SEQUENCES AND TREES 5

COMPUTER SCIENCE 61A

October 2, 2014

1 List Comprehension

List comprehensions are a compact way of applying operations to a sequence. The basic format of a list comprehension is:

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

Let's break down an example of a list comprehension:

```
[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` (in this case the elements 1, 3, and 5). For each element that we keep, we want to apply the map expression `x*x - 3` before adding it to the new list that we are creating, resulting in an output list: `[-2, 6, 22]`

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

1.1 Questions

1. What would Python print?

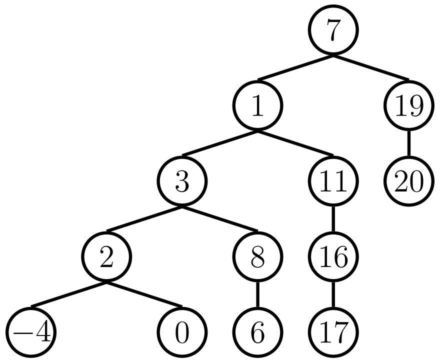
```
>>> [i + 1 for i in [1, 2, 3, 4, 5] if i % 2 == 0]


>>> [(lambda j: j * j)(2 * 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. There is only one root. Because every other node branches directly or indirectly from the root, it is possible to start from the root and reach any other node in the tree. The root is, of course, a parent — it is the only node that is not a child. For example, the node that contains the 7 at the top is the root.

- **Leaf**: Nodes that have no children. For example, the nodes that contain the bottom -4, 0, 6, 17, and 20 are leaves. The node that contains 19 is not a leaf, since it has one child.
- **Subtree**: Notice that each child of a parent is itself the root of a smaller tree (for 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, how many generations away from the root is the specific child node? In the diagram, the node containing 19 has depth 1; the node containing 3 has depth 2. We define the root of a tree to have depth 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.

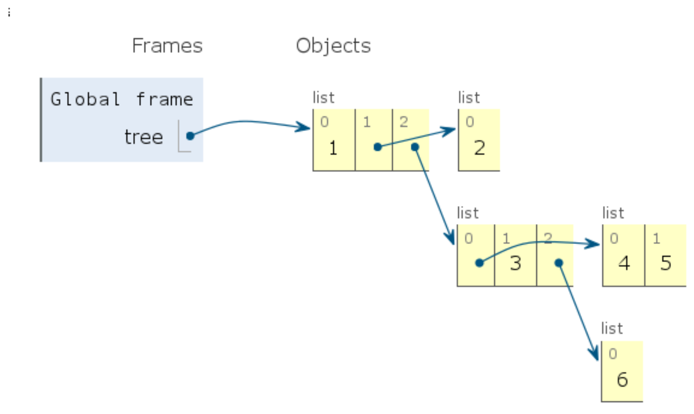
In computer science, there are many different types of trees – some vary in the number of children each node has, and others vary in the structure of the tree.

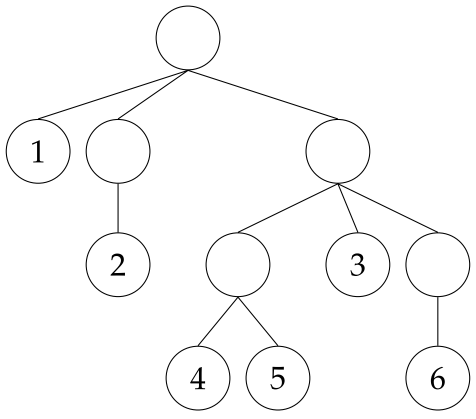
2.1 Our Implementation

In this class, we represent a *tree* as a nested sequence. A tree is a sequence of trees or a non-sequence value, called a *leaf*. For example, consider the following tree with 6 integer leaves:

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

We can visualize this tree with a box-and-pointer diagram, or as a tree shown below. Note that unlike the tree we've seen earlier, this representation of a tree only stores values at its leaves.





When a tree is a sequence, each element in the sequence is called a branch of the tree. Each branch is itself a tree, according to our definition. For now, we will assume a that trees are lists and leaves are non-lists.

```
def is_leaf(tree):
    return type(tree) != list
```

Many of the common patterns we have found for processing sequences also apply to trees. For instance, the function `apply_to_leaves` is similar to `apply_to_all`, but recursively applies a function only to the leaves of a tree, creating another tree as a result.

```
def apply_to_leaves(map_fn, tree):
    """Apply map_fn to all leaves of tree,
    constructing another tree."""
    if is_leaf(tree):
        return map_fn(tree)
    else:
        return [apply_to_leaves(map_fn, branch)
                for branch in tree]
```

2.2 Questions

1. Define a function `square_tree(tree)` that uses `apply_to_leaves` to square every item in `tree`. You can assume that every item is a number.

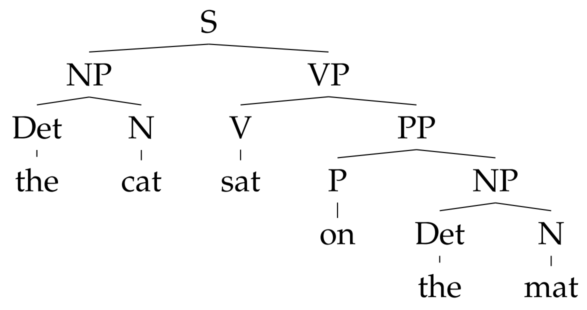
```
def square_tree(tree):
    """Returns a tree containing integers by squaring all of
    TREE's elements
    >>> square_tree([1, [2, [3]], 4])
    [1, [4, [9]], 16]
    """

    return apply_to_leaves(______________________, ________)
```

2. Define a function `height(tree)` that returns the height of a tree. The height of a Tree is defined as the length of the *longest* path from the root node down to a leaf node. In our representation of trees as a nested sequence, the height is the highest level of nesting.

   If it helps, there is a Python built-in function `max` that takes an arbitrarily long sequence of numbers and returns the maximum value in the sequence.

```
def height(tree):
    """Returns the height of a tree
    >>> height(1)
    0
    >>> height([1])
    1
    >>> height([1, [2, [3]], [4, [5]]])
    3
    """
```

3. In linguistics, sentences are parsed according to a grammar into a tree, such as the one below. Write a function `join`, which takes in a tree consisting of strings in its leaves and turns it back into a sentence.



```
def join(tree):
    """ Joins a tree containing strings as leaves into a
    sentence.
    >>> join('hi')
    'hi'
    >>> join(['hello', ' ', 'world'])
    'hello world'
    >>> join([['the', 'cat'], ['sat', ['on', ['the', 'mat
       ']]]]])
    'thecatsatonthemat'
    """
```

4. Define a function `search(tree, x)` that searches a tree for leaves of value `x`. `search` should return the minimum depth of all such leaves, or `False` if no leaves are found.

```
def search(tree, x):
    """Returns the minimum depth of a leaf with value X in
    TREE, False if not found.
    >>> search([2, [3, 4], [1, [5, [3]]]], 3)
    2
    >>> search([2, [3, 4], [1, [5, [3]]]], 6)
    False
    """
```

2.3 Rooted Trees

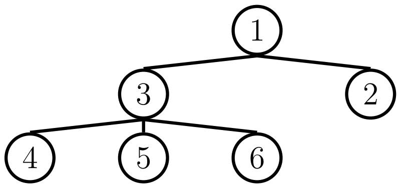
A *rooted tree* has both a sequence of branches and a root value. A rooted tree with no branches is also called a *leaf*. The data abstraction for a rooted tree consists of the constructor `rooted` and the selectors `root` and `branches`.

```
def rooted(value, branches):
    return [value] + list(branches)

def root(tree):
    return tree[0]

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

We introduce convenience methods `leaf` to create leaves and `is_rooted_leaf` to check if a rooted tree is a leaf.

```
def leaf(value):
    return rooted(value, [])

def is_rooted_leaf(tree):
    return branches(tree) == []

>>> t = rooted(1, [leaf(2), leaf(3)])
>>> t
[1, [2], [3]]
>>> root(t)
1
>>> branches(t)
[[2], [3]]
```

With rooted trees, we can associate any root of a subtree with a value. For example, the following tree can be represented as a rooted tree:

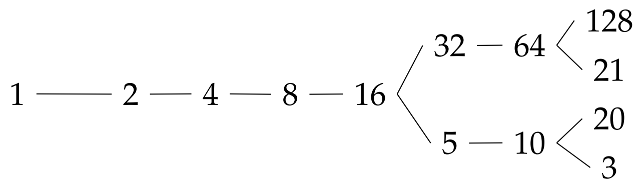


```
rooted(1, [rooted(3, [leaf(4), leaf(5), leaf(6)]), leaf(2)])
```

5. An *expression tree* is a rooted tree, containing 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(leaf(1))
    1
    >>> expr = rooted(mul, [leaf(2), leaf(3)])
    >>> eval_tree(expr)
    6
    >>> eval_tree(rooted(add, [expr, leaf(4)]))
    10
    """
```

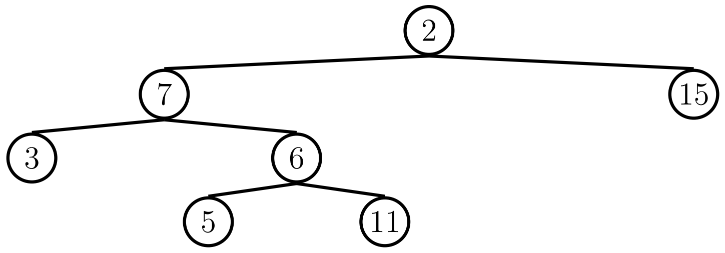
6. We can represent the hailstone sequence as a rooted tree in the figure below, showing the route different numbers take to reach 1. Remember that a hailstone sequence starts with a number n, continuing to $n/2$ if n is even or $n * 3 + 1$ if n is odd, ending with 1. Write a function `hailstone_tree(n,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



```
def hailstone_tree(n, h):
    """Generates a rooted 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]]]]
    """
```

7. 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 `False`. 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):
    """ Returns a path in a tree to a leaf with value X,
    False if such a leaf is not present.
    >>> r, l = rooted, leaf
    >>> t = r(2, [r(7, [l(3), r(6, [l(5), l(11)])]), l(15)])
    >>> find_path(t, 5)
    [2, 7, 6, 5]
    >>> find_path(t, 6)
    [2, 7, 6]
    >>> find_path(t, 10)
    False
    """
```