Recursion and Higher Order Functions

Computer Science Mentors 61A

February 15 to February 19, 2016

1 Lists

1. Draw box-and-pointer diagrams for the following.

```
>>> a = [1, 2, 3]
>>> a
```

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

```
>>> a[2]
```

Solution: 3

```
>>> b = a
>>> a = a + [4, 5]
>>> a
```

Solution:
```
[1, 2, 3, 4, 5]
```

```
>>> b
```

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

```
>>> c = a
>>> a = [4, 5]
>>> a
```

Solution:
```
[4, 5]
```

```
>>> c
```

Solution:
```
[1, 2, 3, 4, 5]
```

Solution: Box and pointer diagram in Python Tutor.

2. 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(num):`

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 Data Abstraction

1. The following is an **Abstract Data Type (ADT)** for elephants. Each elephant keeps track of its name, age, and whether or not it can fly. Given our provided constructor, fill out the selectors:

```
def elephant(name, age, can_fly):
    """
    Takes in a string name, an int age, and a boolean can_fly.
    Constructs an elephant with these attributes.
    >>> dumbo = elephant("Dumbo", 10, True)
    >>> elephant_name(dumbo)
        "Dumbo"
    >>> elephant_age(dumbo)
        10
    >>> elephant_can_fly(dumbo)
        True
    """
    return [name, age, can_fly]
def elephant_name(e):
```

Solution:

```
    return e[0]
```

```
def elephant_age(e):
```

Solution:

```
    return e[1]
```

```
def elephant_can_fly(e):
```

Solution:

```
    return e[2]
```

2. This function returns the correct result, but there's something wrong about its implementation. How do we fix it?

```
def elephant_roster(elephants):
    """
    Takes in a list of elephants and returns a list of their
       names.
    """
    result = []
    for elephant in elephants:
        result = result + [elephant[0]]
    return result
```

Solution:
elephant[0] is a Data Abstraction Violation (DAV).
We should use a selector instead.

3. Fill out the following constructor for the given selectors.

```
def elephant(name, age, can_fly):
```

Solution:

```
    return [[name, age], can_fly]
```

```
def elephant_name(e):
    return e[0][0]
def elephant_age(e):
    return e[0][1]
def elephant_can_fly(e):
    return e[1]
```

4. How can we write the fixed `elephant_roster` function for the constructors and selectors in the previous question?

Solution: No change is necessary to fix `elephant_roster` since using the `elephant` selectors "protects" the roster from constructor definition changes.

5. **(Optional)** Fill out the following constructor for the given selectors.

```
def elephant(name, age, can_fly):
    """
    >>> chris = elephant("Chris Martin", 38, False)
    >>> elephant_name(chris)
        "Chris Martin"
    >>> elephant_age(chris)
        37
    >>> elephant_can_fly(chris)
        False
    """
    def select(command)
```

Solution:

```
        if command == "name":
            return name
        elif command == "age":
            return age
        elif command == "can_fly":
            return can_fly
        return "Breaking abstraction barrier!"
```

```
    return select
def elephant_name(e):
    return e("name")
def elephant_age(e):
    return e("age")
def elephant_can_fly(e):
    return e("can_fly")
```

3 Trees

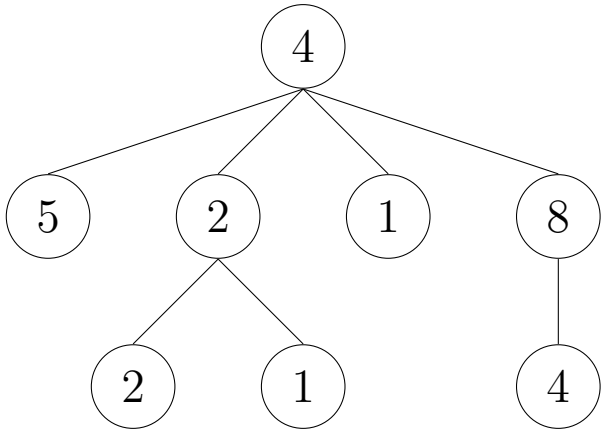
Things to remember

```
def tree(root, branches=[]): # ALWAYS OUTPUTS A TREE
    for branch in branches:
        assert is_tree(branch), 'branches must be trees'
    return [root] + list(branches)

def root(t): # ALWAYS OUTPUTS A NUMBER
    return t[0]

def branches(t): # ALWAYS OUTPUTS A LIST
    return t[1:]
```

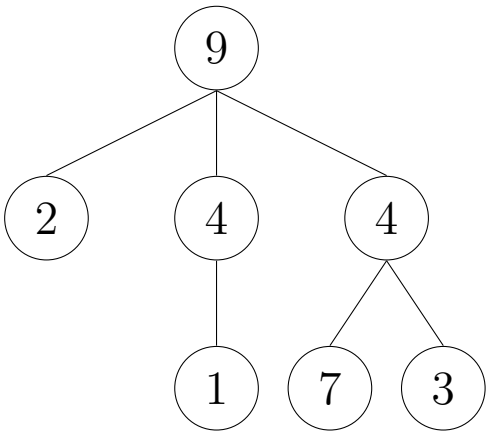
1. Draw the tree that is created by the following statement:

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

Solution:



2. 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, [])])])
```

3. What would this output?

```
>>> root(t)
```

Solution: 9

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

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

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

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

4. Write the Python expression to get the integer 2 from t.

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

5. 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:

```
    sum = 0
    for branch in branches(t):
        sum += sum_of_nodes(branch)
    sum += root(t)
    return sum

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