Exam: CS61A Summer 2020 Midterm

Name: Solution Key

Email: example_key

secure

```
Point breakdown
    q1: 1.0/1

Score:
    Total: 1.0

Reskeletonized solution follows

==================================================
def cat(password, limit):
    " Write a higher-order function `cat` that returns a one-argument\n    funct¶
ion `attempt`. Every time `attempt` is called, it checks to see if its argument\¶
n    matches the password at the corresponding index.\n\n    If the password ent¶
irely matches, return a success string. If more than `limit`\n    number of inco¶
rrect hacks are attempted, you should return an error string.\n    For details, ¶
see the doctest.\n\n\n    Note: to comment out a blank that covers an entire lin¶
e, just put down 'unnecessary' (with quotes)\n\n    >>> hacker = cat([1,2], 2)\n¶
    >>> hacker(1)\n    >>> hacker(2)\n    'Successfully unlocked!'\n    >>> hack¶
er = cat([1,2], 1)\n    >>> hacker(1)\n    >>> hacker(3) # used up attempts to g¶
ain access\n    >>> hacker(2) # correct attempt to gain access, but already lock¶
ed\n    'The safe is now inaccessible!'\n    >>> hacker = cat([1,2], 2)\n    >>>¶
 hacker(1)\n    >>> hacker(3) # 1 attempt left to gain access\n    >>> hacker(2)¶
 # correct attempt to gain access\n    'Successfully unlocked!'\n    "
    num_incorrect = 0
    index = 0

    def attempt(digit):
        nonlocal num_incorrect
        nonlocal index
        if (num_incorrect >= limit):
            return 'The safe is now inaccessible!'
        if (password[index] == digit):
            index += 1
            if (index == len(password)):
                return 'Successfully unlocked!'
        else:
            num_incorrect += 1
    return attempt
==================================================


Original code follows

==================================================
def cat(password, limit):
    """ Write a higher-order function `cat` that returns a one-argument
    function `attempt`. Every time `attempt` is called, it checks to see if its ¶
argument
    matches the password at the corresponding index.

    If the password entirely matches, return a success string. If more than `lim¶
it`
    number of incorrect hacks are attempted, you should return an error string.
```

```
    For details, see the doctest.


    Note: to comment out a blank that covers an entire line, just put down 'unne¶
cessary' (with quotes)

    >>> hacker = cat([1,2], 2)
    >>> hacker(1)
    >>> hacker(2)
    'Successfully unlocked!'
    >>> hacker = cat([1,2], 1)
    >>> hacker(1)
    >>> hacker(3) # used up attempts to gain access
    >>> hacker(2) # correct attempt to gain access, but already locked
    'The safe is now inaccessible!'
    >>> hacker = cat([1,2], 2)
    >>> hacker(1)
    >>> hacker(3) # 1 attempt left to gain access
    >>> hacker(2) # correct attempt to gain access
    'Successfully unlocked!'
    """
    num_incorrect = 0
    index = 0
    def attempt(digit):
        nonlocal num_incorrect
        nonlocal index
        if num_incorrect >= limit:
            return 'The safe is now inaccessible!'
        if password[index] == digit:
            index += 1
            if index == len(password):
                return "Successfully unlocked!"
        else:
            num_incorrect += 1
    return attempt
==================================================
```

schedule

```
Point breakdown
    q2: 1.0/1

Score:
    Total: 1.0

Reskeletonized solution follows

==================================================
def schedule(galaxy, sum_to, max_digit):
    '\n    A \'galaxy\' is a string which contains either digits or \'?\'s.\n\n ¶
   A \'completion\' of a galaxy is a string that is the same as galaxy, except\n¶
    with digits replacing each of the \'?\'s.\n\n    Your task in this question ¶
is to find all completions of the given `galaxy`\n    that use digits up to `max¶
_digit`, and whose digits sum to `sum_to`.\n\n    Note 1: the function int can b¶
e used to convert a string to an integer and str\n        can be used to convert¶
 an integer to a string as such:\n\n        >>> int("5")\n        5\n        >>>¶
 str(5)\n        \'5\'\n\n    Note 2: Indexing and slicing can be used on string¶
s as well as on lists.\n\n        >>> \'evocative\'[3]\n        \'c\'\n        >¶
>> \'evocative\'[3:]\n        \'cative\'\n        >>> \'evocative\'[:6]\n       ¶
 \'evocat\'\n        >>> \'evocative\'[3:6]\n        \'cat\'\n\n\n    >>> schedu¶
le(\'?????\', 25, 5)\n    [\'55555\']\n    >>> schedule(\'???\', 5, 2)\n    [\'1¶
22\', \'212\', \'221\']\n    >>> schedule(\'?2??11?\', 5, 3)\n    [\'0200111\', ¶
\'0201110\', \'0210110\', \'1200110\']\n    '

    def schedule_helper(galaxy, sum_sofar, index):
        if ((index >= len(galaxy)) and (sum_sofar == sum_to)):
            return [galaxy]
        elif ((sum_sofar > sum_to) or (index >= len(galaxy))):
            return []
        elif (galaxy[index] != '?'):
            return schedule_helper(galaxy, (sum_sofar + int(galaxy[index])), (in¶
dex + 1))
        ans = []
        for x in range((max_digit + 1)):
            modified_galaxy = ((galaxy[:index] + str(x)) + galaxy[(index + 1):])¶

            ans += schedule_helper(modified_galaxy, (sum_sofar + x), (index + 1)¶
)
        return ans
    return schedule_helper(galaxy, 0, 0)
==================================================


Original code follows

==================================================
def schedule(galaxy, sum_to, max_digit):
    """
    A 'galaxy' is a string which contains either digits or '?'s.

    A 'completion' of a galaxy is a string that is the same as galaxy, except
    with digits replacing each of the '?'s.
```

```
    Your task in this question is to find all completions of the given `galaxy`
    that use digits up to `max_digit`, and whose digits sum to `sum_to`.

    Note 1: the function int can be used to convert a string to an integer and s¶
tr
        can be used to convert an integer to a string as such:

        >>> int("5")
        5
        >>> str(5)
        '5'

    Note 2: Indexing and slicing can be used on strings as well as on lists.

        >>> 'evocative'[3]
        'c'
        >>> 'evocative'[3:]
        'cative'
        >>> 'evocative'[:6]
        'evocat'
        >>> 'evocative'[3:6]
        'cat'


    >>> schedule('?????', 25, 5)
    ['55555']
    >>> schedule('???', 5, 2)
    ['122', '212', '221']
    >>> schedule('?2??11?', 5, 3)
    ['0200111', '0201110', '0210110', '1200110']
    """
    def schedule_helper(galaxy, sum_sofar, index):
        if index >= len(galaxy) and sum_sofar == sum_to:
            return [galaxy]
        elif sum_sofar > sum_to or index >= len(galaxy):
            return []
        elif galaxy[index] != '?':
            return schedule_helper(galaxy, sum_sofar + int(galaxy[index]), index¶
 + 1)
        ans = []
        for x in range(max_digit + 1):
            modified_galaxy = galaxy[:index] + str(x) + galaxy[index + 1:]
            ans += schedule_helper(modified_galaxy, sum_sofar + x, index + 1)
        return ans

    return schedule_helper(galaxy, 0, 0)
==================================================
```

consume

```
Point breakdown
    q3: 1.0/1

Score:
    Total: 1.0

Reskeletonized solution follows

==================================================
'\nLet a `painting` be a self-referential function that\n    - takes in one inte¶
ger\n    - returns two values, another painting and well as an integer\n\nFor an¶
 example see the function `identity_painting` below.\n\nYou have two tasks in th¶
is assignment, to implement the functions `microscope`\nand `plush`. Both have t¶
heir behavior defined by their doctests.\n\nIt is not necessary to implement `mi¶
croscope` correctly to get the points for\n`plush`. However, the ok test cases f¶
or `plush` will fail if you have not correctly\nimplemented `microscope`.\n'

def identity_painting(x):
    return (identity_painting, x)

def microscope(a=0, s=1):
    '\n    This function returns a painting function that processes a sequence\n¶
    of integers, and returns the alternating sum of all integers seen thus\n    ¶
far (see doctest for an example).\n\n    >>> painting_a = microscope()\n    >>> ¶
painting_b, x = painting_a(2)\n    >>> x                                 # 2\n¶
    2\n    >>> painting_c, x = painting_b(8)\n    >>> x                        ¶
         # 2 - 8\n    -6\n    >>> painting_d, x = painting_c(12)\n    >>> x    ¶
                                # 2 - 8 + 12\n    6\n    >>> painting_e, x = pain¶
ting_d(30)\n    >>> x                                  # 2 - 8 + 12 - 30\n    -¶
24\n    >>> painting_b_again, x = painting_a(100)\n    >>> x                    ¶
            # 100 [note that we are using painting_a not painting_d here]\n    ¶
  100\n    '

    def painting(x):
        return (microscope((a + (s * x)), (- s)), (a + (s * x)))
    return painting

def plush(painting, items):
    '\n    The function `plush` takes in a `painting` and a nonempty list of `it¶
ems` and\n    runs the given `painting` on each of the `items` in turn, returnin¶
g the final\n    numeric result.\n\n    For example, on the items [1, 2, 3, 4, 5¶
] with the painting microscope\n    we return 1 - 2 + 3 - 4 + 5 = 3\n\n    >>> p¶
lush(microscope(), [1, 2, 3, 4, 5])\n    3\n    >>> plush(microscope(), [4000])\¶
n    4000\n    >>> plush(microscope(), [2, 90])\n    -88\n    >>> plush(identity¶
_painting, [2, 90])\n    90\n    '
    (painting, x) = painting(items[0])
    if (len(items) == 1):
        return x
    return plush(painting, items[1:])
==================================================
```

Original code follows

```
==================================================
"""
Let a `painting` be a self-referential function that
    - takes in one integer
    - returns two values, another painting and well as an integer

For an example see the function `identity_painting` below.

You have two tasks in this assignment, to implement the functions `microscope`
and `plush`. Both have their behavior defined by their doctests.

It is not necessary to implement `microscope` correctly to get the points for
`plush`. However, the ok test cases for `plush` will fail if you have not correc¶
tly
implemented `microscope`.
"""

def identity_painting(x):
    return identity_painting, x

def microscope(a=0, s=1):
    """
    This function returns a painting function that processes a sequence
    of integers, and returns the alternating sum of all integers seen thus
    far (see doctest for an example).

    >>> painting_a = microscope()
    >>> painting_b, x = painting_a(2)
    >>> x                                   # 2
    2
    >>> painting_c, x = painting_b(8)
    >>> x                                   # 2 - 8
    -6
    >>> painting_d, x = painting_c(12)
    >>> x                                   # 2 - 8 + 12
    6
    >>> painting_e, x = painting_d(30)
    >>> x                                   # 2 - 8 + 12 - 30
    -24
    >>> painting_b_again, x = painting_a(100)
    >>> x                                   # 100 [note that we are using painti¶
ng_a not painting_d here]
    100
    """
    def painting(x):
        return microscope(a + s * x, -s), a + s * x
    return painting

def plush(painting, items):
    """
    The function `plush` takes in a `painting` and a nonempty list of `items` an¶
d
```

```
    runs the given `painting` on each of the `items` in turn, returning the fina¶
l
    numeric result.

    For example, on the items [1, 2, 3, 4, 5] with the painting microscope
    we return 1 - 2 + 3 - 4 + 5 = 3

    >>> plush(microscope(), [1, 2, 3, 4, 5])
    3
    >>> plush(microscope(), [4000])
    4000
    >>> plush(microscope(), [2, 90])
    -88
    >>> plush(identity_painting, [2, 90])
    90
    """
    painting, x = painting(items[0])
    if len(items) == 1:
        return x
    return plush(painting, items[1:])
==================================================
```

exact_copy

```
Point breakdown
    q4: 1.0/1

Score:
    Total: 1.0

Reskeletonized solution follows

==================================================
def lemon(xv):
    '\n    A lemon-copy is a perfect replica of a nested list\'s box-and-pointer¶
 structure.\n        If an environment diagram were drawn out, the two should be¶
 entirely\n        separate but identical.\n\n    A `xv` is a list that only con¶
tains ints and other lists.\n\n    The function `lemon` generates a lemon-copy o¶
f the given list `xv`.\n\n    Note: The `isinstance` function takes in a value a¶
nd a type and determines\n        whether the value is of the given type. So\n\n¶
        >>> isinstance("abc", str)\n        True\n        >>> isinstance("abc", ¶
list)\n        False\n\n    Here\'s an example, where lemon_y = lemon(y)\n\n\n  ¶
                           +-----+-----+            +-----+-----+-----+\n      ¶
                      |     |     |             |     |     |     |\n         ¶
                 |  +  |  +-------------> | 200 | 300 |  +  |\n        y +-------¶
----------> |  |  |     |             |     |     |  |  |\n                     ¶
       +-----+-----+       +--> +-----+-----+-----+\n        lemon_y +-+        ¶
     |                |       ^           |\n                     |            +--¶
--------------+       |           |\n                   |                          ¶
              +-----------+\n                  |\n                 |          +¶
-----+-----+           +-----+-----+-----+\n                    |          |     ¶
|     |             |     |     |     |\n                  +-------->  |  +  |  +-¶
------------> | 200 | 300 |  +  |\n                              |  |  |     |   ¶
         |     |     |  |  |\n                              +-----+-----+       +¶
--> +-----+-----+-----+\n                             |                |     ¶
  ^           |\n                               +-----------------+       |     ¶
     |\n                                                          +-----------+\¶
n\n    >>> x = [200, 300]\n    >>> x.append(x)\n    >>> y = [x, x]              ¶
# this is the `y` from the doctests\n    >>> lemon_y = lemon(y)      # this is t¶
he `lemon_y` from the doctests\n    >>> # check that lemon_y has the same struct¶
ure as y\n    >>> len(lemon_y)\n    2\n    >>> lemon_y[0] is lemon_y[1]\n    Tru¶
e\n    >>> len(lemon_y[0])\n    3\n    >>> lemon_y[0][0]\n    200\n    >>> lemon¶
_y[0][1]\n    300\n    >>> lemon_y[0][2] is lemon_y[0]\n    True\n    >>> # chec¶
k that lemon_y and y have no list objects in common\n    >>> lemon_y is y\n    F¶
alse\n    >>> lemon_y[0] is y[0]\n    False\n    '
    lemon_lookup = []

    def helper(xv):
        if isinstance(xv, int):
            return xv
        for old_new in lemon_lookup:
            if (old_new[0] is xv):
                return old_new[1]
        new_xv = []
        lemon_lookup.append((xv, new_xv))
        for element in xv:
```

```
            new_xv.append(helper(element))
        return new_xv
    return helper(xv)
==================================================

Original code follows

==================================================
def lemon(xv):
    """
    A lemon-copy is a perfect replica of a nested list's box-and-pointer structu¶
re.
        If an environment diagram were drawn out, the two should be entirely
        separate but identical.

    A `xv` is a list that only contains ints and other lists.

    The function `lemon` generates a lemon-copy of the given list `xv`.

    Note: The `isinstance` function takes in a value and a type and determines
        whether the value is of the given type. So

        >>> isinstance("abc", str)
        True
        >>> isinstance("abc", list)
        False

    Here's an example, where lemon_y = lemon(y)


                             +-----+-----+            +-----+-----+-----+
                             |     |     |            |     |     |     |
                             |  +  |  +-------------> | 200 | 300 |  +  |
        y +----------------> |  |  |     |            |     |     |  |  |
                             +-----+-----+       +--> +-----+-----+-----+
        lemon_y +-+             |                |       ^           |
                  |             +----------------+       |           |
                  |                                      +-----------+
                  |
                  |          +-----+-----+            +-----+-----+-----+
                  |          |     |     |            |     |     |     |
                  +------->  |  +  |  +-------------> | 200 | 300 |  +  |
                             |  |  |     |            |     |     |  |  |
                             +-----+-----+       +--> +-----+-----+-----+
                                |                |       ^           |
                                +----------------+       |           |
                                                         +-----------+

    >>> x = [200, 300]
    >>> x.append(x)
    >>> y = [x, x]              # this is the `y` from the doctests
    >>> lemon_y = lemon(y)      # this is the `lemon_y` from the doctests
    >>> # check that lemon_y has the same structure as y
```

```
    >>> len(lemon_y)
    2
    >>> lemon_y[0] is lemon_y[1]
    True
    >>> len(lemon_y[0])
    3
    >>> lemon_y[0][0]
    200
    >>> lemon_y[0][1]
    300
    >>> lemon_y[0][2] is lemon_y[0]
    True
    >>> # check that lemon_y and y have no list objects in common
    >>> lemon_y is y
    False
    >>> lemon_y[0] is y[0]
    False
    """
    lemon_lookup = []
    def helper(xv):
        if isinstance(xv, int):
            return xv
        for old_new in lemon_lookup:
            if old_new[0] is xv:
                return old_new[1]
        new_xv = []
        lemon_lookup.append((xv, new_xv))
        for element in xv:
            new_xv.append(helper(element))
        return new_xv
    return helper(xv)
==================================================
```

nth_repeating_seq

```
Point breakdown
    q5: 1.0/1

Score:
    Total: 1.0

Reskeletonized solution follows

==================================================
def subsaltshaker(disk):
    "\n    A 'saltshaker' is a sequence of digits of length `d` composed entirel¶
y of the digit `d`. Examples include\n        1\n        4444\n        7777777\n¶
\n    Note that `1 <= d <= 9`; there are no 0-length saltshakers.\n\n    Your ta¶
sk is to implement the `subsaltshaker` function, which takes in an integer `disk¶
` and returns\n        whether `disk` contains a saltshaker as a consecutive sub¶
integer of its digits.\n\n    >>> subsaltshaker(2233) # 22 counts\n    True\n    ¶
 >>> subsaltshaker(2444423) # 4444 counts\n    True\n    >>> subsaltshaker(82223¶
) # 22 counts even if it appears as part of 222\n    True\n    >>> subsaltshaker¶
(234562) # 2...2 does not count if the 2s are not consecutive\n    False\n    >>¶
> subsaltshaker(1) # 1 counts\n    True\n    >>> subsaltshaker(498729879871) # 1¶
 counts\n    True\n    >>> subsaltshaker(149872987987) # 1 counts\n    True\n    ¶
 >>> subsaltshaker(4445555) # no saltshakers in this number\n    False\n    >>> ¶
subsaltshaker(20) # no saltshakers in this number\n    False\n    "
    current_digit = (disk % 10)
    count = 0
    while (disk != 0):
        last = (disk % 10)
        if (current_digit == last):
            count += 1
        else:
            count = 1
            current_digit = last
        if (count == current_digit):
            return True
        disk = (disk // 10)
    return False
==================================================


Original code follows

==================================================
def subsaltshaker(disk):
    """
    A 'saltshaker' is a sequence of digits of length `d` composed entirely of th¶
e digit `d`. Examples include
        1
        4444
        7777777

    Note that `1 <= d <= 9`; there are no 0-length saltshakers.

    Your task is to implement the `subsaltshaker` function, which takes in an in¶
```

```
teger `disk` and returns
        whether `disk` contains a saltshaker as a consecutive subinteger of its ¶
digits.

    >>> subsaltshaker(2233) # 22 counts
    True
    >>> subsaltshaker(2444423) # 4444 counts
    True
    >>> subsaltshaker(82223) # 22 counts even if it appears as part of 222
    True
    >>> subsaltshaker(234562) # 2...2 does not count if the 2s are not consecuti¶
ve
    False
    >>> subsaltshaker(1) # 1 counts
    True
    >>> subsaltshaker(498729879871) # 1 counts
    True
    >>> subsaltshaker(149872987987) # 1 counts
    True
    >>> subsaltshaker(4445555) # no saltshakers in this number
    False
    >>> subsaltshaker(20) # no saltshakers in this number
    False
    """
    current_digit = disk % 10
    count = 0
    while disk != 0:
        last = disk % 10
        if current_digit == last:
            count += 1
        else:
            count = 1
            current_digit = last
        if count == current_digit:
            return True
        disk = disk // 10
    return False
==================================================
```

copycat

```
Point breakdown
    q6: 1.0/1

Score:
    Total: 1.0

Reskeletonized solution follows

==================================================
def copycat(lst1, lst2):
    "\n    Write a function `copycat` that takes in two lists.\n        `lst1` i¶
s a list of strings\n        `lst2` is a list of integers\n\n    It returns a ne¶
w list where every element from `lst1` is copied the\n    number of times as the¶
 corresponding element in `lst2`. If the number\n    of times to be copied is ne¶
gative (-k), then it removes the previous\n    k elements added.\n\n    Note 1: ¶
`lst1` and `lst2` do not have to be the same length, simply ignore\n    any extr¶
a elements in the longer list.\n\n    Note 2: you can assume that you will never¶
 be asked to delete more\n    elements than exist\n\n\n    >>> copycat(['a', 'b'¶
, 'c'], [1, 2, 3])\n    ['a', 'b', 'b', 'c', 'c', 'c']\n    >>> copycat(['a', 'b¶
', 'c'], [3])\n    ['a', 'a', 'a']\n    >>> copycat(['a', 'b', 'c'], [0, 2, 0])\¶
n    ['b', 'b']\n    >>> copycat([], [1,2,3])\n    []\n    >>> copycat(['a', 'b'¶
, 'c'], [1, -1, 3])\n    ['c', 'c', 'c']\n    "

    def copycat_helper(lst1, lst2, lst_so_far):
        if ((len(lst1) == 0) or (len(lst2) == 0)):
            return lst_so_far
        if (lst2[0] >= 0):
            lst_so_far = (lst_so_far + [lst1[0] for _ in range(lst2[0])])
        else:
            lst_so_far = lst_so_far[:lst2[0]]
        return copycat_helper(lst1[1:], lst2[1:], lst_so_far)
    return copycat_helper(lst1, lst2, [])
==================================================


Original code follows

==================================================
def copycat(lst1, lst2):
    """
    Write a function `copycat` that takes in two lists.
        `lst1` is a list of strings
        `lst2` is a list of integers

    It returns a new list where every element from `lst1` is copied the
    number of times as the corresponding element in `lst2`. If the number
    of times to be copied is negative (-k), then it removes the previous
    k elements added.

    Note 1: `lst1` and `lst2` do not have to be the same length, simply ignore
    any extra elements in the longer list.

    Note 2: you can assume that you will never be asked to delete more
```

```
    elements than exist


    >>> copycat(['a', 'b', 'c'], [1, 2, 3])
    ['a', 'b', 'b', 'c', 'c', 'c']
    >>> copycat(['a', 'b', 'c'], [3])
    ['a', 'a', 'a']
    >>> copycat(['a', 'b', 'c'], [0, 2, 0])
    ['b', 'b']
    >>> copycat([], [1,2,3])
    []
    >>> copycat(['a', 'b', 'c'], [1, -1, 3])
    ['c', 'c', 'c']
    """
    def copycat_helper(lst1, lst2, lst_so_far):
        if len(lst1) == 0 or len(lst2) == 0:
            return lst_so_far
        if lst2[0] >= 0:
            lst_so_far = lst_so_far + [lst1[0] for _ in range(lst2[0])]
        else:
            lst_so_far = lst_so_far[:lst2[0]]
        return copycat_helper(lst1[1:], lst2[1:], lst_so_far)
    return copycat_helper(lst1, lst2, [])
==================================================
```

flatmap_tree

```
Point breakdown
    q7: 1.0/1

Score:
    Total: 1.0

Reskeletonized solution follows

==================================================
def village(apple, t):
    '\n    The `village` operation takes\n        a function `apple` that maps a¶
n integer to a tree where\n            every label is an integer.\n        a tre¶
e `t` whose labels are all integers\n\n    And applies `apple` to every label in¶
 `t`.\n\n    To recombine this tree of trees into a a single tree,\n        simp¶
ly copy all its branches to each of the leaves\n        of the new tree.\n\n    ¶
For example, if we have\n        apple(x) = tree(x, [tree(x + 1), tree(x + 2)])\¶
n    and\n        t =         10\n                  /                    20    ¶
 30\n\n    We should get the output\n\n        village(apple, t)\n          =   ¶
                10\n                           /                              ¶
/                                 11                12\n                    /  ¶
  \\           /                     20      30       20      30\n             ¶
    / \\     /  \\     /  \\    /                    21 22  31   32   21   22  31  ¶
32\n    >>> t = tree(10, [tree(20), tree(30)])\n    >>> apple = lambda x: tree(x¶
, [tree(x + 1), tree(x + 2)])\n    >>> print_tree(village(apple, t))\n    10\n  ¶
    11\n        20\n            21\n            22\n        30\n            31\n      ¶
    32\n      12\n        20\n            21\n            22\n        30\n            ¶
31\n         32\n    '

    def graft(t, bs):
        '\n        Grafts the given branches `bs` onto each leaf\n        of the¶
 given tree `t`, returning a new tree.\n        '
        if is_leaf(t):
            return tree(label(t), bs)
        new_branches = [graft(b, bs) for b in branches(t)]
        return tree(label(t), new_branches)
    base_t = apple(label(t))
    bs = [village(apple, b) for b in branches(t)]
    return graft(base_t, bs)

def tree(label, branches=[]):
    'Construct a tree with the given label value and a list of branches.'
    for branch in branches:
        assert is_tree(branch), 'branches must be trees'
    return ([label] + list(branches))

def label(tree):
    'Return the label value of a tree.'
    return tree[0]

def branches(tree):
    'Return the list of branches of the given tree.'
    return tree[1:]
```

```
def is_tree(tree):
    'Returns True if the given tree is a tree, and False otherwise.'
    if ((type(tree) != list) or (len(tree) < 1)):
        return False
    for branch in branches(tree):
        if (not is_tree(branch)):
            return False
    return True

def is_leaf(tree):
    "Returns True if the given tree's list of branches is empty, and False\n    ¶
otherwise.\n    "
    return (not branches(tree))

def print_tree(t, indent=0):
    'Print a representation of this tree in which each node is\n    indented by ¶
two spaces times its depth from the entry.\n    '
    print((('  ' * indent) + str(label(t))))
    for b in branches(t):
        print_tree(b, (indent + 1))
==================================================

Original code follows

==================================================
def village(apple, t):
    """
    The `village` operation takes
        a function `apple` that maps an integer to a tree where
            every label is an integer.
        a tree `t` whose labels are all integers

    And applies `apple` to every label in `t`.

    To recombine this tree of trees into a a single tree,
        simply copy all its branches to each of the leaves
        of the new tree.

    For example, if we have
        apple(x) = tree(x, [tree(x + 1), tree(x + 2)])
    and
        t =         10
                  /    \
                20      30

    We should get the output

        village(apple, t)
          =                   10
                            /       \
                         /             \
                       11                12
                     /    \            /    \
```

```
                 20      30       20      30
                / \     /  \     /  \    /  \
              21 22  31   32   21  22  31  32
    >>> t = tree(10, [tree(20), tree(30)])
    >>> apple = lambda x: tree(x, [tree(x + 1), tree(x + 2)])
    >>> print_tree(village(apple, t))
    10
      11
        20
          21
          22
        30
          31
          32
      12
        20
          21
          22
        30
          31
          32
    """
    def graft(t, bs):
        """
        Grafts the given branches `bs` onto each leaf
        of the given tree `t`, returning a new tree.
        """
        if is_leaf(t):
            return tree(label(t), bs)
        new_branches = [graft(b, bs) for b in branches(t)]
        return tree(label(t), new_branches)
    base_t = apple(label(t))
    bs = [village(apple, b) for b in branches(t)]
    return graft(base_t, bs)

def tree(label, branches=[]):
    """Construct a tree with the given label value and a list of branches."""
    for branch in branches:
        assert is_tree(branch), 'branches must be trees'
    return [label] + list(branches)

def label(tree):
    """Return the label value of a tree."""
    return tree[0]

def branches(tree):
    """Return the list of branches of the given tree."""
    return tree[1:]

def is_tree(tree):
    """Returns True if the given tree is a tree, and False otherwise."""
    if type(tree) != list or len(tree) < 1:
        return False
```

```
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True

def is_leaf(tree):
    """Returns True if the given tree's list of branches is empty, and False
    otherwise.
    """
    return not branches(tree)

def print_tree(t, indent=0):
    """Print a representation of this tree in which each node is
    indented by two spaces times its depth from the entry.
    """
    print('  ' * indent + str(label(t)))
    for b in branches(t):
        print_tree(b, indent + 1)
==================================================
```