CS61A NOTE4 Trees,Data Abstraction

Data Abstraction

Data abstraction is a powerful concept in computer science that allows programmers to treat code as objects. 数据抽象允许程序员把代码当作对象。

Data abstraction mimics how we think about the world.数据抽象模仿了我们对世界的思考方式 If you want to drive a car, you don't need to know how the engine was built or what kind of material the tires are made of to do so. You just have to know how to use the car for driving itself, such as how to turn the wheel or press the gas pedal.

A data abstraction consists of two types of functions:数据抽象包含两种类型函数

- **Constructors**: functions that build the abstract data type.构造器：构建抽象数据类型的函数
- **Selectors**: functions that retrieve information from the data type.选择器：从数据类型中检索信息的函数。

Programmers design data abstractions to abstract away how information is stored and calculated such that the end user does *not* need to know how constructors and selectors are implemented. 数据抽象为抽象出信息的存储和计算方式，使得用户不需要知道 constructors 和 selectors 如何实现。The nature of *abstraction* allows whoever uses them to assume that the functions have been written correctly and work as described.抽象使用户假定函数编写正确，并按照描述工作。

eg:

Say we have an abstract data type for cities. A city has a name, a latitude coordinate, and a longitude coordinate.

data abstraction has one **constructor**:

- `make_city(name, lat, lon)`: Creates a city object with the given name, latitude, and longitude.

selectors in order to get the information for each city:

- `get_name(city)`: Returns the city's name
- `get_lat(city)`: Returns the city's latitude
- `get_lon(city)`: Returns the city's longitude

```
纯文本
>>> berkeley = make_city('Berkeley', 122, 37)
>>> get_name(berkeley)
'Berkeley' #带引号
>>> get_lat(berkeley)
122
>>> new_york = make_city('New York City', 74, 40)
>>> get_lon(new_york)
40
```

All of the selector and constructor functions can be found in the lab file.The point of data abstraction is that we do not need to know how an abstract data type is implemented, but rather just how we can interact with and use the data type.

Lab 05 Distance,Closer city

```
纯文本
#distance
def distance(city_a, city_b):
    return sqrt((get_lat(city_a)-get_lat(city_b))**2+(get_lon(city_a)-get_l

#takes a latitude, longitude, and two cities, and returns the name of close
def closer_city(lat, lon, city_a, city_b):
    if (lat - get_lat(city_a)) ** 2+(lon - get_lon(city_a)) ** 2 <(lat - ge
        return get_name(city_a) #注意返回name
    else:
        return get_name(city_b)
```

hw4 Arms-length recursion 臂长递归

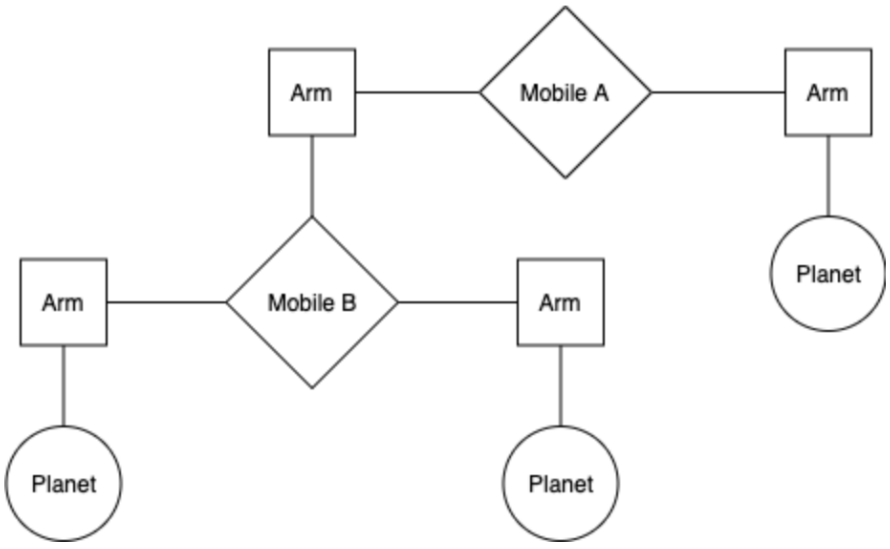
Return the number of steps in the shortest path to a leaf in tree t.到叶子的距离，避免多余

纯文本

```
def min_depth(t):
    if is_leaf(t):
        return 0
    for b in branches(t):  # CAN BE REMOVED
        if is_leaf(b):     # CAN BE REMOVED
            return 1       # CAN BE REMOVED
    return 1 + min([min_depth(b) for b in branches(t)])
```

hw4 Q1 Mobiles

We are making a planetarium mobile 天文装置. A mobile is a type of hanging sculpture 悬挂雕塑. A binary mobile 二进制装置 consists of two arms 两臂. Each arm is a rod of a certain length, from which hangs either a planet or another mobile 一定长度的杆挂着行星或另一个装置. 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.



represent a binary mobile using the data abstractions below.二进制装置

- A `mobile` must have both a left `arm` and a right `arm`.类似树

- An `arm` has a positive length and must have something hanging at the end, either a `mobile` or `planet`. 手臂正长度 + 悬挂东西
- A `planet` has a positive mass, and nothing hanging from it.行星正质量且不挂东西，类似 leaf

Below are the implementations of the various data abstractions used in mobiles. The `mobile` and `arm` data abstractions have been completed for you.

纯文本

```
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] #提示planet构造

def is_mobile(m): #是不是装置
    """Return whether m is a mobile."""#提示后面
    return type(m) == list and len(m) == 3 and m[0] == 'mobile'

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

def right(m): #选右
    """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): #建立手臂，输入长度和装置/行星
    """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] #提示

def is_arm(s): #是不是手臂
    """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] #提示
```

```
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]
```

hw4 Q2: Weights

Implement the `planet` data abstraction 行星数据抽象 by completing the `planet` constructor 创造器 and the `mass` selector 属性选择器 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 mass.给了 planet 创造器的提示，有'planet'和 mass

纯文本

```
def planet(mass):
    """Construct a planet of some mass."""
    assert mass > 0
    return ['planet', mass]
def mass(w):
    """Select the mass of a planet."""
    assert is_planet(w), 'must call mass on a planet'
    "*** YOUR CODE HERE ***"
    return w[1]
def is_planet(w):
    """Whether w is a planet."""
    return type(w) == list and len(w) == 2 and w[0] == 'planet'
```

纯文本

```
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.
```

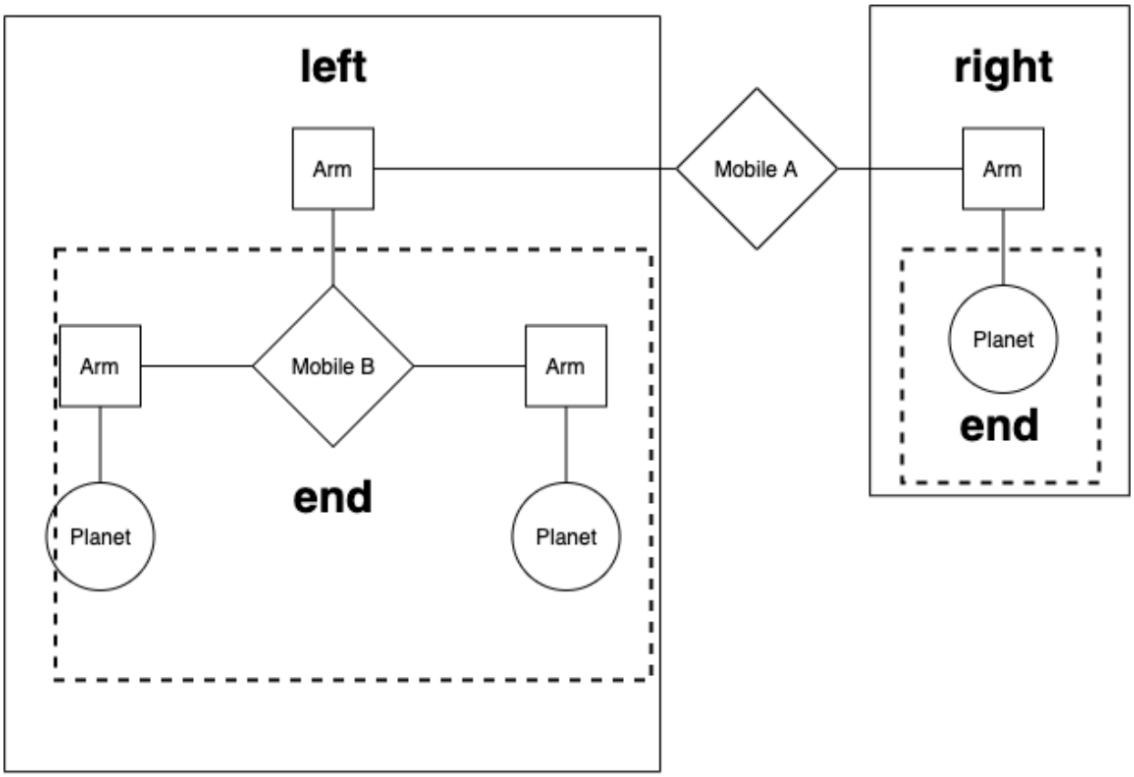
```
    >>> t, u, v = examples()
    >>> total_weight(t)
    3
    >>> total_weight(u)
    6
    >>> total_weight(v)
    9
    """
    if is_planet(m):
        return mass(m)
    else:
        assert is_mobile(m), "must get total weight of a mobile or a planet
        return total_weight(end(left(m)))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

hw4 Q3:Balanced 要全部平衡

Implement the `balanced` function, which returns whether `m` is a balanced mobile. A mobile is balanced if both of the following conditions are met:返回是否平衡

1. The torque 矩 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 weight hanging from that rod 长度乘质量. 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 weight `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.行星平衡




```
def balanced(m):
    if is_planet(m):
        return True
    else:
        torque_left=total_weight(end(left(m)))*length(left(m))
        torque_right=total_weight(end(right(m)))*length(right(m))
        return torque_left==torque_right and balanced(end(left(m))) and bal
```

hw4 Q4 Totals 变成树

Implement totals_tree, which takes in a mobile or planet and returns a tree whose root label is the total weight of the input. For a planet, totals_tree should return a leaf. For a mobile, totals_tree should return a tree whose label is that mobile's total weight, and whose branches are totals_trees for the ends of its arms. As a reminder, the description of the tree data abstraction can be found here.实现 totals_tree，它输入 mobil 或 planet，返回 tree，根标签是输入总重量。若是星球 totals_tree 应该返回叶子。对于 mobile，totals_tree 返回一棵树，label 是该移动物体的总重量，其分支是其手臂末端的 totals_tree。

```
def totals_tree(m):
    if is_planet(m):
```

```
        return tree(mass(m))
    else:
        branches=[totals_tree(end(i)) for i in [left(m),right(m)]]
        #return tree(sum([label(b) for b in branches]),branches)
        return tree(label(totals_tree(end(left(m))))+label(totals_tree(end(
        #end是下方整体，需要变成totals_tree后取label相加
        #branches用到递归思想，将左边右边变成两个树再放入[branches]
```

hw04 Q7 Interval Abstraction

"interval" that has two endpoints: a lower bound and an upper bound,given the endpoints of an interval, she can create the interval using data abstraction.Using this constructor and the appropriate selectors, she defines the following operations:

纯文本

```
def str_interval(x): #constructor
    """Return a string representation of interval x."""
    return '{0} to {1}'.format(lower_bound(x), upper_bound(x))

def add_interval(x, y):
    """Return an interval that contains the sum of any value in interval x
    any value in interval y."""
    lower = lower_bound(x) + lower_bound(y)
    upper = upper_bound(x) + upper_bound(y)
    return interval(lower, upper)
```

纯文本

```
def interval(a, b):
    """Construct an interval from a to b."""
    assert a <= b, 'Lower bound cannot be greater than upper bound'
    return [a, b]
def lower_bound(x):
    """Return the lower bound of interval x."""
    "*** YOUR CODE HERE ***"
    return x[0]
def upper_bound(x):
    """Return the upper bound of interval x."""
    "*** YOUR CODE HERE ***"
    return x[1]
```

hw04 Q8 interval Arithmetic

纯文本

```
def mul_interval(x, y):
    """Return the interval that contains the product of any value in x and
    value in y."""
    p1 = lower_bound(x) * lower_bound(y)
    p2 = lower_bound(x) * upper_bound(y)
    p3 = upper_bound(x) * lower_bound(y)
    p4 = upper_bound(x) * upper_bound(y)
    return [min(p1, p2, p3, p4), max(p1, p2, p3, p4)]

def sub_interval(x, y):
    """Return the interval that contains the difference between any value i
    and any value in y."""
    "*** YOUR CODE HERE ***"
    return add_interval(x, interval(-upper_bound(y),-lower_bound(y)))
    #Lower bound cannot be greater than upper bound

def div_interval(x, y):
    """Return the interval that contains the quotient of any value in x div
    any value in y. Division is implemented as the multiplication of x by t
    reciprocal of y."""
    "*** YOUR CODE HERE ***" #题目加assert防止interval spans zero
    assert not (lower_bound(y)<=0<=upper_bound(y))
    reciprocal_y = interval(1 / upper_bound(y), 1 / lower_bound(y))
    return mul_interval(x, reciprocal_y)
```

小标题

Trees: Data Abstractiona 的特例

In computer science, **trees** are recursive data structures that are widely used in various settings and can be implemented in many ways. 树是一种递归数据结构

- **Parent Node**: A node that has at least one branch.父节点，至少有一个分支
- **Child Node**: A node that has a parent. A child node can only have one parent.
- **Root**: The top node of the tree.顶部节点

- **Label**: The value at a node. In our example, every node's label is an integer.节点的标签
- **Leaf**: A node that has no branches. 没有分支的最下面节点
- **Branch**: A subtree of the root. Trees have branches, which are trees themselves: this is why trees are *recursive* data structures.根的子树
- **Depth**: How far away a node is from the root. We define this as the number of edges between the root to the node. As there are no edges between the root and itself, the root has depth 0.节点离根有多远
- **Height**: The depth of the lowest (furthest from the root) leaf.最低（离根最远）叶子的深度

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 a data abstraction, our choice to use lists is just an implementation detail.树有 root 的 value 和一列 branches(也是树)用 list 实现树(因为数是数据抽象)

- The arguments to the constructor `tree` are the value for the root node and an optional list of branches. *If no branches parameter is provided, the default value* `[]` *is used.*构造函数 tree 的参数是根节点的值和一个可选的分支列表。如果没有提供分支参数，将使用默认值[]。
- The selectors for these are `label` and `branches`.选择器是 label 和 branches 选择器是标签和分支

Remember 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.分支返回的事数的链表而不是树，树与树的链表不同

```
纯文本
def tree(label, branches=[]): 通过给定的label value和branches链表构造树
    """Construct a tree with the given label value and a list of branches."
    return [label] + list(branches)

def label(tree):返回root的标签，也是树的标签
    """Return the label value of a tree."""
    return tree[0]

def branches(tree):返回的是branches的list
```

```
    """Return the list of branches of the given tree."""
    return tree[1:]

def is_leaf(tree): #常用，判断是否为树
    """Returns True if the given tree's list of branches is empty, and Fals
    otherwise.
    """
    return not branches(tree)
```

disc Q2:Height

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.用 recursion 写 height

纯文本

```
def height(t):#用recursion写height，会用到is_lea和branches和列表表达
   if is_leaf(t): #自己到自己返回0
        return 0
    return 1 + max([height(i) for i in branches(t)])
```

disc Q3:Maximum Path Sum

Write a function that takes in a tree and returns the maximum sum of the values along any path in the tree. Recall that a path is from the tree's root to any leaf.

纯文本

```
def max_path_sum(t): #树的递归都是从上往下
    if is_leaf(t):
        return label(t)
    else:
        return label(t)+max([max_path_sum(i) for i in branches(t)])
```

disc Q4:Find Path

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 None. Assume that the entries of the tree are unique.输入树和一个值返回从根到节点 x 的节点 list，节点数字唯一，若 x 不存在返回 None

```
def find_path(t, x):
    if label(t)==x: #都要考虑一开始的情况
        return [label(t)] #注意别忘了[]是返回一个[]
    for i in branches(t): #不再是之前简单的max而是要一个一个branches找
        if find_path(i,x):
            return [label(t)]+find_path(i,x) #可以[]+[]，不加的话成数字求和了
    #不用在这里return None，因为for循环是有限步，没找到就直接return None
```

disc Q5:Perfectly Balanced

Part A: Implement `sum_tree`, which returns the sum of all the labels in tree `t`.

Part B: Implement `balanced`, which returns whether every branch of `t` has the same total sum and that the branches themselves are also balanced.每个 branches 上的求和是否相同，不仅是最上的 branches

Challenge: Solve both of these parts with just 1 line of code each.一行代码

```
def sum_tree(t):
    total = 0
    for b in branches(t): #branches可能是几个子树
        total += sum_tree(b)
    return label(t) + total #别忘了加label(t)

def sum_tree(t):
    return label(t)+sum([sum_tree(i) for i in branches(t)])
```

```
def balanced(t):
    if is_leaf(t):
        return True
    sum=sum_tree(branches(t)[0]) #这里取了第一个branch的sum值，毕竟两两比较太麻烦
    for i in branches(t):
```

```
        if sum_tree(i) != sum or balanced(i): #不仅根sum比，还要递归子branch是
            return False
    return True #这里的return注意缩进，如果前面一直不return跳出则为真

def balanced(t): #一行很有挑战
    return False not in [balanced(i) and sum_tree(i)==sum_tree(branches(t)[
#我自己没想到False not in，如果只写后边[]，是一对奇怪的False/True，只有后面没有一个F
```

disc Q6: Sprout Leaves 发芽

Define a function `sprout_leaves` that takes in a tree t, and a list of leaves. It produces a new tree that is identical to t, but where each old leaf node has new branches, one for each leaf in leaves.输入树 t 和叶子 list，产生一个相同的新树，每个旧叶子有新枝——叶子 list

纯文本

```
def sprout_leaves(t, leaves):
    if is_leaf(t): #一开始就是叶子就不用递归，易错：返回两个指标的树，branches是一堆
        return tree(label(t),[tree(i) for i in leaves])
    return tree(label(t),[sprout_leaves(j,leaves) for j in branches(t)])
这里易错tree的表达，每次向下一行删一个branch的[],又是一个或多个新的tree( ,[])
```

disc Q7: Hailstone Tree 冰雹树，略难

We can represent the hailstone sequence as a tree in the figure below, showing the route different numbers take to reach 1. 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.产生一个高 h，到达 n（最上 label 是 n）的冰雹树，第一步构造冰雹树，第二步 print

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?当(n−1)%3==0andn≠1andn≠4 时产生 2branches

纯文本

```
def hailstone_tree(n, h): #难
    """Generates a tree of hailstone numbers that will reach N, with height
    >>> print_tree(hailstone_tree(1, 0))
```

```
    1
    >>> print_tree(hailstone_tree(1, 4))
    1
      2
        4
          8
            16
    >>> print_tree(hailstone_tree(8, 3))
    8
      16
        32
          64
        5
          10
    """
    if h==0:
        return tree(n)
    branches = [hailstone_tree(n*2, h-1)] #因为无论如何都有*2这条路
    if n-1%3==0 and n!=4 and n!=1: #hint提示了此时加路
        branches += [hailstone_tree((n-1)//3, h-1)] #可[]+[],branches里面是-
    return tree(n, branches) #生成树

def print_tree(t): #这啥
    def helper(i, t):
        print("    " * i + str(label(t)))
        for b in branches(t):
            helper(i + 1, b)
    helper(0, t)
```

hw4 Q5:Replace Loki at Leaf

Define `replace_loki_at_leaf`, which takes a tree `t` and a value `lokis_replacement`. `replace_loki_at_leaf` returns a new tree that's the same as `t` except that every leaf label equal to `"loki"` has been replaced with `lokis_replacement`.当叶子等于 loki 时替换，否则都与之前一样，对树进行操作而不是 sum 或 max，不断向下操作 branches 里的树，并定义最后叶子的操作即可

纯文本

```
def replace_loki_at_leaf(t, lokis_replacement):
    if is_leaf(t) and label(t)=='loki':
```

```
        return tree(lokis_replacement)
    return tree(label(t), [replace_loki_at_leaf(b, lokis_replacement) for b i
```

disc Q6: Has Path

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, such as autocomplete). You may assume that every node's `label` is exactly one character.输入树 t 和 string word，返回是否有路路从根到 word，叫 trie，可以应用于自动完成

纯文本

```
def has_path(t, word):
    if is_leaf(t) or len(word)==1: #与别的尽头不一样，这里有两面，要么t到了根，要么
        return label(t)==word
    elif label(t)!=word[0]: #把头部定义好
        return False
    for b in branches(t):
        if has_path(b, word[1:]):
            return True
    return False #注意缩进，若后移，会返回很多False
```