MATTHEW'S WEEK 4 PRACTICE PROBLEMS 1

COMPUTER SCIENCE 61A

July 17, 2014

1 Nonlocal Environment Diagram

1. (From Fall 2013 Midterm 2) Fill in the environment diagram that results from executing the code below until the entire program is finished or an error occurs.
   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 miley(ray):
...     def cy():
...         def rus(billy):
...             nonlocal cy
...             cy = lambda: billy + ray
...             return (1, billy)
...         if len(rus(2)) == 1:
...             return (3, 4)
...         else:
...             return (cy(), 5)
...     return cy()[1]
>>> billy = 6
>>> miley(7)
```

2 Lists in Environment Diagrams

1. (From Spring 2014 Midterm 2) Draw the environment diagram (with box-and-pointer diagrams) for the following code:

```
>>> q = [1, 2]
>>> s = [1, 2, [3]]
>>> t = [4, [s, 5], 6]
>>> u = [t]
>>> u.append(u)
```

3 Trees

1. Implement the function `avg_element` which returns the average of all of the elements in the tree `t`. *Hint: you might want to consider using a helper function, similar to how we approached* `deep_inhexing` *from midterm 1.*

```
def avg_element(t):
    """
    >>> new_tree = tree(1, [tree(3, [tree(4)]), \
    tree(5), tree(6, [tree(7), tree(9)])])
    >>> avg_element(new_tree)
    5.0
    >>> first_subtree = children(new_tree)[0]
    >>> avg_element(first_subtree)
    3.5
    """
```

2. Suppose that we want to use trees to represent a person's lineage. Suppose that Rohin has 3 children, Ajay, Ravali, and Taylor. Now suppose that Ravali has two children, Leslie and Alex. We can represent Rohin's family using the following tree:

```
rohin = tree('Rohin', [tree('Ajay'), tree('Ravali', \
[tree('Leslie'), tree('Alex')]), tree('Taylor')])
```

We want to write a function `tree_to_dict` that takes in a lineage tree `t` and returns a dictionary with keys corresponding to the name of a person and values corresponding to a list of that person's children's names.

```
def tree_to_dict(t):
    """
    >>> rohin = tree('Rohin', [tree('Ajay'), tree('Ravali', \
    [tree('Leslie'), tree('Alex')]), tree('Taylor')])
    >>> new_dict = tree_to_dict(rohin)
    >>> new_dict
    {'Rohin': ['Ajay', 'Ravali', 'Taylor'], \
    'Ajay': [], 'Ravali': ['Leslie', 'Alex'],
    'Taylor': [], 'Leslie': [], 'Alex': []}
    """
```

4 List Comprehensions

1. Recall the function `inhexing` from midterm 1. Now that we've learned list comprehensions, we can solve the `inhexing` problem with just one line of code! Remember that the function `inhexing` takes in a Python list of numbers `lst`, a function `hex`, and an integer `n`, and returns a new list where every n^{th} element is replaced by the result of calling `hex` on that element.

```
def inhexing(lst, hex, n):
    """
    >>> inhexing([1, 2, 3, 4, 5], lambda x: 'Poof!', 2)
    [1, 'Poof!', 3, 'Poof!', 5]
    >>> inhexing([2, 3, 4, 5, 6, 7, 8], lambda x: x + 10, 3)
    [2, 3, 14, 5, 6, 17, 8]
    """
```

2. Now write the function `deep_inhexing` from midterm 1 in one line, using list comprehensions.

```
def deep_inhexing(lst, hex, n):
    """
    >>> deep_inhexing([1, 2, 3, 4, 5, 6], lambda x: x + 10, 3)
    [1, 2, 13, 4, 5, 16]
    >>> deep_inhexing([1, [[2]], [3, 4, [5]]], lambda x: 'Poof!', 1)
    ['Poof!', [['Poof!']], ['Poof!', 'Poof!', ['Poof!']]]
    >>> deep_inhexing([1, [2], 3], lambda x: 'Poof!', 2)
    [1, [2], 3]
```