1 List Comprehensions

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.

\[
\text{[<map exp> for <name> in <iter exp> if <filter exp>]}\]

It might be helpful to note that we can rewrite a list comprehension as an equivalent for statement. See the example to the right.

Let’s break down an example:

\[
[x * x - 3 \text{ for } x \text{ in } [1, 2, 3, 4, 5] \text{ if } x \% 2 == 1] \]

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.

Questions

1.1 What would Python display?

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

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

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

In computer science, trees are recursive data structures that are widely used in various settings. The diagram to the right is an example of a 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 branches. Parent nodes can have multiple branches.

- **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.

- **Label**: The value at a node. In our example, all of the integers are values.

- **Leaf**: A node that has no branches. In our example, the nodes that contain $-4$, $0$, $6$, $17$, and $20$ are leaves.

- **Branch**: A subtree of the root. Note that trees have branches, which are trees themselves: this is why trees are recursive data structures.

- **Depth**: How far away a node is from the root. In other words, the number of edges between the root of the tree to the node. In the diagram, the node containing $19$ has depth $1$; the node containing $3$ has depth $2$. Since there are no edges between the root of the tree and itself, the depth of the root 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 branches each node has; others vary in the structure of the tree.
Implementation

A tree has both a value for the root node and a sequence of branches, which are also trees. In our implementation, we represent the branches as a list of trees. Since a tree is an abstract data type, our choice to use lists is just an implementation detail.

- The arguments to the constructor `tree` are the value for the root node and a list of branches.
- The selectors for these are `label` and `branches`.

Note that `branches` returns a list of trees and not a tree directly. It’s important to distinguish between working with a tree and working with a list of trees.

We have also provided a convenience function, `is_leaf`.

Let’s try to create the tree below.

```python
# Example tree construction
t = tree(1,
    [tree(3,
        [tree(4),
        tree(5),
        tree(6)])],
    tree(2))
```

```python
# Constructor
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

# Selectors
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]

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

2.1 Write a function that returns the largest number in a tree.

```python
def tree_max(t):
    """Return the maximum label in a tree."
    
    >>> t = tree(4, [tree(2, [tree(1)]), tree(10)])
    >>> tree_max(t)
    10
    """
```

2.2 Write a function 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.

```python
def height(t):
    """Return the height of a tree."
    
    >>> t = tree(3, [tree(5, [tree(1)]), tree(2)])
    >>> height(t)
    2
    """
```

2.3 Write a function that takes in a tree and squares every value. It should return a new tree. You can assume that every item is a number.

```python
def square_tree(t):
    """Return a tree with the square of every element in t"""
```
2.4 Write a function that takes in a tree and a value \(x\) and returns a list containing the nodes along the path required to get from the root of the tree to a node containing \(x\).

If \(x\) is not present in the tree, return \textit{None}. Assume that the entries of the tree are unique.

For the following tree, \texttt{find\_path(t, 5)} should return \([2, 7, 6, 5]\)

```
def find_path(tree, x):
    """
    >>> 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, 10) # returns None
    """

    if ________________________________:
        return __________________________
        ________________________________:
        path = __________________________
        if ________________________________:
            return __________________________
```

2.5 Write a function that takes in a tree and a depth \(k\) and returns a new tree that contains only the first \(k\) levels of the original tree.

For example, if \(t\) is the tree shown in the previous question, then \texttt{prune(t,}
2) should return the following tree.

```
Tree representation

def prune(t, k):

26 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 \( n \).

*Hint:* A node of a hailstone tree will always have at least one, and at most
two branches (which are also hailstone trees). Under what conditions do you
add the second branch?

```

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]]]]
```