# CS 61A Fall 2017

# Structure and Interpretation of Computer Programs

MIDTERM 2

# INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except two hand-written  $8.5" \times 11"$  crib sheets of your own creation and the two official CS 61A midterm study guides.
- Mark your answers on the exam itself. We will not grade answers written on scratch paper.

Last name	
First name	
Student ID number	
0.10 / 1 / 21 / 2 / 1	
CalCentral email (_@berkeley.edu)	
TLA	
TA	
Name of the person to your left	
Name of the person to your right	
All the work on this exam is my own. (please sign)	

#### POLICIES & CLARIFICATIONS

- If you need to use the restroom, bring your phone and exam to the front of the room.
- Before asking a question, read the announcements on the screen/board. We will not answer your question directly. If we decide to respond, we'll add our response to the screen/board so everyone can see the clarification.
- For fill-in-the blank coding problems, we will only grade work written in the provided blanks. You may only write one Python statement per blank line, and it must be indented to the level that the blank is indented.
- Unless otherwise specified, you are allowed to reference functions defined in previous parts of the same question.

# 1. (12 points) By Any Other Name

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. The interactive interpreter displays the repr string of the value of a successfully evaluated expression, unless it is None. If an error occurs, write "Error", but include all output displayed before the error. The first row has been provided as an example.

Assume that you have started python3 and executed the code shown on the left first, then you evaluate each expression on the right in order. Statements and expressions sent to the interpreter have a cumulative effect.

```
class Plant:
    k = 1
    kind = "green"
    def __init__(self):
        self.k = Plant.k
        Plant.k = self.k + 1
        if self.k > 3:
            Plant.name = lambda t: "tree"
            Plant.k = 6
    def name(self):
        return kind
    def __repr__(self):
        s = self.name() + " "
        return s + str(self.k)
class Flower(Plant):
    kind = "pretty"
    def __repr__(self):
        s = self.smell() + " "
        return s + Plant.__repr__(self)
    def smell(self):
        return "bad"
class Rose(Flower):
    def name(self):
        return "rose"
    def smell(self):
        return "nice"
class Garden:
    def __init__(self, kind):
        self.name = kind
        self.smell = kind().smell
    def smell(self):
        return self.name.kind
f1 = Flower()
f2 = Flower()
```

Expression	Interactive Output
[2, 3]	[2, 3]
(1 pt) f1.name()	
(1 pt) f1.k	
(1 pt) Plant().k	
(1 pt) Rose.k	
(2 pt) Plant()	
(2 pt) Rose()	
(2 pt) Garden(Flower).smell()	
(2 pt) Garden(Flower).name()	

Name:

### 2. (8 points) Buy Local

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:

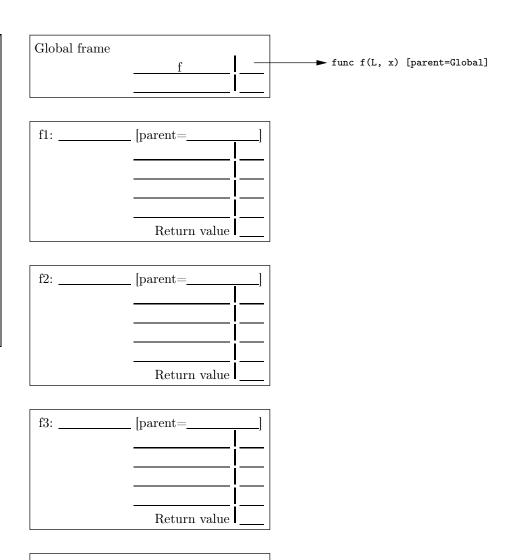
- Add all missing names and parent annotations to frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.
- Use box-and-pointer notation for list values. You do not need to write index numbers or the word "list".

[parent=\_\_\_\_

Return value

Important: The slash on line 11 means that the return expression continues on the next line.

```
1
    def f(L, x):
 2
      L[1] = L
 3
       def g(c, h, b):
 4
         nonlocal x
 5
         x = c
 6
         if c == 0:
 7
           b.append(5)
 8
           b = L + b
 9
           return h
10
         else:
11
           return \
12
             g(c-1,
13
               lambda: [c, x],
14
15
       p = g(1, None, [4])
16
       x += 3
17
       return p()
18
    r = f([0, 0], 1)
```



3. (	(10)	points	Pumpkin	Splice	Latte

`	• / •
	(2 pt) Implement splice, which takes two lists a and b and a non-negative integer k that is less than or equal to the length of a. It returns the result of <i>splicing</i> b into a at k. That is, it returns a new list containing the first k elements of a, then all elements of b, then the remaining elements of a.
	splice(a, b, k): """Return a list of the first k elements of a, then all of b, then the rest of a.
	>>> splice([2, 3, 4, 5], [6, 7], 2) [2, 3, 6, 7, 4, 5] """
	return
1	(3 pt) Implement all_splice, which returns a list of all the non-negative integers k such that splicing list b into list a at k creates a list with the same contents as c. Assume that splice is implemented correctly.
	all_splice(a, b, c): """Return a list of all k such that splicing b into a at position k gives c.
	>>> all_splice([1, 2], [3, 4], [1, 3, 4, 2]) [1] >>> all_splice([1, 2, 1, 2], [1, 2], [1, 2, 1, 2, 1, 2]) [0, 2, 4]
	return
1	(5 pt) Implement splink, which takes two Link instances a and b and a non-negative integer k that is less than or equal to the length of a. It returns a Link instance containing the first k elements of a, then all elements of b, then the remaining elements of a. The Link class is defined on the midterm 2 study guide.  Important: You may not use len, in, for, list, slicing, element selection, addition, or list comprehensions
	<pre>splink(a, b, k): """Return a Link containing the first k elements of a, then all of b, then the rest of a.</pre>
	>>> splink(Link(2, Link(3, Link(4, Link(5)))), Link(6, Link(7)), 2) Link(2, Link(3, Link(6, Link(7, Link(4, Link(5)))))) """
	if
	return a
	elifelif
	return

return \_\_\_\_\_

$N_{amo}$ .	5
Name:	

4.	(11)	points)	) Both	Ways
----	------	---------	--------	------

(a) (4 pt) Implement both, which takes two sorted linked lists composed of Link objects and returns whether some value is in both of them. The Link class is defined on the midterm 2 study guide.

Important: You may not use len, in, for, list, slicing, element selection, addition, or list comprehensions.

def both(a, b):

"""Return whether there is any value that appears in both a and b, two sorted Link instances.

```
>>> both(Link(1, Link(3, Link(5, Link(7)))), Link(2, Link(4, Link(6))))
False
>>> both(Link(1, Link(3, Link(5, Link(7)))), Link(2, Link(7, Link(9)))) # both have 7
True
>>> both(Link(1, Link(4, Link(5, Link(7)))), Link(2, Link(4, Link(5)))) # both have 4 and 5
True
if _____:
  return False
if _____:
  a, b = b, a
return _____
```

- (b) (2 pt) Circle the  $\Theta$  expression that describes the minimum number of comparisons (e.g., <, >, <=,==, or >= expressions) required to verify that two sorted lists of length n contain no values in common.
  - $\Theta(n)$  $\Theta(n^2)$  $\Theta(2^n)$  $\Theta(1)$  $\Theta(\log n)$ None of these
- (c) (5 pt) Implement ways, which takes two values start and end, a non-negative integer k, and a list of oneargument functions actions. It returns the number of ways of choosing functions  $f_1, f_2, ..., f_j$  from actions, such that  $f_1(f_2(...(f_i(start))))$  equals end and  $j \le k$ . The same action function can be chosen multiple times. If a sequence of actions reaches end, then no further actions can be applied (see the first example below).

def ways(start, end, k, actions):

"""Return the number of ways of reaching end from start by taking up to k actions.

```
>>> ways(-1, 1, 5, [abs, lambda x: x+2])  # abs(-1) or -1+2, but not abs(abs(-1))
>>> ways(1, 10, 5, [lambda x: x+1, lambda x: x+4]) # 1+1+4+4, 1+4+4+1, or 1+4+1+4
>>> ways(1, 20, 5, [lambda x: x+1, lambda x: x+4])
>>> ways([3], [2, 3, 2, 3], 4, [lambda x: [2]+x, lambda x: 2*x, lambda x: x[:-1]])
3
11 11 11
if _____:
   return 1
```

elif \_\_\_\_\_:

return 0

return \_\_\_\_\_([\_\_\_\_\_\_ for f in actions])

5. (9 points) Autumn Leaves

return s

**Definition.** A pile (of leaves) for a tree t with no repeated leaf labels is a dictionary in which the label for each leaf of t is a key, and its value is the path from that leaf to the root. Each path from a node to the root is either an empty tuple, if the node is the root, or a two-element tuple containing the label of the node's parent and the rest of the path (i.e., the path to the root from the node's parent).

(a) (5 pt) Implement pile, which takes a tree constructed using the tree data abstraction. It returns a pile for that tree. You may use the tree, label, branches, and is\_leaf functions from the midterm 2 study guide. def pile(t):

```
"""Return a dict that contains every path from a leaf to the root of tree t.
   >>> pile(tree(5, [tree(3, [tree(1), tree(2)]), tree(6, [tree(7)])]))
   \{1: (3, (5, ())), 2: (3, (5, ())), 7: (6, (5, ()))\}
   p = \{\}
   def gather(_____):
       if is_leaf(u):
       for b in branches(u):
   return p
(b) (4 pt) Implement Path, a class whose constructor takes a tree t constructed by tree and a leaf_label.
   Assume all leaf labels of t are unique. When a Path is printed, labels in the path from the root to the leaf of
   t with label leaf_label are displayed, separated by dashes. Assume pile is implemented correctly.
class Path:
   """A path through a tree from the root to a leaf, identified by its leaf label.
   >>> a = tree(5, [tree(3, [tree(1), tree(2)]), tree(6, [tree(7)])])
   >>> print(Path(a, 7), Path(a, 2))
   5-6-7 5-3-2
   11 11 11
   def __init__(self, t, leaf_label):
       self.pile, self.end = pile(t), leaf_label
   def __str__(self):
       path, s = ______, ______,
       while path:
          path, s = _____, , _____
```