CS 61A Structure and Interpretation of Computer Programs

Fall 2012

Final Examination Solutions

INSTRUCTIONS

- You have 3 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 three official 61A 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)**	

For staff use only

Q. 1	Q. 2	Q. 3	Q. 4	Total
/20	/16	/30	/14	/80

THIS PAGE INTENTIONALLY LEFT BLANK

1. (20 points) Bank Rewrite

(a) (10 pt) For each of the following expressions, write the `repr` string of the value to which the expression evaluates. Special cases: If an expression evaluates to a function, write FUNCTION. If evaluation would never complete, write FOREVER. If an error would occur, write ERROR.

Assume that the expressions are evaluated in order. Evaluating the first may affect the value of the second, etc.

Assume that you have started Python 3 and executed the following statements:

```
jan = [1, 3, 5]
feb = [3, 5, 7]

def mar(apr, may):
    if not apr or not may:
        return []
    if apr[0] == may[0]:
        return mar(apr[1:], may[1:]) + [apr[0]]
    elif apr[0] < may[0]:
        return mar(apr[1:], may)
    else:
        return mar(apr, may[1:])
```

Expression	Evaluates to
`5*5`	25
`feb[jan[0]]`	5
`mar(jan, feb)`	[5, 3]
`jan`	[1, 3, 5]
`next(iter(jan))`	1
`len(mar(range(5, 50), range(20, 200)))`	30

(b) (10 pt) For each of the following expressions, write the `repr` string of the value to which the expression evaluates. Special cases: If an expression evaluates to a function, write FUNCTION. If evaluation would never complete, write FOREVER. If an error would occur, write ERROR.

Assume that you have started Python 3 and executed the following statements **after executing the Stream class statement from the Final Exam Study Guide**:

```
from operator import add, mul

def stone(a):
    return Stream(a, lambda: stone(a+1))
rock = stone(3)

def lava(x, y, z):
    def magma():
        return lava(x.rest, y.rest, z)
    volcano = z(x.first, y.first)
    return Stream(volcano, magma)
fire = lava(rock, rock.rest, mul)

def hot():
    crater = Stream(0, lambda: lava(crater, rock, add))
    return crater
ash = hot()
```

Expression	Evaluates to
`(1, rock.first)`	(1, 3)
`(rock.rest.first, rock.rest.rest.first)`	(4, 5)
`(fire.first, fire.rest.first)`	(12, 20)
`fire.rest is fire.rest`	True
`(ash.first, ash.rest.first)`	(0, 3)
`ash.rest.rest.rest.first`	12

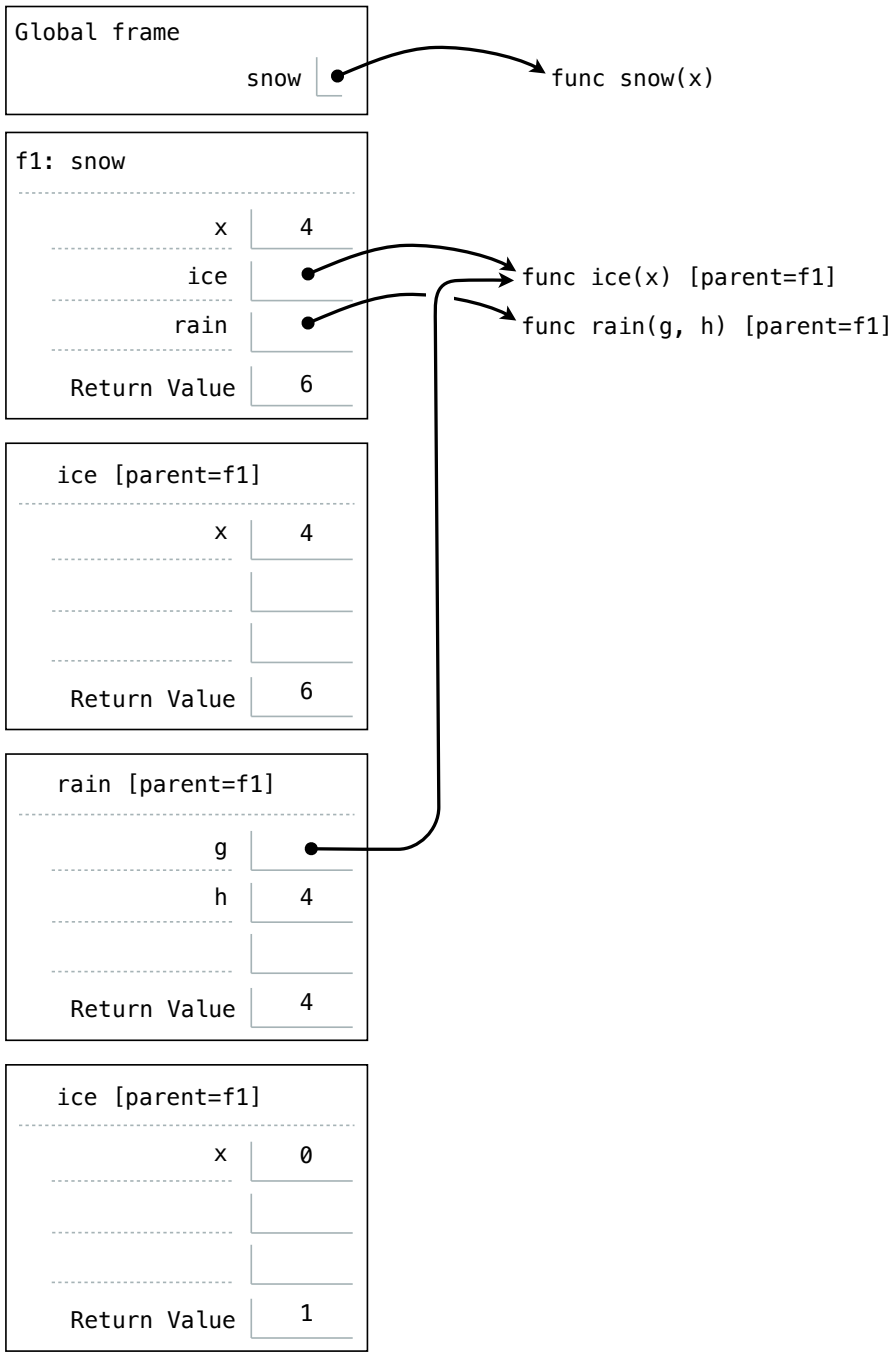
2. (16 points) Web Rater Ink

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

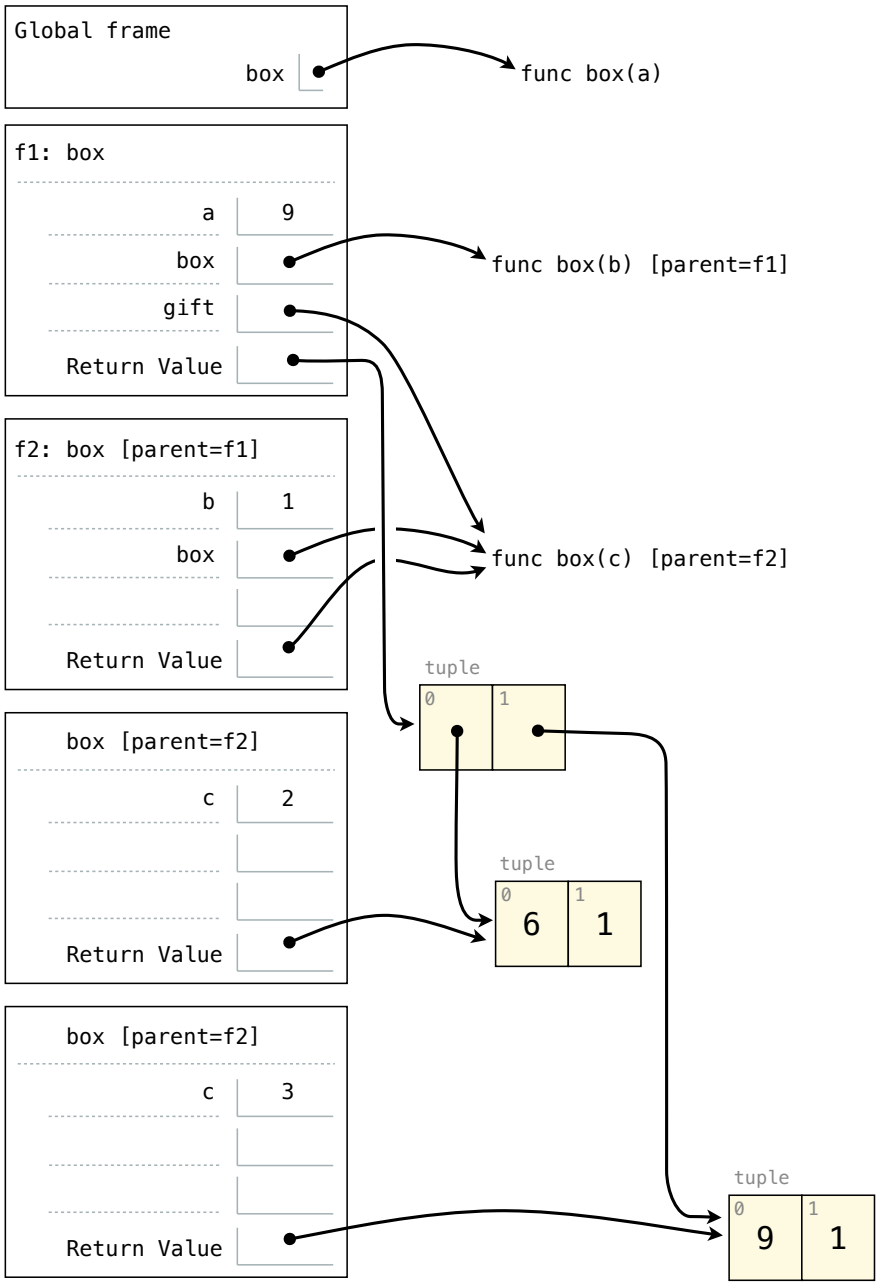
```
def snow(x):
    def ice(x):
        if x == 0:
            return 1
        return 2 + rain(ice, x)
    def rain(g, h):
        return 3 + g(h-x)
    return ice(x)
snow(4)
```



(b) **(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, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.

```
def box(a):
    def box(b):
        def box(c):
            nonlocal a
            a = a + c
            return (a, b)
        return box
    gift = box(1)
    return (gift(2), gift(3))
box(4)
```



3. (30 points) Twin Breaker

(a) (8 pt) Run-length encoding (RLE) is a technique used to compress sequences that contain repeated elements. For example, the sequence $1, 1, 1, 4, 2, 2, 2, 2$ would be encoded as three 1's, one 4, and four 2's. Fill in the blanks in the `RLE` class below, so that all doctests pass.

```
class RLE(object):
    """A run-length encoding of a sequence.

    >>> RLE([2, 2, 2, 2, 2, 7]).runs
    [(5, 2), (1, 7)]
    >>> s = RLE([1, 1, 1, 4, 2, 2, 2, 2])
    >>> s.runs
    [(3, 1), (1, 4), (4, 2)]
    >>> len(s)
    8
    >>> s[2], s[3], s[4], s[5]
    (1, 4, 2, 2)
    """
    def __init__(self, elements):

        last, count = None, 0

        self.runs = []

        for elem in elements:

            if last != elem and count > 0:

                self.runs.append((count, last))

            if last != elem:

                last, count = elem, 1

            else:

                count += 1

        self.runs.append((count, last))

    def __len__(self):

        return sum(pair[0] for pair in self.runs)

    def __getitem__(self, k):

        run = 0

        while k >= self.runs[run][0]:

            k, run = k-self.runs[run][0], run+1

        return self.runs[run][1]
```

(b) (6 pt) A path through a tree is a sequence of connected nodes in which each node appears at most once. The height of a tree is the longest path starting at the root. Fill in the blanks in the calls to `max` below, so that all doctests pass. Write each operand expression on a separate line. **You may not need to use all of the blank lines.** This question uses the `Tree` class statement from the Midterm 2 Study Guide. You may assume that `height` works correctly when implementing `longest`.

```
s = Tree(0, Tree(1, Tree(2, Tree(3), Tree(4))))
t = Tree(5, Tree(6, Tree(7, s, Tree(8)), Tree(9, None, Tree(10, s))))

def height(tree):
    """Return the length of the longest path from the root to a leaf.

    >>> height(None)
    0
    >>> height(s)
    4
    >>> height(t)
    8
    """

    if tree is None:

        return 0

    return 1 + max(height(tree.left),

                   height(tree.right)

                   )

def longest(tree):
    """Return the length of the longest path from any two nodes.

    >>> longest(None)
    0
    >>> longest(Tree(5))
    1
    >>> [longest(b) for b in (s.left.left, s.left, s)]
    [3, 3, 4]
    >>> longest(t)
    12
    """

    if tree is None:

        return 0

    return max(longest(tree.left),

               longest(tree.right),

               1 + height(tree.left) + height(tree.right)

               )
```

(c) (8 pt) Given a set of unique positive integers `s` and a maximum sum `m`, the `pack` function returns a subset of `s` with the largest sum less than or equal to `m`. Fill in the blanks below, so that all doctests pass. *Assume that sets are printed in sorted order, regardless of how they are constructed.*

```
def pack(s, m):
    """Return the subset of s with the largest sum up to m.

    >>> s = [4, 1, 3, 5]
    >>> pack(s, 7)
    {3, 4}
    >>> pack(s, 6)
    {1, 5}
    >>> pack(s, 11)
    {1, 4, 5}
    """
    if len(s) == 0:

        return set()

    if s[0] > m:

        return pack(s[1:], m)

    with_s0 = {s[0]}.union(pack(s[1:], m-s[0]))

    without = pack(s[1:], m)

    if sum(with_s0) > sum(without):

        return with_s0

    else:

        return without
```

(d) **(8 pt)** Cross out lines from the implementation of the `IterableTree` class below so that all doctests pass **and the implementation contains as few lines of code as possible**. Don't cross out any docstrings or doctests.

The `__iter__` generator for this class should yield the entries of the tree (and each subtree) starting with the root, and yield all of the entries of the left branch before any of the entries of the right branch. This question uses the `Tree` class statement from the Midterm 2 Study Guide.

```
class IterableTree(object):

class IterableTree(Tree):

    def __init__(self, entry, left=None, right=None):

        Tree.__init__(entry, left, right)

        Tree.__init__(self, entry, left, right)

    def __iter__(self):
        """Yield the entries of this tree.

        >>> T = IterableTree
        >>> t = T('A', T(2, T('C'), T(4)), T('E', None, T(6)))
        >>> list(t)
        ['A', 2, 'C', 4, 'E', 6]
        """

        yield self.entry

        yield entry

        for branch in (self.left, self.right):

            if branch:

            if self.branch:

                branch = iter(branch)

                for entry in branch:

                for entry in branch():

                    yield self.entry

                    yield entry

        yield self.entry

        yield entry
```

4. (14 points) Winter Break

(a) (2 pt) Write the value of the Scheme expression `(f 7)` after evaluating the define expressions below?

```
(define j
  (mu (c k)
    (if (< c n)
        (j (+ c 2) (- k 1))
        k)))
(define (f n)
  (define c n)
  (j 0 0))
```

-4

(b) (2 pt) Circle (*True* or *False*): Every call to `j` above is a tail call.

(c) (4 pt) In your project 4 implementation, how many total calls to `scheme_eval` and `scheme_apply` would result from evaluating the following **two expressions**? Assume that you **are not** using the tail call optimized scheme_eval_optimized function for evaluation.

```
(define (square x) (* x x))
(+ (square 3) (- 3 2))
```

Calls to scheme_eval (circle one): 2 5 14 24

Calls to scheme_apply (circle one): 1 2 3 4

(d) **(4 pt)** Fill in two facts below to complete the definitions of the relations `reversed` and `palindrome`. The `reversed` relation indicates that the first list contains the same elements as the second, but in reversed order. The `palindrome` relation indicates that a list is the same backward and forward.

```
logic> (fact (append-to-form () ?x ?x))

logic> (fact (append-to-form (?a . ?r) ?y (?a . ?z))
             (append-to-form ?r ?y ?z))

logic> (fact (reversed () ()))

logic> (fact (reversed (?a . ?r) ?s)
             (reversed ?r ?rev)

             (append-to-form ?rev (?a) ?s))

logic> (query (reversed ?x (a b c d)))
Success!
x: (d c b a)

logic> (fact (palindrome ?s)

             (reversed ?s ?s))

logic> (query (palindrome (a b ?x d e ?y ?z))
Success!
x: e    y: b    z: a
```

(e) **(2 pt)** Define a simple mathematical function $f(n)$ such that calling `m(n)` on **positive integer** `n` prints $\Theta(f(n))$ **lines of output**.

```
def m(n):
    g(n)
    if n <= 2:
        print('The')
    else:
        m(n//3)

def g(n):
    if n == 42:
        print('Last')
    if n <= 0:
        print('Question')
    else:
        g(n-1)
```

$f(n) = \log_3(n)$