计算机程序的构造与解释（SICP）
期中试题

姓 名：_____

学 号：_____

题目	分数
1	
2	
3	
4	
5	
6	
7	
8	

1. **(18 points) What Would Python Display**

For each of the expressions in the table below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. Each expression has at least one line of output.

- If an error occurs, write **Error**, but include all output displayed before the error.
- To display a function value, write **Function**.
- If an expression would take forever to evaluate, write **Forever**.

The interactive interpreter displays the value of a successfully evaluated expression, unless it is None. Assume that you have started python3 and executed the code shown on the left. Expressions evaluated by the interpreter have a cumulative effect.

```
def foo(x, f):
    def bar(g):
        return f(x) or g
    return bar

def my_all(it):
    for x in it:
        if not x:
            return False
    return True

def gen_p():
    n = 2
    while True:
        if my_all([n % i != 0
                   for i in range(2, n)]):
            yield n
        n += 1

a2 = lambda x : 2 + x
m10 = lambda x : 10 * x

def com(n):
    if n == 1:
        return lambda f: lambda x: f(x)
    return lambda f: lambda g: com(n - 1)(lambda x: f(g(x)))
```

Expression	Interactive Output
`print(2, 0) + 1`	2 0 **Error**
`2**11 - 28` `True and not 1/0`	

`min, max = max, min(2, 0)`
`print(min)`
`print(max)`	

`t = (0, [1])`
`t[0] = 2`
`s = t[1]`
`t[1].append(2)`
`s is t[1]`
`t[1][1:] + [t[0]] + [2, 0]`	

`foo('SICP', print)(2020)`	

`my_all(True, False)`
`my_all([])`
`my_all([True, False])`	

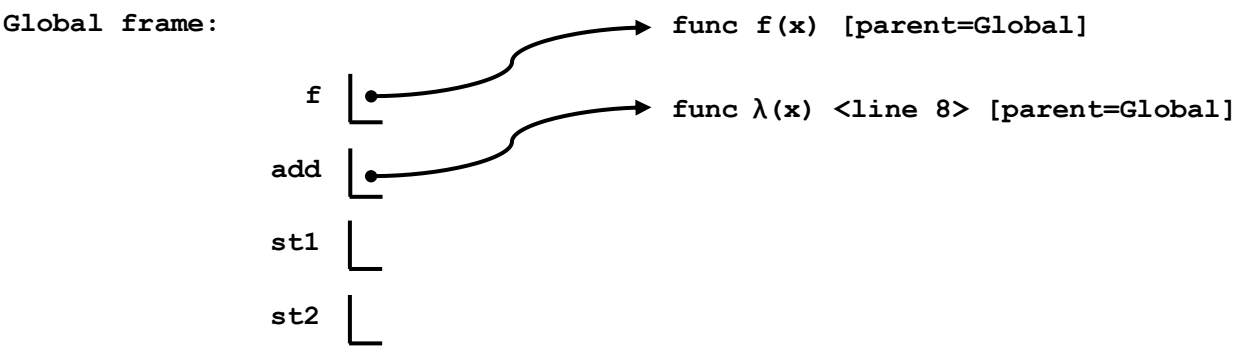
`p = gen_p()`
`for _ in range(3):`
`  print(next(p))`
`p is gen_p()`	

`com(3)(m10)(a2)(m10)(20)`	

2. (12 points) Environment Diagram

Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. You may not need to use all of the spaces or frames. A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.
- The diagram should reflect the result at the end of the program (you don't need to show intermediate results).



f1: _____ [parent = _____]

_____

_____

Return Value

f2: _____ [parent = _____]

_____

Return Value

f3: _____ [parent = _____]

_____

Return Value

f4: _____ [parent = _____]

_____

Return Value

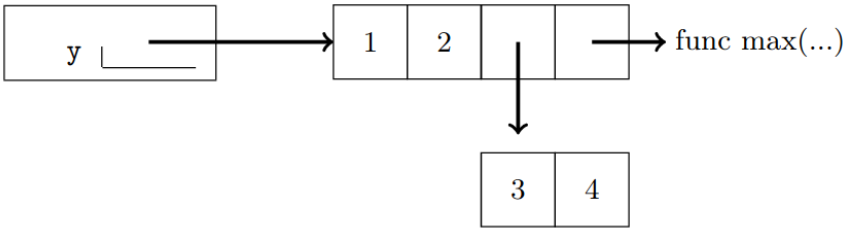
```
1  def f(x):
2      def mut(f):
3          nonlocal x
4          x = f(x)
5          return mut
6      return mut
7
8  add = lambda x: lambda y: x + y
9  st1 = f(5)
10 st2 = st1(add(4))
```

3. (12 points) Boxing Day

Draw box-and-pointer diagrams for the state of the lists after executing each block of code below. Each box should contain a value or a pointer to an object, as in an environment diagram. You don't need to write index numbers or the word `list`. Please erase or cross out any boxes or pointers that are not part of a final diagram.

An example is given below:

```
1  y = [1, 2, [3, 4], max]
```



You **only** need to draw diagrams for the variables **whose names are already provided in the table**. You can write `empty list` at the end of a pointer to indicate an empty list.

(a, 3 points)

```
lst1 = [0, [1], [2]]
lst2 = [lst1[2], 0, lst1[1][:]]
lst3 = [x in lst2 for x in lst1]
```

lst1

lst2

lst3

(b, 4 points)

```
lst1 = [1, 2]
lst2 = [lst1, lst1.append(3)]
lst3 = []
while lst1:
    lst3 = [lst1[0], lst3]
    lst1 = lst1[1:]
```

lst1

lst2

lst3

(c, 2 points)

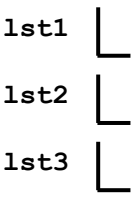
```
lst1 = [-1, 0, True, 'None']
lst2 = []
lst1, k = lst1[1:], lst1[0]
popper = lambda lst:
  lambda: lst[k] and lst.pop()
popper = popper(lst1)
for _ in range(len(lst1)):
    lst2.append(popper())
```

lst1

lst2

(d, 3 points)

```
lst1 = [[2], 0]
lst2 = lst1[1:] + [lst1[0]]
lst3 = list(lst2)
for item in lst3:
    if item:
        lst2.append(item.pop() + 1)
    else:
        item -= 1
        lst1.append(item)
```



4. (5 points) Reverse Digits

Define a function `reverse_digit` that takes an input integer `n` as argument and returns another integer with an reversed order of the digits of `n`.

```
def reverse_digit(n):
    """
    >>> reverse_digit(1240)
    421
    """
    result = _____________________________

    while _____________________________:

        result = _____________________________

        n = _____________________________

    return result
```

5. (5 points) Simple Math (NOT Church numeral!)

```
plus = lambda x, y: x + y # auxiliary function used to define mult
def mult(m, n):
    """
    mult(m, n) returns the value of m*n, where both m and n are positive integers.
    Just put one name or constant in each of the blank space.
    Do NOT write anything more.
    """
    if m == 0:

        return _____________________

    m1 = m - 1

    return plus(__________, ( __________( __________, __________ ) ) )
```

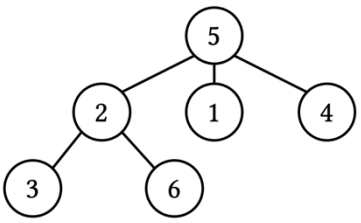
6. (27 points) Trees

The tree data abstraction is provided here for your reference. Do not violate the abstraction barrier!

```
def tree(label, branches=[]):
    return [label] + list(branches)

def label(t):
    return t[0]

def is_leaf(t);
    return not branches(t)

def branches(t):
    return t[1:]
```

(a, 4 points) Define the tree `t1` shown on the right using the `tree` constructor (supposing that the node with label `5` is the root node of `t1`):

`t1`



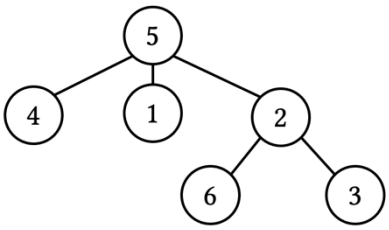

```
t1 =
```

(b, 6 points) Define the function `label_sum(t)`, which returns the summation of all the nodes' labels. For instance, given the `t1` defined above, `label_sum(t1)` returns `21`.

```
def label_sum(t):
    if _____________________ :
        return _____________________
    return ________ + sum([_____________________ for ________ in _____________________ ])
```

(c, 5 points) Define the function `reverse_tree(t)`, which forms a new tree, where the branch order of each node is the **reverse** of the branch order of the corresponding node on `t`. For instance, `reverse_tree(t1)` for the `t1` defined above generates the tree `t3`.

`t3`




```
def reverse_tree(t):
    if _____________________ :
        return ____________________________________
    reversed_branches = ____________________________________________________
    return ____________________________________________________________
```

(d, 12 points) It is often the case that we want to fold a tree into a value, and we just did it in the `label_sum` function. Now, it is time to define a function `fold_tree` to abstract such a process!

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

Read the definition above and make sure that you understand for what `base_func` and `merge_func` are defined. Here is an example for the usage of the `fold_tree`, where we count the number of leaves by it.

```
def count_leaves(t):
    return fold_tree(t, lambda v: 1, lambda v, vs: sum(vs))
```

Now implement label_sum again using our fold_tree function.

```
def label_sum(t):
    return fold_tree(t, _____________, _____________________________)
```

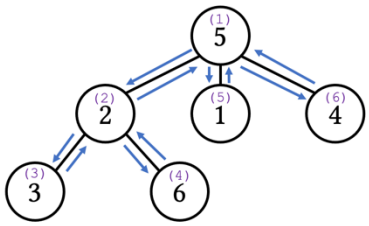
The height of a tree **t** is defined recursively as follows: if **t** is a leaf, its height is 1; otherwise its height is the maximum height of sub-trees plus 1. For example, the height of **t1** from question (a) is 3. Now using the `fold_tree` to implement the height function below to compute the height of a tree. You may use the python built-in function **max**.

```
def height(t):
    return fold_tree(t, _____________, _____________________________)
```

Recall that we defined a `preorder` function in our homework which takes in a tree as an argument and returns a list of all the labels in the tree in an order that the label of the root node is printed first, then are the branches. The picture below shows you what is the preorder of **t1** from question (a). Now reimplement the `preorder` function using the `fold_tree`. We give you the definition of **reduce** function, which may do you a favor. The printed order of each node in `t1` during `preorder` is shown on the right.

```
def reduce(f, base, it):
    result = base
    for x in it:
        result = f(result, x)
    return result

def preorder(t):
    return fold_tree(t, ____________________,
        ____________________________________________________________)
```

t1 (preorder)



7. (21 points) Automatic Function Composition

In our half semester of programming learning, we often use given functions and variables to write a program, in order to complete some task. Well, have you ever thought of throwing these functions and variables to your computer and urging it to automatically write the program itself? Although it sounds crazy, we are now going to implement a toy version!

Automatic Function Composition Problem

Given: We are given an integer pair (x, y) (also called input-output pair), a list of defined functions $[f_1, f_2, \dots, f_n]$ (also called function list), and a list of integers $[t_1, t_2, \dots, t_n]$ (also called times list), where each function f_i $(i = 1 \dots n)$ takes exactly one integer argument and returns exactly one integer value.

Goal: We want to automatically find a non-empty list of m functions $[f_{i_1}, f_{i_2}, \dots, f_{i_m}]$ (where $m > 0$, $i_j \in \{1, \dots, n\}$, for each $1 \le j \le m$) where each function is chosen from the original n functions, and compose these chosen ones as $g = \text{lambda } x\text{: } f_{i_m}(f_{i_{m-1}}(\dots(f_{i_1}(x))))$, which satisfies that $g(x)$ equals y. Note that each function $\mathrm{f_i}$ in the original function list can only be chosen at most $\mathrm{t_i}$ ($\mathrm{t_i}$ is an integer greater than 0) times and at least 0 times (0-times means not chosen). For example, $[f_1, f_1]$ would be a solution to the problem with input-output pair (x_1, y_1), function list $[f_1, f_2, f_3]$, times list $[3,3,3]$, if $f_1(f_1(x_1))$ equals y_1. In this example, f_1 is chosen twice (2 times), while both f_2 and f_3 are not chosen (0 times). These chosen times ($2, 0, 0$ times) are all in the range from 0 to 3.

The following two examples help you understand this problem.

Example 1: We are given an input-output pair (i.e., 2-tuple) `(4, 6)`, a function list `[dec, mul2]`, and a times list `[2, 2]`, where `dec = lambda x: x-1` and `mul2 = lambda x: 2*x`. A solution is `[mul2, dec, dec]`, which means the composed function `g = lambda x: dec(dec(mul2(x)))` makes `g(4) == 6` true. However, `[dec, mul2]` is also a solution, since `mul2(dec(4)) == 6` is true. The example suggests that we may have one or more solutions to the automatic function composition problem. In fact, the solution list to this problem is `[[dec, mul2], [mul2, dec, dec]]`.

Example 2: We are given an input-output pair `(4, 6)`, a function list `[mul2, mul3]`, and a times list `[1000, 1000]`, where `mul2 = lambda x: 2*x` and `mul3 = lambda x: 3*x`. We cannot find any solution to this concrete problem (the solution list is `[]`), which suggests that we may have no solutions to the automatic function composition problem. No need to be surprised because the given function list and times list restrict the power of composing the program you want.

(a, 4 points) Do the following problems have solutions? If so, write ALL the solutions in a list (each solution is a function list looking just like `[mul2,dec,dec]` in Example 1, and the order between solutions does not matter). If not, just write "No solution".

(1) Input-output pair: `(11, 3)`
Function list: `[div2, inc]`, where `div2 = lambda x: x // 2` and `inc = lambda x: x + 1`
Times list: `[2,1]`

(2) Input-output pair: `(5, 1)`
Function list: `[div3, add3]`, where `div3 = lambda x: x // 3` and `add3 = lambda x: x + 3`
Times list: `[1,1]`

(b, 12 points) Implement the functions `satisfy` and `afc` whose frameworks are shown below. `afc` takes in an input-output pair `io_pair`, a function list `func_list`, and a times list `times_list` with integer elements. `afc` returns a list storing ALL the solutions (the order between solutions does not matter) to the corresponding automatic function composition problem with these given arguments. If it had no solution, the returned solution list would be an empty list. You may use the function `satisfy` in your implementation of `afc`.

```
def satisfy(io_pair, sol):
    """Returns True if sol satisfies io_pair, otherwise returns False.
    """
    input_x = io_pair[0]
    output_y = io_pair[1]
    eval_result = _____________________________
    for _____________________________:
        eval_result = _____________________________
    return _____________________________________

def afc(io_pair, func_list, times_list):
    """Returns the solution list of the automatic function composition problem
    given io_pair, func_list, and times_list.
    """
    assert len(func_list) == len(times_list)
    solutions = []
    def sol_search(remain_times_list, cur_sol):
        # cur_sol is a list of currently chosen functions
        # Every solution should NOT be empty
        if _____________________________:
            solutions.append(cur_sol)
        for _____________________________:
            if _____________________________________________:
                new_times_list = ____________________________________________________
                sol_search(new_times_list, _____________________________________________)
    sol_search(times_list, [])
    return solutions
```

(c, 5 points) From now on, this problem will be extended a bit. Input-output pair is now extended to input-output pair list $[(x_1, y_1), (x_2, y_2), \ldots, (x_p, y_p)]$, which is a list containing p input-output pairs. Now a solution g to this extended problem should satisfy that FOR EACH input-output pair (x_i, y_i) (where $i \in \{1, \ldots, p\}$), we have $g(x_i)$ equals y_i.

Example 3: We are given an input-output pair list `[(2,0), (3,3)]`, a function list `[dec, mul3]`, and a times list `[2, 2]`, where `dec = lambda x: x-1` and `mul3 = lambda x: 3*x`. The solution list to this problem is `[[dec, dec, mul3]]`, since `mul3(dec(dec(2))) == 0` and `mul3(dec(dec(3))) == 3`. Although `[dec, dec]` satisfies the input-output pair `(2, 0)`, it fails to satisfy `(3, 3)`, leading to its absence from the final solution list.

Please implement the function `extended_afc`, where you should use the functions `afc` and `satisfy` defined in question (b).

```
def extended_afc(io_pair_list, func_list, times_list):
    """Returns the solution list of the automatic function composition problem
    given io_pair, func_list, and times_list.
    """
    assert len(io_pair_list) > 0

    solutions = ____________________________________________________________________

    return my_filter(_____________________________________________________, solutions)
```

Hint:

(1) You only need to use `afc` once.

(2) You may need to use `my_filter` function and `my_all` function. Their implementations are shown below.

```
def my_filter(f, lst):
    return [element for element in lst if f(element)]

def my_all(lst):
    for bool_elem in lst:
        if not bool_elem:
            return False
    return True
```

8. (Extra 3 points) Questions about SICP

(a) The full name of SICP is _____. (in English)

(b) The two instructors' names are _____ and _____, respectively. (in Chinese)

(c) Write the names of at least two teaching assistants: _____. (in Chinese)