7 Scheme (Disc 10 & beyond)

7.1 Write a function that takes a procedure and applies to every element in a given nested list.

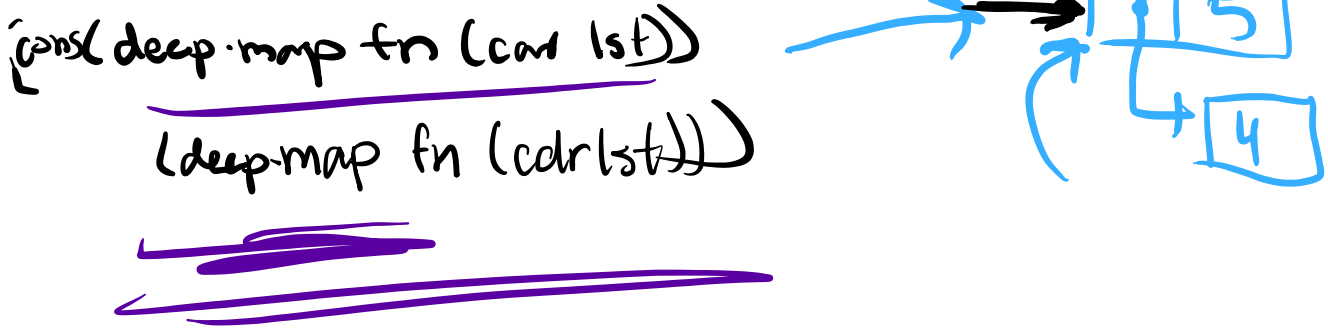
The result should be a nested list with the same structure as the input list, but with each element replaced by the result of applying the procedure to that element.

Use the built-in `list?` procedure to detect whether a value is a list.

```
(define (deep-map fn lst)
```

(cond((null? lst) nil)
((list? (car lst))
(else (cons (fn (car lst))
(deep-map fn (cdr lst)))))

```
scm> (deep-map (lambda (x) (* x x)) '(1 2 3))
(1 4 9)
scm> (deep-map (lambda (x) (* x x)) '(1 ((4) 5) 9))
(1 ((16) 25) 81)
```



(cons(deep-map fn (car lst))
(deep-map fn (cdr lst)))

```
(define (map fn lst) ; return new list where each
  (if (null? lst)      element is fn applied to the
      nil              corresponding elem in lst.
      (cons (fn (car lst))
            (map fn (cdr lst)))
  )
)
```

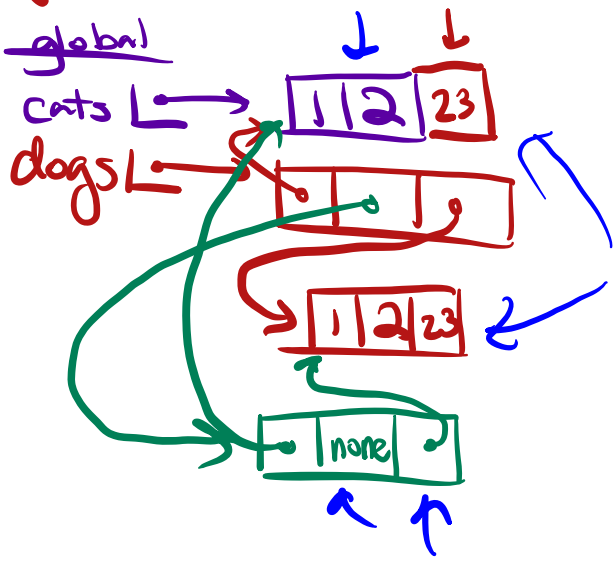
3 Mutation (Disc 04)

3.1 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
```

print, extend, append → none

[1, 2, 23]




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

[[1, 2, 23], none, [1, 2, 23]]

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

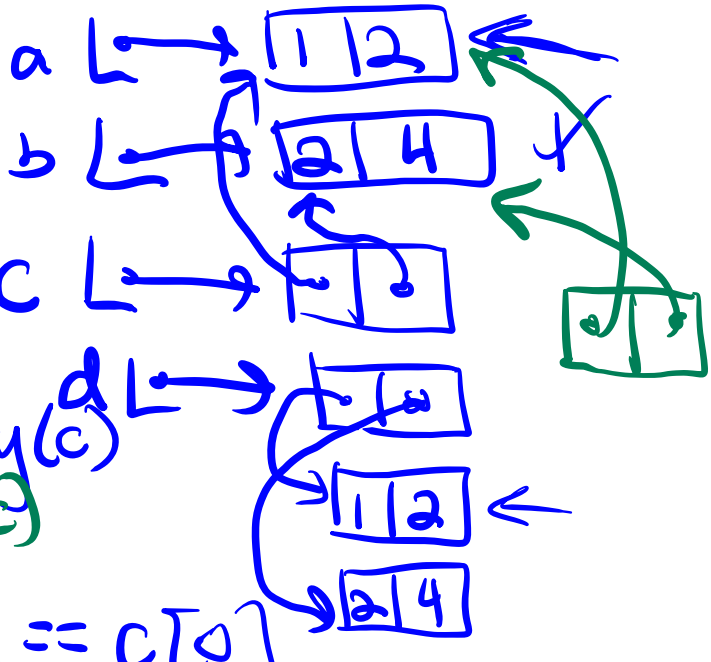
```
>>> cats[1::2]
```

```
>>> cats[:3]
```

1 2

1 2

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



a == b

a is b

a.equals(b)

d[0] == c[0] → false

e[0] == c[0] → true

```
>>> dogs
```

5 Mutable Linked Lists and Trees

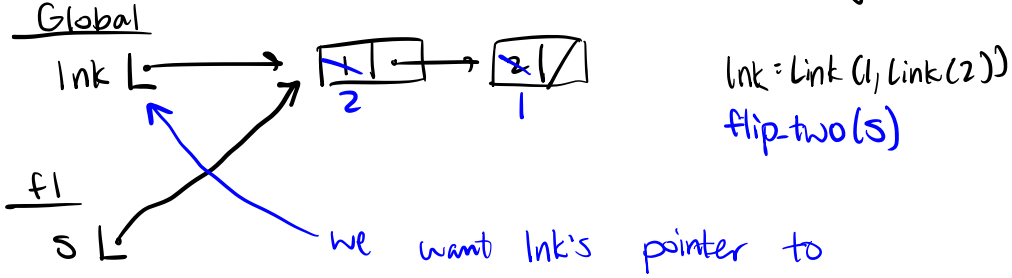
5.1 Write a function that takes a sorted linked list of integers and mutates it so that all duplicates are removed.

```
def remove_duplicates(lnk):
    """
    >>> lnk = Link(1, Link(1, Link(1, Link(1, Link(5)))))
    >>> remove_duplicates(lnk)
    >>> lnk
    Link(1, Link(5))
    """
```

```
def flip_two(s):
    if s is Link.empty or s.rest is Link.empty:
        return s
    s.rest.first, s.rest = s.rest, s.rest.first
    flip_two(s.rest.rest)
```

Disc 07 #6: Flip Two

note that we're switching the values in the boxes & not the boxes → why?



lnk = Link(1, Link(2))
flip_two(s)

we want lnk's pointer to still point @ the beginning of the flipped list

6 Generators (Disc 06)

6.1 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.

Iterators & Generators
- bookmarks
- generators can be fancy
- when call a generator func
  - returns a generator w/o reading body of func
  - which yields items (until it runs out)
  - yield
    - like return but doesn't end the function
    - like a "pause"
    - calling next hits "play"
    - when you run out of tape, StopIteration

```
def repeated(f):
    """
    >>> double = lambda x: 2 * x
    >>> funcs = repeated(double)
    >>> identity = next(funcs)
    >>> double = next(funcs)
    >>> quad = next(funcs)
    >>> oct = next(funcs)
    >>> quad(1)
    4
    >>> oct(1)
    8
    >>> [g(1) for _, g in
    ...  zip(range(5), repeated(lambda x: 2 * x))]
    [1, 2, 4, 8, 16]
    """


    g = ____________________________________________________________________

    while True:


        ____________________________________________________________________


        ____________________________________________________________________
```

6.2 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))
```

2 Trees

2.1 Implement `long_paths`, which returns a list of all *paths* in a tree with length at least `n`. A path in a tree is a linked list of node values that starts with the root and ends at a leaf. Each subsequent element must be from a child of the previous value's node. The *length* of a path is the number of edges in the path (i.e. one less than the number of nodes in the path). Paths are listed in order from left to right. See the doctests for some examples.

```
def long_paths(tree, n):
    """Return a list of all paths in tree with length at least n.

    >>> t = Tree(3, [Tree(4), Tree(4), Tree(5)])
    >>> left = Tree(1, [Tree(2), t])
    >>> mid = Tree(6, [Tree(7, [Tree(8)]), Tree(9)])
    >>> right = Tree(11, [Tree(12, [Tree(13, [Tree(14)])])])
    >>> whole = Tree(0, [left, Tree(13), mid, right])
    >>> for path in long_paths(whole, 2):
    ...     print(path)
    ...
    <0 1 2>
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 6 9>
    <0 11 12 13 14>
    >>> for path in long_paths(whole, 3):
    ...     print(path)
    ...
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 11 12 13 14>
    >>> long_paths(whole, 4)
    [Link(0, Link(11, Link(12, Link(13, Link(14)))))]
    """
```

```
def find_paths(t, entry):
    paths = []
    if t.label == entry:
        paths.append([t.label])
    for b in t.branches:
        for path in find_paths(b, entry):
            paths.append([t.label] + path)
    return paths
```

Disc 07 #8: Find Paths

2.2 Write a function that takes a `Tree` object and returns the elements at the depth with the most elements.

In this problem, you may find it helpful to use the second optional argument to `sum`, which provides a starting value. All items in the sequence to be summed will be concatenated to the starting value. By default, start will default to 0, which allows you to sum a sequence of numbers. We provide an example of sum starting with a list, which allows you to concatenate items in a list.

```
def widest_level(t):
    """
    >>> sum([[1], [2]], [])
    [1, 2]
    >>> t = Tree(3, [Tree(1, [Tree(1), Tree(5)]),
    ...              Tree(4, [Tree(9, [Tree(2)])])])
    >>> widest_level(t)
    [1, 5, 9]
    """
    levels = []
    x = [t]

    while ________________________________________________:

        ____________________________________________________

        __________ = sum(_________________________________, [])

    return max(levels, key=_________________________________)
```

CS 61A
Spring 2021

Final Review

Discussion 14: April 28, 2021

1 Recursion

1.1 (Adapted from Fall 2013) Fill in the blanks in the implementation of `paths`, which takes as input two positive integers `x` and `y`. It returns a list of paths, where each path is a list containing steps to reach `y` from `x` by repeated incrementing or doubling. For instance, we can reach 9 from 3 by incrementing to 4, doubling to 8, then incrementing again to 9, so one path is `[3, 4, 8, 9]`

```
def paths(x, y):
    """Return a list of ways to reach y from x by repeated
    incrementing or doubling.
    >>> paths(3, 5)
    [[3, 4, 5]]
    >>> sorted(paths(3, 6))
    [[3, 4, 5, 6], [3, 6]]
    >>> sorted(paths(3, 9))
    [[3, 4, 5, 6, 7, 8, 9], [3, 4, 8, 9], [3, 6, 7, 8, 9]]
    >>> paths(3, 3) # No calls is a valid path
    [[3]]
    """
    if x == y:
        return [[y]]
    elif x > y:
        return []
    else:
        a = [[x] + path for path in paths(x+1, y)]
        b = [[x] + path for path in paths(2*x, y)]
        return a + b
```

1.2 We will now write one of the faster sorting algorithms commonly used, known as *merge sort*. Merge sort works like this:

1. If there is only one (or zero) item(s) in the sequence, it is already sorted!
2. If there are more than one item, then we can split the sequence in half, sort each half recursively, then merge the results, using the `merge` procedure described below. The result will be a sorted sequence.

Using the algorithm described, write a function `mergesort(seq)` that takes an unsorted sequence and sorts it.

Recall the `merge` procedure is as follows:

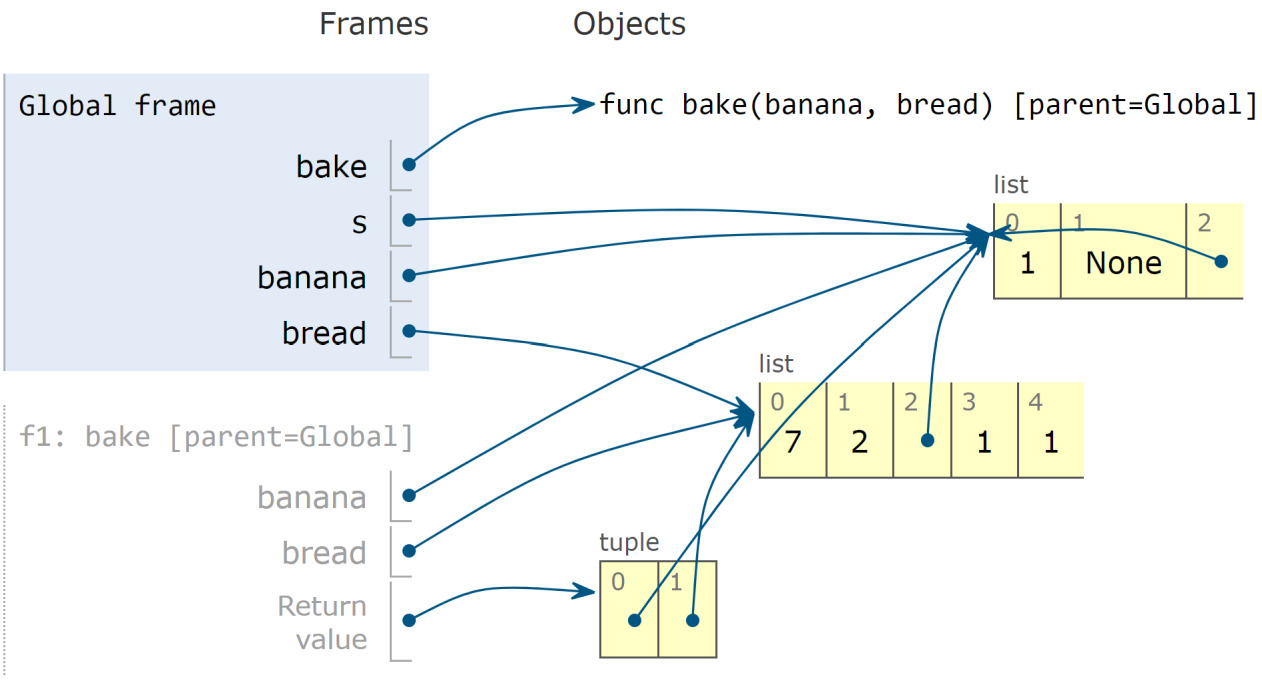
```
def merge(s1, s2):
    """ Merges two sorted lists """
    if len(s1) == 0:
        return s2
    elif len(s2) == 0:
        return s1
    elif s1[0] < s2[0]:
        return [s1[0]] + merge(s1[1:], s2)
    else:
        return [s2[0]] + merge(s1, s2[1:])
```

```
def mergesort(seq):
```

3.2 Fill in the following lines so that the code creates the environment diagram shown below. **You may only use** `append`, `extend`, `1`, `banana`, **and** `bread` **in your solution.**

```
def bake(banana, bread):
    _____________(_____________(__________)) # This line is Multiple Choice
    # Select all correct answers for the blank above
    # A. banana.append(bread.append(1))
    # B. bread.append(banana.append(1))
    # C. banana.extend([bread.extend([1])])
    # D. bread.extend([banana.extend([1])])
    bread += banana[: (len(_____________) - _____________)]
    banana._____________(bread[___________[_____________]])
    return ___________, _____________
```

```
s = [1]
banana, bread = bake(s, [7, 2, s])
```

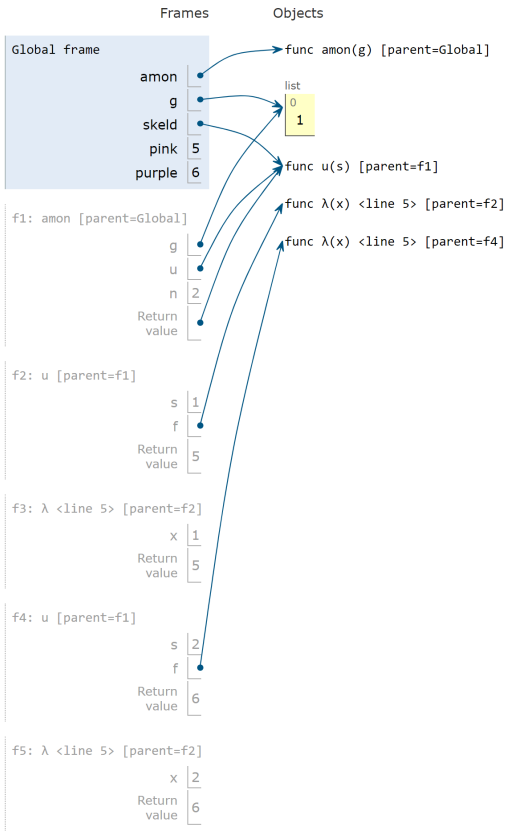


Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

3.3 Fill in the following lines so that the code creates the environment diagram shown below.

```
def amon(g):
    ____________________
            (a)
    def u(s):
        ______________________
                (b)
        f = lambda x: x + g._________ + n
                                (c)
        _________________
                (d)
        return f(s)
    return u

g = [1, 2, 3]
skeld = amon(g)
pink = skeld(1)
purple = skeld(2)
```



4 OOP

4.1 Fill in the classes `Emotion`, `Joy`, and `Sadness` below so that you get the following output from the Python interpreter.

```
>>> Emotion.num
0
>>> joy = Joy()
>>> sadness = Sadness()
>>> Emotion.num # number of Emotion instances created
2
>>> joy.power
5
>>> joy.catchphrase() # Print Joy's catchphrase
Think positive thoughts
>>> sadness.catchphrase() #Print Sad's catchphrase
I'm positive you will get lost
>>> sadness.power
5
>>> joy.feeling(sadness) # When both Emotions have same power value, print "Together"
Together
>>> sadness.feeling(joy)
Together
>>> joy.power = 7
>>> joy.feeling(sadness) # Print the catchphrase of the more powerful feeling before the less
    powerful feeling
Think positive thoughts
I'm positive you will get lost
>>> sadness.feeling(joy)
Think positive thoughts
I'm positive you will get lost
```

```
class Emotion(_______):



    def __init__(self):




    def feeling(self, other):
```

```
class Joy(_______):

    def catchphrase(self):
```

```
class Sadness(_______):

    def catchphrase(self):
```

6.3 Implement `accumulate`, which takes in an `iterable` and a function `f` and yields each accumulated value from applying `f` to the running total and the next element.

```
from operator import add, mul

def accumulate(iterable, f):
    """
    >>> list(accumulate([1, 2, 3, 4, 5], add))
    [1, 3, 6, 10, 15]
    >>> list(accumulate([1, 2, 3, 4, 5], mul))
    [1, 2, 6, 24, 120]
    """
    it = iter(iterable)


    ____________________________________________________________________________


    ____________________________________________________________________________

    for ____________________________________________________________________________:

        ____________________________________________________________________________

        ____________________________________________________________________________
```