CS 61A Spring 2019 — 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" × 11" crib sheets of your own creation and the official CS 61A 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	
CalCentral email (`_@berkeley.edu`)	
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.
- You may use built-in Python functions that do not require import, such as `min`, `max`, `pow`, `len`, `abs`, `sum`, `next`, `iter`, `list`, `tuple`, `map`, `filter`, `zip`, `all`, and `any`.
- You **may not** use example functions defined on your study guides unless a problem clearly states you can.
- 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.
- You may use the `Tree`, `Link`, and `BTree` classes defined on Page 2 (left column) of the Midterm 2 Study Guide.

1. (12 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. If an error occurs, write "Error", but include all output displayed before the error. If evaluation would run forever, write "Forever". To display a function value, write "Function". The first two rows have been provided as examples.

The interactive interpreter displays the value of a successfully evaluated expression, unless it is `None`.

Assume that you have first started `python3` and executed the statements on the left.

```
items, n = [], 2

class Airpods:
    cost, k = 200, 0
    f = lambda self: print(self)

    def __init__(self):
        Airpods.k += 1
        Airpods.f(self)
        items.extend([self])

    def __repr__(self):
        return (Airpods.k < 2 and "lonely") or "pair"

class TwoAirpods(Airpods):
    def __init__(self):
        self.k = 2
        Airpods.__init__(self)
        Airpods.__init__(self)

def discount(a):
    a.cost //= 2

def u(w, u):
    return [print(u) for u in [w, u]]

discount(Airpods)
```

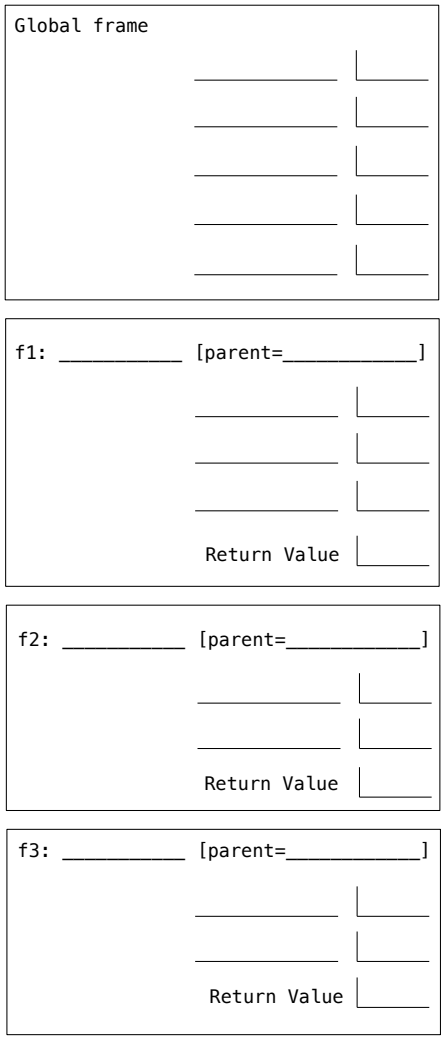
Expression	Interactive Output
`pow(10, 2)`	100
`print(Link(2, Link(3)))`	`<2, 3>`
`TwoAirpods.cost`	
`lost = Airpods()`	
`willbelost = TwoAirpods()`	
`str(lost)`	
`[item.k for item in items]`	
`u(lost, willbelost)`	

2. (10 points) Ultimate

Fill in the environment diagram that results from executing the code on the right 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:

- Use box-and-pointer notation for all lists.
- Add missing names and parents to all local frames.
- Add missing values created or referenced during execution.
- Show the return value for each local frame.

```
1  defeated = ['luigi']
2  def start():
3      chal = 'falco'
4      def approach(new_chal):
5          nonlocal chal
6          defeated.append(chal)
7          chal = new_chal
8      def revive():
9          return defeated.pop()
10     return [approach, revive]
11 gameover = start()[:]
12 gameover.pop(0)('marth')
13 restart = gameover[0]()
```



func start() [parent=Global]

func approach(new_chal) [parent= _____]

func revive() [parent= _____]

3. (5 points) Deep lists

Implement `in_nested` which takes in a value `v` and a nested list or an individual value `L` and returns whether the value is contained in the list.

Hint: The built-in function `type` takes an object and returns the type of that object.

```
def in_nested(v, L):
    """
    >>> in_nested(5, [1, 2, [[3], 4]])
    False
    >>> in_nested(9, [[[1], [6, 4, [5, [9]]], 7], 7, 7])
    True
    >>> in_nested(1, 1)
    True
    """
    if _____:

        return _____

    else:

        return _____
```

4. (8 points) A data structure by any other form would smell just as sweet...

(a) (6 pt) Implement `link_to_dict` which takes a linked list encoding a "flattened" dictionary (in which elements are key1 → value1 → key2 → value2 → key3 → value3, etc), **removes all the values**, and returns the equivalent dictionary. The input and returned list **may** include duplicate keys, as in the example below. You may assume the linked list always contains an even number of elements. The `Link` class is provided below.

```
class Link:
   empty = ()

   def __init__(self, first, rest=empty):
       self.first = first
       self.rest = rest

   def __str__(self):
       string = '<'
       while self.rest is not Link.empty:
           string += str(self.first) + ', '
           self = self.rest
       return string + str(self.first) + '>'

def link_to_dict(L):
    """
    >>> L = Link(1, Link(2, Link(3, Link(4, Link(1, Link(5))))))
    >>> print(L)
    <1, 2, 3, 4, 1, 5>
    >>> link_to_dict(L)
    {1: [2, 5], 3: [4]}
    >>> print(L)
    <1, 3, 1>
    """
    D = {}

    while _____:

        key, value = _____

        if _____:

            _____

        else:

            _____

        _____

    return D
```

(b) (2 pt) Circle the Θ expression that describes the number of iterations of the while loop in `link_to_dict` where `n` is the length of the list.

$\Theta(1)$ $\Theta(\log n)$ $\Theta(n)$ $\Theta(n^2)$ $\Theta(2^n)$ None of these

5. (10 points) I speak for the Trees

Execute each line of code in order. If a line errors, assume we didn't type that line. For each line, indicate whether it was:

- (E) an error
- (D) a Data Abstraction Violation
- (OK) perfectly fine code.

Please fill in the bubbles completely.

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

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def print_tree_adt(t, indent=0):
    print('  ' * indent + str(label(t)))
    for b in branches(t):
        print_tree_adt(b, indent + 1)


class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)

    def __str__(self):
        return '\n'.join(self.indented())

    def indented(self, k=0):
        ind = []
        for b in self.branches:
            for line in b.indented(k + 1):
                ind.append('  ' + line)
        return [str(self.label)] + ind
```

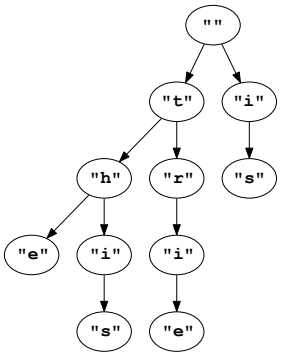
(E)	(D)	(OK)	`t = Tree(3, [Tree(1), Tree(2)])`
(E)	(D)	(OK)	`L = [tree(1)]`
(E)	(D)	(OK)	`L.append(tree(2))`
(E)	(D)	(OK)	`s = tree(3, L)`
(E)	(D)	(OK)	`s`
(E)	(D)	(OK)	`print(s)`
(E)	(D)	(OK)	`t`
(E)	(D)	(OK)	`t.label`
(E)	(D)	(OK)	`print(t)`
(E)	(D)	(OK)	`label(s)`
(E)	(D)	(OK)	`t.label = 4`
(E)	(D)	(OK)	`label(s) = 4`
(E)	(D)	(OK)	`s[0] = 5`
(E)	(D)	(OK)	`print_tree_adt([5])`
(E)	(D)	(OK)	`print_tree_adt(s)`
(E)	(D)	(OK)	`t.branches[0].label = s`
(E)	(D)	(OK)	`print_tree_adt(t.branches[0])`
(E)	(D)	(OK)	`print_tree_adt(t.branches[0].label)`
(E)	(D)	(OK)	`t`
(E)	(D)	(OK)	`print(t)`

6. (15 points) Trie this

A Trie is a Tree where every node in the tree contains a single letter except for the root which is always the empty string. Every path from the root to a leaf forms a word. You may assume no words are substrings of other words in the trie (e.g., "hi" and "him"). The figure below is a trie generated by storing the words ["this", "is", "the", "trie"]. The `Tree` class is defined on Page 2 of the Midterm 2 Study Guide.

(a) (7 pt) Implement `add_word` which takes a Trie and a word and adds the word to the trie.

```
def make_trie(words):
    """ Makes a tree where every node is a letter of a word.
        All words end as a leaf of the tree.
        words is given as a list of strings.
    """
    trie = Tree('')
    for word in words:
        add_word(trie, word)
    return trie
```




```
def add_word(trie, word):

    if _____:
        return

    branch = _____

    for _____:

        if _____:

            branch = _____

    if _____:

        branch = _____

        _____

    _____
```

(b) (8 pt) Implement `get_words`, which takes a Trie and returns a list of all the words the Trie is storing.

```
def get_words(trie):
    """
    >>> get_words(make_trie(['this', 'is', 'the', 'trie']))
    ['this', 'the', 'trie', 'is']
    """
    if _____:

        return _____

    return sum(_____, [])
```

No more questions.