《计算机程序的构造和解释》
Lab 04: Data Abstraction, Trees, and Mutable Values

助教：李晨曦、李煦阳、吴羽、徐鼎坤、张天昀

2022 年 10 月 26 日

Problem1: Mobiles

图: Mobile(sculpture)

Problem1.2 Balanced

平衡的（Balanced）

1. length(left(m)) * total_weight(end(left(m))) == length(right(m)) * total_weight(end(right(m)))
2. m 的左右子节点各自是平衡的

Generalize and fold

```
length(left(m)) * total_weight(end(left(m)))
---------------------------------------------- generalize
def torque(a):
    assert (is_arm(a))
    return length(a) * total_weight(end(a))
```

```
length(left(m)) * total_weight(end(left(m)))
---------------------------------------------- fold
torque(left(m))
```

```
torque(left(m)) == torque(right(m))
```

辅助函数

小问题

m 的子节点有可能是 planet

```
def pre_balanced(m):
    return is_planet(m) or balanced(m)

def balanced(m):
    return torque(left(m)) == torque(right(m)) and
        pre_balanced(end(left(m))) and
        pre_balanced(end(right(m)))
```

小提示

在递归函数中避免判断子节点的类型（除非真的需要）

Problem 1.3 totals_tree

哪里错了？

```
def totals_tree(m):
    assert is_mobile(m) or is_planet(m)
    if is_planet(end(left(m))):
        ...
```

Contracts

Contracts

A programming contract consists of:

1. **Preconditions**: requirements for the input. If they do not hold, we blame the caller.
2. **Postconditions**: promises for the output. If they do not hold, we blame the library.

- Python 没有完整的 Contracts 支持（有 assert），但必须在编程时必须时刻注意一个函数的 preconditions 和 postconditions.
- ”静态”的 contracts – 类型系统、霍尔逻辑……
- 完整 Contracts 支持 – Racket，一个 scheme 方言。

Contracts

`hello` 的前件（preconditions）是什么？

```
def hello(n):
    for i in range(0, n):
        print("hello,_world")
```

Contracts

abs 的前件和后件（postconditions）是什么？

```
def abs(n):
    return n if n > 0 else -n
```

Problem 2 Preorder

一个递归函数分为两部分：

1. 基本情况
2. 递归步（组合更小输入的解）


```
preorder(tree(1)) = [1]
preorder(tree(2)) = [2]
preorder(tree(10, [tree(1), tree(2)]))
= [10, 1, 2]
= [10] + [1] + [2]
= [10] + preorder(tree(1)) + preorder(tree(2))
```

Problem 2 Preorder

```
def preorder(t):
    result_list = [label(t)]
    for b in branches(t):
        result_list += preorder(b)
    return result_list
```

Problem 3 Trie

```
has_path(tree('a'), 'a') = True
has_path(tree('a'), 'b') = False

has_path(tree('a', [tree('b'), tree('c')]), 'ab')
= has_path(tree('b'), 'b') or has_path(tree('c'), 'b')
```

A small trick

```
def f(l):
    for i in l:
        if i == 0:
            return True
    return False
```

```
def f(l):
    return any([i == 0 for i in l])
```

Problem 3 Trie

```
def has_path(t, w):
    if w[0] == label(t):
        return True
    else: # not w[0] == label(t)
        if len(w) == 1 or is_leaf(t):
            return False
        else:
            return any(
                [has_path(st, w[1:]) for st in branches(t)]
```

Problem 4

小提示

时刻警惕有副作用的操作

```
def insert_items(lst, entry, elem):
    a = 0
    m = []

    for i in lst:
        if i == entry:
            m = m + [a]
        a += 1

    for n in m:
        lst.insert(n + 1,elem)

    return lst
```

Problem 4

```
def insert_items(lst, entry, elem):
    i = 0
    for i in lst:
        if i == entry:
            lst.insert(i + 1, elem)
            if entry == elem:
                i += 1
        i += 1
    return lst
```

Problem 3.3 add_trees

小提示

branches 可以对叶子节点使用。

```
branches(tree(1)) = []
```

```
def add_trees(t1, t2):
    br_1 = branches(t1)
    br_2 = branches(t2)
    new_subtrees = add_branches(br_1, br_2)
    return tree(label(t1) + label(t2), new_subtrees)
```

```
add_branches([], l1) = l1
add_branches(l2, []) = l2
```

Problem 3.3 add_trees

列表

列表也可以看作一种“类似于树”的结构

```
[]
[]

lst(1, lst(2, []))
[1, 2]

lst(1, lst(2, lst(3, lst(4, []))))
[1, 2, 3, 4]

lst( 1,    lst(...) )
     ----   --------
      hd       tl
```

Problem 3.3 add_trees

```
def hd(l):
    return l[0]

def tl(l):
    return l[1:]

def lst(h, t):
    return [h] + t

hd([1, 2, 3]) = 1
tl([1, 2, 3]) = [2, 3]
```

```
def sum(l):
    if l == []:
        return 0
    else:
        return hd(l) +  sum(tl(l))
```

Problem 3.3 add_trees

```
def add_branches(l1, l2):
    if l1 == []:
        return l2
    elif l2 == []:
        return l1
    else:
        return lst(add_trees(hd(l1), hd(l2)),
            add_branches(tl(h1), tl(h2)))
```

Problem 3.3 add_trees

$$0 \leq i < \min(\text{len}(br_1), \text{len}(br_2))$$

$$\min(\text{len}(br_1), \text{len}(br_2)) \leq i < \max(\text{len}(br_1), \text{len}(br_2))$$

```
def add_branches(br_1, br_2):
    new_subtrees = []
    br = br_1 if len(br_1) < len(br_2) else br_2

    for i in range(max(len(br_1), len(br_2))):
        if i < min(len(br_1), len(br_2)):
            new_subtrees.append(
                add_trees(br_1[i], br_2[i]))
        else:
            new_subtrees.append(br[i])

    return new_subtrees
```

Problem 3.3 add_trees

另一种思路

把两个列表填成一样长的

小提示

不要修改 `branches()` 返回的结果

问题

填什么？

- `None`
- `tree(0)`

Problem 3.3 add_trees

```
def fill(b1, n):
    return b1 + [None] * (n - len(b1))

def add_trees(t1, t2):
    if t1 == None:
        return t2
    elif t2 == None:
        return t1
    else:
        br_1, br_2 = branches(t1), branches(t2)
        m = max(len(br_1), len(br_2))
        b1, b2 = fill(br_1, m), fill(br_2, m)

        new_subtrees = [add_trees(st1, st2) for
                        (st1, st2) in couple(b1, b2)]
        return tree(label(t1) + label(t2), new_subtrees)
```

Problem 3.4 big_path

```
def all_path(t):
    k = label(t)
    s = []
    for i in branches(t):
        s.append([ [k] + l for l in all_path(i) ])
    return [[k]] + s

def big_path(t, n):
    return len([i for i in all_path(t) if sum(i) >= n])
```

Problem 3.4 big_path

```
def all_big_path(t, n):
    k = label(t)
    s = []
    for i in branches(t):
        s.append([ [n] + l for l in all_big_path(i, n - k)
    return [[n]] + s if k >= n else s

def big_path(t, n):
    return len(all_big_path(t, n))
```

Problem 3.4 big_path

```
def big_path(t, n):
    k = label(t)
    s = 0
    for i in branches(t):
        s += big_path(t, n - k)
    return 1 + s if k >= n else s
```

```
def big_path(t, n):
    k = label(t)
    s = sum([ big_path(i, n - k) for i in branches(t) ])
    return 1 + s if k >= n else s
```

Problem 3.5 bigger_path

1. `[1, 2]`
2. `[1, 3]`
3. `[3]`

Problem 3.5 bigger_path

```
def bigger_path(t, n):
    k = label(t)
    s = 0
    for i in branches(t):
        s += bigger_path(i, n - k)
        s += bigger_path(i, n)
    return 1 + s if k >= n else s
```

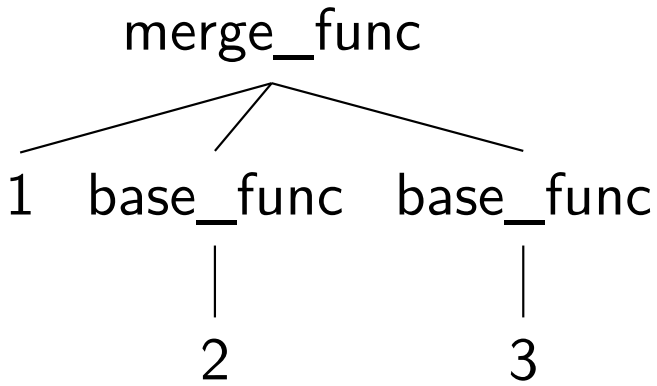
问题

bigger_path(i, n - k) 得到的路径都以 *i* 为起点吗？

Problem 3.5 bigger_path

```
def bigger_path(t, n):
    k = big_path(t, n)
    return k + sum([bigger_path(t, n) for t
                    in branches(t)])
```

Problem 4.1 tree fold



Problem 4.1 tree fold

```
def fold_tree(t, base_func, merge_func):
    if is_leaf(t):
        return base_func(label(t))
    return merge_func(label(t),
                      [fold_tree(b, base_func, merge_func)
                       for b in branches(t)])

def preorder(t):
    return fold_tree(t,
        lambda v: [v],
        lambda v, vs: [v] + flatten(vs))

flatten([[1, 2], [3, 4]]) = [1, 2, 3, 4]
```

Problem 4.1 tree fold

fold to a function:

```
def has_int(tree, i):
    def base_func(l):
        return lambda i: l == i
    def merge_func(l, bs):
        return lambda i: l == i or any([b(i) for b in bs])

    return fold_tree(tree, base_func, merge_func)(i)
```