Lab 04: Data Abstraction, Trees, and Mutable Values

1. Instructions

> Please download lab materials `lab04.zip` from our QQ group if you don't have one.

In this lab, you have one task:

- Complete the required problems described in section 3 and submit your code to our OJ website as instructed in lab00. The starter code for these problems is provided in `lab04.py`, which is distributed as part of the lab materials in the `code` directory.

Submission: As instructed above, you just need to submit your answer for problems described in section 3 to our OJ website. You may submit more than once before the deadline; only the final submission will be scored. See lab00 for more instructions on submitting assignments.

Readings: You might find the following reference to the textbook useful:

- Section 2.2
- Section 2.3
- Section 2.4

2. Review

2.1 Mutability

We say that an object is **mutable** if its state can change as code is executed. The process of changing an object's state is called **mutation**. Examples of mutable objects include lists and dictionaries. Examples of objects that are not mutable include tuples and functions.

We have seen how to use the `==` operator to check if two expressions evaluate to *equal* values. We now introduce a new comparison operator, `is`, that checks whether two expressions evaluate to the *same* values.

Wait, what's the difference? For primitive values, there is none:

```
>>> 2 + 2 == 3 + 1
True
>>> 2 + 2 is 3 + 1
True
```

This is because all primitives have the same *identity* under the hood. However, with non-primitive values, such as lists and large numbers, each object has its own identity. That means you can construct two objects that may look exactly the same but have different identities.

```
>>> large_num1 = 23333333333333333333
>>> large_num2 = 23333333333333333333
>>> large_num1 == large_num2
True
>>> large_num1 is large_num2
False
>>> lst1 = [1, 2, 3, 4]
>>> lst2 = [1, 2, 3, 4]
>>> lst1 == lst2
True
>>> lst1 is lst2
False
```

Here, although the lists referred to by `lst1` and `lst2` have *equal* contents, they are not the *same* object. In other words, they are the same in terms of equality, but not in terms of identity.

This is important in our discussion of mutability because when we mutate an object, we simply change its state, not its identity.

```
>>> lst1 = [1, 2, 3, 4]
>>> lst2 = lst1
>>> lst1.append(5)
>>> lst2
[1, 2, 3, 4, 5]
>>> lst1 is lst2
True
```

> You can think of the **name** in Python as the **pointer variable** in the C language, and the **identity** of an object in Python as the **address** of an object in the C language. In such an analogy,
>
> - assigning an object to a name is similar to assigning the address of this object to a pointer variable,
> - the `==` operator compares whether the two **pointed values** are equal,
> - and the `is` operator compares whether the two **pointer** are equal.

You can use the built-in function `id` to fetch the identity of an object, which differs during different runnings of the Python interpreter.
In fact, the expression `a is b` is equivalent to `id(a) == id(b)`.

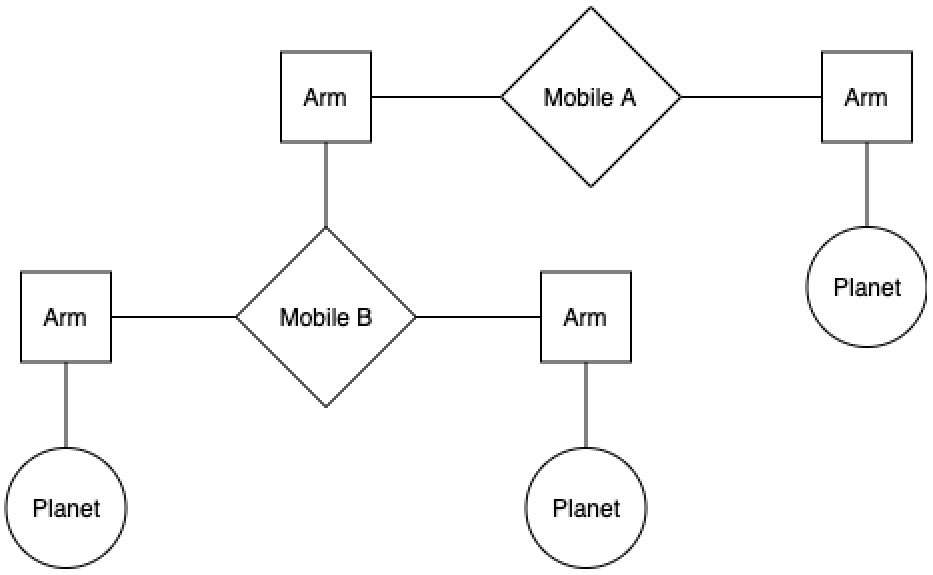
```
>>> lst = [1, 2, 3, 4]
>>> id(lst)
2624875298056 # It's different on your machine
>>> lst.append(5)
>>> id(lst)
2624875298056
```

3. Required Problems

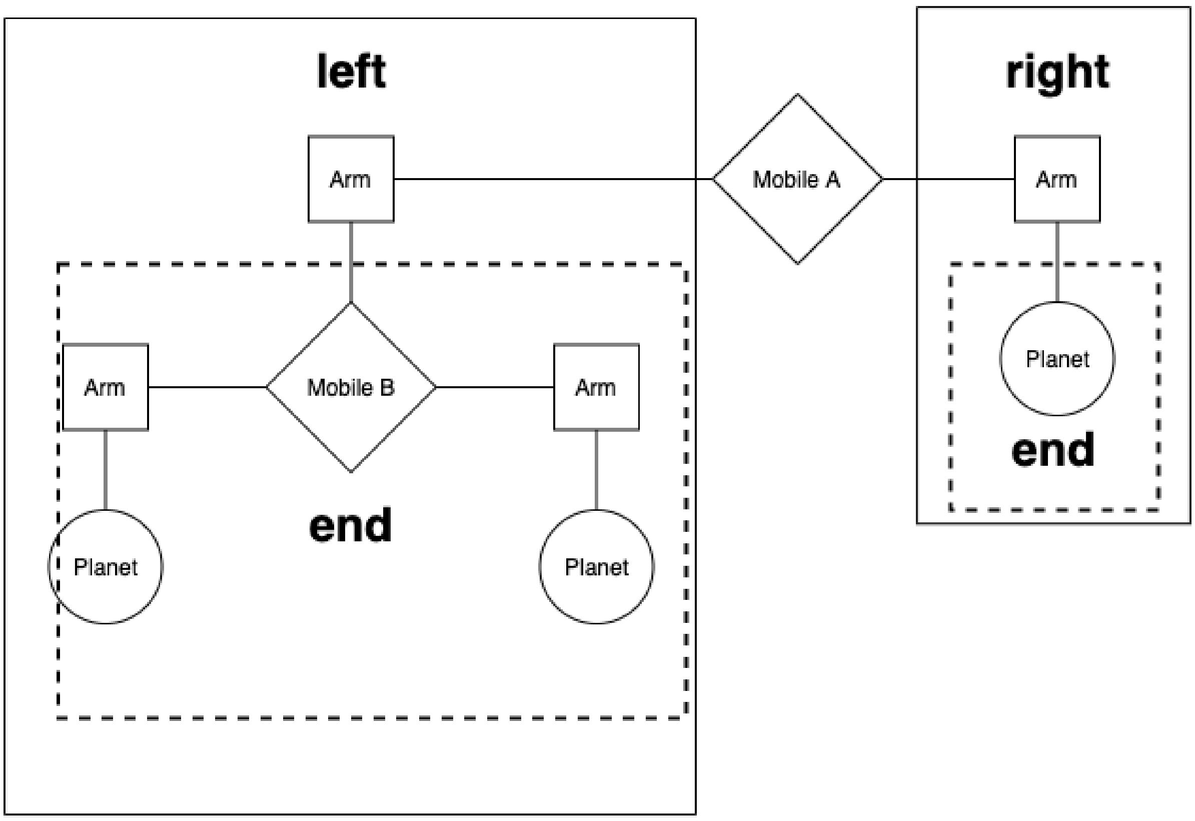
In this section, you are required to complete the problems below and submit your code to `Contest lab04` in our OJ website as instructed in lab00 to get your answer scored.

Problem 1: Mobiles (100+100+200pts)

Acknowledgements. This mobile example is based on a classic problem from Structure and Interpretation of Computer Programs, Section 2.2.2.



We are making a planetarium mobile. A mobile 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.



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

- A `mobile` has a left `arm` and a right `arm`.
- An `arm` has a positive length and something hanging at the end, either a `mobile` or `planet`.
- A `planet` has a positive size.

Problem 1.1 Weights

Implement the `planet` data abstraction by completing the `planet` constructor and the `size` selector so that a planet is represented using a two-element list where the first element is the string `'planet'` and the second element is its `size`. The `total_weight` example is provided to demonstrate use of the mobile, arm, and planet abstractions.

```python
# The constructor and selectors of the mobile

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]
```

```python
# The constructor and selectors of the arm

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]
```

```python
# The constructor and selectors of the planet

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

def size(w):
    """Select the size of a planet."""
    assert is_planet(w), 'must call size 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 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 size(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)))
```

Problem 1.2: Balanced

> Hint: for more information on this problem (with more pictures!), please refer to [this document](#).

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

1. The torque applied by its left arm is equal to that applied by its right arm. Torque of the left arm is the length of the left rod multiplied by the total weight hanging from that rod. Likewise for the right.
2. Each of the mobiles hanging at the end of its arms is balanced.

Planets themselves are balanced.

```
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
    """
    "*** YOUR CODE HERE ***"
```

Problem 1.3 Mobile to Tree

Implement `totals_tree`, which takes a `mobile` (or `planet`) and returns a `tree` whose root is the total weight of the input. For a `planet`, the result should be a leaf. For a `mobile`, the result's branches should be `totals_trees` for the ends of its arms.

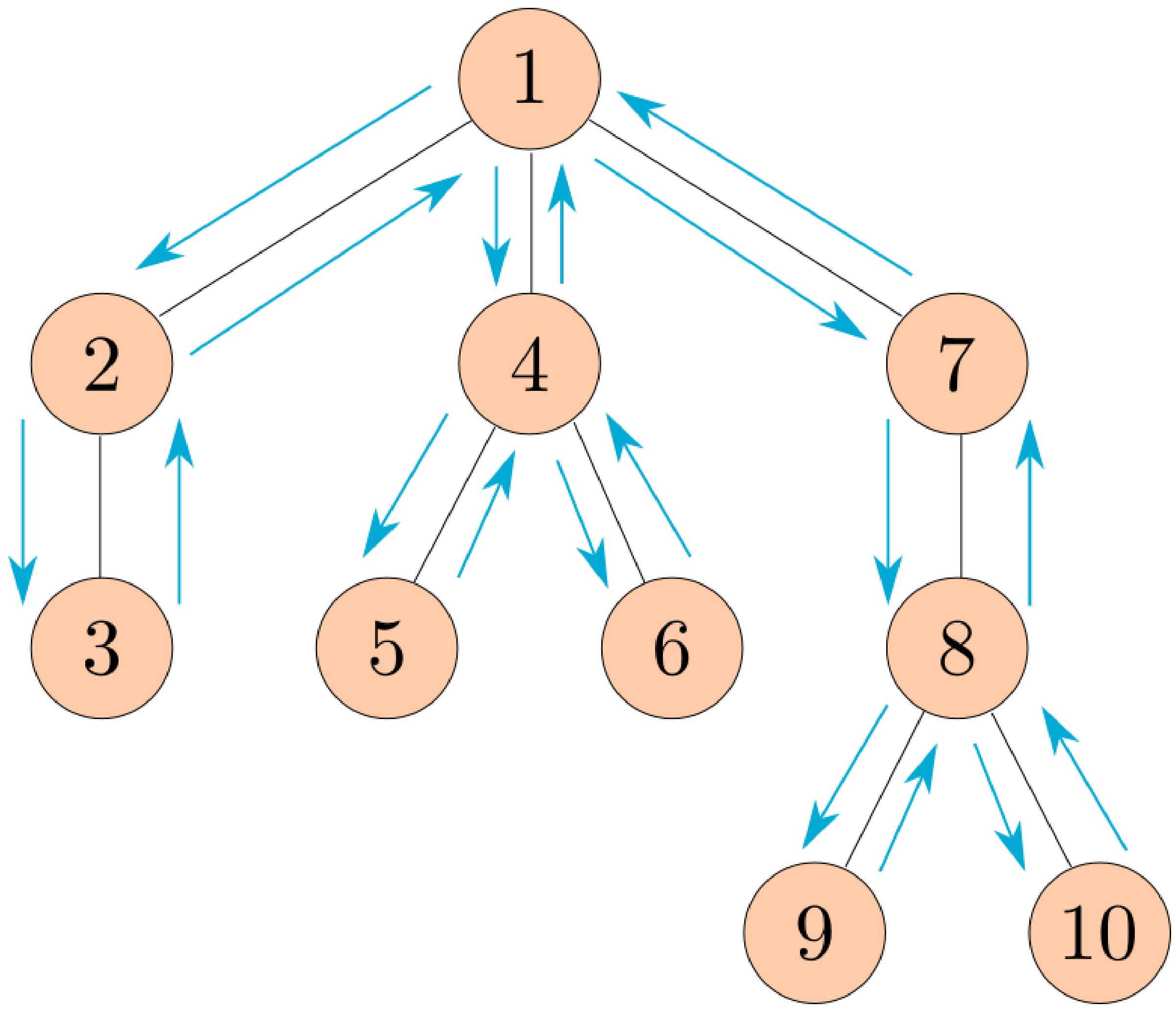
```
def totals_tree(m):
    """Return a tree representing the mobile with its total weight at the root.

    >>> 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
    """
    "*** YOUR CODE HERE ***"
```

Problem 2 Preorder (100pts)

Define the function `preorder`, which takes in a tree as an argument and returns a list of all the entries in the tree in the order that `print_tree` would print them.

The following diagram shows the order that the nodes would get printed, with the arrows representing function calls.



> *Note*: This ordering of the nodes in a tree is called a preorder traversal.

```
def preorder(t):
    """Return a list of the entries in this tree in the order that they
    would be visited by a preorder traversal (see problem description).

    >>> numbers = tree(1, [tree(2), tree(3, [tree(4), tree(5)]), tree(6,
[tree(7)])])
    >>> preorder(numbers)
    [1, 2, 3, 4, 5, 6, 7]
    >>> preorder(tree(2, [tree(4, [tree(6)])]))
    [2, 4, 6]
    """
    "*** YOUR CODE HERE ***"
```

Problem 3: Trie (200pts)

Write a function `has_path` that takes in a tree `t` and a string `word`. It returns `True` if there is a path that starts from the root where the entries along the path spell out the `word`, and `False` otherwise. (This data structure is called a trie, and it has a lot of cool applications!---think autocomplete). You may assume that every node's `label` is exactly one character.

```
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
    """
    assert len(word) > 0, 'no path for empty word.'
    "*** YOUR CODE HERE ***"
```

Problem 4: Insert (100pts)

Write a function which takes in a list `lst`, an argument `entry`, and another argument `elem`. This function will check through each item present in `lst` to see if it is equivalent with `entry`. Upon finding an equivalent entry, the function should modify the list by placing `elem` into the list right after the found entry. At the end of the function, the modified list should be returned. See the doctests for examples on how this function is utilized. Use list mutation to modify the original list, no new lists should be created or returned.

Be careful in situations where the values passed into `entry` and `elem` are equivalent, so as not to create an infinitely long list while iterating through it. If you find that your code is taking more than a few seconds to run, it is most likely that the function is in a loop of inserting new values.

> *Hint*: You may use the `lst.insert(ind, obj)` to insert an element `obj` to a position indexed by `ind`. Search the website for more information about its usage.

```
def insert_items(lst, entry, elem):
    """
    >>> test_lst = [1, 5, 8, 5, 2, 3]
    >>> new_lst = insert_items(test_lst, 5, 7)
    >>> new_lst
    [1, 5, 7, 8, 5, 7, 2, 3]
    >>> large_lst = [1, 4, 8]
    >>> large_lst2 = insert_items(large_lst, 4, 4)
    >>> large_lst2
    [1, 4, 4, 8]
    >>> large_lst3 = insert_items(large_lst2, 4, 6)
    >>> large_lst3
    [1, 4, 6, 4, 6, 8]
    >>> large_lst3 is large_lst
    True
    """
    "*** YOUR CODE HERE ***"
```