CS 61A Structure and Interpretation of Computer Programs

Spring 2013

FINAL 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 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	Q. 5	Q. 6	Q. 7	Total
/12	/16	/8	/7	/8	/16	/13	/80

1. (12 points) We Are Binary Tree Huggers

This problem makes use of the `Tree` class from lecture; its definition is contained in the Midterm 2 Study Guide, attached to the end of this exam.

(a) The *depth* of a tree is defined as the number of nodes encountered in the longest path from the root to a leaf. Complete the function definition below to compute the depth of a binary tree.

```
def depth(tree):
    """Compute the depth of binary a tree.

    >>> t = Tree(6, Tree(2, Tree(1)), Tree(7))
    >>> depth(t)
    3
    >>> t.left.right = Tree(4, Tree(3), Tree(5))
    >>> depth(t)
    4
    """
    if tree:
        return 1 + max(depth(tree.left), depth(tree.right))
    return 0
```

(b) A binary tree is *balanced* if for every node, the depth of its left subtree differs by at most 1 from the depth of its right subtree. Fill in the definition of the `is_balanced` function below to determine whether or not a binary tree is balanced. You may assume that the `depth` function works correctly for this part.

```
def is_balanced(tree):
    """Determine whether or not a binary tree is balanced.

    >>> t = Tree(6, Tree(2, Tree(1)), Tree(7))
    >>> is_balanced(t)
    True
    >>> t.left.right = Tree(4, Tree(3), Tree(5))
    >>> is_balanced(t)
    False
    """
    if not tree:
        return True
    return (is_balanced(tree.left) and is_balanced(tree.right) and
           -1 <= depth(tree.left) - depth(tree.right) <= 1)
```

(c) For the following class definition, cross out any incorrect or unnecessary lines in the following code so that the doctests pass. **Do not cross out class declarations, doctests, or docstrings.** You can cross out anything else, including method declarations, and your final code should be as compact as possible. **Make sure to cross out the entire line for anything you wish to remove.** You may assume that the `depth` and `is_balanced` functions are defined correctly in the global environment.

```
class STree(Tree):
    """A smart tree that knows its depth and whether or not it is
    balanced.

    >>> s = STree(6, STree(2, STree(1)), STree(7))
    >>> s.depth
    3
    >>> s.is_balanced
    True
    >>> s.left.right = STree(4, STree(3), STree(5))
    >>> s.depth
    4
    >>> s.is_balanced
    False
    """
    @property
    def depth(self):
        return depth(self)

    @property
    def is_balanced(self):
        return is_balanced(self)
```

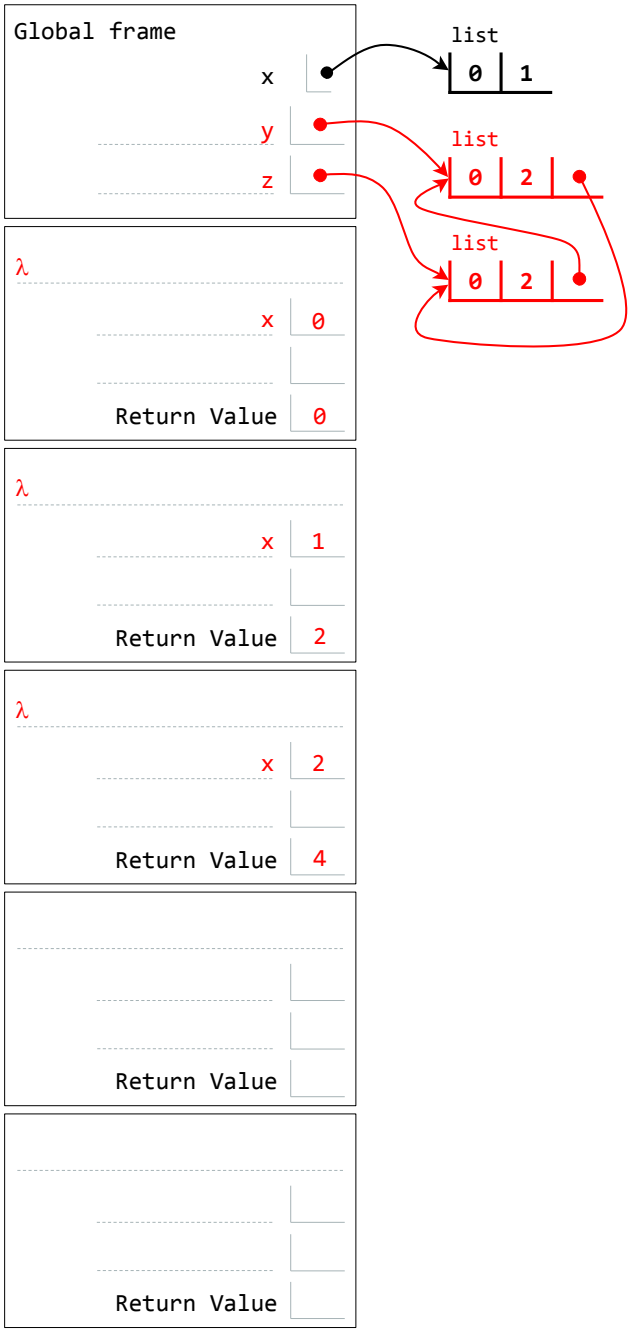
2. (16 points) Binary Tree Huggers are Environmentalists

(a) (7 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 need only show the final state of each frame. *You may not need to use all of the spaces or frames.* You may draw objects that are created but are not accessible from the environment, if you wish. Make sure to reflect every call to a user-defined function in the environment diagram.

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.

```
x = [0, 1]
y = list(map(lambda x: 2*x,
             x + [2]))
y[2] = y
z = y[:]
z[2][2] = z
```



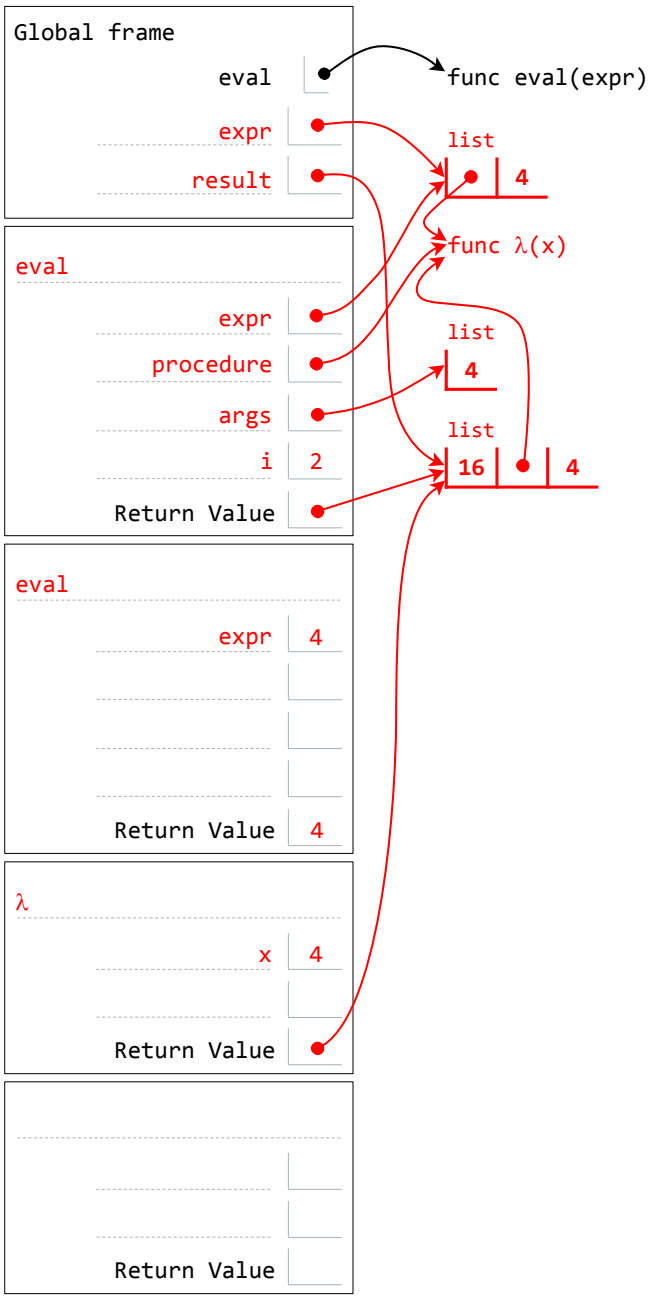
(b) **(9 pt)** Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. You need only show the final state of each frame. *You may not need to use all of the spaces or frames.* You may draw objects that are created but are not accessible from the environment, if you wish. Make sure to reflect every call to a user-defined function in the environment diagram.

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 eval(expr):
    if type(expr) in (int, float):
        return expr
    procedure = expr[0]
    args, i = [], 1
    while i < len(expr):
        args.append(eval(expr[i]))
        i += 1
    return procedure(*args)

expr = [lambda x: [x * x] + expr, 4]
result = eval(expr)
```



3. (8 points) We Recurse at Hard Problems

The following are the first six rows of *Pascal's triangle*:

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}
$$

The first and last element in each row is 1, and each of the other elements is equal to the sum of the element above it and to the left and the element above it and to the right. For example, the third element in the last row is $4 + 6 = 10$, since 4 and 6 are the elements above it and to the left and right.

(a) Define a function `pascal` that takes a row index n and an element index k as arguments and computes the kth element in row n, with indexing beginning at 0 for both n and k. *Do* ***not*** *compute factorial or any other combinatorial expression as part of your solution.*

```
def pascal(n, k):
    """Compute the kth element of the nth row in Pascal's triangle.

    >>> pascal(5, 0)
    1
    >>> pascal(5, 2)
    10
    """
    if k == 0 or k == n:
        return 1
    return pascal(n-1, k-1) + pascal(n-1, k)
```

(b) Fill in the `pascal_gen` function below, which returns an iterator over the elements in the nth row of Pascal's triangle. Your solution should be self-contained; *you may* ***not*** *use the* `pascal` *function defined in part (a).*

```
def pascal_gen(n):
    """Return an iterator over all the elements in the nth row of
    Pascal's triangle.

    >>> list(pascal_gen(5))
    [1, 5, 10, 10, 5, 1]
    """
    if n != 0:
        last = 0
        for num in pascal_gen(n-1):
            yield last + num
            last = num
    yield 1
```

4. (7 points) We are Functionally Lazy

(a) Fill in the function below to match its docstring description, so that all doctests pass.

```
def squares(num):
    """Return the square of num and a function to compute subsequent
    squares.

    >>> s, f = squares(1)
    >>> s
    1
    >>> s, f = f()
    >>> s
    4
    >>> s, f = f()
    >>> s
    9
    """
    squared = num * num
    func = lambda: squares(num + 1)
    return squared, func
```

(b) Fill in the function below to match its docstring description, so that all doctests pass.

```
def make_countdown(start):
    """Return a function that will count down from start to 1,
    returning the next value each time it is called, and returning
    'GO!' when it is done.

    >>> countdown = make_countdown(3)
    >>> countdown()
    3
    >>> countdown()
    2
    >>> countdown()
    1
    >>> countdown()
    'GO!'
    >>> countdown()
    'GO!'
    """
    def countdown():
        nonlocal start
        value = start
        start -= 1
        return 'GO!' if value <= 0 else value
    return countdown
```

5. (8 points) We are Objectively Lazy

Suppose we wish to define a new lazily evaluated list type called `LazyList`. A `LazyList` does not hold elements directly; instead, it holds 0-argument functions to compute each element. The first time an element is accessed, the `LazyList` calls the stored function to compute that element. Subsequent accesses to the same element do not call the stored function. See the docstring for `LazyList` for examples of how to use it.

(a) Fill in the class definition of `LazyList` below to match its docstring description, so that all doctests pass.

```
class LazyList(object):
    """A lazy list that stores functions to compute an element. Calls
    the appropriate function on first access to an element; never
    calls an element's function more than once.

    >>> def compute_number(num):
    ...     print('computing', num)
    ...     return num
    ...
    >>> s = LazyList()
    >>> s.append(lambda: compute_number(1))
    >>> s.append(lambda: compute_number(2))
    >>> s.append(lambda: compute_number(3))
    >>> s[1]
    computing 2
    2
    >>> s[1]
    2
    >>> s[0]
    computing 1
    1
    >>> for item in s: print(item)
    1
    2
    computing 3
    3
    """
    def __init__(self):
        self._list = []
        self._computed_indices = set()

    def append(self, item):
        self._list.append(item)

    def __getitem__(self, index):
        if index not in self._computed_indices:
            self._computed_indices.add(index)
            self._list[index] = self._list[index]()
        return self._list[index]

    def __iter__(self):
        for index in range(len(self._list)):
            yield self[index]
```

(b) Assuming a correct definition of `LazyList`, what value would be bound to `result` after executing the following code? Circle the value below, or "other" if the value is not one of the choices provided.(*Hint*: Draw the environment diagram for `mystery` and the functions defined within it.)

```
def mystery(n):
    s = LazyList()
    i = 0
    while i < n:
        s.append(lambda: i)
        i += 1
    return s

result = mystery(4)[1]
```

0 1 2 3 4 other

6. (16 points) We Love to Scheme; Muahahaha!

(a) Assume that you have started the Scheme interpreter and defined the following procedures:

```
(define x (lambda (y)
            (if (= y 0)
                1
                (+ (x (- y 1)) y))))

(define y (mu (x)
            (if (= x 0)
                1
                (- (y (- x 1)) x))))

(define (z x y) (x y))
(define (r x y r) (x r))
```

For each of the following expressions, write the value to which it evaluates. If the value is a function value, write FUNCTION. If evaluation causes an error, write ERROR. If evaluation would run forever, write FOREVER. Otherwise, write the resulting value as the interactive interpreter would display it. The first two rows have been provided as examples:

Expression	Evaluates to
`(+ 1 4)`	5
`(+ 1 car)`	ERROR
`(cons 1 (cons 2 3))`	(1 2 . 3)
`(cdr '(1 (2) 3))`	((2) 3)
`(z car (list 1 (2) 3))`	Error
`(z z z)`	Error
`(z x 3)`	7
`(z y 3)`	Error
`(r x y 3)`	7
`(r y x 3)`	1

(b) Write a Scheme function `insert` that creates a new list that would result from inserting an item into an existing list at the given index. Assume that the given index is between 0 and the length of the original list, inclusive.

```
(define (insert lst item index)
  (if (= index 0)
      (cons item lst)
      (cons (car lst) (insert (cdr lst) item (- index 1)))))
```

(c) Suppose a tree abstract data type is defined as follows:

```
;;; An empty tree.
(define empty-tree nil)

;;; Determine if a tree is empty.
(define (empty? tree) (null? tree))

;;; Construct a tree from an element and left and right subtrees.
(define (tree elem left right) (list elem left right))

;;; Retrieve the element stored at the given tree node.
(define (elem tree) (car tree))

;;; Retrieve the left subtree of a tree.
(define (left tree) (car (cdr tree)))

;;; Retrieve the right subtree of a tree.
(define (right tree) (car (cdr (cdr tree))))
```

Fill in the `contains` procedure below, which determines whether or not a number is contained in a set represented by the tree data structure above.

```
(define (contains tree num)
  (if (empty? tree)
      false
      (if (= (elem tree) num)
          true
          (if (< (elem tree) num)
              (contains (right tree) num)
              (contains (left tree) num)))))
```

7. (13 points) Our Schemes are Logical, and Our Logic is Schemy

(a) Assume that you have started the Logic interpreter and defined the following relations:

```
(fact (append () ?x ?x))
(fact (append (?a . ?r) ?s (?a . ?t))
      (append ?r ?s ?t))

(fact (foo () () ()))
(fact (foo (?a . ?r) (?b . ?s) ((?a ?b) . ?t))
      (foo ?r ?s ?t))

(fact (bar (?a . ?r) ?s)
      (append ?r (?a) ?s))

(fact (baz ?rel () ()))
(fact (baz ?rel (?a . ?r) (?b . ?s))
      (?rel ?a ?b)
      (baz ?rel ?r ?s))
```

For each of the following expressions, write the output that the interactive Logic interpreter would produce. The first two rows have been provided as examples:

Expression	Interactive Output
`(query (append (1 2) 3 (1 2 3)))`	**Failed.**
`(query (append (1 2) (3 4) ?what))`	**Success!** **what: (1 2 3 4)**
`(query (foo 1 3 (1 3)))`	Failed.
`(query (foo (1) (3) (1 3)))`	Failed.
`(query (foo (1 2) (3 4) ?what))`	Success! what: ((1 3) (2 4))
`(query (bar () ()))`	Failed.
`(query (bar (1 2 3 4) ?what))`	Success! what: (2 3 4 1)
`(query (baz bar ((1 2 3) (4 5) (6)) ?what))`	Success! what: ((2 3 1) (5 4) (6))

(b) Write a relation `sorted` that is true if the given list is sorted in increasing order. Assume that you have a `<=` relation that relates two items if the first is less than or equal to the second. Here are some sample facts and queries:

```
logic> (fact (<= a a))
logic> (fact (<= a b))
logic> (fact (<= a c))
logic> (fact (<= b b))
logic> (fact (<= b c))
logic> (fact (<= c c))
logic> (query (sorted ()))
Success!
logic> (query (sorted (a b b c)))
Success!
logic> (query (sorted (b a c)))
Failed.

(fact (sorted ()))
(fact (sorted (?a)))
(fact (sorted (?a ?b . ?r))
      (<= ?a ?b)
      (sorted (?b . ?r)))
```

(c) Fill in the `all<=all` relation below, which relates two lists if every element in the first list is `<=` every element in the second list. You may use the `<=all` relation defined below. Here are some sample queries:

```
logic> (query (all<=all (a b c) (a b c)))
Failed.
logic> (query (all<=all (a b) (b b c)))
Success!

(fact (<=all ?x ()))
(fact (<=all ?x (?a . ?r))
      (<= ?x ?a)
      (<=all ?x ?r))

(fact (all<=all () ?x))
(fact (all<=all (?a . ?r) ?s)
      (<=all ?a ?s)
      (all<=all ?r ?s))
```

Comments

Problem 1

Part A

Grading: 3 pts

Common Mistakes: Incorrect condition or computation for base case.

Part B

Grading: 5 pts

Common Mistakes: Not checking if the subtrees are also balanced. Incorrect base case.

Comments: Some students attempted to solve this problem iteratively. Tree-structured data is almost always processed recursively.

Part C

Grading: 4 pts

Comments: The doctests were very important here. First, they illustrate the syntax of accessing `depth` and `is_balanced`; they are accesses without call expression syntax, so they must be non-functional attributes or property methods. Second, the doctests demonstrate that `depth` and `is_balanced` should work correctly even after the tree is mutated. This implies that they must be property methods.

It was also important in this question to understand the lookup procedure for attributes. An instance attribute that is a function *is not* automatically bound to an instance, while a class attribute that is a function *is* automatically bound to an instance. Thus, the property methods must be defined as class attributes, not as instance attributes.

Problem 2

Part A

Grading: 7 pts

Common Mistakes: Neglecting to draw frames for calls to the lambda function; it should be evident that `map` calls the given function once for each element in the input iterable. Drawing frames for `list` and `map`. (Frames should only be drawn for user-defined functions, and all such frames should be drawn. It's impossible to draw frames for built-in functions, since the contents of such a frame are hidden below an abstraction barrier.) Drawing whole lists inside other lists rather than just references. Drawing new lists for operations that mutate an existing list rather than create a new one.

Part B

Grading: 9 pts

Common Mistakes: Placing the lambda function itself inside a list, rather than a reference to the function. Incorrectly appending the two lists inside the lambda call. Ignoring or incorrectly handling the second call to `eval`. Incorrect pointers in the global frame or the return values in local frames.

Problem 3

Part A

Grading: 3 pts

Common Mistakes: Neglecting the `k == n` base case.

Comments: A recurrence relation is defined here, and the most natural way to implement a recurrence relation is using recursion (not to mention the hint in the title of this problem). Unfortunately, some students still attempted to use iteration; few such solutions worked.

Part B

Grading: 5 pts

Common Mistakes: Incorrect base case. Attempting to iterate over a non-iterable. Attempting to index an iterator.

Problem 4

Part A

Grading: 3 pts

Common Mistakes: Not assigning a function to `func`. Failing to return both required values.

Comments: As usual, the doctests provide a lot of information about how the function should work. The problem title also provided a substantial hint, that computation should be lazy.

Part B

Grading: 4 pts

Common Mistakes: Defining a generator function. Incorrect handling of the start and end cases.

Comments: Again, the doctests provide a lot of information. The interface demonstrated in the doctests is contrary to the interface for a generator. Instead, the doctests for `make_countdown` (not to mention its name) show that it produces a function that evolves over time. This is exactly the situation where we use nonlocal assignment.

Problem 5

Part A

Grading: 6 pts

Common Mistakes: Failing to call the item-computing function in `__getitem__`. Failing to do so in `__iter__`, either explicitly or by relying on `__getitem__`. Failing to define `__iter__` as a generator function.

Part B

Grading: 2 pts

Comments: This is an example where intuition fails, and we need to walk through the environment diagram to see what happens. The diagram demonstrates that the frame for the lambda function has the frame for `mystery` as its parent, and that `i` has no binding in the lambda frame but is bound to 4 in the parent frame. Thus, the lambda function returns 4.

This is why we have environment diagrams, so that we can keep track of what is happening. Intuition often leads us to the wrong answer.

Problem 6

Part A

Grading: 1 pt each

Part B

Grading: 3 pts. No points were deducted for syntax errors, as long as the computational logic was clear.

Common Mistakes: Using `list` or `append` instead of `cons`.

Part C

Grading: 5 pts. No points were deducted for syntax errors or failing to take advantage of the BST property.

Common Mistakes: Incorrect base cases. Abstraction violations.

Problem 7

Part A

Grading: 1 pt each. No points were deducted for missing `Success!` or variable names in the output.

Part B

Grading: 4 pts

Common Mistakes: Missing the second base case. Recursing on just `?r` instead of `(?b . ?r)`. Attempting to relate two lists instead of just one; the sample queries demonstrate that the `sorted` relation takes in only one list.

Part C

Grading: 3 pts

Common Mistakes: Comparing the two lists pairwise, rather than comparing all elements of the first list to all elements in the second. Assuming that the second list has at least one element, by using something like `(?b . ?s)`; no points were deducted for this error.