CS 61A
Spring 2019

Structure and Interpretation of Computer Programs

Midterm 2 Practice

INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except one hand-written 8.5" × 11" crib sheet of your own creation and the 2 official 61A midterm study guides attached to the back of this exam.
- Mark your answers ON THE EXAM ITSELF. If you are not sure of your answer you may wish to provide a *brief* explanation.

Last name	
First name	
SID	
Login	
TA & section time	
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)**	

1. (12 points) Class Hierarchy

For each row below, write the output displayed by the interactive Python interpreter when the expression is evaluated. Expressions are evaluated in order, and **expressions may affect later expressions**.

Whenever the interpreter would report an error, write Error. You *should* include any lines displayed before an error. *Reminder*: The interactive interpreter displays the `repr` string of the value of a successfully evaluated expression, unless it is `None`. Assume that you have started Python 3 and executed the following:

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting
class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'My job is to gather wealth'
class Proletariat(Worker):
    greeting = 'Comrade'
    def work(self, other):
        other.greeting = self.greeting + ' ' + other.greeting
        other.work() # for revolution
        return other
jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

Expression	Interactive Output
`5*5`	25
`1/0`	Error
`Worker().work()`	
`jack`	
`jack.work()`	

Expression	Interactive Output
`john.work()[10:]`	
`Proletariat().work(john)`	
`john.elf.work(john)`	

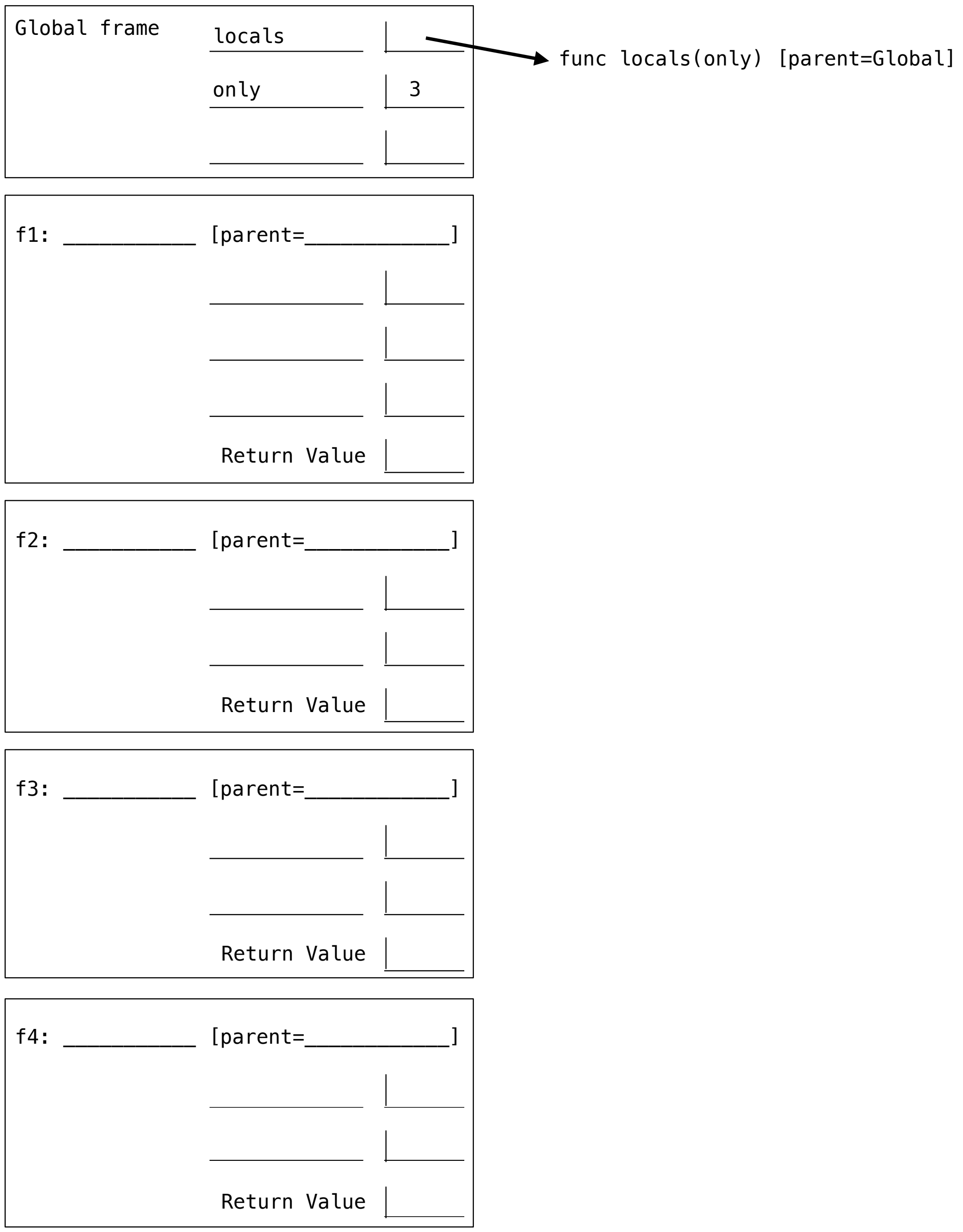
2. (14 points) Space

(a) (8 pt) 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:

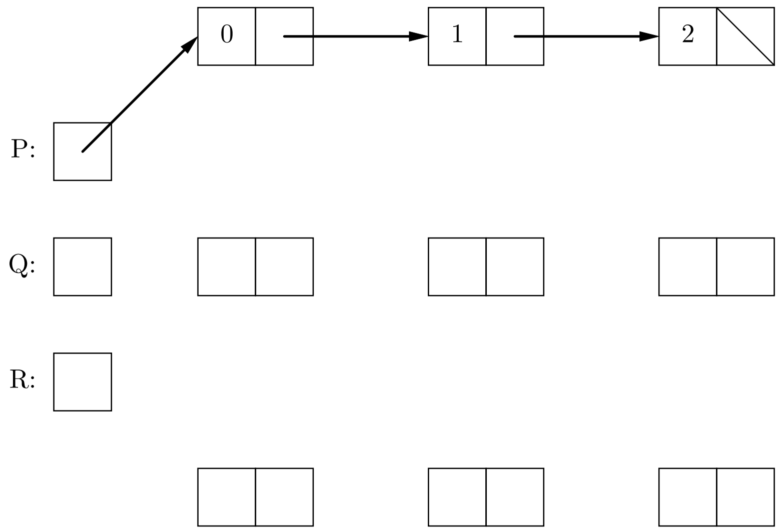
- Add all missing names and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.

```
def locals(only):
    def get(out):
        nonlocal only
        def only(one):
            return lambda get: out
        out = out + 1
        return [out + 2]
    out = get(-only)
    return only

only = 3
earth = locals(only)
earth(4)(5)
```



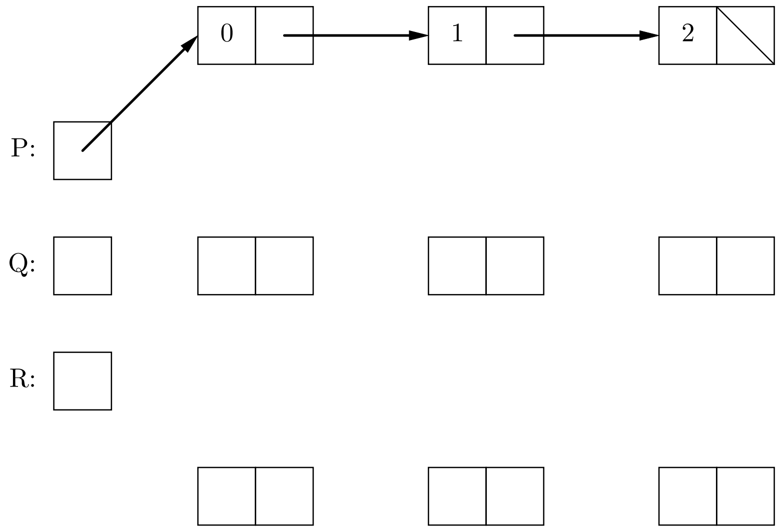
(c) (3 pt) For the next two problems, show the result of executing the code on the left on the initial conditions displayed on the right. We've done the first statement for you in each case, so that the diagrams on the right show the state at the point marked `# START`. Use the empty object skeletons only for newly created `Link` objects. If any pointer is modified, **neatly cross out** the original pointer and draw in the replacement. Show only the final state, not any intermediate states.

```
P = Link(0, Link(1, Link(2)))

# START
def crack1(L):
    if L is Link.empty:
        return (Link.empty, Link.empty)
    L1, L2 = crack1(L.rest)
    return (Link(L.first, L2), L1)
Q, R = crack1(P)
```



(d) (3 pt)

```
P = Link(0, Link(1, Link(2)))

# START
def crack2(L):
    if L is Link.empty:
        return (Link.empty, Link.empty)
    L1, L2 = crack2(L.rest)
    L.rest = L2
    return (L, L1)
Q, R = crack2(P)
```



3. (8 points) This One Goes to Eleven

(a) (4 pt) Fill in the blanks of the implementation of `sixty_ones` below, a function that takes a `Link` instance representing a sequence of integers and returns the number of times that 6 and 1 appear consecutively.

```
def sixty_ones(s):
    """Return the number of times that 1 directly follows 6 in linked list s.

    >>> once = Link(4, Link(6, Link(1, Link(6, Link(0, Link(1))))))
    >>> twice = Link(1, Link(6, Link(1, once)))
    >>> thrice = Link(6, twice)
    >>> apply_to_all(sixty_ones, [Link.empty, once, twice, thrice])
    [0, 1, 2, 3]
    """
    if ________________________________________________________________:

        return 0

    elif ______________________________________________________________:


        return 1 + ____________________________________________________:

    else:

        return ________________________________________________________
```

(b) (4 pt) Fill in the blanks of the implementation of `no_eleven` below, a function that returns a list of all distinct length-`n` lists of ones and sixes in which 1 and 1 do not appear consecutively.

```
def no_eleven(n):
    """Return a list of lists of 1's and 6's that do not contain 1 after 1.

    >>> no_eleven(2)
    [[6, 6], [6, 1], [1, 6]]
    >>> no_eleven(3)
    [[6, 6, 6], [6, 6, 1], [6, 1, 6], [1, 6, 6], [1, 6, 1]]
    >>> no_eleven(4)[:4] # first half
    [[6, 6, 6, 6], [6, 6, 6, 1], [6, 6, 1, 6], [6, 1, 6, 6]]
    >>> no_eleven(4)[4:] # second half
    [[6, 1, 6, 1], [1, 6, 6, 6], [1, 6, 6, 1], [1, 6, 1, 6]]
    """
    if n == 0:

        return ____________________________________________________________

    elif n == 1:

        return ____________________________________________________________

    else:

        a, b = no_eleven(__________________), no_eleven(__________________)


        return [________________ for s in a] + [________________ for s in b]
```

4. (8 points) Tree Time

(a) (4 pt) A `GrootTree` g is a binary tree that has an attribute `parent`. Its parent is the `GrootTree` in which g is a branch. If a `GrootTree` instance is not a branch of any other `GrootTree` instance, then its `parent` is `BinaryTree.empty`.

`BinaryTree.empty` should not have a `parent` attribute. Assume that every `GrootTree` instance is a branch of at most one other `GrootTree` instance and not a branch of any other kind of tree.

Fill in the blanks below so that the `parent` attribute is set correctly. You may not need to use all of the lines. Indentation is allowed. You *should not* include any `assert` statements. Using your solution, the doctests for `fib_groot` should pass. The `BinaryTree` class appears on your study guide.

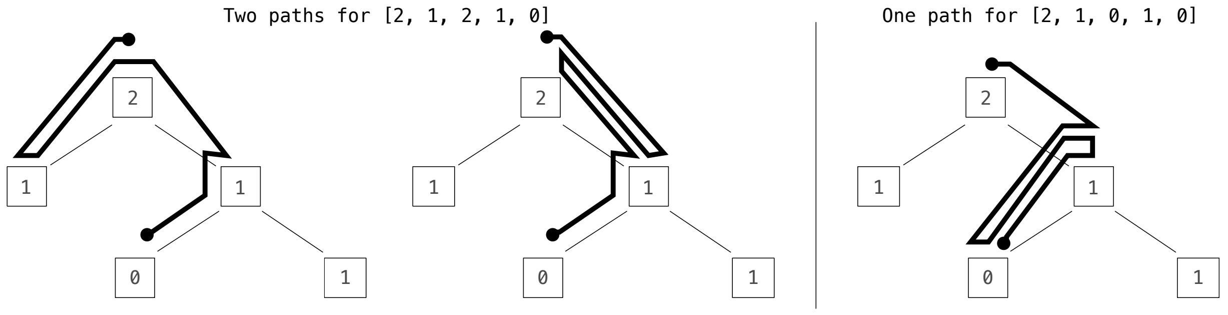
Hint: A picture of `fib_groot(3)` appears on the next page.

```
class GrootTree(BinaryTree):
    """A binary tree with a parent."""

    def __init__(self, entry, left=BinaryTree.empty, right=BinaryTree.empty):
        BinaryTree.__init__(self, entry, left, right)


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


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


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


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


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


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

def fib_groot(n):
    """Return a Fibonacci GrootTree.

    >>> t = fib_groot(3)
    >>> t.entry
    2
    >>> t.parent.is_empty
    True
    >>> t.left.parent.entry
    2
    >>> t.right.left.parent.right.parent.entry
    1
    """
    if n == 0 or n == 1:
        return GrootTree(n)
    else:
        left, right = fib_groot(n-2), fib_groot(n-1)
        return GrootTree(left.entry + right.entry, left, right)
```

(b) **(4 pt)** Fill in the blanks of the implementation of `paths`, a function that takes two arguments: a `GrootTree` instance `g` and a list `s`. It returns the number of paths through `g` whose entries are the elements of `s`. A path through a `GrootTree` can extend either to a branch or its `parent`.

You may assume that the `GrootTree` class is implemented correctly and that the list `s` is non-empty.

The two paths that have entries `[2, 1, 2, 1, 0]` in `fib_groot(3)` are shown below (left). The one path that has entries `[2, 1, 0, 1, 0]` is shown below (right).



```
def paths(g, s):
    """The number of paths through g with entries s.

    >>> t = fib_groot(3)
    >>> paths(t, [1])
    0
    >>> paths(t, [2])
    1
    >>> paths(t, [2, 1, 2, 1, 0])
    2
    >>> paths(t, [2, 1, 0, 1, 0])
    1
    >>> paths(t, [2, 1, 2, 1, 2, 1])
    8
    """

    if g is BinaryTree.empty ______________________________________________:

        return 0

    elif __________________________________________________________________:

        return 1

    else:

        xs = [_______________________________________________________________]

        return sum([ __________________________________________ for x in xs])
```

5. (8 points) Return of the Digits

(a) (4 pt) Implement `complete`, which takes a `Tree` instance `t` and two positive integers `d` and `k`. It returns whether `t` is *`d`-`k`-complete*. A tree is *`d`-`k`-complete* if every node at a depth less than `d` has exactly `k` branches and every node at depth `d` is a leaf. *Notes:* The depth of a node is the number of steps from the root; the root node has depth 0. The built-in `all` function takes a sequence and returns whether all elements are true values: `all([1, 2])` is `True` but `all([0, 1])` is `False`. `Tree` appears on the Midterm 2 Study Guide.

```
def complete(t, d, k):
    """Return whether t is d-k-complete.

    >>> complete(Tree(1), 0, 5)
    True
    >>> u = Tree(1, [Tree(1), Tree(1), Tree(1)])
    >>> [ complete(u, 1, 3)  ,  complete(u, 1, 2)  ,  complete(u, 2, 3) ]
    [True, False, False]
    >>> complete(Tree(1, [u, u, u]), 2, 3)
    True
    """
    if not t.branches:

        return ______________________________________________________________

    bs = [____________________________________________________________________]

    return ______________________________________________________ and all(bs)
```

(b) (4 pt) Implement `adder`, which takes two lists `x` and `y` of digits representing positive numbers. It mutates `x` to represent the result of adding `x` and `y`. *Notes:* The built-in `reversed` function takes a sequence and returns its elements in reverse order. Assume that `x[0]` and `y[0]` are both positive.

```
def adder(x, y):
    """Adds y into x for lists of digits x and y representing positive numbers.

    >>> a = [3, 4, 5]
    >>> adder(a, [5, 5])           #  345 +    55 =   400
    [4, 0, 0]
    >>> adder(a, [8, 3, 4])        #  400 +   834 =  1234
    [1, 2, 3, 4]
    >>> adder(a, [3, 3, 3, 3, 3])  # 1234 + 33333 = 34567
    [3, 4, 5, 6, 7]
    """
    carry, i = 0, len(x)-1
    for d in reversed([0] + y):


        if __________________________________________________________________:

            x.insert(0, 0)
            i = 0
        d = carry + x[i] + d

        ____________________________________________________________________

    if x[0] == 0:
        x.remove(0)
    return x
```

Post-Exam Reflection

Which problem type did you lose the majority of points on?
(e.g. What Would Python Print, Environment Diagram, Coding, etc.)

For each problem that you did not receive full credit for, please answer the following questions (You can either answer for an entire problem, or for each problem subsection individually- it's up to you. Under each prompt, you can list your responses for each problem consecutively.):

1. After looking over the solution can you explain why the solution is correct?

2. Are any parts of your original answer on the right track? If so, which parts?

3. What led you to lose points on this problem? (In other words, what did you get stuck on or what did you miss?)

4. What are key ideas here that could be helpful in future problems?