

# CS 61A

## Fall 2014

# Structure and Interpretation of Computer Programs

MIDTERM 2 SOLUTIONS

## 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"  $\times$  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	
<i>All the work on this exam is my own. (please sign)</i>	

## For staff use only

Q. 1	Q. 2	Q. 3	Q. 4	Q. 5	Total
/12	/14	/8	/8	/8	/50

Blank Page

### 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'
```

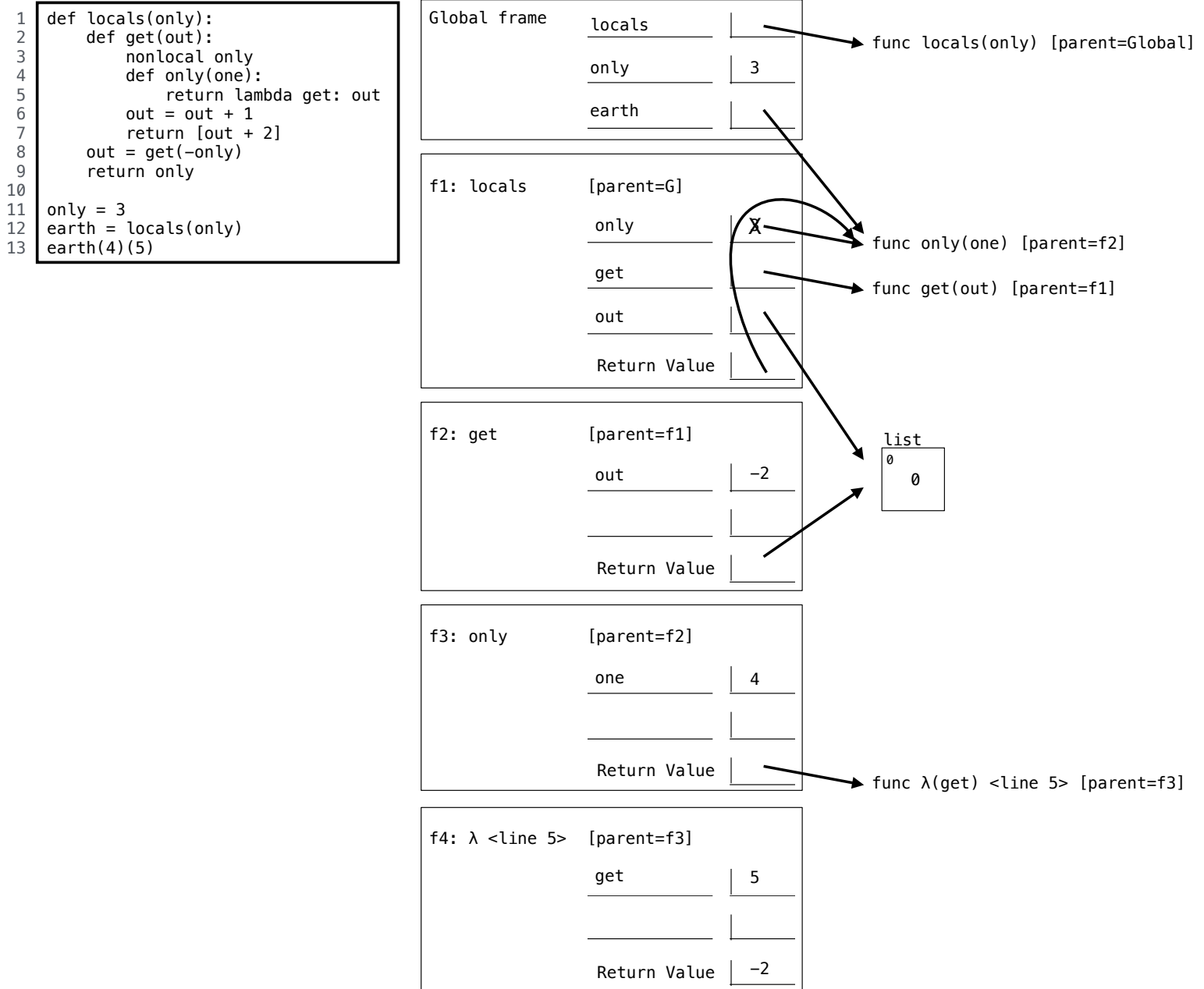
Expression	Interactive Output	Expression	Interactive Output
5*5	25	john.work()[10:]	Peon, I work 'to gather wealth'
1/0	ERROR		
Worker().work()	'Sir, I work'	Proletariat().work(john)	Comrade Peon, I work Peon
jack	Peon	john.elf.work(john)	'Comrade Peon, I work'
jack.work()	'Maam, I work'		

## 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.



- (b) (6 pt) Fill in the blanks with the shortest possible expressions that complete the code in a way that results in the environment diagram shown. You can use only brackets, commas, colons, and the names `luke`, `spock`, and `yoda`. You **\*cannot\*** use integer literals, such as 0, in your answer! You also cannot call any built-in functions or invoke any methods by name.

```

spock, yoda = 1, 2

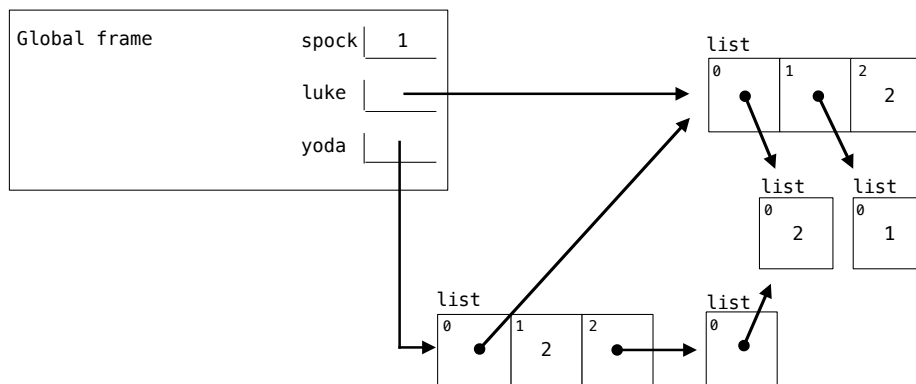
luke = [[yoda], [spock], yoda]

yoda = 0

yoda = [luke, luke[yoda][yoda]]

yoda.append(luke[:spock])

```



### 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 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 s is Link.empty or s.rest is Link.empty:

        return 0

    elif s.first == 6 and s.rest.first == 1:

        return 1 + sixty_ones(s.rest.rest)

    else:

        return sixty_ones(s.rest)
```

- (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 [[6], [1]]

    else:

        a, b = no_eleven(n-1), no_eleven(n-2)

        return [[6] + s for s in a] + [[1, 6] + s 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)

        self.parent = BinaryTree.Empty

        for b in [left, right]:

            if b is not BinaryTree.empty:

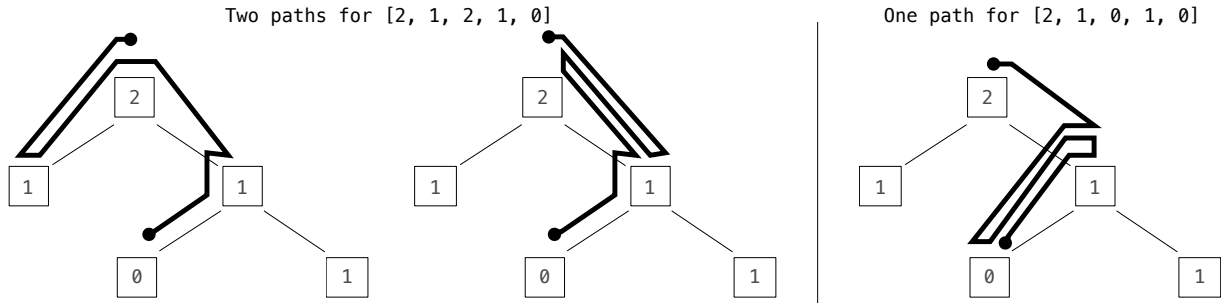
                b.parent = self

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.entry
    1
    >>> 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)
```

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 or s == [] or g.entry != s[0]:

        return 0

    elif len(s) == 1 and g.entry == s[0]:

        return 1

    else:

        extensions = [g.left, g.right, g.parent]

        return sum(paths(x, s[1:]) for x in extensions)
```



## 5. (8 points) Abstraction and Growth

- (a) (6 pt) Your project partner has invented an abstract representation of a sequence called a **slinky**, which uses a **transition** function to compute each element from the previous element. A **slinky** explicitly stores only those elements that cannot be computed by calling **transition**, using a **starts** dictionary. Each entry in **starts** is a pair of an index key and an element value. See the doctests for examples.

Help your partner fix this implementation by crossing out as many lines as possible, but leaving a program that passes the doctests. Do not change the doctests. The program continues onto the following page.

```
def length(slinky):
    return slinky[0]
def starts(slinky):
    return slinky[1]
def transition(slinky):
    return slinky[2]

def slinky(elements, transition):
    """Return a slinky containing elements using transition.

    >>> s = slinky(range(3, 10), lambda x: x+1)
    >>> length(s)
    7
    >>> starts(s)
    {0: 3}
    >>> get(s, 2)
    5
    >>> t = slinky([2, 4, 10, 20, 40], lambda x: 2*x)
    >>> starts(t)
    {0: 2, 2: 10}
    >>> get(t, 3)
    20
    >>> slinky([], abs)
    [0, {}, <built-in function abs>]
    >>> slinky([5, 4, 3], abs)
    [3, {0: 5, 1: 4, 2: 3}, <built-in function abs>]
    """
    starts = {}
    for index in range(len(elements)):
        if index == 0 or elements[index] != transition(elements[index-1]):
            starts[index] = elements[index]
    return [len(elements), starts, transition]
```

```
def get(slinky, index):
    """Return the element at index of slinky."""
    start = index
    while start not in starts(slinky):
        start = start - 1
    value = starts(slinky)[start]
    while start < index:
        value = transition(slinky)(value)
        start = start + 1
    return value
```

(b) (2 pt) Circle the  $\Theta$  expression below that describes the number of operations required to compute `slinky(elements, transition)`, assuming that

- $n$  is the initial length of `elements`,
- $d$  is the final length of the `starts` dictionary created,
- the `transition` function requires constant time,
- the `pop` method of a list requires constant time,
- the `len` function applied to a `list` requires linear time,
- the `len` function applied to a `range` requires constant time,
- adding or updating an entry in a dictionary requires constant time,
- getting an element from a list by its index requires constant time,
- creating a list requires time that is proportional to the length of the list.

$\Theta(1)$        $\Theta(n)$        $\Theta(d)$        $\Theta(n^2)$        $\Theta(d^2)$        $\Theta(n \cdot d)$