CS 61A Fall 2015

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" × 11" crib sheet 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	
BearFacts email (`_@berkeley.edu`)	
TA	
All the work on this exam is my own. **(please sign)**	

1. (12 points) `ok --submit`

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 more than 3 lines are displayed, just write the first 3. If an error occurs, write "Error". If evaulation would run forever, write "Forever".

The first two rows have been provided as examples.

Assume that you have started `python3` and executed the following statements (which do not cause errors):

```
class Ok:
    py = [3.14]
    def __init__(self, py):
        self.ok = self.py
        Ok.py.append(3 * py)
    def my(self, eye):
        print(self.my(eye))
        return self.ok.pop()
    def __str__(self):
        return str(self.ok)[:4]

class Go(Ok):
    def my(self, help):
        return [help+3, len(Ok.py)]

oh = Go(5)
Go.py = [3, 1, 4]
oh.no = {'just': Go(9)}
```

Expression	**Interactive Output**
`'z' * 3`	`'zzz'`
`print(4, 5) + 1`	4 5 Error
`oh.py`	[3, 1, 4]
`oh.my(3)`	[6, 3]
`oh.ok + oh.no['just'].ok`	[3.14, 15, 27, 3, 1, 4]
`print(oh)`	[3.1
`Ok('go').my(5)`	"Error"
`Ok.my(oh, 5)`	[8, 4] `'gogogo'`

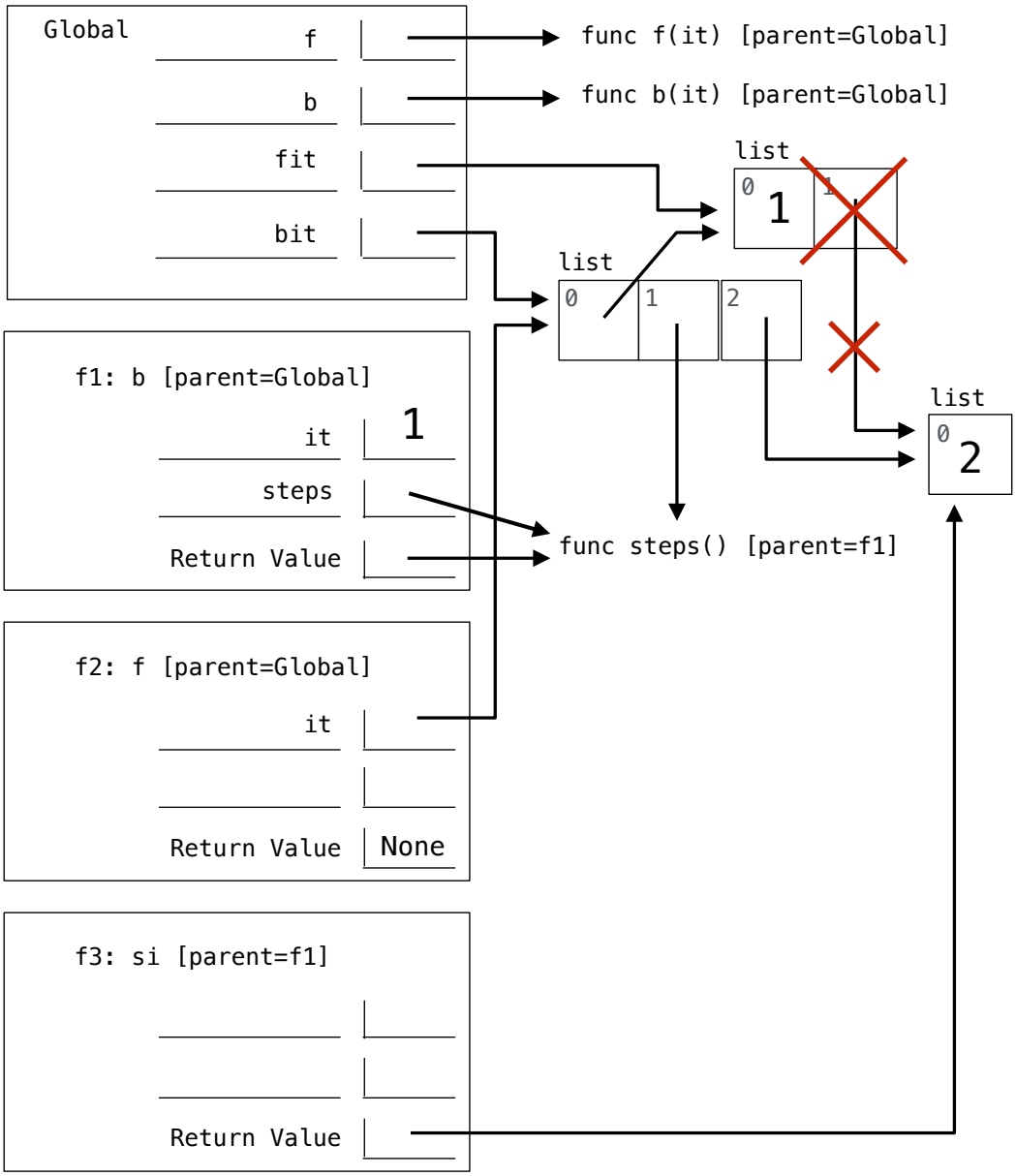
2. (14 points) Exercises

(a) (6 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 or referenced during execution.
- Show the return value for each local frame.

You are not required to write index numbers in list boxes.

```
1  def f(it):
2      it.append(it[1]())
3
4  def b(it):
5      def steps():
6          nonlocal it
7          it = fit[0]
8          return fit.pop()
9      return steps
10
11 fit = [1, [2]]
12 bit = [fit, b(fit[1])]
13 f(bit)
```



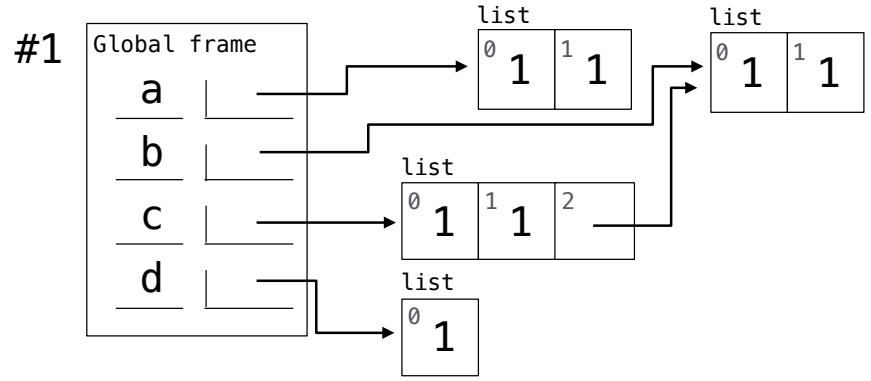
(b) (8 pt) In each box next to a line of code below, write the **number** of the environment diagram that would result **after** executing that line **and** complete the diagram by adding all missing values (boxes and arrows). *Suggestion*: You may want to write out the whole diagram elsewhere (scratch paper or the bottom left of this page) before filling in the answer area.

Write numbers (1, 2, or 3) in these boxes

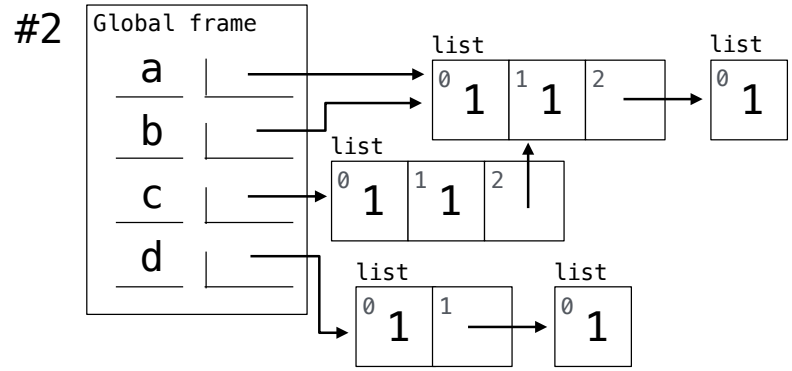
```
a = [1, 1]
b = [1, 1]
c = a + [b]
d = c[1:2]
while a:
    b.extend([[a.pop()]])
    d, b = b, d
a = b
b[2][0], d = c, b[2]
```

Complete these diagrams

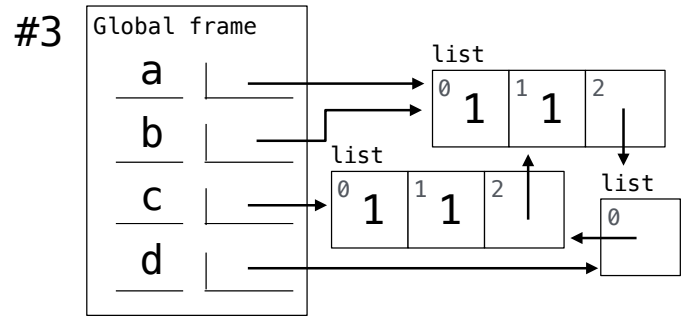
#1



#2



#3



3. (24 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 d == 0

    bs = [complete(b, d-1, k) for b in t.branches]

    return len(t.branches) == k 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]
    >>> b = [9, 9, 9, 4, 5]
    >>> adder(b, [5, 5])           #  945 +    55 =   400
    [1, 0, 0, 0, 0, 0]
    >>> c = [9, 4, 5]
    >>> adder(c, [5, 5])           #  945 +    55 =   400
    [1, 0, 0, 0]
    >>> d = [5, 5]
    >>> adder(d, [9, 9, 9, 4, 5])
    [1, 0, 0, 0, 0, 0]
    """
    carry, i = 0, len(x)-1
    for d in reversed([0] + y):
        if i == -1:
            x.insert(0, 0)
            i = 0
        d = carry + x[i] + d
        carry, x[0 if len(x)-i==len(y)+1 and d>=10 else i:i+1], i = d // 10, adder(x[0:i+1], [0] * i
    if x[0] == 0:
        x.remove(0)
    return x
```

(c) (6 pt) Implement `multiples`, which takes a positive integer `k` and a linked list `s` of digits **greater than 0 and less than 10**. It returns a linked list of all positive `n` that are multiples of `k` greater than `k` *and* made up of digits only from `s`. The digits in each `n` must appear in the same order as they do in `s`, and each digit from `s` can appear only once in each `n`. The `Link` class appears on the Midterm 2 Study Guide.

```
def multiples(k, s):
    """Return a linked list of all multiples of k selected from digits in s.

    >>> odds = Link(1, Link(3, Link(5, Link(7, Link(9)))))
    >>> multiples(5, odds)
    Link(135, Link(15, Link(35)))
    >>> multiples(7, odds)
    Link(1379, Link(357, Link(35)))
    >>> multiples(9, odds)
    Link(1359, Link(135))
    >>> multiples(2, odds)
    ()
    """

    t = Link.empty

    def f(n, s):


        nonlocal t

        if s is Link.empty:


            if n > k and n % k == 0:


                t = Link(n, t)

        else:


            f(n, s.rest)

            f(n*10+s.first, s.rest)

    f(0, s)

    return t
```

(d) (2 pt) Circle the Θ expression that describes the length of the linked list returned by `multiples(1, s)` for an input list of length `n`. Assume `multiples` is implemented correctly.

$\Theta(1)$ $\Theta(\log n)$ $\Theta(n)$ $\Theta(n^2)$ $\Theta(2^n)$

Note: $\Theta(1)$ was also accepted if accompanied by the justification that `s` would not contain repeated digits.

(e) **(8 pt)** Implement `int_set`, which is a higher-order function that takes a list of non-negative integers called `contents`. It returns a function that takes a non-negative integer `n` and returns whether `n` appears in `contents`. Your partner left you this clue: Every integer can be expressed uniquely as a sum of powers of 2. E.g., 5 equals 1 + 4 equals `pow(2, 0) + pow(2, 2)`. The `bits` helper function encodes a list of `nums` using sequences of 0's and 1's that tell you whether each power of 2 is used, starting with `pow(2, 0)`.
Note: You may *not* use built-in tests of list membership, such as an `in` expression or a list's `index` method.

```
def bits(nums):
    """A set of nums represented as a function that takes 'entry', 0, or 1.

    >>> t = bits([4, 5]) # Contains 4 and 5, but not 2
    >>> t(0)(0)(1)('entry') # 4 = 0 * pow(2, 0) + 0 * pow(2, 1) + 1 * pow(2, 2)
    True
    >>> t(0)(1)('entry')    # 2 = 0 * pow(2, 0) + 1 * pow(2, 1)
    False
    >>> t(1)(0)(1)('entry') # 5 = 1 * pow(2, 0) + 0 * pow(2, 1) + 1 * pow(2, 2)
    True
    """
    def branch(last):

        if last == 'entry':

            return 0 in nums

        return bits([k // 2 for k in nums if k % 2 == last])

    return branch

def int_set(contents):
    """Return a function that represents a set of non-negative integers.

    >>> int_set([1, 2])(1) ,  int_set([1, 2])(3)  # 1 in [1, 2] but 3 is not
    (True, False)
    >>> s = int_set([1, 3, 4, 7, 9])
    >>> [s(k) for k in range(10)]
    [False, True, False, True, True, False, False, True, False, True]
    """

    index = bits(contents)

    def contains(n):

        t = index

        while n:

            last, n = n % 2, n // 2

            t = t(last)

        return t('entry')

    return contains
```

4. (0 points) Back to the Future

Draw a picture of what would happen if Fibonacci traveled through time to October 22, 2015.