CS 61A Lists, Mutability, ADTs, and Trees

Summer 2019 Guerrilla Section 2: July 12, 2019

1 Sequences

Questions

1.1 What would Python display?

```
lst = [1, 2, 3, 4, 5]
lst[1:3]
|\begin{solution}
[2, 3]
\end{solution}|
lst[0:len(lst)]
|\begin{solution}
[1, 2, 3, 4, 5]
\end{solution}|
lst[-4:]
|\begin{solution}
[2, 3, 4, 5]
\end{solution}|
lst[3:]
|\begin{solution}
[4, 5]
\end{solution}|
lst[1:4:2]
|\begin{solution}
[2, 4]
\end{solution}|
lst[:4:2]
|\begin{solution}
[1, 3]
\end{solution}|
lst[1::2]
|\begin{solution}
[2, 4]
\end{solution}|
lst[::-1]
|\begin{solution}
[5, 4, 3, 2, 1]
\end{solution}|
lst + 100
|\begin{solution}
Error (These aren't numpy arrays)
```

```
\end{solution}|
lst3 = [[1], [2], [3]]
lst + lst3
|\begin{solution}
[1, 2, 3, 4, 5, [1], [2] , [3]]
\end{solution}|
```

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)
```

1.3 Implement a function *map_mut* that takes a list as an argument and maps a function f onto each element of the list. You should mutate the original lists, without creating any new lists. Do NOT return anything.

```
def map_mut(f, L):
    >>> L = [1, 2, 3, 4]
    >>> map_mut(lambda x: x**2, L)
    >>> L
    [1, 4, 9, 16]
```

1.4 Check your understanding

1 When copying the list, when are you copying a pointer of the list vs. copying the actual value inside of a list?

2 How would you make a deep copy of a list?

2 Mutability

Questions

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

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

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

```
a = 1
b = 2
dt = {a: 1, b: 2}
|\begin{solution}
No -- a and b are both immutable, so we can use them as Dictionary keys.
\end{solution}|

a = [1]
b = [2]
dt = {a: 1, b: 2}
|\begin{solution}
Yes -- a and b are mutable, so we can't use them as Dictionary keys.
\end{solution}|
```

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
|\begin{solution}
[2, 3]
\end{solution}|
a.append(c)
a
|\begin{solution}
[1, [2, 3], 4, [2, 3]]
\end{solution}|
c[0] = 0
c
|\begin{solution}
[0, 3]
\end{solution}|
a
|\begin{solution}
[1, [0, 3], 4, [0, 3]]
\end{solution}|
a.extend(c)
```

```
c[1] = 9
a
|\begin{solution}
[1, [0, 9], 4, [0, 9], 0, 3]
\end{solution}|
list1 = [1, 2, 3]
list2 = [1, 2, 3]
list1 == list2
|\begin{solution}
True
\end{solution}|
list1 is list2
|\begin{solution}
False
\end{solution}|
```

2.5 Check your understanding:

1 What is the difference between the append function, extend function, and the '+' operator?

2 Given the below code, answer the following questions: a = [1, 2, [3, 4], 5]
b = a[:]
b[1] = 6
b[2][0] = 7

What does b evaluate to?

What does a evaluate to? Are a and b the same? Please explain your reasoning.

3 Data Abstraction

Questions

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

3.2 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)
```

3.3 Check your understanding

1 How do we know what we are breaking an abstraction barrier?

2 What are the benefits to Data Abstraction?

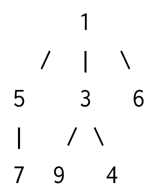
4 Trees

Questions

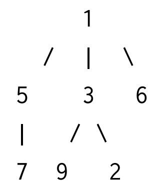
4.1 Fill in this implementation of the Tree ADT.

```
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):
    |\begin{solution}
    \begin{verbatim}
    return tree[0]
\end{verbatim}
\end{solution}|

def branches(tree):
    |\begin{solution}
    \begin{verbatim}
    return tree[1:]
\end{verbatim}
\end{solution}|

def is_leaf(tree):
    |\begin{solution}
    \begin{verbatim}
    return not branches(tree)
\end{verbatim}
\end{solution}|
```

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:



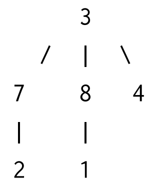
However, the following tree is not a min-heap because the node with value 3 has a value greater than one of its children:



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):
    |\begin{solution}[0.75in]
    \begin{verbatim}
    for b in branches(t):
        if label(t) > label(b) or not is_min_heap(b):
            return False
    return True
    \end{verbatim}
    \end{solution}|
```

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.



```
        3
      / | \
    7   8   4
    |   |
    2   1
```

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
    """
    |\begin{solution}[1in]
    \begin{verbatim}
    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)
    \end{verbatim}
    \end{solution}|
```

4.4 Check your understanding:

1 Given the first tree in 4.2, write the corresponding python call to create the tree

2 What is the benefit of using a tree as a data structure, rather than a list or linked list?

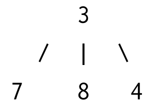
3 Below is the function contains, which takes in an input of a tree, t and a value, e. The function returns true if e exists as a label inside t. However, the function does not work properly, debug this code and find the error(s).

```
def contains(t, e):
     if is_leaf(t):
        return False
     elif e == label(t):
        return True
     else:
        for b in branches(t):
            return contains(b, e)
        return True
```

4 Implement a function max_tree, which takes a tree t. It returns a new tree with the exact same structure as t; at each node in the new tree, the entry is the largest number that is contained in that node's subtrees or the corresponding node in t.

```
def max_tree(t):
>>> max_tree(tree(1, [tree(5, [tree(7)]),tree(3,[tree(9),tree(4)]),tree(6)]))
    tree(9, [tree(7, [tree(7)]),tree(9,[tree(9),tree(4)]),tree(6)])
    if ___________:
        return _________________
    else:
        new_branches= ______________________________
        new_label = __________________________________
        return ______________________
```

4.5 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):
|\begin{solution}
\begin{verbatim}
#iterative solution
def level_order(tree)
    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

\end{verbatim}
\end{solution}|
```

4.6 Challenge Question: Write a function **all_paths** which will return a list of lists of all the possible paths of an input tree, t. When the function is called on the same tree as the problem above, the function would return: $[[3, 7], [3, 8], [3, 4]]$

```
def all_paths(t):
    if _______________:
        ____________________
    else:
        ___________________
        ____________________
            ________________________
                ______________________
```

____