LISTS AND TREES

COMPUTER SCIENCE MENTORS CS 61A

February 19 to February 21, 2018

1 Lists

1. What would Python display? If an error occurs, write "Error". If a function is displayed, write "Function". If nothing is returned, write "Nothing". Also, draw box-and-pointer diagram for each list created.

```
>>> a = [1, 2]
>>> a[1]
```

Solution:
2

```
>>> a.append([3, 4])
>>> a
```

Solution:
[1, 2, [3, 4]]

```
>>> b = a[1:]
>>> a[1] = 5
>>> a[2][0] = 6
>>> b
```

Solution:
[2, [6, 4]]

```
>>> a.extend([7])
>>> a += [8]
```

```
>>> a
```

Solution:
```
[1, 5, [6, 4], 7, 8]
```

```
>>> a += 9
```

Solution:
```
TypeError: 'int' object is not iterable
```

Solution: Box-and-pointer diagram: https://goo.gl/YJfNgf

2. Draw the environment diagram that results from running the code.

```
def reverse(lst):
    if len(lst) <= 1:
        return lst
    return reverse(lst[1:]) + [lst[0]]

lst = [1, [2, 3], 4]
rev = reverse(lst)
```

Solution: https://goo.gl/6vPeX9

CS61A Spring 2019: Catherine Cang and Alex Stennet, with
Albert Xu, Albert Yu, Alex Yang, Allan Yu, Amy Hung, Chris Niu, Danelle Nachum, Evan Corriere, Jack Coyle, Jade Singh, Jason Chang, Jemmy Zhou, Kevin Wang, Lucas Pan, Mary Xu, Michelle Cheung, Olivia Shiah, Patricia Ouyang, Richard Roggenkemper, Samy Cherfaoui, Shreya Sahoo, Varun Jhunjhunwalla

3. Write a function that takes in a list `nums` and returns a new list with only the primes from `nums`. Assume that `is_prime(n)` is defined. You may use a `while` loop, a `for` loop, or a list comprehension.
   `def all_primes(nums):`

Solution:

```
    result = []
    for i in nums:
        if is_prime(i):
            result = result + [i]
    return result

    List comprehension:
    return [x for x in nums if is_prime(x)]
```

2 Trees

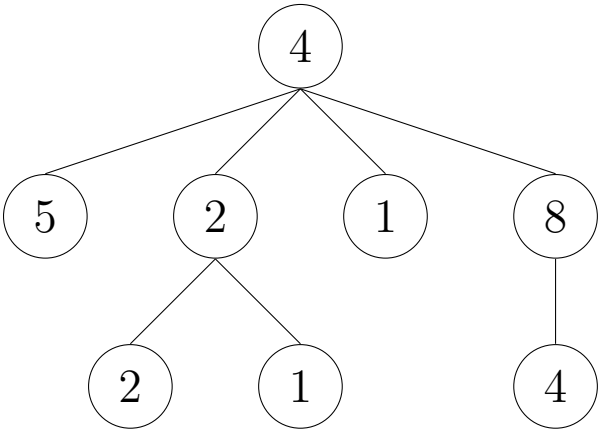
Things to remember:

```
def tree(label, branches=[]):
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:] #returns a list of branches
```

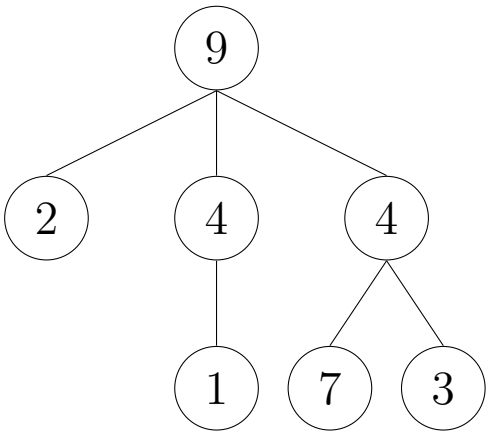
As shown above, the tree constructor takes in a label and a list of branches (which are themselves trees).

```
tree(4,
    [tree(5),
     tree(2,
        [tree(2),
         tree(1)]),
     tree(1),
     tree(8,
        [tree(4)])])
```

This creates a tree that looks like this:



1. Construct the following tree and save it to the variable `t`.



Solution:
```
t = tree(9, [tree(2, []),
             tree(4, [tree(1, [])]),
             tree(4, [tree(7, []),
                      tree(3, [])])])
```

2. What do the following expressions evaluate to? If the expressions evaluates to a tree, format your answer as `tree(... , ...)`. (Note that the python interpreter wouldn't display trees like this. We ask you to do this in order to think about trees as an ADT instead of worrying about their implementation.)

```
>>> label(t)
```

Solution: 9

```
>>> branches(t)[2]
```

Solution:
```
tree(4, [tree(7), tree(3)])
```

```
>>> branches(branches(t)[2])[0]
```

Solution:
```
tree(7)
```

3. Write the Python expression to return the integer 2 from t.

Solution:
```
label(branches(t)[0])
```

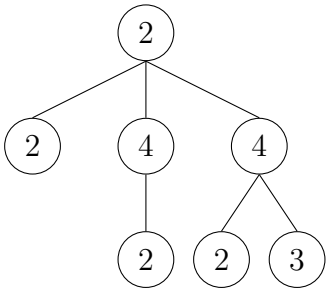
4. Write the function `sum_of_nodes` which takes in a tree and outputs the sum of all the elements in the tree.

```
def sum_of_nodes(t):
    """
    >>> t = tree(...) # Tree from question 2.
    >>> sum_of_nodes(t) # 9 + 2 + 4 + 4 + 1 + 7 + 3 = 30
    30
    """
```

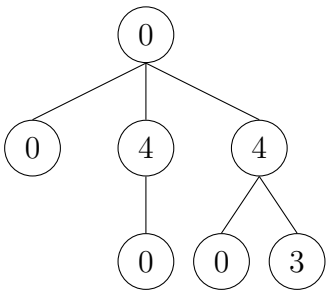
Solution:
```
    total = label(t)
    for branch in branches(t):
        total += sum_of_nodes(branch)
    return total

    Alternative solution:
    return label(t) +\
           sum([sum_of_nodes(b) for b in branches(t)])
```

5. Write a function, `replace_x` that takes in a tree, `t`, and returns a new tree with all labels `x` replaced with 0.

   For example, if we called `replace_x(t, 2)` on the following tree:



   We would expect it to return




```
def replace_x(t, x):
```

Solution:

```
    new_branches = [replace_x(b, x) for b in branches(t)]
    if label(t) == x:
        return tree(0, new_branches)
    return tree(label(t), new_branches)
```

Here, we construct and return a new tree. First, we make a new list of branches where each branch is the same as the previous branch but all occurrences of x have been replaced with 0 (this is what the output of `replace_x` is defined to be.)

If the label of our tree is equal to x, we will additionally need to "replace" our current label with 0 in the tree we return. Otherwise, we can keep our previous label.

These two steps guarantee that each occurrence of x is replaced.

We do not need a base case here, as if we are at a leaf, the list comprehension we use to create the new branches will evaluate to an empty list. Then we will either return `tree(0, [])` or `tree(label(t), [])` as appropriate.