CS 61A Fall 2018 — 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) A/B Test *(All are in Scope: WWPD, Object-Oriented Programming, Linked Lists)*

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.

```
x = [1, 2]

class A:
    x = 3
    y = 4

    def __init__(self, y):
        self.a = y
        self.x = b
        self.__str__ = lambda: str(y)

    def __str__(self):
        return 'BB'

class B(A):
    x = [5, 6]

    def __init__(self, y):
        self.a = x[1]
        y[1] = 8

b = B(x)
a = A(6)

def dash(x):
    return print(self.x)

elastigirl = Link(7, Link(8))
elastigirl.first = elastigirl.rest
```

Expression	Interactive Output
`pow(10, 2)`	100
`print(Link(2, Link(3)))`	`<2, 3>`
`[c.a for c in [a, b]]`	
`print(a.x, b.x)`	
`print(b.y, x)`	
`a.__str__()`	
`dash(b)`	
`print(elastigirl)`	

2. (3 points) *(All are in Scope: Python Lists, Lambda Expressions)* Implement `lowest`, which takes a list of numbers `s` and returns a list of only the elements of `s` with the smallest absolute value. **You may only write a single name in each blank.**

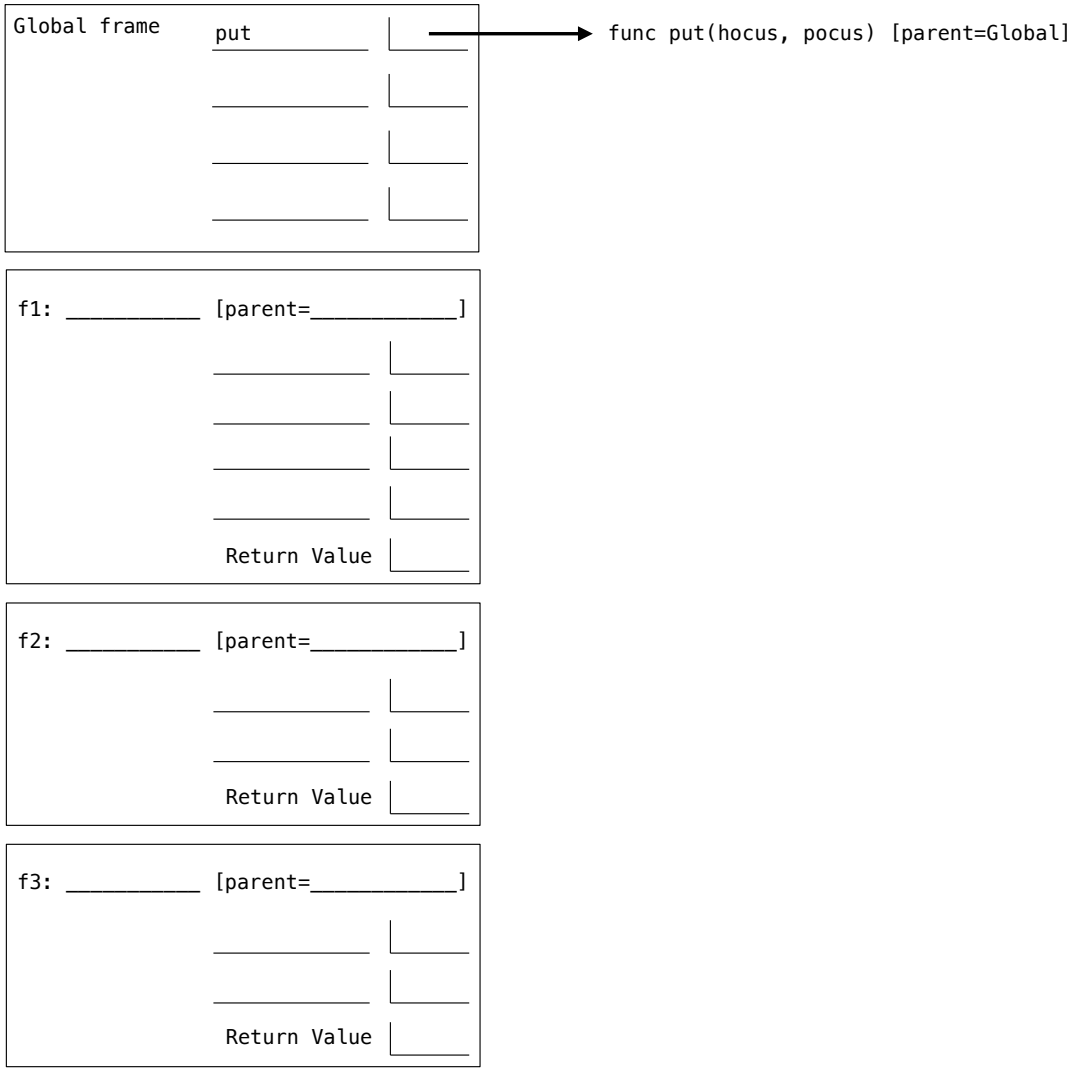
```
def lowest(s):
    """Return a list of the elements in s with the smallest absolute value.

    >>> lowest([3, -2, 2, -3, -4, 2, 3, 4])
    [-2, 2, 2]
    >>> lowest(range(-5, 5))
    [0]
    """
    return (lambda y: [x for ______ in ______ if abs(______)==y])(abs(______(______, key=______)))
```

3. (8 points) Hocus Pocus *(At least one of these is out of Scope: Environment Diagrams, Nonlocal, Python Lists, Mutability)*

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  def put(hocus, pocus):
2      hocus = 2
3      you = pocus[hocus]
4      def pocus():
5          nonlocal hocus
6          if type(spell.pop()) == list:
7              you.append(hocus)
8              return [pocus(), hocus]
9          else:
10             hocus = 3
11             return spell[0:1]
12     return pocus
13 spell = [6, 5, [4]]
14 you = spell
15 put(spell, you)()
```



4. (10 points) Nonplussed

Definition. A *plus expression* for a non-negative integer `n` is made by inserting `+` symbols in between digits of `n`, such that there are **never more than two consecutive digits** in the resulting expression. For example, one plus expression for 2018 is `20+1+8`, and its value is 29. Assume that a two-digit number starting with 0 evaluates to its one's digit. For example, another plus expression for 2018 is `2+01+8`, and its value is 11.

(a) (3 pt) *(All are in Scope: Control, Tree Recursion)* Implement `plus`, which takes a non-negative integer `n`. It returns the largest value of any plus expression for `n`.

```
def plus(n):
    """Return the largest sum that results from inserting +'s into n.

    >>> plus(123456)     # 12 + 34 + 56 = 102
    102
    >>> plus(1604)       # 1 + 60 + 4 = 65
    65
    >>> plus(160450)     # 1 + 60 + 4 + 50 = 115
    115
    """
    if n:

        return ____________________________________________________________________________

    return 0
```

(b) (5 pt) *(All are in Scope: Tree Recursion)* Implement `plusses`, which takes non-negative integers `n` and `cap`. It returns the number of plus expressions for `n` that have a value less than `cap`.

```
def plusses(n, cap):
    """Return the number of plus expressions for n with values below cap.

    >>> plusses(123, 16)  # 1+2+3=6 and 12+3=15, but 1+23=24 isn't below cap.
    2
    >>> plusses(2018, 38) # 2+0+1+8, 20+1+8, 2+0+18, and 2+01+8, but not 20+18.
    4
    >>> plusses(1, 2)
    1
    """
    if ____________________________________________________________________________________:

        return 1

    elif ____________________________________________________________________________________:

        return 0

    else:

        return ____________________________________________________________________________
```

(c) (2 pt) *(All are in Scope: Asymptotic Notation)* Circle the Θ expression that describes how many addition operations are required to evaluate a plus expression for a positive integer n.

$\Theta(1)$ $\Theta(\log n)$ $\Theta(n)$ $\Theta(n^2)$ $\Theta(2^n)$ $\Theta(10^n)$ None of these

5. (8 points) Midterm Elections

(a) (6 pt) *(All are in Scope: Object-Oriented Programming, Mutability)* Implement the `Poll` class and the `tally` function, which takes a choice `c` and returns a list describing the number of votes for `c`. This list contains pairs, each with a name and the number of times `vote` was called on that `choice` at the `Poll` with that name. Pairs can be in any order. Assume all `Poll` instances have distinct names. *Hint*: the dictionary `get(key, default)` method (MT 2 guide, page 1 top-right) returns the value for a `key` if it appears in the dictionary and `default` otherwise.

```
class Poll:
    s = []
    def __init__(self, n):

        self.name = ____________________________________________________________________________

        self.votes = {}

        ____________________________________________________________________________________________

    def vote(self, choice):

        self._________________________ = ____________________________________________________

def tally(c):
    """Tally all votes for a choice c as a list of (poll name, vote count) pairs.

    >>> a, b, c = Poll('A'), Poll('B'), Poll('C')
    >>> c.vote('dog')
    >>> a.vote('dog')
    >>> a.vote('cat')
    >>> b.vote('cat')
    >>> a.vote('dog')
    >>> tally('dog')
    [('A', 2), ('C', 1)]
    >>> tally('cat')
    [('A', 1), ('B', 1)]
    """

    return ____________________________________________________________________________________
```

(b) (2 pt) *(All are in Scope: Object-Oriented Programming)* Implement the `vote` method of the `Crooked` class, which only records every other `vote` call for each `Crooked` instance. Only odd numbered calls to `vote` are recorded, e.g., first, third, fifth, etc.

```
class Crooked(Poll):
    """A poll that ignores every other call to vote.

    >>> d = Crooked('D')
    >>> for s in ['dog', 'cat', 'dog', 'cat', 'cat']:
    ...     d.vote(s)
    >>> d.votes
    {'dog': 2, 'cat': 1}
    """
    record = True
    def vote(self, choice):
        if self.record:

            _________________ . ____________________________(____________________________________)

        self._________________ = _____________________________________________________________________
```

6. (4 points) Dr. Frankenlink *(All are in Scope: Linked Lists)*

Implement `replace`, which takes two non-empty linked lists `s` and `t`, as well as **positive** integers `i` and `j` with `i < j`. It mutates `s` by removing elements with indices from `i` to `j` (removing element i but not removing element `j`) and replacing them with `t`. Afterward, `s` contains all of the objects in `t`, so a change to `t` would be reflected in `s` as well. `t` may change as a result of calling `replace`. Assume `s` has at least `j` elements.

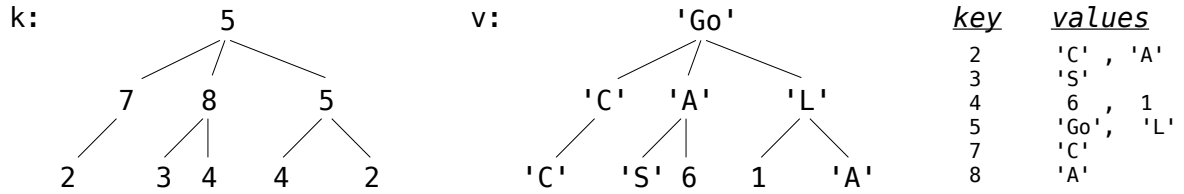
```
def replace(s, t, i, j):
    """Replace the slice of s from i to j with t.

    >>> s, t = Link(3, Link(4, Link(5, Link(6, Link(7))))), Link(0, Link(1, Link(2)))
    >>> replace(s, t, 2, 4)
    >>> print(s)
    <3, 4, 0, 1, 2, 7>
    >>> t.rest.first = 8
    >>> print(s)
    <3, 4, 0, 8, 2, 7>
    """
    assert s is not Link.empty and t is not Link.empty and i > 0 and i < j

    if i > 1:

        ------------------------------------------------------------------------------

    else:

        for k in range(j - i):

            _______________ = ____________________________________________________

        end = t

        while ____________________________________________________________________________:

            end = end.rest

        s.rest, end.rest = ______________________________, ______________________________
```

7. (5 points) Trictionary or Treat *(All are in Scope: Iterators, Generators, Trees)*

Definitions. A *trictionary* is a pair of `Tree` instances `k` and `v` that have identical structure: each node in `k` has a corresponding node in `v`. The labels in `k` are called *keys*. Each key may be the label for multiple nodes in `k`, and the *values* for that key are the labels of all the corresponding nodes in `v`.

A *lookup function* returns one of the values for a key. Specifically, a lookup function for a node in `k` is a function that takes `v` as an argument and returns the label for the corresponding node in `v`.

Implement the generator function `lookups`, which takes a `Tree` instance `k` and some `key`. It yields all *lookup functions* for nodes in `k` that have `key` as their label. The `new_lookup` function is part of the implementation.




```
k = Tree(5, [Tree(7, [Tree(2)]), Tree(8, [Tree(3), Tree(4)]), Tree(5, [Tree(4), Tree(2)])])
v = Tree('Go', [Tree('C', [Tree('C')]), Tree('A', [Tree('S'), Tree(6)]), Tree('L', [Tree(1), Tree('A')])])

def lookups(k, key):
    """Yield one lookup function for each node of k that has the label key.

    >>> [f(v) for f in lookups(k, 2)]
    ['C', 'A']
    >>> [f(v) for f in lookups(k, 3)]
    ['S']
    >>> [f(v) for f in lookups(k, 6)]
    []
    """

    if ____________________________________________________________________________________________:

        yield lambda v: ____________________________________________________________________________

    for i in range(len(k.branches)):

        ____________________________________________________________________________________________:

            yield new_lookup(i, lookup)

def new_lookup(i, f):

    def g(v):

        return ____________________________________________________________________________________

    return g
```

No more questions.