
    >>> l = [1, 4, 5, 1, 4]
    >>> mutable_reverse(l)
    >>> l
    [4, 1, 5, 4, 1]
    >>> l = [1, 4, 5, 1, 4, 5]
    >>> mutable_reverse(l)
    >>> l
    [5, 4, 1, 5, 4, 1]
    """
    if len(lst) > 0:
        item = lst.pop()
        mutable_reverse(lst)
        lst.insert(0, item)
```

Recursive Data Structures

1. Given the immutable, tuple-representation of RLists, construct the following rlists.

```
empty_rlist = None
def rlist(first, rest):
    return (first, rest)
def first(rlist):
    return rlist[0]
def rest(rlist):
    return rlist[1]
```

a. <1>

```
rlist(1, empty_rlist)
```

b. <1, 2, 3>

```
rlist(1, rlist(2, rlist(3, empty_rlist)))
```

c. <1, <2, 3>, 4> # <2, 3> is a nested rlist

```
rlist(1, rlist(rlist(2, rlist(3, empty_rlist)), rlist(4,
empty_rlist)))
```

d. <1, (2, 3), 4> # (2, 3) is a tuple

```
rlist(1, rlist( (2, 3) , rlist(4, empty_rlist)))
```

2. Write a function that, given a rlist, returns a sorted rlist. You may want to use insertion sort.

```
def sort_rlist(r):
    """ Sort an rlist in ascending order.

    This is done by taking one element at a time out of r and
    placing it into a sorted rlist that you build up from an empty
    rlist.

    It starts like this:
        r = stuff
        sorted = empty

    Take first(r) and place it in sorted, in sorted order:
        r = rest(stuff)
        sorted = rlist(first(stuff), empty_rlist)

    Take first(r) and place it in sorted, in sorted order:
        r = rest(rest(stuff))
        sorted = place the next element in the correct position in
                 the previous sorted

    Repeat until r is empty

    >>> r = rlist(3, rlist(5, rlist(1, rlist(9, rlist(7,
                                            empty_rlist)))))
    >>> sort_rlist(r)
    (1, (3, (5, (7, (9, None)))))
    """
    if r == empty_rlist:
        return r
    return insert(first(r), sort_rlist(rest(r)))
```

```
def insert(item, r):
    """ Inserts item into r in sorted order.  Assume that r is
        sorted.
    >>> r = rlist(4, rlist(6, rlist(8, None)))
    >>> insert(5, r)
    (4, (5, (6, (8, None))))
    """
    if r == empty_rlist:
        return rlist(item, empty_rlist)
    if item <= first(r):
        return rlist(item, r)
    return rlist(first(r), insert(item, rest(r)))
```

3. Given the mutable, object-oriented representation of RLists (shown on the Midterm 2 Study Guide), write the following Rlist constructors.
   a. <1>

      ```
      Rlist(1)
      ```

   b. <1, 2, 3>

      ```
      Rlist(1, Rlist(2, Rlist(3)))
      ```

   c. <1, <2, 3>, 4> # <2, 3> is a nested rlist

      ```
      Rlist(1, Rlist(Rlist(2, Rlist(3)), Rlist(4)))
      ```

   d. <1, (2, 3), 4> # (2, 3) is a tuple

      ```
      Rlist(1, Rlist( (2, 3) , Rlist(4)))
      ```

4. Immutable vs. Mutable Rlists. For comparison, implement the following things, one for tuple Rlists, and another as a mutating function for object oriented Rlists.
   a. Change the first item of the rlist to something else

      ```
      rlst = rlist(4, empty_rlist)
      rlst = rlist(2, rest(rlst))

      rlst2 = Rlist(4)
      rlst2.first = 5
      ```

   b. Add one item to the front of the rlist

      ```
      rlst = rlist(4, empty_rlist)
      rlst = rlist(3, rlst)

      rlst2 = Rlist(4)
      rlst2.rest = Rlist(rlst2.first, rlst2.rest)
      rlst2.first = item
      ```

5. Given an Rlist, write a function that eliminates successive repeated elements.

```
def squish(r):
    """ Given an Rlist, use mutation to eliminate repeat elements

    Hint: Look at 'Yummy in my Summy" from Summer 2012 Midterm 2.

    >>> r = sequence_to_rlist((1, 1, 2, 3, 3, 3, 4, 4))
    ...          # this function should look familiar.
    ...
    >>> squish(r)
    >>> r
    RList(1, RList(2, RList(3, RList(4))))
    """
    prev = r
    curr = r.rest
    while curr is not Rlist.empty:
        if curr.first == prev.first:
            prev.rest = curr.rest
        else:
            prev = curr
        curr = curr.rest
```

Trees and Tree-Structured Sets

1. Looking at the given code for a Tree class on the Midterm 2 Study Guide, write the constructor you would use to construct each of the given trees.
   a. (1)

   ```
   Tree(1)
   ```

   b.

   ```
      ( 2 )
     /     \
   ( 1 )  ( 3 )
   Tree(2, Tree(1), Tree(3))
   ```

   c.

   ```
      ( 2 )
       /
    ( 1 )
   Tree(2, Tree(1))
            # Also acceptable: Tree(2, Tree(1), None) ---> Why?
   ```

d.
```
    ( 1 )
      \
      ( 3 )
```
```
Tree(1, None, Tree(3))
```

2. Review the notes on sets on the Final Exam Study Guide. (Warmup: write a function that implements membership testing in tree sets, aka binary search trees.) Write a function, `build_binary`, that will build a binary search tree out of a list of elements, eliminating repeats. This is also the same as converting a list into a tree-set.

```
def build_binary(ls):
    """
    Write a function that will build a binary _search_ tree
    out of a list of elements. (Which is also a tree set)

    Hint: you may want to write a helper function for adjoining
    an element to a tree.
    >>> build_binary([2, 1, 3])
    Tree(2, Tree(1, None, None), Tree(3, None, None))
    """
    tree = None
    for elem in ls:
        tree = adjoin(tree, elem)
    return tree


# Helper for Exercise 1
def adjoin(tree, elem):
    """
    Returns a new tree with the given element adjoined to the
    given tree.  Does not mutate the original tree.
    """
    if tree == None:
        return Tree(elem)
    if tree.entry == elem:
        return tree
    if tree.entry > elem:
        return Tree(tree.entry,
```

```
                    adjoin(tree.left, elem), tree.right)
    return Tree(tree.entry, tree.left, adjoin(tree.right, elem))
```

3. Look at Fall 2011 Midterm 2, for “pruned,” and solve that one first.
    a. Write a function that will return whether or not a given tree is a subtree of another tree.

```
def isSubtree(s, t):
    """
    Return whether or not Tree s is a subtree of Tree t.
    The entries must match, and the subtree can be anywhere
    in the tree (think of it this way: you must be able to
    take some branch of t, and are allowed to trim it, that
    is identical to s).
    Assume no repeated elements in either of the trees.

    >>> tree = build_binary([2,4,7,1,3,5])
    >>> subtree = build_binary([7, 5])
    >>> isSubtree(subtree, tree)
    True
    >>> isSubtree(tree, subtree)
    False
    >>> t1 = build_binary([5, 2, 1, 4, 3, 8, 7, 12, 9, 15])
    >>> t2 = build_binary([2, 8])
    >>> isSubtree(t2, t1)
    False
    >>> t2 = build_binary([100, 101])
    >>> isSubtree(t2, t1)
    False
    >>> isSubtree(t1, t2)
    False
    """
```

```
    if s == None:
        return True
    if t == None:
        return False
    if s.entry == t.entry:
        return isSubtree(s.left, t.left) and
               isSubtree(s.right, t.right)
    if s.entry < t.entry:
               # subtree must be on the left side
        return isSubtree(s, t.left)
    return isSubtree(s, t.right)
```

b. Like part a, write a function that will return whether or not a given tree is a subtree, but the subtree must be identical to one of the children in the first tree.

```
def is_subtree(s, t):
    """
    Return whether or not Tree s is a subtree of Tree t.  The
    entries must match, and the subtree can be anywhere in the
    tree (think of it this way: you must be able to take some
    branch of t, without trimming it, that is identical to s).
    Assume no repeated elements in either of the trees.

    >>> tree = build_binary([2,4,7,1,3,5])
    >>> subtree = build_binary([7, 5])
    >>> is_subtree(subtree, tree)
    True
    >>> is_subtree(tree, subtree)
    False
    >>> t1 = build_binary([5, 2, 6])
    >>> t2 = build_binary([5, 2])
    >>> is_subtree(t2, t1)
    False
    >>> t2 = build_binary([5, 2, 3])
    >>> is_subtree(t2, t1)
    False
```

```
    >>> is_subtree(t1, t2)
    False
    """
    if s == None:
        return True
    if t == None:
        return False
    if s.entry == t.entry:
        return tree_equal(s.left, t.left) and
               tree_equal(s.right, t.right)
    if s.entry < t.entry:
        return is_subtree(s, t.left)
    return is_subtree(s, t.right)
```

```
# Exercise 2 helper
def tree_equal(t, s):
    if s == None and t == None:
        return True
    if s == None or t == None:
        return False
    if t.entry == s.entry:
        return tree_equal(t.left, s.left) and
               tree_equal(t.right, s.right)
    return False
```

Orders of Growth

1. Give a Θ approximation for each of the following expressions.
    a. $3x^2 + 2x - 100$
    b. $3^x + 3x^5 + x$
    c. 38
    d. $log(x) + log(x^2) + log(x^3)$
    e. $log(x^{2x})$

2. State whether the following expressions are true.
    a. $n+1 \in \Theta(n+2)$
    b. $n^{0.6} \in \Theta(\sqrt{n})$
    c. $\sqrt{n} \in O(n^{0.6})$
    d. $n \in \Theta(nlog(n))$
    e. $n! \in \Theta(n^n)$
3. Analyze the runtime of the following program:

```
def mystery1(l, x, y, z):
    if y < z:
        return False
    w = (y + z) // 2
    if l[w] > x:
        return mystery1(l, x, y, w-1)
    if l[w] < x:
        return mystery1(l, x, w+1, z)
    return True
```

4. Analyze the runtime of the following program. Hint: `t` is a tree. Write the answer in terms of `n`, the number of nodes in the tree.

```
def mystery2(t, v):
    q = []
    q.append(v)
    while q:
        s = q.pop(0)
        if s == v:
             return True
        for child in s.children:
            q.append(v)
    return False
```

5. Analyze the runtime of the following program. Hint: `t` is a tree. Write the answer in terms

of n, the number of nodes in the tree.

```
def mystery3(t, v):
    if not t:
        return False
    if t.val == v:
        return True
    for child in t.children:
        if mystery3(child, v):
            return True
    return False
```

Scheme

1. Write a scheme function that sums the values of all vertices in a binary tree given the following implementation:

```
(define (tree val left right)
    (list val left right))
(define (val tree)
    (car tree))
(define (left-child tree)
    (cadr tree))
(define (right-child tree)
    (caddr tree))
(define (sum-vertices tree)
    (if (null? tree)
        0
        (+ (val tree)
           (sum-vertices (left-child tree))
           (sum-vertices (right-child tree)))))
```

2. Now write a scheme function that finds the average value of all vertices in a binary tree.

```
(define (count-vertices tree)
    (if (null? tree)
        0
        (+ 1
           (count-vertices (left-child tree))
           (count-vertices (right-child tree)))))
(define (average-value tree)
    (if (null? tree)
        0
        (/ (sum-vertices tree) (count-vertices tree))))
```

Scheme lists and parsing

(see http://www-inst.eecs.berkeley.edu/~cs61a-tk/finalreview.html for answers to this section)

1) What are the following in standard scheme list notation:

```
stk> '(1 . (2 . ((4 . (3 . (2 . ())))) . ((2 . (4 . 2)) . (3 .
()))))))

stk> '(1 2 3 . ((4 . 5) . (5 6 . 7))

>>> P  = Pair #for brevity
>>> P(1, P(2, P(3, P(P(4, P(5, nil)), P(P(6, P(7, nil)),
nil)))))
```

2) Write a scheme function "deep-map" that takes a scheme list which contains elements that may be lists themselves, and maps the function f on each element (and sub-element) of the list.

```
(define (deep-map lst f)
    ;;; YOUR CODE HERE ;;;
)
;;; test code
(print (deep-map '(1 2 (3 4 (5 6) 7 ) 8 9) (lambda (x) (* x
x))))
;;;should print (1 4 (9 16 (25 36) 49) 64 81)
(print 'done)
(exit)
```

Tail calls

(see http://www-inst.eecs.berkeley.edu/~cs61a-tk/finalreview.html for answers to this section)

1) Write the following python function tail-recursively in scheme:

```
def blum(i):
    n = 3
    M = 11*19
    for _ in range(0, i):
        n = n**2 % M
    return n
```

```
(define (blum i)
    ;;;YOUR CODE HERE;;;
)
```

2) Consider the following series that converges to pi:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \frac{4}{13} - \cdots$$

Now write a function that calculates the sum of n terms of this series tail recursively

```
(define (calc-pi n)
   ;;; YOUR CODE HERE ;;;
)
```

3) Which of the following are tail-recursive?

```
(define (f x)
  (begin
  (display x)
  (if (= x 0)
      (display 'blastoff)
      (f (- x 1))
  )
  )
)

(define (g x)
  (if (= x 0) (display 'blastoff)
      (display (f (- x 1)))
  )
)

(define (h x)
  (if (= x 0) (display 'blastoff)
      (begin (display x) (f (- x 1)) )
  )
)
```

Parallelism & MapReduce

1. Let's say two threads running in parallel both try to execute code that accesses and writes to the same variable. Give the possible outcomes for the print statement.

   ```
   x = 0
   ```

   Thread 1: `x += 1`

   Thread 2: `x += 2`

   ```
   print(x)
   ```

   `1, 2, or 3`

2. Explain how you would use MapReduce to implement a filter. Just explain conceptually what the mapper and the reducer would do. (There are multiple possibilities.)

   mapper:

   ```
   for every value in the sequence:
       if pred_function(value):
           emit(value, 0)
   ```

   reducer:

   ```
   nothing
   ```

   *Note that the mapper emits a (key, value) pair. The value actually isn't needed, so I just put 0 as a placeholder.

3. [Challenge Question] Now let's analyze friend relationships on Facebook. More importantly, given a dictionary of users and their friends, give a dictionary of pairs of users and their mutual friends. The output dictionary should only contain pairs of users who are already friends.
   For example, given:

```
{A: (B, C, D),
 B: (A, C, D, E),
 C: (A, B, D, E),
 D: (A, B, C, E),
 E: (B, C, D)}
```

   the output of MapReduce should be

```
{(A, B): (C, D),
 (A, C): (B, D),
 (A, D): (B, C),
 (B, C): (A, D, E),
 (B, D): (A, C, E),
 (B, E): (C, D),
 (C, D): (A, B, E),
 (C, E): (B, D),
 (D, E): (B, C)}
```

   pseudocode given

```
mapper:
    for every input user, tuple_of_friends in input:
        for every friend in tuple_of_friends:
            emit(sort(user, friend), tuple_of_friends)
reducer:
    for every friends_pair, tuple_of_friends_iterator in input:
        emit(friends_pair, intersection(tuple_of_friends))
```

   The mapper emits every possible (friend1, friend2) pair twice, with the pair serving as the key. The value is going to be either the tuple of friend1's friends, or friend2's friends. Thus, I should see (friend1, friend2) appear twice in the output of the mapper.
   The reducer then takes the intersection of the list of friends because the intersection is necessarily the mutual friends of friend1 and friend2. For a more detailed explanation, visit this awesome website: http://stevekrenzel.com/finding-friends-with-mapreduce

Generators and generator functions

1. Write a generator that takes in two arguments a and b, and returns the sequence that starts with 1,1 and has the recurrence relation: x_n = a*x_{n-2} + b*x_{n-1}

```
def recurrence_gen(a,b):
    #Begin Solution
    yield 1
    prev, curr = 1,1
    while True:
        yield curr
        prev, curr = curr, prev*a + curr*b
    #End Solution
```

2. Write a generator that takes in an iterator and goes over it twice. If the original iterator's next calls return 1,2,3,4, this generator should return 1,1,2,2,3,3,4,4.

```
def twice_iter(iterator):
    #Begin Solution
    while True:
        a = next(iterator)
        yield a
        yield a
```

3. Given an iterator, write a generator that yields the first n elements of the iterator in a list. The first value of this generator should be the empty list.

```
def list_gen(iterator):
    #Begin Solution
    a = list()
    while True:
        yield a
        a.append(next(iterator))
```

Streams

1. Write a function that takes in an iterator and outputs a stream that lists the elements of the iterator in order.

```
def iter_to_stream(iterator):
    #Begin Solution
    def compute_rest():
        return iter_to_stream(iterator)
    try:
        return Stream(next(iterator), compute_rest)
    except:
        return Stream.empty
    #End Solution
```

2. Write a function that takes in a stream, a combiner function of 2 arguments and a starting value and returns a stream that uses the combiner to combine the first k elements of the sequence. For example, if the iterator returns the elements 1,2,3, and 4, your combiner is “add” and your start value is 0, the stream should be a stream of 1,3,6,10.

```
def combine_stream(combiner, stream, start):
    #Begin Solution
    def compute_rest():
        return combine_stream(combiner, stream.rest, first)
    if stream != Stream.empty:
        first = combiner(stream.first, start)
        return Stream(first, compute_rest)
    return Stream.empty
```

Logic http://bit.ly/sp13finalLogic

Learning stuff

In Logic, there are 2 things we can do.
We can either establish facts about our “universe”, or we can issue queries about the “universe”.
All of this is done using Scheme syntax.

Facts

We define facts using the following syntax:

```
(define (fact <conclusion>) <hypothesis> ...)
```

For example:

```
(fact (parent abraham barack))
(fact (parent abraham clinton))
(fact (parent fillmore abraham))
```

These are *simple* facts, which means that they are “naively” true. We don’t have to satisfy any hypotheses for them to be true. All these facts say is that the *parent* relation is true for the symbols “delano” and “herbert”, in that order, and “abraham” and “barack”, again in that order.
We can also have *complex* facts. These are facts which have 1 or more hypotheses.
For example:

```
(fact (grandparent ?x ?y)
      (parent ?x ?z)
      (parent ?z ?y))
```

This fact states that two things (call them *x* and *y*) satisfy the grandparent relation iff the hypotheses hold. These hypotheses state that there must be some variable *z* who’s parent is *x*, and who is the parent of *y*.

Queries

Using the previously defined facts, we can then ask the interpreter questions about the “universe” we’ve defined, and it will try to answer them for us.
The format of a query is

```
(query <relation1> <relation2> ...)
```

So, we can use our previously defined facts to ask the interpreter some questions:

```
logic> (query (parent abraham ?who))
Success!
who: barack
who: clinton
logic> (query (parent ?who clinton))
```

```
Success!
who: abraham
logic> (query (grandparent ?x ?y))
Success!
x: fillmore y: barack
x: fillmore y: clinton
logic> (query (grandparent fillmore ?who))
Success!
who: barack
who: clinton
logic> (query (parent ?x ?z) (parent ?z ?y))
Success!
x: fillmore z: abraham  y: barack
x: fillmore z: abraham  y: clinton
```

Problems

1. Repeat (from the Summer 2012 CS61A final)

For this problem, we will use the unary representation of numbers as described in homework, where a number is represented as a list of the same number of a's. So, for example, the number 3 is represented as <a a a> and the number 0 is represented as <>. Write the rules and associated facts for repeat, which relates three lists in the following manner:

```
logic> (query (repeat (foo bar) ((a a a) (a a)) (foo foo foo bar bar)))
Success!
logic> (query (repeat (foo bar garply) ((a a) () (a a a)) ?what))
Success!
what: (foo foo garply garply garply)
logic> (query (repeat (foo bar) ?what (foo bar bar bar bar)))
Success!
what: ((a) (a a a a))
logic> (query (repeat (foo bar) ?what (bar foo foo))
Failed.
```

2. Reversed and Palindrome (Fall 2012 final)

Fill in two facts below to complete the definitions of the relations reversed and palindrome. The reversed relation indicates that the first list contains the same elements as the second, but in

reversed order. The palindrome relation indicates that a list is the same backward and forward.

```
logic > (fact (append-to-form () ?x ?x))
logic > (fact (append-to-form (?a . ?r) ?y (?a . ?z))
            (append-to-form ?r ?y ?z))
logic > (fact (reversed () ()))
logic > (fact (reversed (?a . ?r) ?s)
            (reversed ?r ?rev)
            ____________________________________________________________)
logic > (query (reversed ?x (a b c d)))
Success!
x: (d c b a)
logic > (fact (palindrome ?s)
            ____________________________________________________________)
logic > (query (palindrome (a b ?x d e ?y ?z))
Success!
x: e y: b z: a
```

3. Pig Latin!

```
def pig_latin(w):
      """Return the Pig Latin equivalent of English word w."""
      if starts_with_a_vowel(w):
            return w + 'ay'
      return pig_latin(w[1:] + w[0])

def starts_with_a_vowel(w):
      """Return whether w begins with a vowel."""
      return w[0].lower() in 'aeiou'
```

Our version of pig latin says that, if a word starts with a vowel, we stick “ay” on the end, and if it doesn’t, we stick the first character at the end and try again. Now, we want to write this using Logic!

We’ll represent words in Logic as lists of single characters. Assume that you have an `append-to-form` that works correctly. You will need to come up with a way to decide if a letter is a vowel or consonant. You don’t need to write these all out, just describe what they would look like. We want the following queries to work:

```
logic> (query (pig-latin (b i g) ?what))
```

```
Success!
what: (i g b a y)
logic> (query (pig-latin (i g) ?what))
Success!
what: (i g a y)
```

Iterators and Iterables

1. I want an iterator that when each next method is called, returns an iterator over natural numbers starting with next number in the iteration.

```
>>> a = NaturalIteratorIterator(4)
>>> b = next(a)
>>> next(b)
1
>>> next(b)
2
>>> b = next(a)
>>> next(b)
2
>>> next(next(a))
3
>>> next(next(a))
3

class NaturalIteratorIterator:
    """YOUR CODE HERE"""
```

2. Write a generator function that returns a building list of natural numbers and terminates after hitting a given number.

```
def natural_list_gen(end):
     """
     >>> n = natural_list_gen(2)
            >>> next(n)
     [1]
     >>> next(n)
     [1 2]
     >>> next(n)
     Traceback (most recent call last):
         ...
     StopIteration
     >>> for i in natural_list_gen(3):
             ...     print(i)
     [1]
     [1 2]
     [1 2 3]
     """
     """YOUR CODE HERE"""
```

3. Write a generator function that returns the intersection of two given ascending iterators.

```
def intersect_iterators(iter1, iter2):
      """
      >>> n1 = NaturalIterator() #1,2,3,...
      >>> def n2(): #0,2,4,6,...
            count = 0
            while True:
                  yield count * 2
                  count += 1
      >>> i = intersect_iterators(n1, n2)
      >>> next(i)
      2
      >>> next(i)
      4
      """
      """YOUR CODE HERE"""
```