Homework 3

● Graded

Student
Sangwon Ji

Total Points
2 / 2 pts

Autograder Score
2.0 / 2.0

Autograder Results

```
=====================================================================
Assignment: Homework 3
OK, version v1.18.1
=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Scoring tests

---------------------------------------------------------------------
Doctests for filter

>>> from hw03 import *
>>> original_list = [5, -1, 2, 0]
>>> filter(lambda x: x % 2 == 0, original_list)
>>> original_list
[2, 0]
Score: 1.0/1

---------------------------------------------------------------------
Doctests for deep_map_mut

>>> from hw03 import *
>>> l = [1, 2, [3, [4], 5], 6]
>>> deep_map_mut(lambda x: x * x, l)
>>> l
[1, 4, [9, [16], 25], 36]
Score: 1.0/1

---------------------------------------------------------------------
Doctests for has_path

>>> from hw03 import *
>>> greetings = tree('h', [tree('i'),
... tree('e', [tree('l', [tree('l', [tree('o')])]),
... tree('y')])])
>>> print_tree(greetings)
h
i
e
```

```
l
l
o
y
>>> has_path(greetings, 'h')
True
>>> has_path(greetings, 'i')
False
>>> has_path(greetings, 'hi')
True
>>> has_path(greetings, 'hello')
True
>>> has_path(greetings, 'hey')
True
>>> has_path(greetings, 'bye')
False
>>> has_path(greetings, 'hint')
False
Score: 1.0/1

---------------------------------------------------------------------
Point breakdown
filter: 1.0/1
deep_map_mut: 1.0/1
has_path: 1.0/1

Score:
Total: 3.0

Cannot backup when running ok with --local.
---------------------------------------------------------------------
Final Score:2.0
```

Submitted Files

hw03.py Download

```
def filter(pred, lst):
    """Filters lst with pred using mutation.
    >>> original_list = [5, -1, 2, 0]
    >>> filter(lambda x: x % 2 == 0, original_list)
    >>> original_list
    [2, 0]
    """
    new_list =[]
    while len(lst) >0:
        element = lst.pop(0)
        if pred(element):
            new_list.append(element)
    lst.extend(new_list)


def deep_map_mut(func, lst):
    """Deeply maps a function over a Python list, replacing each item
    in the original list object.

    Does NOT create new lists by either using literal notation
    ([1, 2, 3]), +, or slicing.

    Does NOT return the mutated list object.

    >>> l = [1, 2, [3, [4], 5], 6]
    >>> deep_map_mut(lambda x: x * x, l)
    >>> l
    [1, 4, [9, [16], 25], 36]
    """
    for i in range(len(lst)):
        if type(lst[i]) == list:
            deep_map_mut(func, lst[i])
        else:
            lst[i] = func(lst[i])


def has_path(t, word):
    """Return whether there is a path in a tree where the entries along the path
    spell out a particular word.

    >>> greetings = tree('h', [tree('i'),
    ...                        tree('e', [tree('l', [tree('l', [tree('o')])]),
    ...                                   tree('y')])])
    >>> print_tree(greetings)
    h
      i
      e
        l
```

```
        l
          o
       y
    >>> has_path(greetings, 'h')
    True
    >>> has_path(greetings, 'i')
    False
    >>> has_path(greetings, 'hi')
    True
    >>> has_path(greetings, 'hello')
    True
    >>> has_path(greetings, 'hey')
    True
    >>> has_path(greetings, 'bye')
    False
    >>> has_path(greetings, 'hint')
    False
    """
    assert len(word) > 0, 'no path for empty word.'
    if word[:1] != label(t):
        return False
    elif len(word) == 1:
        return True
    else:
        for b in branches(t):
            if has_path(b, word[1:]):
                return True
        return False


HW_SOURCE_FILE = __file__


def mobile(left, right):
    """Construct a mobile from a left arm and a right arm."""
    assert is_arm(left), "left must be a arm"
    assert is_arm(right), "right must be a arm"
    return ['mobile', left, right]


def is_mobile(m):
    """Return whether m is a mobile."""
    return type(m) == list and len(m) == 3 and m[0] == 'mobile'


def left(m):
    """Select the left arm of a mobile."""
    assert is_mobile(m), "must call left on a mobile"
    return m[1]


def right(m):
```

```
    """Select the right arm of a mobile."""
    assert is_mobile(m), "must call right on a mobile"
    return m[2]


def arm(length, mobile_or_planet):
    """Construct a arm: a length of rod with a mobile or planet at the end."""
    assert is_mobile(mobile_or_planet) or is_planet(mobile_or_planet)
    return ['arm', length, mobile_or_planet]


def is_arm(s):
    """Return whether s is a arm."""
    return type(s) == list and len(s) == 3 and s[0] == 'arm'


def length(s):
    """Select the length of a arm."""
    assert is_arm(s), "must call length on a arm"
    return s[1]


def end(s):
    """Select the mobile or planet hanging at the end of a arm."""
    assert is_arm(s), "must call end on a arm"
    return s[2]


def planet(mass):
    """Construct a planet of some mass."""
    assert mass > 0
    "*** YOUR CODE HERE ***"


def mass(w):
    """Select the mass of a planet."""
    assert is_planet(w), 'must call mass on a planet'
    "*** YOUR CODE HERE ***"


def is_planet(w):
    """Whether w is a planet."""
    return type(w) == list and len(w) == 2 and w[0] == 'planet'


def examples():
    t = mobile(arm(1, planet(2)),
               arm(2, planet(1)))
    u = mobile(arm(5, planet(1)),
               arm(1, mobile(arm(2, planet(3)),
                             arm(3, planet(2)))))
    v = mobile(arm(4, t), arm(2, u))
```

```
    return t, u, v


def total_weight(m):
    """Return the total weight of m, a planet or mobile.

    >>> t, u, v = examples()
    >>> total_weight(t)
    3
    >>> total_weight(u)
    6
    >>> total_weight(v)
    9
    """
    if is_planet(m):
        return mass(m)
    else:
        assert is_mobile(m), "must get total weight of a mobile or a planet"
        return total_weight(end(left(m))) + total_weight(end(right(m)))


def balanced(m):
    """Return whether m is balanced.

    >>> t, u, v = examples()
    >>> balanced(t)
    True
    >>> balanced(v)
    True
    >>> w = mobile(arm(3, t), arm(2, u))
    >>> balanced(w)
    False
    >>> balanced(mobile(arm(1, v), arm(1, w)))
    False
    >>> balanced(mobile(arm(1, w), arm(1, v)))
    False
    >>> from construct_check import check
    >>> # checking for abstraction barrier violations by banning indexing
    >>> check(HW_SOURCE_FILE, 'balanced', ['Index'])
    True
    """
    "*** YOUR CODE HERE ***"


def totals_tree(m):
    """Return a tree representing the mobile with its total weight at the root.

    >>> t, u, v = examples()
    >>> print_tree(totals_tree(t))
    3
      2
      1
```

```
    >>> print_tree(totals_tree(u))
    6
      1
      5
        3
        2
    >>> print_tree(totals_tree(v))
    9
      3
        2
        1
      6
        1
        5
          3
          2
    >>> from construct_check import check
    >>> # checking for abstraction barrier violations by banning indexing
    >>> check(HW_SOURCE_FILE, 'totals_tree', ['Index'])
    True
    """
    "*** YOUR CODE HERE ***"


# Tree ADT

def tree(label, branches=[]):
    """Construct a tree with the given label value and a list of branches."""
    for branch in branches:
        assert is_tree(branch), 'branches must be trees'
    return [label] + list(branches)


def label(tree):
    """Return the label value of a tree."""
    return tree[0]


def branches(tree):
    """Return the list of branches of the given tree."""
    return tree[1:]


def is_tree(tree):
    """Returns True if the given tree is a tree, and False otherwise."""
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True

```

```
258
259 def is_leaf(tree):
260     """Returns True if the given tree's list of branches is empty, and False
261     otherwise.
262     """
263     return not branches(tree)
264
265
266 def print_tree(t, indent=0):
267     """Print a representation of this tree in which each node is
268     indented by two spaces times its depth from the root.
269
270     >>> print_tree(tree(1))
271     1
272     >>> print_tree(tree(1, [tree(2)]))
273     1
274       2
275     >>> numbers = tree(1, [tree(2), tree(3, [tree(4), tree(5)]), tree(6, [tree(7)])])
276     >>> print_tree(numbers)
277     1
278       2
279       3
280         4
281         5
282       6
283         7
284     """
285     print('  ' * indent + str(label(t)))
286     for b in branches(t):
287         print_tree(b, indent + 1)
288
289
290 def copy_tree(t):
291     """Returns a copy of t. Only for testing purposes.
292
293     >>> t = tree(5)
294     >>> copy = copy_tree(t)
295     >>> t = tree(6)
296     >>> print_tree(copy)
297     5
298     """
299     return tree(label(t), [copy_tree(b) for b in branches(t)])
300
```