SEQUENCES AND TREES 4

COMPUTER SCIENCE 61A

February 19, 2015

1 List Comprehension

A **list comprehension** is a compact way to create a list whose elements are the results of applying a fixed expression to elements in another sequence.

```
[<map exp> for <name> in <iter exp> if <filter exp>]
```

Let's break down an example:

```
[x * x - 3 for x in [1, 2, 3, 4, 5] if x % 2 == 1]
```

In this list comprehension, we are creating a new list after performing a series of operations to our initial sequence `[1, 2, 3, 4, 5]`. We only keep the elements that satisfy the filter expression `x % 2 == 1` (`1`, `3`, and `5`). For each retained element, we apply the map expression `x*x - 3` before adding it to the new list that we are creating, resulting in the output `[-2, 6, 22]`.

Note: The `if` clause in a list comprehension is optional.

1.1 Questions

1. What would Python print?

```
>>> [i + 1 for i in [1, 2, 3, 4, 5] if i % 2 == 0]
```

```
>>> [i * i for i in [5, -1, 3, -1, 3] if i > 2]
```

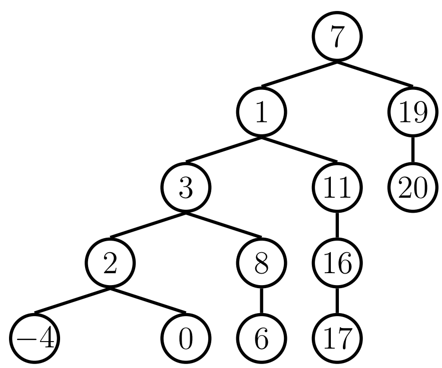
```
>>> [[y * 2 for y in [x, x + 1]] for x in [1, 2, 3, 4]]
```

2. Define a function `foo` that takes in a list `lst` and returns a new list that keeps only the even-indexed elements of `lst` and multiples each of those elements by the corresponding index.

```
def foo(lst):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> foo(x)
    [0, 6, 20]
    """

    return [________________________________________________]
```

2 Trees

In computer science, **trees** are recursive data structures that are widely used in various settings. This is a diagram of a simple tree.



Notice that the tree branches downward – in computer science, the **root** of a tree starts at the top, and the **leaves** are at the bottom.

Some terminology regarding trees:

- **Parent node**: A node that has children. Parent nodes can have multiple children.
- **Child node**: A node that has a parent. A child node can only belong to one parent.
- **Root**: The top node of the tree. In our example, the node that contains 7 is the root.
- **Leaf**: A node that has no children. In our example, the nodes that contain -4, 0, 6, 17, and 20 are leaves.
- **Subtree**: Notice that each child of a parent is itself the root of a smaller tree. In our example, the node containing 1 is the root of another tree. This is why trees are *recursive* data structures: trees are made up of subtrees, which are trees themselves.

- **Depth**: How far away a node is from the root. In other words, the length of the path from the root to the node. In the diagram, the node containing 19 has depth 1; the node containing 3 has depth 2. We define the depth of the root of a tree is 0.
- **Height**: The depth of the lowest leaf. In the diagram, the nodes containing -4, 0, 6, and 17 are all the "lowest leaves," and they have depth 4. Thus, the entire tree has height 4.

In computer science, there are many different types of trees – some vary in the number of children each node has, and others vary in the structure of the tree.

2.1 Implementation

A tree has both a root value and a sequence of branches. In our implementation, we represent the branches as lists of subtrees.

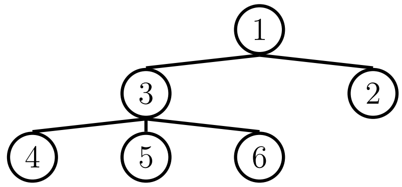
- The arguments to the constructor, `tree`, as a value for the root and a list of branches.
- The selectors are `root` and `branches`.

```
# Constructor
def tree(value, branches=[]):
    for branch in branches:
        assert is_tree(branch), 'branches must be trees'
    return [value] + list(branches)

# Selectors
def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]
```

We have also provided two convenience functions, `is_leaf` and `is_tree`:

```
def is_leaf(tree):
    return not branches(tree)

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

It's simple to construct a tree. Let's try to create the following tree:




```
t = tree(1,
         [tree(3,
               [tree(4),
                tree(5),
                tree(6)]),
          tree(2)])
```

The use of whitespace can help with legibility, but it is not required.

2.2 Questions

1. Define a function `square_tree(t)` that squares every item in the tree `t`. It should return a new tree. You can assume that every item is a number.

```
def square_tree(t):
    """Return a tree with the square of every element in t"""
```

2. Define a function `height(t)` that returns the height of a tree. Recall that the height of a tree is the length of the longest path from the root to a leaf.

```
def height(t):
    """Return the height of a tree"""
```

CS 61A Spring 2015: John DeNero, with
Soumya Basu, Kevin Chen, Rohan Chitnis, Timothy Greeno, Jessica Gu, Brian Hou, Robert Huang, Andrew Huang, Joy Jeng, Austin Le, Youri Park, Harold Pimentel, Sumukh Sridhara, Steven Tang, Michelle Tian, Alana Tran, Dickson Tsai, Max Wolffe, Albert Wu, Marvin Zhang

3. Define a function `tree_size(t)` that returns the number of nodes in a tree.

```
def tree_size(t):
    """Return the size of a tree."""
```

4. Define a function `tree_max(t)` that returns the largest number in a tree.

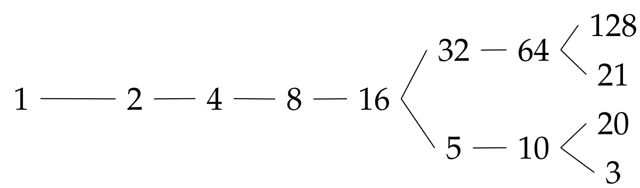
```
def tree_max(t):
    """Return the max of a tree."""
```

2.3 Extra Questions

1. An **expression tree** is a tree that contains a function for each non-leaf root, which can be either `add` or `mul`. All leaves are numbers. Implement `eval_tree`, which evaluates an expression tree to its value. You may find the functions `reduce` and `apply_to_all`, introduced during lecture, useful.

```
def reduce(fn, s, init):
    reduced = init
    for x in s:
        reduced = fn(reduced, x)
    return reduced

def apply_to_all(fn, s):
    return [fn(x) for x in s]

from operator import add, mul
def eval_tree(tree):
    """Evaluates an expression tree with functions as root
    >>> eval_tree(tree(1))
    1
    >>> expr = tree(mul, [tree(2), tree(3)])
    >>> eval_tree(expr)
    6
    >>> eval_tree(tree(add, [expr, tree(4)]))
    10
    """
```

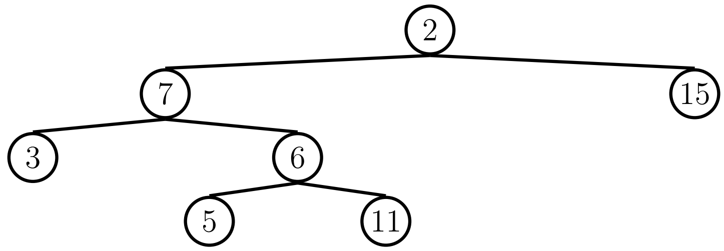
2. We can represent the hailstone sequence as a tree in the figure below, showing the route different numbers take to reach 1. Remember that a hailstone sequence starts with a number n, continuing to $n/2$ if n is even or $3n + 1$ if n is odd, ending with 1. Write a function `hailstone_tree(n, h)` which generates a tree of height `h`, containing hailstone numbers that will reach `n`.




```
def hailstone_tree(n, h):
    """Generates a tree of hailstone numbers that will reach N
       , with height H.
    >>> hailstone_tree(1, 0)
    [1]
    >>> hailstone_tree(1, 4)
    [1, [2, [4, [8, [16]]]]]
    >>> hailstone_tree(8, 3)
    [8, [16, [32, [64]], [5, [10]]]]
    """
```

3. Define the procedure `find_path(tree, x)` that, given a rooted tree `tree` and a value `x`, returns a list containing the nodes along the path required to get from the root of `tree` to a node `x`. If `x` is not present in `tree`, return `None`. Assume that the elements in `tree` are unique.

   For the following tree, `find_path(t, 5)` should return `[2, 7, 6, 5]`




```
def find_path(tree, x):
    """ Return a path in a tree to a leaf with value X,
    None if such a leaf is not present.
    >>> t = tree(2, [tree(7, [tree(3), tree(6, [tree(5), tree
        (11)])]), tree(15)])
    >>> find_path(t, 5)
    [2, 7, 6, 5]
    >>> find_path(t, 6)
    [2, 7, 6]
    >>> find_path(t, 10)
    """
```