Homework 4 Solutions **hw04.zip (hw04.zip)**

Solution Files

You can find the solutions in hw04.py (hw04.py).

Required Questions

Sequences

Q1: Shuffle

Implement `shuffle`, which takes a sequence `s` (such as a list or range) with an even number of elements. It returns a new list that *interleaves* the elements of the first half of `s` with the elements of the second half. It does not modify `s`.

To *interleave* two sequences `s0` and `s1` is to create a new list containing the first element of `s0`, the first element of `s1`, the second element of `s0`, the second element of `s1`, and so on. For example, if `s0 = [1, 2, 3]` and `s1 = [4, 5, 6]`, then interleaving `s0` and `s1` would result in `[1, 4, 2, 5, 3, 6]`.

```
def shuffle(s):
    """Return a shuffled list that interleaves the two halves of s.

    >>> shuffle(range(6))
    [0, 3, 1, 4, 2, 5]
    >>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
    >>> shuffle(letters)
    ['a', 'e', 'b', 'f', 'c', 'g', 'd', 'h']
    >>> shuffle(shuffle(letters))
    ['a', 'c', 'e', 'g', 'b', 'd', 'f', 'h']
    >>> letters  # Original list should not be modified
    ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
    """
    assert len(s) % 2 == 0, 'len(seq) must be even'
    half = len(s) // 2
    shuffled = []
    for i in range(half):
        shuffled.append(s[i])
        shuffled.append(s[half + i])
    return shuffled
```

Use Ok to test your code:

```
python3 ok -q shuffle
```

Q2: Deep Map

Definition: A *nested list of numbers* is a list that contains numbers and lists. It may contain only numbers, only lists, or a mixture of both. The lists must also be *nested lists of numbers*. For example: `[1, [2, [3]], 4]`, `[1, 2, 3]`, and `[[1, 2], [3, 4]]` are all *nested lists of numbers*.

Write a function `deep_map` that takes two arguments: a nested list of numbers `s` and a one-argument function `f`. It modifies `s` **in place** by applying `f` to each number within `s` and replacing the number with the result of calling `f` on that number.

`deep_map` returns `None` and should not create any new lists.

> **Hint:** `type(a) == list` will evaluate to `True` if `a` is a list.

```
def deep_map(f, s):
    """Replace all non-list elements x with f(x) in the nested list s.

    >>> six = [1, 2, [3, [4], 5], 6]
    >>> deep_map(lambda x: x * x, six)
    >>> six
    [1, 4, [9, [16], 25], 36]
    >>> # Check that you're not making new lists
    >>> s = [3, [1, [4, [1]]]]
    >>> s1 = s[1]
    >>> s2 = s1[1]
    >>> s3 = s2[1]
    >>> deep_map(lambda x: x + 1, s)
    >>> s
    [4, [2, [5, [2]]]]
    >>> s1 is s[1]
    True
    >>> s2 is s1[1]
    True
    >>> s3 is s2[1]
    True
    """
    for i in range(len(s)):
        if type(s[i]) == list:
            deep_map(f, s[i])
        else:
            s[i] = f(s[i])
```

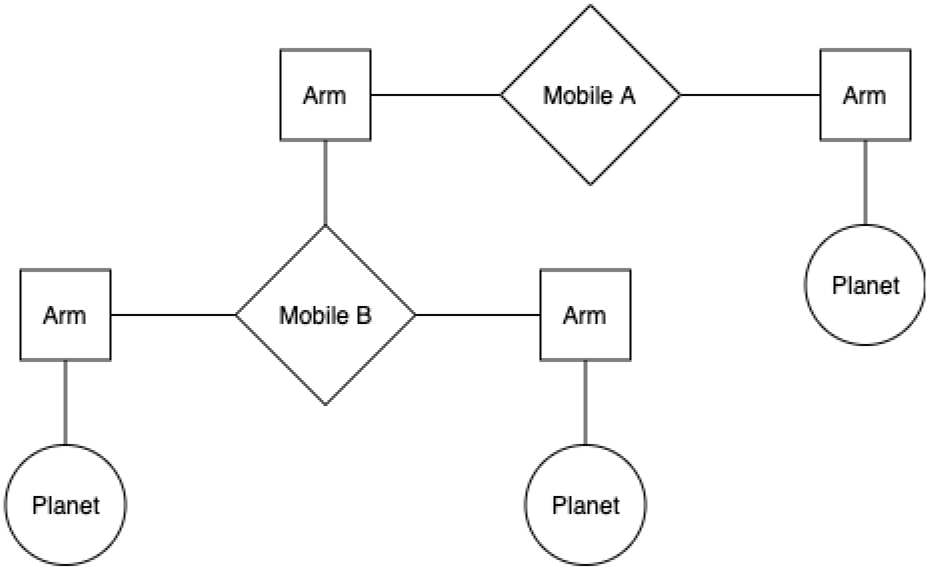
Use Ok to test your code:

```
python3 ok -q deep_map
```

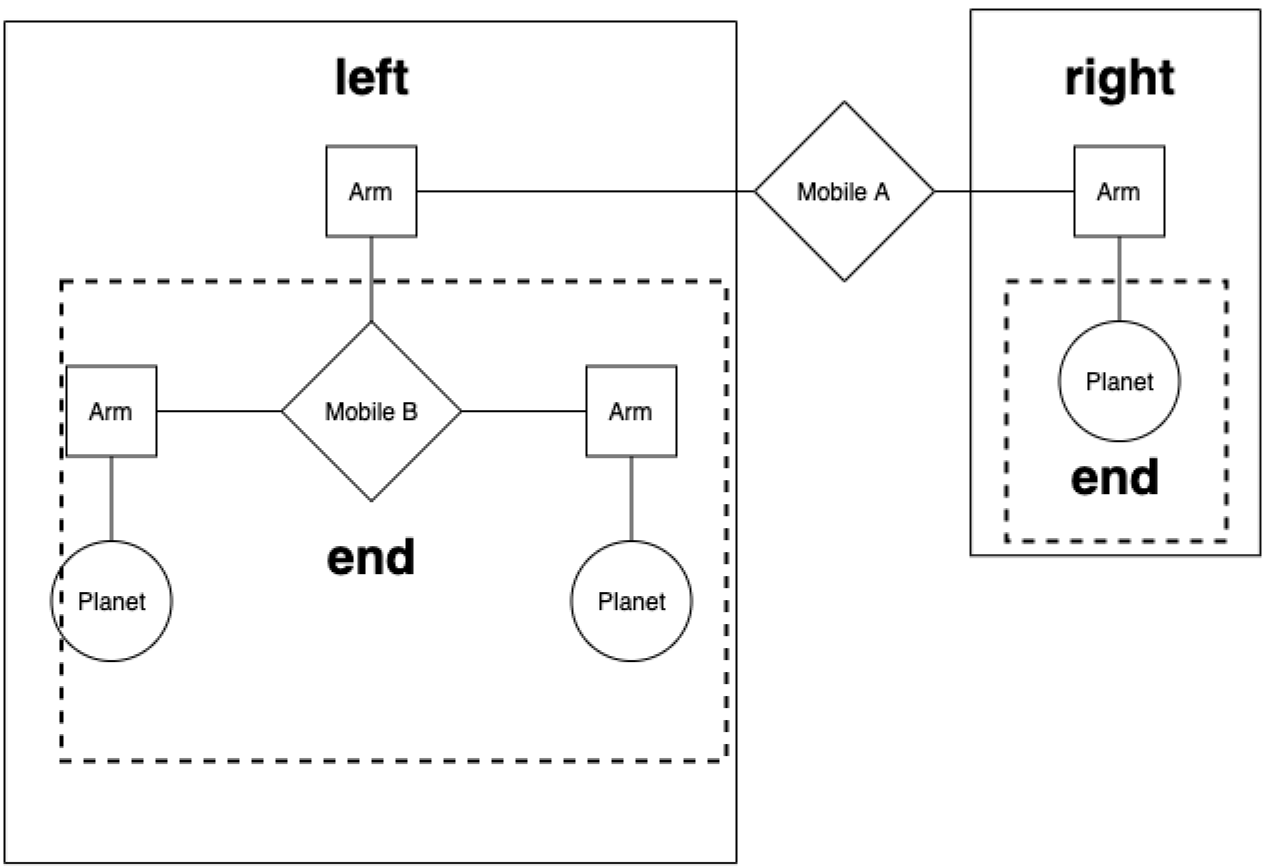
Data Abstraction

Mobiles

This problem is based on one from Structure and Interpretation of Computer Programs Section 2.2.2 (https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/full-text/book/book-Z-H-15.html#%_sec_2.2.2).



We are making a planetarium mobile. A [mobile (https://www.northwestnatureshop.com/wp-content/uploads/2015/04/AMSolarSystem.jpg)](https://www.northwestnatureshop.com/wp-content/uploads/2015/04/AMSolarSystem.jpg) is a type of hanging sculpture. A binary mobile consists of two arms. Each arm is a rod of a certain length, from which hangs either a planet or another mobile. For example, the below diagram shows the left and right arms of Mobile A, and what hangs at the ends of each of those arms.



We will represent a binary mobile using the data abstractions below.

- A `mobile` must have both a left `arm` and a right `arm`.
- An `arm` has a positive length and must have something hanging at the end, either a `mobile` or `planet`.
- A `planet` has a positive mass, and nothing hanging from it.

Below are the various constructors and selectors for the `mobile` and `arm` data abstraction. They have already been implemented for you, though the code is not shown here. As with any data abstraction, you should focus on what the function does rather than its specific implementation. You are free to use any of their constructor and selector functions in the Mobiles coding exercises.

Mobile Data Abstraction (**for your reference, no need to do anything here**):

```
def mobile(left, right):
    """
    Construct a mobile from a left arm and a right arm.

    Arguments:
        left: An arm representing the left arm of the mobile.
        right: An arm representing the right arm of the mobile.

    Returns:
        A mobile constructed from the left and right arms.
    """
    pass

def is_mobile(m):
    """
    Return whether m is a mobile.

    Arguments:
        m: An object to be checked.

    Returns:
        True if m is a mobile, False otherwise.
    """
    pass

def left(m):
    """
    Select the left arm of a mobile.

    Arguments:
        m: A mobile.

    Returns:
        The left arm of the mobile.
    """
    pass

def right(m):
    """
    Select the right arm of a mobile.

    Arguments:
        m: A mobile.

    Returns:
```

```
        The right arm of the mobile.
    """
    pass
```

Arm Data Abstraction (**for your reference, no need to do anything here**):

```
def arm(length, mobile_or_planet):
    """
    Construct an arm: a length of rod with a mobile or planet at the end.

    Arguments:
        length: The length of the rod.
        mobile_or_planet: A mobile or a planet at the end of the arm.

    Returns:
        An arm constructed from the given length and mobile or planet.
    """
    pass

def is_arm(s):
    """
    Return whether s is an arm.

    Arguments:
        s: An object to be checked.

    Returns:
        True if s is an arm, False otherwise.
    """
    pass

def length(s):
    """
    Select the length of an arm.

    Arguments:
        s: An arm.

    Returns:
        The length of the arm.
    """
    pass

def end(s):
    """
    Select the mobile or planet hanging at the end of an arm.

    Arguments:
        s: An arm.

    Returns:
```

```
        The mobile or planet at the end of the arm.
    """
    pass
```

Q3: Mass

Implement the `planet` data abstraction by completing the `planet` constructor and the `mass` selector. A planet should be represented using a two-element list where the first element is the string `'planet'` and the second element is the planet's mass. The `mass` function should return the mass of the `planet` object that is passed as a parameter.

```
def planet(mass):
    """Construct a planet of some mass."""
    assert mass > 0
    return ['planet', mass]

def mass(p):
    """Select the mass of a planet."""
    assert is_planet(p), 'must call mass on a planet'
    return p[1]

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

The `total_mass` function demonstrates the use of the mobile, arm, and planet abstractions. It has been implemented for you. *You may use the `total_mass` function in the following questions.*

```
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_mass(m):
    """Return the total mass of m, a planet or mobile.

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

Run the `ok` tests for `total_mass` to make sure that your `planet` and `mass` functions are implemented correctly.

Use Ok to test your code:

```
python3 ok -q total_mass
```

Q4: Balanced

Implement the `balanced` function, which returns whether `m` is a *balanced* mobile. A mobile is *balanced* if **both** of the following conditions are met:

1. The *torque* applied by its left arm is equal to the *torque* applied by its right arm. The *torque* of the left arm is the length of the left rod multiplied by the total mass hanging from that rod. Likewise for the right. For example, if the left arm has a length of `5`, and there is a `mobile` hanging at the end of the left arm of total mass `10`, the torque on the left side of our mobile is `50`.

2. Each of the mobiles hanging at the end of its arms is itself *balanced*.

Planets themselves are balanced, as there is nothing hanging off of them.

> **Reminder:** You may use the `total_mass` function above. **Don't violate abstraction barriers.** Instead, use the selector functions that have been defined.

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

    >>> t, u, v = examples()
    >>> balanced(t)
    True
    >>> balanced(v)
    True
    >>> p = mobile(arm(3, t), arm(2, u))
    >>> balanced(p)
    False
    >>> balanced(mobile(arm(1, v), arm(1, p)))
    False
    >>> balanced(mobile(arm(1, p), arm(1, v)))
    False
    >>> from construct_check import check
    >>> # checking for abstraction barrier violations by banning indexing
    >>> check(HW_SOURCE_FILE, 'balanced', ['Index'])
    True
    """
    if is_planet(m):
        return True
    else:
        left_end, right_end = end(left(m)), end(right(m))
        torque_left = length(left(m)) * total_mass(left_end)
        torque_right = length(right(m)) * total_mass(right_end)
        return torque_left == torque_right and balanced(left_end) and balanced(right_end)
```

Use Ok to test your code:

```
python3 ok -q balanced
```

The fact that planets are balanced is important, since we will be solving this recursively like many other tree problems (even though this is not explicitly a tree).

- **Base case:** if we are checking a planet, then we know that this is balanced. Why is this an appropriate base case? There are two possible approaches to this:

1. Because we know that our data structures so far are trees, planets are the simplest possible tree since we have chosen to implement them as leaves.
2. We also know that from a data abstraction standpoint, planets are the terminal item in a mobile. There can be no further mobile structures under this planet, so it makes sense to stop check here.

- **Otherwise:** note that it is important to do a recursive call to check if both arms are balanced. However, we also need to do the basic comparison of looking at the total mass of both arms as well as their length. For example if both arms are a planet, trivially, they will both be balanced. However, the torque must be equal in order for the entire mobile to balanced (i.e. it's insufficient to just check if the arms are balanced).

Trees

Q5: Finding Berries!

The squirrels on campus need your help! There are a lot of trees on campus and the squirrels would like to know which ones contain berries. Define the function `berry_finder`, which takes in a tree and returns `True` if the tree contains a node with the value `'berry'` and `False` otherwise.

> **Hint**: To iterate through each of the branches of a particular tree, you can consider using a `for` loop to get each branch.

```
def berry_finder(t):
    """Returns True if t contains a node with the value 'berry' and
    False otherwise.

    >>> scrat = tree('berry')
    >>> berry_finder(scrat)
    True
    >>> sproul = tree('roots', [tree('branch1', [tree('leaf'), tree('berry')]), tree('brar
    >>> berry_finder(sproul)
    True
    >>> numbers = tree(1, [tree(2), tree(3, [tree(4), tree(5)]), tree(6, [tree(7)])])
    >>> berry_finder(numbers)
    False
    >>> t = tree(1, [tree('berry',[tree('not berry')])])
    >>> berry_finder(t)
    True
    """
    if label(t) == 'berry':
        return True
    for b in branches(t):
        if berry_finder(b):
            return True
    return False

# Alternative solution
def berry_finder_alt(t):
    if label(t) == 'berry':
        return True
    return True in [berry_finder(b) for b in branches(t)]
```

Use Ok to test your code:

```
python3 ok -q berry_finder
```

Q6: Maximum Path Sum

Write a function that takes in a tree and returns the maximum sum of the values along any *root-to-leaf path* in the tree. A *root-to-leaf path* is a sequence of nodes starting at the root and proceeding to some leaf of the tree. You can assume the tree will have positive numbers for its labels.

```
def max_path_sum(t):
    """Return the maximum root-to-leaf path sum of a tree.
    >>> t = tree(1, [tree(5, [tree(1), tree(3)]), tree(10)])
    >>> max_path_sum(t) # 1, 10
    11
    >>> t2 = tree(5, [tree(4, [tree(1), tree(3)]), tree(2, [tree(10), tree(3)])])
    >>> max_path_sum(t2) # 5, 2, 10
    17
    """
    # Non-list comprehension solution
    if is_leaf(t):
        return label(t)
    highest_sum = 0
    for b in branches(t):
        highest_sum = max(max_path_sum(b), highest_sum)
    return label(t) + highest_sum

    # List comprehension solution
    if is_leaf(t):
      return label(t)
    else:
      return label(t) + max([max_path_sum(b) for b in branches(t)])
```

Use Ok to test your code:

```
python3 ok -q max_path_sum
```

Check Your Score Locally

You can locally check your score on each question of this assignment by running

```
python3 ok --score
```

This does NOT submit the assignment! When you are satisfied with your score, submit the assignment to Gradescope to receive credit for it.

Submit Assignment

Submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** Lab 00 (../../lab/lab00/#submit-with-gradescope) has detailed instructions.

Exam Practice

Homework assignments will also contain prior exam-level questions for you to take a look at. These questions have no submission component; feel free to attempt them if you'd like a challenge!

1. Summer 2021 MT Q4: Maximum Exponen-tree-ation (https://cs61a.org/exam/su21/midterm/61a-su21-midterm.pdf#page=10)
2. Summer 2019 MT Q8: Leaf It To Me (https://inst.eecs.berkeley.edu/~cs61a/sp20/exam/su19/mt/61a-su19-mt.pdf#page=9)
3. Summer 2017 MT Q9: Temmie Flakes (https://inst.eecs.berkeley.edu//~cs61a/su17/assets/pdfs/61a-su17-mt.pdf#page=11)