CS 61A Spring 2019 — Lists, Mutability, ADTs, and Trees

Guerrilla Section 2: March 2, 2019 Solutions

1 Sequences

Questions

1.1 What would Python display?

```
lst = [1, 2, 3, 4, 5]
lst[1:3]
```

[2, 3]

```
lst[0:len(lst)]
```

[1, 2, 3, 4, 5]

```
lst[-4:]
```

[2, 3, 4, 5]

```
lst[:3]
```

[1, 2, 3]

```
lst[3:]
```

[4, 5]

```
lst[:]
```

[1, 2, 3, 4, 5]

```
lst[1:4:2]
```

[2, 4]

```
lst[0:4:3]
```

[1, 4]

```
lst[:4:2]
```

[1, 3]

```
lst[1::2]
```

```
[2, 4]
```

```
lst[::2]
```

```
[1, 3, 5]
```

```
lst[::-1]
```

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

```
lst2 = [6, 1, 0, 7]
lst + lst2
```

```
[1, 2, 3, 4, 5, 6, 1, 0, 7]
```

```
lst + 100
```

```
Error
```

```
lst3 = [[1], [2], [3]]
lst + lst3
```

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

1.2 Draw the environment diagram that results from running the code below

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

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

https://goo.gl/6vPeX9

2 Mutability

Questions

2.1 Name two data types that are mutable. What does it mean to be mutable?

Dictionaries, Lists. Being mutable means we can modify them after they've been created.

2.2 Name at least two data types at are not mutable.

Tuples, functions, int, float

2.3 Will the following code error? If so, why?

```
a = 1
b = 2
dt = {a: 1, b: 2}
```

No -- a and b are both immutable, so we can use them as Dictionary keys.

```
a = [1]
b = [2]
dt = {a: 1, b: 2}
```

Yes -- a and b are mutable, so we can't use them as Dictionary keys.

2.4 Fill in the output and draw a box-and-pointer diagram for the following code. If an error occurs, write Error, but include all output displayed before the error.

```
a = [1, [2, 3], 4]
c = a[1]
c
```

[2, 3]

```
a.append(c)
a
```

[1, [2, 3], 4, [2, 3]]

```
c[0] = 0
c
```

[0, 3]

```
a
```

[1, [0, 3], 4, [0, 3]]

```
a.extend(c)
```

```
c[1] = 9
a
```

```
[1, [0, 9], 4, [0, 9], 0, 3]
```

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]
list1 == list2
```

```
True
```

```
list1 is list2
```

```
False
```

3 Data Abstraction

Questions

3.1 Why are Abstract Data Types useful?

More readable code.
Constructors and selectors have human-readable names.
Makes collaboration easier.
Other programmers don't have to worry about implementation details.
Prevents error propagation.
Fix errors in a single function rather than all over your program.

3.2 What are the two types of functions necessary to make an Abstract Data Type? What do they do?

Constructors make the ADT.
Selectors take instances of the ADT and output relevant information stored in it.

3.3 What is a Data Abstraction Violation? Why is it a terrible sin to commit?

Put simply, a Data Abstraction Violation is when you bypass the constructors and selectors for an ADT, and directly use how its implemented in the rest of your code, thus assuming that its implementation will not change.
We cannot assume we know how the ADT is constructed except by using constructors and likewise, we cannot assume we know how to access details of our ADT except through selectors. The details are supposed to be abstracted away by the constructors and selectors. If we bypass the constructors and selectors and access the details directly, any small change to the implementation of our ADT could break our entire program.

3.4 Assume that **rational**, **numer**, **denom**, and **gcd** run without error and behave as described below. Can you identify where the abstraction barrier is broken? Come up with a scenario where this code runs without error and a scenario where this code would stop working.

```
def rational(num, den): # Returns a rational number ADT
    #implementation not shown
def numer(x): # Returns the numerator of the given rational
    #implementation not shown
def denom(x): # Returns the denominator of the given rational
    #implementation not shown
def gcd(a, b): # Returns the GCD of two numbers
    #implementation not shown

def simplify(f1): #Simplifies a rational number
    g = gcd(f1[0], f1[1])
    return rational(numer(f1) // g, denom(f1) // g)

def multiply(f1, f2): # Multiples and simplifies two rational numbers
```

```
    r = rational(numer(f1) * numer(f2), denom(f1) * denom(f2))
    return simplify(r)

x = rational(1, 2)
y = rational(2, 3)
multiply(x, y)
```

The abstraction barrier is broken inside simplify(f1) when calling gcd(f1[0], f1[1]). This assumes rational returns a type that can be indexed through. i.e. This would work if rational returned a list. However, this would not work if rational returned a dictionary.

The correct implementation of simplify would be

```
def simplify(f1):
    g = gcd(numer(x), denom(x))
    return rational(numer(f1) // g, denom(f1) // g)
```

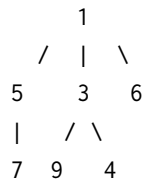
4 Trees

Questions

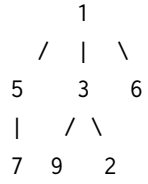
4.1 Fill in this implementation of the Tree ADT.

```
def tree(label, branches = []):
    for b in branches:
        assert is_tree(b), 'branches must be trees'
    return [label] + list(branches)

def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for b in branches(tree):
        if not is_tree(b):
            return False
    return True

def label(tree):

    return tree[0]

def branches(tree):

    return tree[1:]

def is_leaf(tree):

    return not branches(tree)
```

4.2 A min-heap is a tree with the special property that every nodes value is less than or equal to the values of all of its children. For example, the following tree is a min-heap:



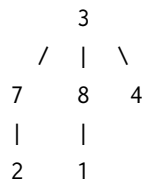
However, the following tree is not a min-heap because the node with value 3 has a value greater than one of its children:



Write a function **is_min_heap** that takes a tree and returns True if the tree is a min-heap and False otherwise.

```
def is_min_heap(t):

    for b in branches(t):
        if label(t) > label(b) or not is_min_heap(b):
            return False
    return True
```

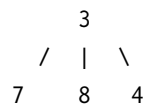
4.3 Write a function **largest_product_path** that finds the largest product path possible. A product path is defined as the product of all nodes between the root and a leaf. The function takes a tree as its parameter. Assume all nodes have a non-negative value.



For example, calling **largest_product_path** on the above tree would return 42, since 3 * 7 * 2 is the largest product path.

```
def largest_product_path(tree):
    """
    >>> largest_product_path(None)
    0
    >>> largest_product_path(tree(3))
    3
    >>> t = tree(3, [tree(7, [tree(2)]), tree(8, [tree(1)]), tree(4)])
    >>> largest_product_path(t)
    42
    """

    if not tree:
        return 0
    elif is_leaf(tree):
        return label(tree)
    else:
        paths = [largest_product_path(t) for t in branches(tree)]
        return label(tree) * max(paths)
```

4.4 Challenge Question: The level-order traversal of a tree is defined as visiting the nodes in each level of a tree before moving onto the nodes in the next level. For example, the level order of the following tree is: 3 7 8 4



Write a function **level_order** that takes in a tree as the parameter and returns a list of the values of the nodes in level order.

```
def level_order(tree):

#iterative solution
   if not tree:
       return []
   current_level, next_level = [label(tree)], [tree]
   while next_level:
       find_next= []
       for b in next_level:
           find_next.extend(branches(b))
       next_level = find_next
       current_level.extend([label(t) for t in next_level])
   return current_level


#recursive solution
   def find_next(current_level):
           if current_level == []:
               return []
           else:
               next_level = []
               for b in current_level:
                   next_level.extend(branches(b))
            return [label(t) for t in next_level] + find_next(next_level)
   return [label(tree)] + find_next([tree])
```