

# Review of Lab04, Hw04, Lab05

Tianyun Zhang



# Lab04p2: Preorder

```
def preorder(t):
```

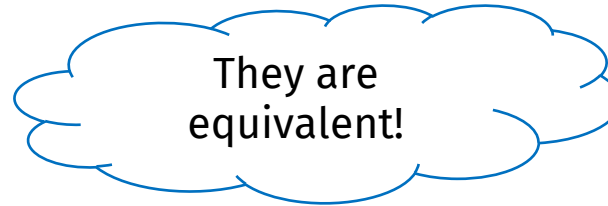
```
    Return a list of the entries in this tree in the order that they would be  
    visited by a preorder traversal.
```

# Lab04p2: Preorder

```
def preorder(t):  
    if is_leaf(t):  
        return [label(t)]
```

# Lab04p2: Preorder

```
def preorder(t):  
    if is_leaf(t):  
        return [label(t)]  
    else:  
        result = [label(t)]  
        for b in branches(t):  
            result.extend(preorder(b))  
    return result
```



# Lab04p2: Preorder

```
def preorder(t):
```

```
    result = [label(t)]
```

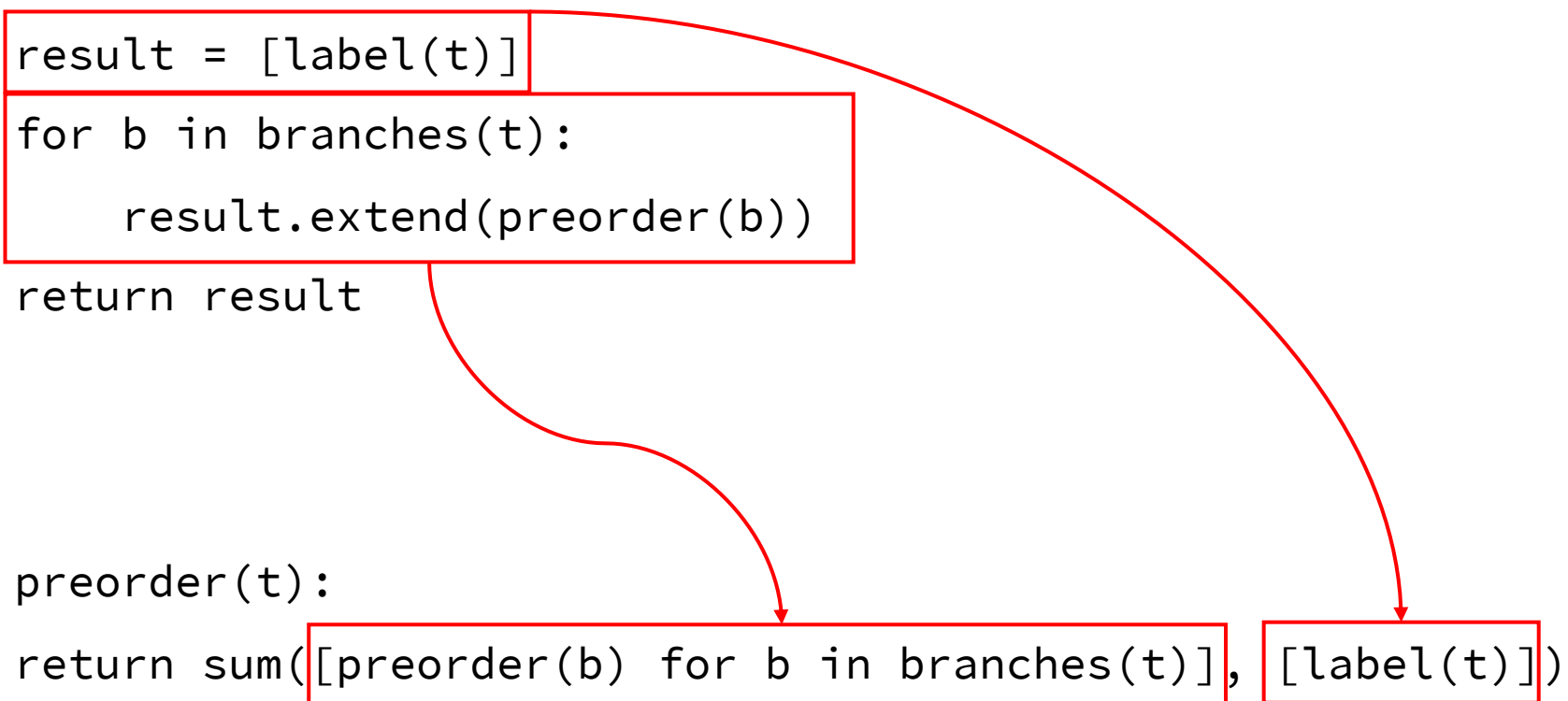
```
    for b in branches(t):
```

```
        result.extend(preorder(b))
```

```
    return result
```

```
def preorder(t):
```

```
    return sum([preorder(b) for b in branches(t)], [label(t)])
```

The diagram illustrates the transformation of a recursive preorder function into a sum-based function. Red boxes highlight the key components: the initial list construction [label(t)] in the first function, the recursive call preorder(b) within a loop, and the list comprehension [preorder(b) for b in branches(t)] and the initial list [label(t)] in the second function. Red arrows show the mapping: one arrow points from the loop's recursive call to the list comprehension, and another points from the initial list construction to the second list argument.

# Hw04p3.1: Nut Finder

```
def nut_finder(t):
```

```
    Returns True if T contains a node with the value 'nut' and False otherwise.
```

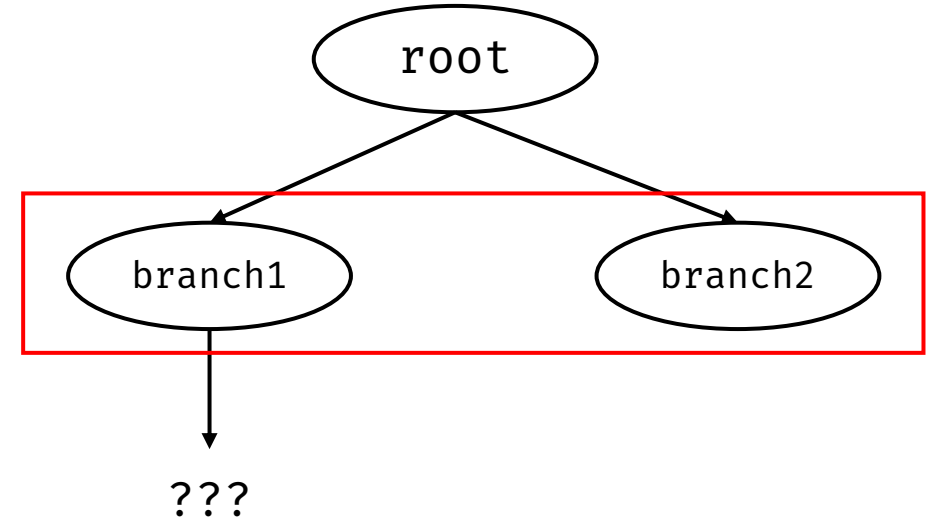
# Hw04p3.1: Nut Finder

```
def nut_finder(t):  
    if label(t) == 'nut':  
        return True
```



# Hw04p3.1: Nut Finder

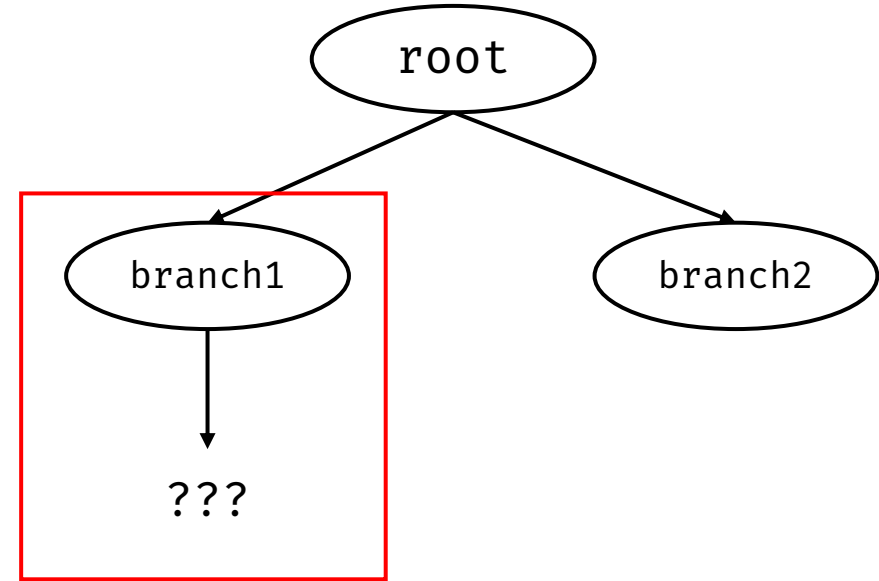
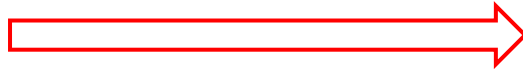
```
def nut_finder(t):  
    if label(t) == 'nut':  
        return True  
    for b in branches(t):  
        if nut_finder(b):  
            return True
```





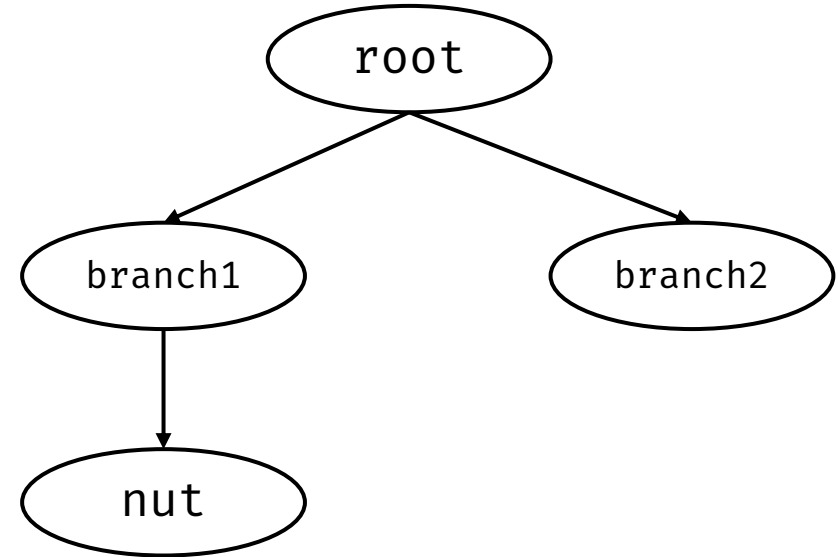
# Hw04p3.1: Nut Finder

```
def nut_finder(t):  
    if label(t) == 'nut':  
        return True  
    for b in branches(t):  
        if nut_finder(b):  
            return True
```



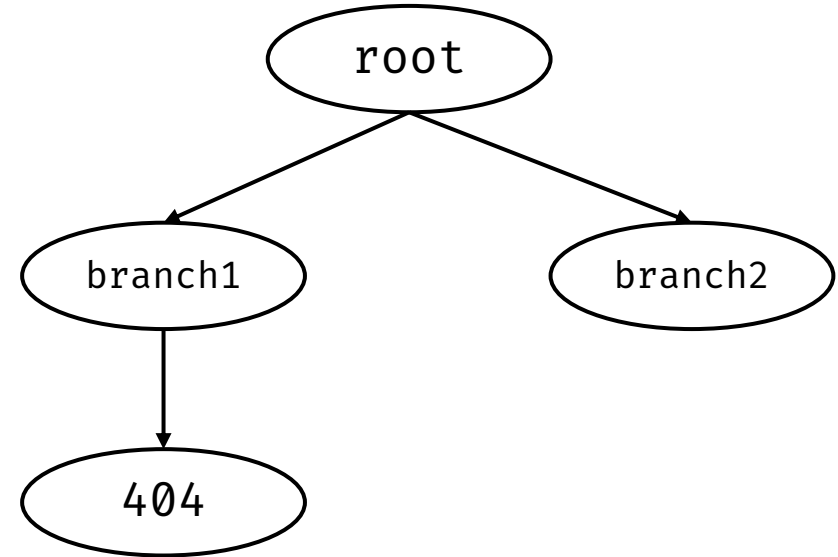
# Hw04p3.1: Nut Finder

```
def nut_finder(t):  
    if label(t) == 'nut':  
        return True  
    for b in branches(t):  
        if nut_finder(b):  
            return True
```



# Hw04p3.1: Nut Finder

```
def nut_finder(t):  
    if label(t) == 'nut':  
        return True  
    for b in branches(t):  
        if nut_finder(b):  
            return True  
    return False
```



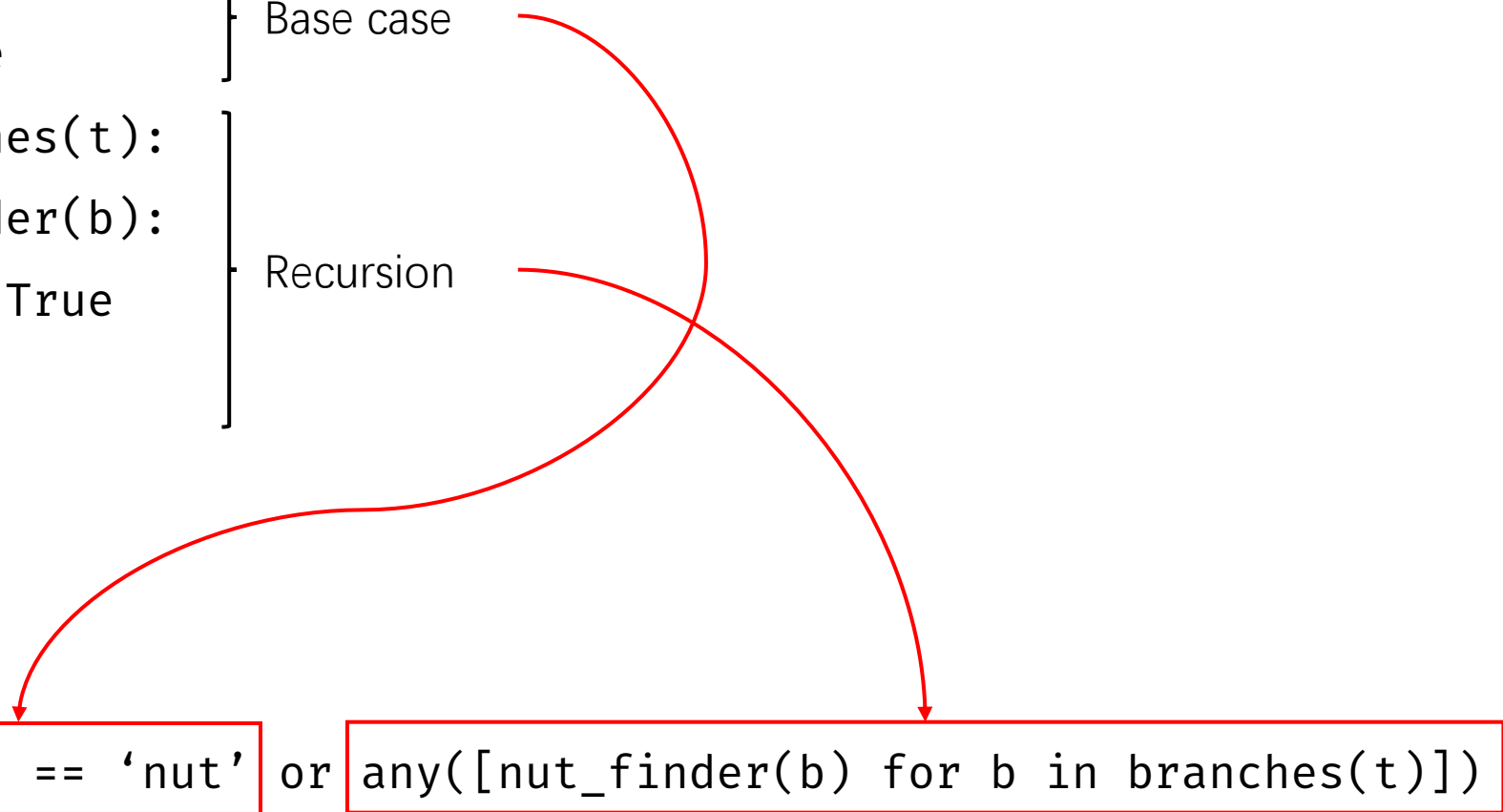
# Hw04p3.1: Nut Finder

```
def nut_finder(t):  
    if label(t) == 'nut':  
        return True  
    for b in branches(t):  
        if nut_finder(b):  
            return True  
    return False
```

} Base case

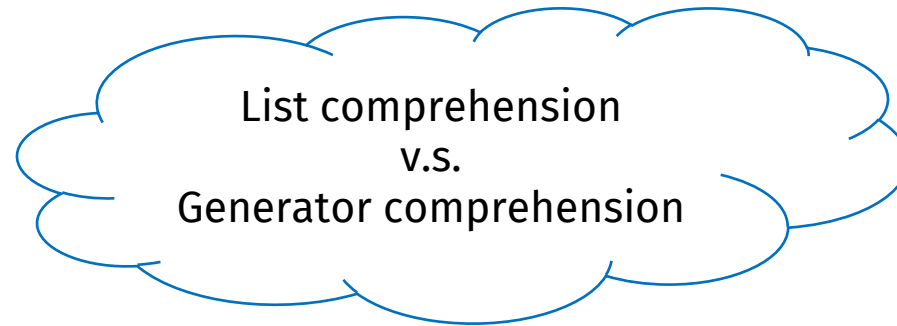
} Recursion

```
def nut_finder(t):  
    return label(t) == 'nut' or any([nut_finder(b) for b in branches(t)])
```



# Hw04p3.1: Nut Finder

```
def nut_finder(t):  
    return label(t) == 'nut' or any([nut_finder(b) for b in branches(t)])
```



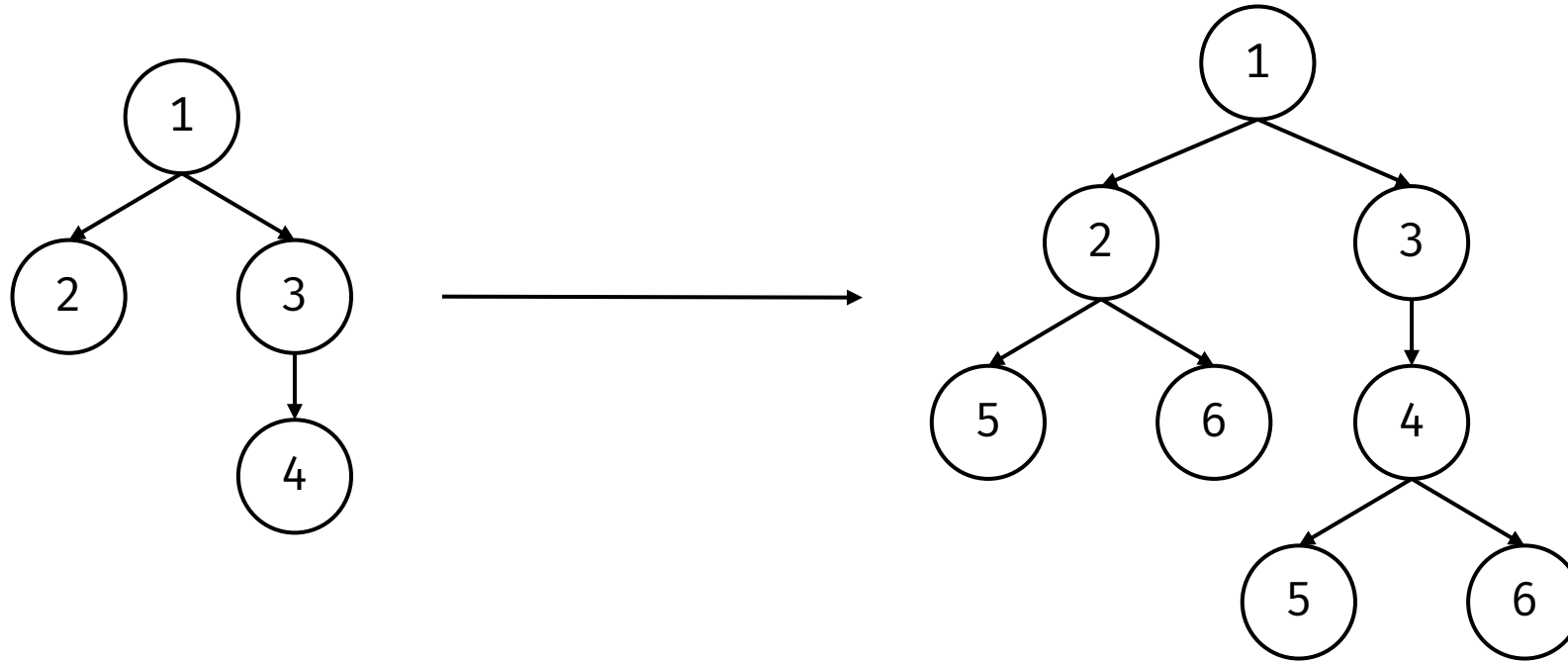
```
def nut_finder(t):  
    return label(t) == 'nut' or any(nut_finder(b) for b in branches(t))
```

Maybe Faster

# Hw04p3.2: Sprout Leaves

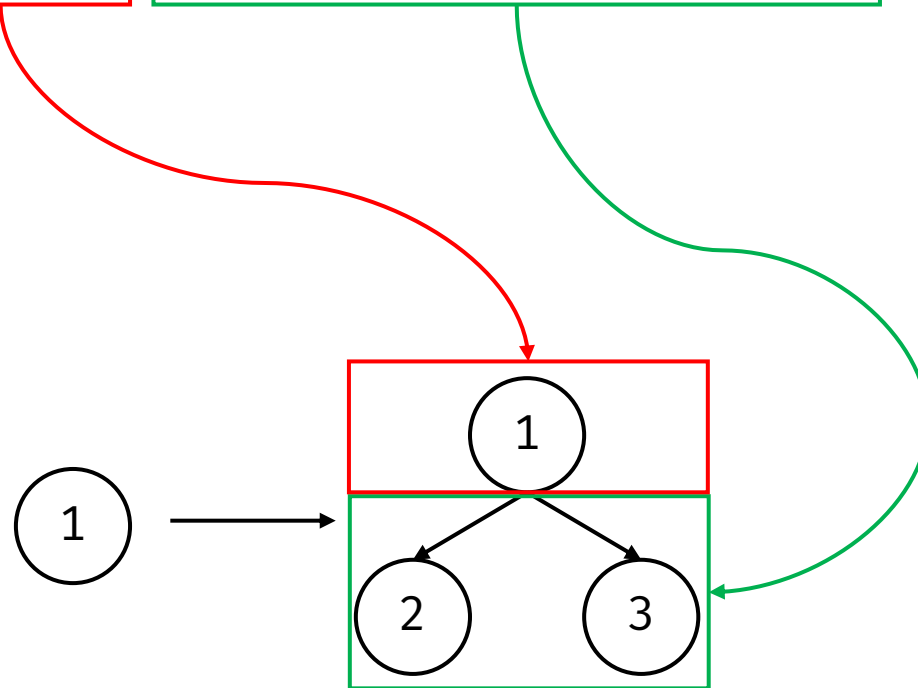
```
def sprout_leaves(t, values):
```

Sprout new leaves containing the data in VALUES at each leaf in the original tree T and return the resulting tree.



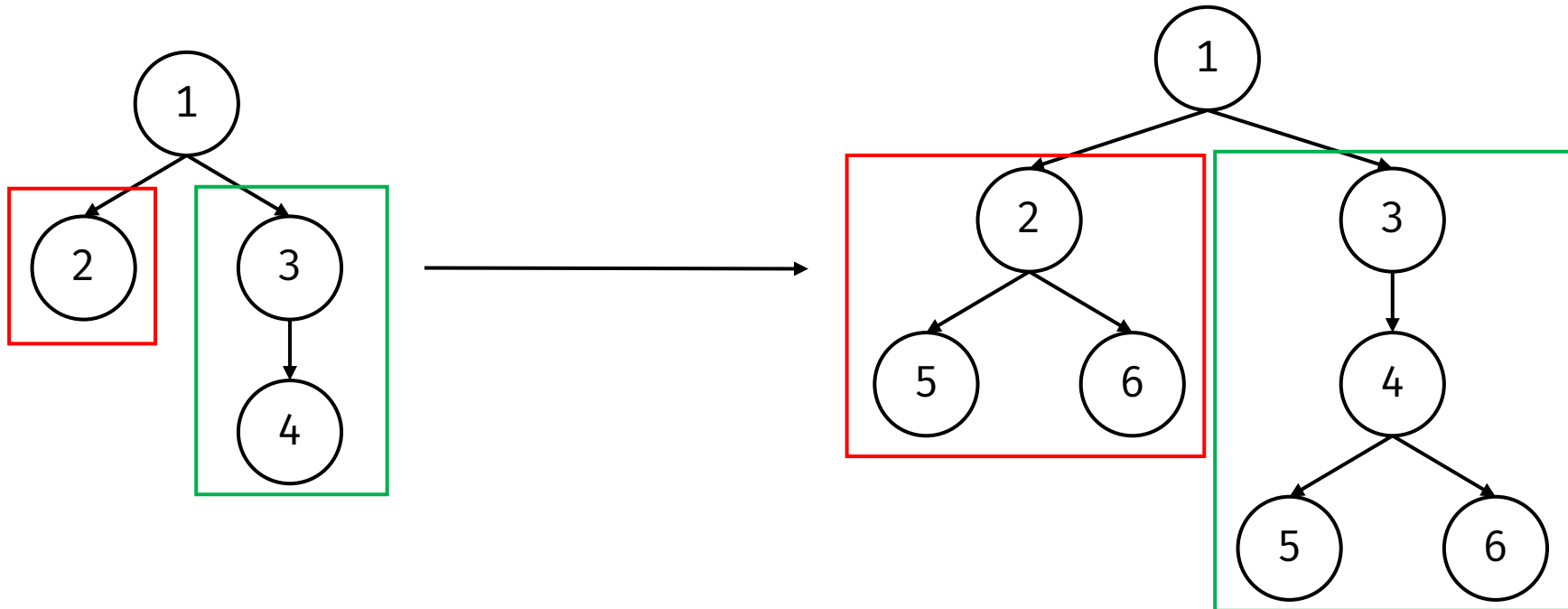
# Hw04p3.2: Sprout Leaves

```
def sprout_leaves(t, values):  
    if is_leaf(t):  
        return tree(label(t), [tree(v) for v in values])
```



# Hw04p3.2: Sprout Leaves

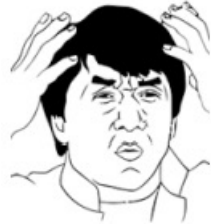
```
def sprout_leaves(t, values):  
    if is_leaf(t):  
        return tree(label(t), [tree(v) for v in values])  
    return tree(label(t), [sprout_leaves(b, values) for b in branches])
```





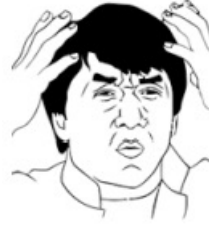
# Hw04p3.2: Sprout Leaves

```
def sprout_leaves(t, values):  
    if is_leaf(t):  
        new = list(values)  
        for x in new:  
            t.append([x])  
        return t  
    else:  
        i = 0  
        while i < len(branches(t)):  
            branches(t)[i] = sprout_leaves(branches(t)[i], values)  
            i = i + 1  
        return t
```



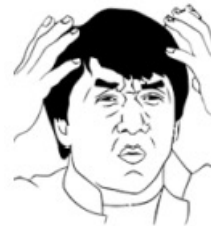
# Hw04p3.2: Sprout Leaves

```
def sprout_leaves(t, values):  
    def helper(t):  
        if is_leaf(t):  
            return tree(label(t), [[i] for i in values])  
        else:  
            return tree(label(t), [helper(i) for i in branches(t)])  
    return helper(t)
```



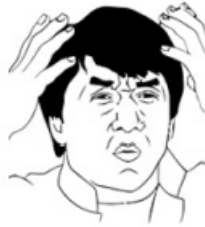
# Hw04p3.2: Sprout Leaves

```
def sprout_leaves(t, values):
    c=[]
    for d in values:
        c=c+[tree(d)]
    if branches(t)==[]:
        return tree(label(t),c)
    else:
        a=label(t)
        e=[]
        for b in branches(t):
            e=e+tree(sprout_leaves(b,values))
        a=tree(a,e)
    return a
```



# Hw04p3.2: Sprout Leaves

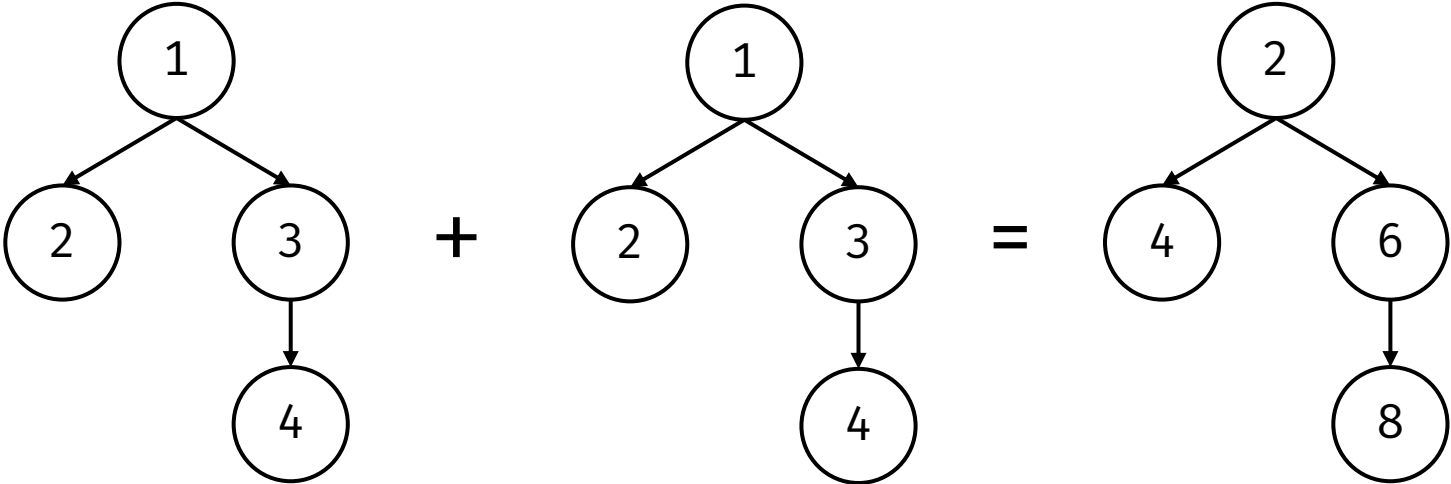
```
def sprout_leaves(t, values):  
    if is_leaf(t):  
        for x in values:  
            t.append(tree(x))  
    else:  
        for b in branches(t):  
            sprout_leaves(b, values)  
    return t
```



# Hw04p3.3: Add Trees

```
def add_trees(t1, t2):
```

Returns a new tree where each corresponding node from T1 is added with the node from T2.



# Hw04p3.3: Add Trees

```
def add_trees(t1, t2):
```

```
    def merge_branches(branches1, branches2):
        merged_branches = []
        for i in range(max(len(branches1), len(branches2))):
            if i < len(branches1) and i < len(branches2):
                merged_branches.append(add_trees(branches1[i], branches2[i]))
            elif i < len(branches1):
                merged_branches.append(branches1[i])
            else:
                merged_branches.append(branches2[i])
        return merged_branches
```

```
    return tree(label(t1) + label(t2), merge_branches(branches(t1), branches(t2)))
```

# Hw04p3.3: Add Trees

```
def add_trees(t1, t2):
    def merge_branches(branches1, branches2):
        merged_branches = []
        for i in range(max(len(branches1), len(branches2))):
            if i < len(branches1) and i < len(branches2):
                merged_branches.append(add_trees(branches1[i], branches2[i]))
            elif i < len(branches1):
                merged_branches.append(branches1[i])
            else:
                merged_branches.append(branches2[i])
        return merged_branches
    return tree(label(t1) + label(t2), merge_branches(branches(t1), branches(t2)))
```

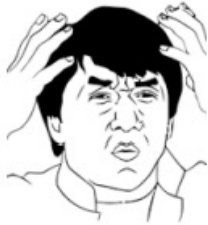
# Hw04p3.3: Add Trees

```
def add_trees(t1, t2):
    def merge_branches(branches1, branches2):
        merged_branches = []
        for i in range(max(len(branches1), len(branches2))):
            if i < len(branches1) and i < len(branches2):
                merged_branches.append(add_trees(branches1[i], branches2[i]))
            elif i < len(branches1):
                merged_branches.append(branches1[i])
            else:
                merged_branches.append(branches2[i])
        return merged_branches
    return tree(label(t1) + label(t2), merge_branches(branches(t1), branches(t2)))
```



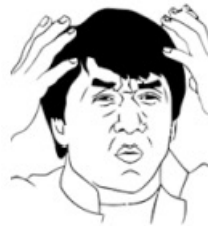
# Hw04p3.3: Add Trees

```
def add_trees(t1, t2):  
    if is_leaf(t1) or is_leaf(t2):  
        return tree(label(t1)+label(t2),branches(t1)+branches(t2))  
    lst=[]  
    n=1  
    l=min(len(t1),len(t2))  
    while l>n:  
        lst+=[(t1[n],t2[n])]  
        n+=1  
    left=t1[n:]+t2[n:]  
    return tree(label(t1)+label(t2),[add_trees(x,y) for x,y in lst]+left)
```



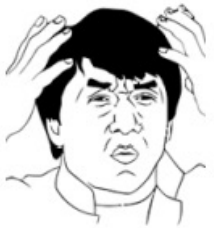
# Hw04p3.3: Add Trees

```
def add_trees(t1, t2):
    len_1, len_2 = len(branches(t1)), len(branches(t2))
    if len_1 == len_2:
        return tree(label(t1) + label(t2), \
                    [add_trees(b1, b2) for b1, b2 in zip(branches(t1), branches(t2))])
    elif len_1 < len_2:
        branches_t1 = branches(t1) + [tree(0) for _ in range(len_2 - len_1)]
        new_t1 = tree(label(t1), branches_t1)
        return add_trees(new_t1, t2)
    else:
        return add_trees(t2, t1)
```



# Hw04p3.3: Add Trees

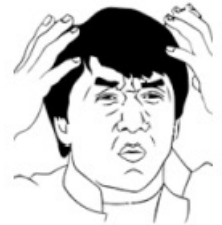
```
def add_trees(t1, t2):
    new_label = label(t1) + label(t2)
    if is_leaf(t2):
        return tree(new_label, branches(t1))
    if is_leaf(t1):
        return tree(new_label, branches(t2))
    else:
        new_branch = []
        for i in range(0, max(len(branches(t1)), len(branches(t2)))):
            if i >= len(branches(t1)):
                new_branch += [branches(t2)[i]]
            if i >= len(branches(t2)):
                new_branch += [branches(t1)[i]]
            else:
                new_branch += [add_trees(branches(t1)[i], branches(t2)[i])]
        return tree(new_label, new_branch)
```



# Hw04p3.3: Add Trees

```
def add_trees(t1, t2):
    if is_leaf(t1) and is_leaf(t2):
        return tree(label(t1) + label(t2))
    if is_leaf(t1) and not is_leaf(t2):
        return tree(label(t1) + label(t2), branches(t2))
    if is_leaf(t2) and not is_leaf(t1):
        return tree(label(t1) + label(t2), branches(t1))
    b1, b2 = branches(t1), branches(t2)
    if len(b1) > len(b2):
        t1, t2 = t2, t1
        b1, b2 = b2, b1
    nb = []
    for i in range(len(b1)):
        nb.append(add_trees(b1[i], b2[i]))
    nb += b2[len(b1):]
    return tree(label(t1) + label(t2), nb)
```

没想到吧，加法可以  
不满足交换律！



# Hw04p3.4: Big Path

```
def big_path(t, n):
```

```
    Return the number of paths in T that have a sum larger or equal to N.
```

```
    The path starts from root of T.
```

```
def bigger_path(t, n):
```

```
    Return the number of paths in T that have a sum larger or equal to N.
```

```
    The path might not start from root of T.
```

# Hw04p3.4: Big Path

```
def big_path(t, n):  
    return (1 if label(t) >= n else 0) + \  
           sum([big_path(b, n - label(t)) for b in branches(t)])
```

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

# Lab05p3: Scale

```
def scale(it, multiplier):
```

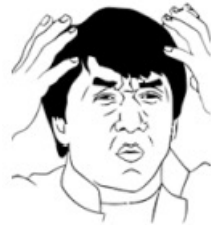
```
    Yield elements of the iterable IT scaled by a number MULTIPLIER.
```

```
    for value in it:
```

```
        yield it * multiplier
```

# Lab05p3: Scale

```
def scale(it, multiplier):  
    iter_it=iter(it)  
    a=next(iter_it)  
    while True:  
        yield a*multiplier  
        a=next(iter_it)
```



StopIteration



# Lab05p3: Scale

```
def scale(it, multiplier):
```

```
    Yield elements of the iterable IT scaled by a number MULTIPLIER.
```

```
    yield from [value * multiplier for value in it]
```



# Lab05p3: Scale

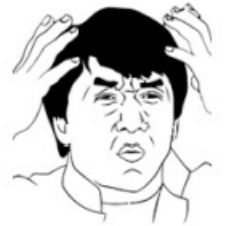
```
def scale(it, multiplier):
```

```
    Yield elements of the iterable IT scaled by a number MULTIPLIER.
```

```
    yield from [value * multiplier for value in it]
```

List comprehension must be **evaluated** first!

[1, 2, 3, 4, 5, 6, 7, 8, ...] -> infinite loop



# Lab05p3: Scale

```
def scale(it, multiplier):
```

```
    Yield elements of the iterable IT scaled by a number MULTIPLIER.
```

```
    yield from (value * multiplier for value in it)
```



# Lab05p3: Scale

```
def scale(it, multiplier):
```

```
    Yield elements of the iterable IT scaled by a number MULTIPLIER.
```

```
    yield from (value * multiplier for value in it)
```

Generator comprehension must be **evaluated** first!

A generator!



# Lab05p3: Scale

```
def scale(it, multiplier):
```

```
    Yield elements of the iterable IT scaled by a number MULTIPLIER.
```

```
    yield from map(lambda value: multiplier * value, it)
```



# Lab05p3: Scale

```
def scale(it, multiplier):
```

```
    Yield elements of the iterable IT scaled by a number MULTIPLIER.
```

```
    yield from map(lambda value: multiplier * value, it)
```

Function call must be **evaluated** first!

map returns an iterable class!



# Lab05p3: Scale

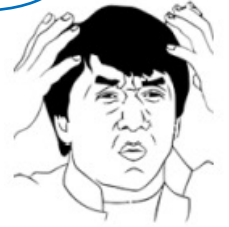
```
def scale(it, multiplier):  
    yield from [value * multiplier for value in it]
```

```
def scale(it, multiplier):  
    yield from (value * multiplier for value in it)
```

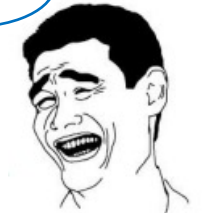
“Lazy Evaluation”

```
def scale(it, multiplier):  
    yield from map(lambda value: multiplier * value, it)
```

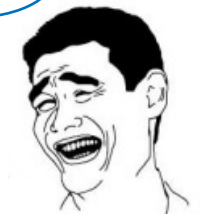
List  
comprehension



Generator  
comprehension



Iterable class



# Lab05p5: Hailstone

```
def hailstone(n):
```

```
    Return a generator that outputs the hailstone sequence.
```

```
    yield n
```

```
    if n == 1:
```

```
        return
```

```
    elif n % 2 == 0:
```

```
        yield from hailstone(n // 2)
```

```
    else:
```

```
        yield from hailstone(n * 3 + 1)
```



# Lab05p5: Hailstone

```
def hailstone(n):
```

```
    while n!=1 :
```

```
        yield n
```

```
        if n%2==0:
```

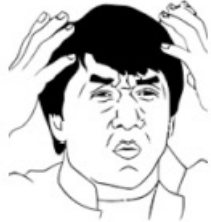
```
            n = n//2
```

```
            if (n==1):
```

```
                yield 1
```

```
        else:
```

```
            n = 3*n + 1
```



```
def hailstone(n):
```

```
    while n!=1 :
```

```
        yield n
```

```
        if n%2==0:
```

```
            n = n//2
```

```
        else:
```

```
            n = 3*n + 1
```

```
    yield 1
```



# Something about yield

```
def func():  
    a = [1, 2]  
    yield a  
    a = [2, 1]  
    yield a
```

The diagram illustrates the execution of the function `func()`. It shows two `yield` statements. The first `yield a` statement is connected by a red arrow to the output tuple `1, 2`. The second `yield a` statement is connected by a red arrow to the output tuple `2, 1`. The arrows originate from the `yield a` text and point to the corresponding output tuple, showing the sequence of values returned by the generator.

# Something about yield

```
def func():  
    a = [1, 2]  
    yield a  
    a = [2, 1]  
    yield a
```

The diagram illustrates the execution of a function with two yield statements. The first yield statement is connected by a red arrow to the value [1, 2], and the second yield statement is connected by a red arrow to the value [2, 1].

```
def func():  
    a = [1, 2]  
    yield a  
    a[0], a[1] = a[1], a[0]  
    yield a
```

The diagram illustrates the execution of a function with a yield statement. A red arrow connects the yield statement to the value [1, 2].

# Something about yield

```
def func():  
    a = [1, 2]  
    yield a  
    a = [2, 1]  
    yield a
```

The diagram illustrates the execution of a function with two yield statements. The first yield statement is connected by a red arrow to the value [1, 2], and the second yield statement is connected by a red arrow to the value [2, 1].

```
def func():  
    a = [1, 2]  
    yield a  
    a[0], a[1] = a[1], a[0]  
    yield a
```

The diagram illustrates the execution of a function with a swap operation. The first yield statement is connected by a red arrow to the value [1, 2]. The second yield statement is connected by a red arrow to the value [2, 1]. A red arrow also points from the swap operation to the second yield statement, indicating that the swap operation is performed before the second yield statement.

# Something about `yield`

```
def func():  
    a = [1, 2]  
    yield a  
    a[0], a[1] = a[1], a[0]  
    yield a
```

`1, 2`

```
>>> it = func()  
>>> next(it)  
[1, 2]
```

The diagram illustrates the execution of the `func()` function. A red arrow originates from the `yield a` statement in the first block of the function and points to the text `1, 2`. A second red arrow originates from the `next(it)` call in the interactive prompt and points to the output `[1, 2]`. This visualizes that the first call to `next()` on the iterator returns the list `[1, 2]`.

# Something about yield

```
def func():
```

```
    a = [1, 2]
```

```
    yield a
```

```
    a[0], a[1] = a[1], a[0]
```

```
    yield a
```

2, 1

```
>>> it = func()
```

```
>>> next(it)
```

```
[1, 2]
```

```
>>> next(it)
```

```
[2, 1]
```

# Something about yield

```
def func():
```

```
    a = [1, 2]
```

```
    yield a
```

```
    a[0], a[1] = a[1], a[0]
```

```
    yield a
```

2, 1

```
>>> it = func()
```

```
>>> next(it)
```

```
[1, 2]
```

```
>>> next(it)
```

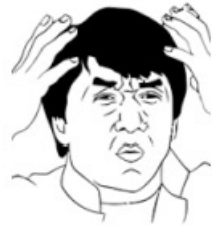
```
[2, 1]
```

```
>>> sorted(list(it))
```

```
[[2, 1], [2, 1]]
```

# Something about lambda

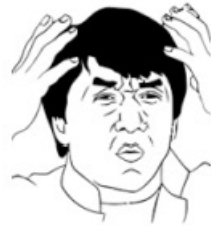
```
i = 0
def helper(x):
    print(i)
while i < n:
    yield helper
    i += 1
```





# Something about lambda

```
i = 0
def helper(x):
    print(i)
while i < n:
    yield helper
    i += 1
```



```
i = 0
def helper(x, i):
    print(i)
while i < n:
    yield helper(bar, i)
    i += 1
```

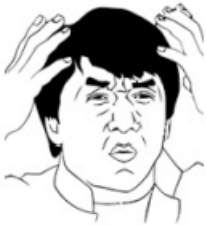
Function call must  
be evaluated

Parameter i must  
be evaluated

But we don't want to  
call the function right now!

# Something about lambda

```
i = 0  
def helper(x):  
    print(i)  
while i < n:  
    yield helper  
    i += 1
```



```
i = 0  
def helper(x, i):  
    print(i)  
while i < n:  
    yield helper(bar, i)  
    i += 1
```

Function call must be evaluated

Parameter i must be evaluated

```
i = 0  
def helper(x, i):  
    print(i)  
while i < n:  
    yield (lambda i: lambda x: helper(x, i))(i)  
    i += 1
```



Function call must be evaluated

# Questions?

恭喜你又变的更强了👍