FINAL REVIEW 11

COMPUTER SCIENCE 61A

December 1, 2016

1 Mutable Sequences

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

```
>>> cats = [1, 2]
>>> dogs = [cats, cats.append(23), list(cats)]
>>> cats


>>> dogs[1] = list(dogs)
>>> dogs[1]


>>> dogs[0].append(2)
>>> cats


>>> dogs[2].extend([list(cats).pop(0), 3])
>>> dogs[3]


>>> dogs
```

2 Environment Diagram

1. (Fall 2012) Draw the environment diagram.

```
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)
```

3 Object-Oriented Programming

1. Assume these commands are entered in order. What would Python output?

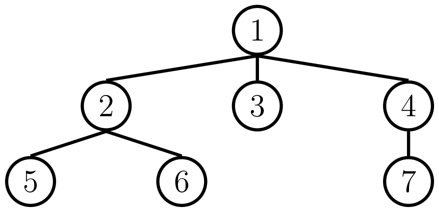
```
>>> class Foo:
...     def __init__(self, a):
...         self.a = a
...     def garply(self):
...         return self.baz(self.a)
>>> class Bar(Foo):
...     a = 1
...     def baz(self, val):
...         return val
>>> f = Foo(4)
>>> b = Bar(3)
>>> f.a


>>> b.a


>>> f.garply()


>>> b.garply()

>>> b.a = 9
>>> b.garply()

>>> f.baz = lambda val: val * val
>>> f.garply()
```

4 Mutable Linked Lists and Trees

1. Write a recursive function `flip_two` that takes as input a linked list `lnk` and mutates `lnk` so that every pair is flipped.

```
def flip_two(lnk):
    """
    >>> one_lnk = Link(1)
    >>> flip_two(one_lnk)
    >>> one_lnk
    Link(1)
    >>> lnk = Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> flip_two(lnk)
    >>> lnk
    Link(2, Link(1, Link(4, Link(3, Link(5)))))
    """
```

2. Write a function `flatten` that given a Tree `t`, will return a linked list of the elements of `t`, ordered by level. Entries on the same level should be ordered from left to right. For example, the following tree will return the linked list `<1 2 3 4 5 6 7>`.



```
def flatten(t):
```

5 Scheme

1. Consider the following Scheme tree data abstraction.

```
(define (make-tree root branches) (cons root branches))
(define (root tree) (car tree))
(define (branches tree) (cdr tree))
(define tree 'below-example)
;                5
;       +--------+--------+
;       |        |        |
;       6        7        2
;    +--+--+     |     +--+--+
;    |     |     |     |     |
;    9     8     1     6     4
;                      |
;                      |
;                      3
```

Write a procedure `tree-sums` that takes a tree of numbers (like the one above) and outputs a list of sums from following each possible path from root to leaf.

Hint: You may find the `flatten` procedure helpful.

```
(define (flatten lst)
  (cond ((null? lst) nil)
        ((list? (car lst)) (append (flatten (car lst)) (
           flatten (cdr lst))))
        (else (cons (car lst) (flatten (cdr lst))))))

(define (tree-sums tree)

  (if _______________________________________________________

      _______________________________________________________

      (map (lambda (x) _____________________________________)

      ____________________________________________________)))

scm> (flatten '(0 (1) ((2)) (3 ((4)))))
(0 1 2 3 4)
scm> (tree-sums tree)
(20 19 13 16 11)
```

6 Streams

1. Implement the `unique-stream` procedure, which takes in a stream and returns new stream that contains each element of the input stream once. Only the first occurrence of each number should be included such that it is in the order that it appears in the original stream. You may want to use `filter-stream` defined below.

```
(define (filter-stream f s)
  (cond
    ((null? s) nil)
    ((f (car s))
      (cons-stream (car s)
                   (filter-stream f (cdr-stream s))))
    (else (filter-stream f (cdr-stream s)))))

(define (unique-stream s)
```

`take` is a procedure that returns a Scheme list containing the first `n` elements a stream `s`.

```
(define (take n s)
  (if (or (= n 0) (null? s))
    nil
    (cons (car s) (take (- n 1) (cdr-stream s)))))

scm> (take 10 (unique-stream (lst-to-stream '(1 3 2 3 4 2))))
(1 3 2 4)
scm> (take 10 (unique-stream (lst-to-stream '(4 4 5 5 6 5))))
(4 5 6)
```

7 Generators

1. Write a generator function that yields functions that are repeated applications of a one-argument function `f`. The first function yielded should apply `f` 0 times (the identity function), the second function yielded should apply `f` once, etc.

```
def repeated(f):
    """
    >>> [g(1) for _, g in
    ...  zip(range(5), repeated(double))]
    [1, 2, 4, 8, 16]
    """

    g = ______________________________

    while True:

        ________________________________________

        ________________________________________
```

2. Ben Bitdiddle proposes the following alternate solution. Does it work?

```
def ben_repeated(f):
    g = lambda x: x
    while True:
        yield g
        g = lambda x: f(g(x))
```

8 SQL

1. You're trying to re-organize your music library! The table `tracks` below contains song titles and the corresponding album. Create another table `tracklist` with two columns: the album and a comma-separated list of all songs from that album in alphabetical.

```
create table tracks as
  select "Human" as title, "The Definition" as album union
  select "Simple and Sweet", "The Definition"        union
  select "Paper Planes", "Translations Through Speakers";

create table tracklist as
  with

    songs(album, total) as (

      _______________________________________________________

    ),

    ____________________________________________________ as (

      _______________________________________________________

      _______________________________________________________

      _______________________________________________________

    )

    select __________________________________________________

      where _______________________________________________;

sqlite3> select * from tracklist order by album;
The Definition|Human, Simple and Sweet
Translations Through Speakers|Paper Planes
```