CS 61A Structure and Interpretation of Computer Programs

Summer 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” $\times$ 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)**	

1. **(5 points) Orange you glad it's not "What Will Python Print"?**

   (a) **(3 pt)** Assume the following definitions have been made:

```
def foo(num):
    return x + num

def bar(x):
    return foo(x + 3)

def banana(x):
    def orange(num):
        return num + x * 3
    return orange

def apple(x):
    return banana(x - 2)(4)
```

Write what each of the function calls below would return in lexical and dynamic scope. If a function call would result in an error, write Error instead.

```
>>> bar(3)
```

- Lexical scope: Error
- Dynamic scope: 9

```
>>> apple(7)
```

- Lexical scope: 19
- Dynamic scope: 25

   (b) **(2 pt)** Alan Kay drew parallels to common user interface design principles from which of the following sports? Write an X on the line next to your answer.

______	Basketball
______	Bowling
______	Football
______	Golf
______	Soccer
X	Tennis
______	Volleyball

2. (4 points) Abstract away your worries

Assume we are provided the following implementation of the rlist abstract data type:

```
empty_rlist = None

def rlist(first, rest):
    return (first, rest)

def first(s):
    return s[0]

def rest(s):
    return s[1]
```

In the following functions, clearly circle any data abstraction violations. Draw one circle for *each* data abstraction violation.

```
def interleave(s1, s2):
    if not s1:
        return s2
    elif not s2:
        return s1
    recursive = interleave(s1[1], s2[1])
    return rlist(first(s1), (first(s2), recursive))

def filter(pred, s):
    if not s:
        return None
    elif pred(first(s)):
        return (first(s), filter(pred, s[1]))
    return filter(pred, s[1])
```

3. (12 points) Lambdas and llamas

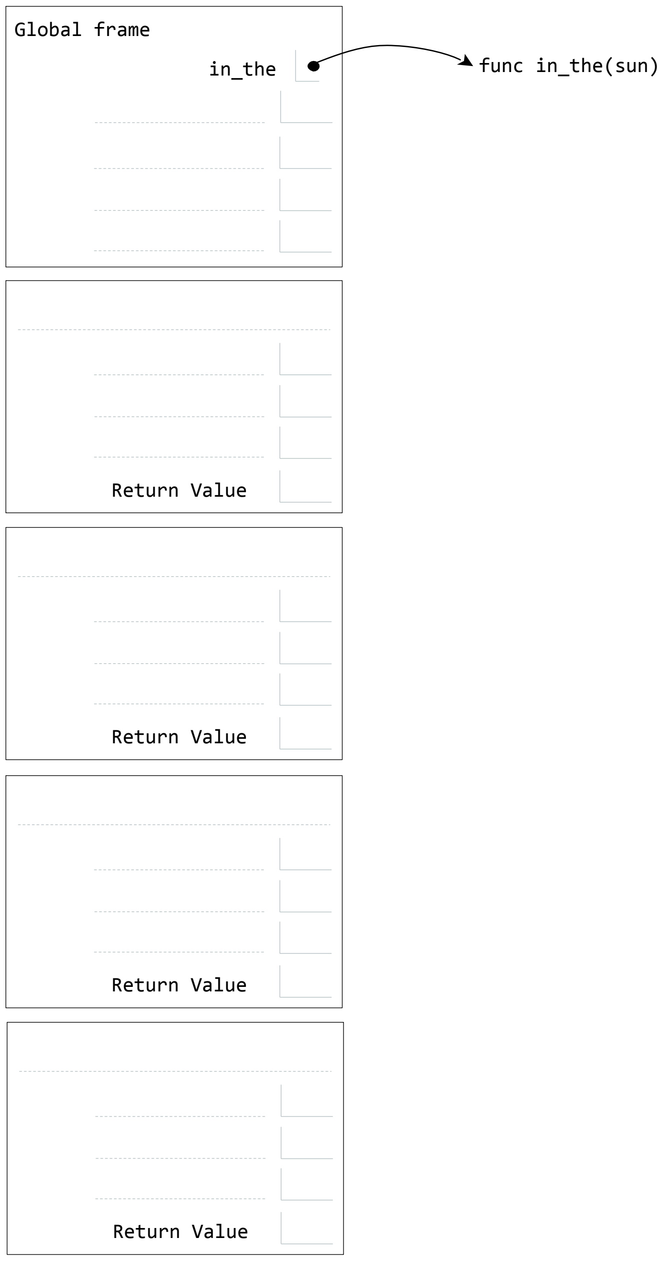
(a) (6 pt) Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. You need only show the final state of each frame. *You may not need to use all of the spaces or frames.*

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 in_the(sun):
    def sun(fun):
        def beach(water):
            nonlocal sun
            if water == 0:
                return fun
            sun = beach
            return sun(water - 1)
        return beach
    return sun

boat = [in_the]
boat.append(boat)
summer = boat[0](boat)
summer(2)(1)
```



http://goo.gl/Oe58E5

(b) **(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 need only show the final state of each frame. *You may not need to use all of the spaces or frames.*

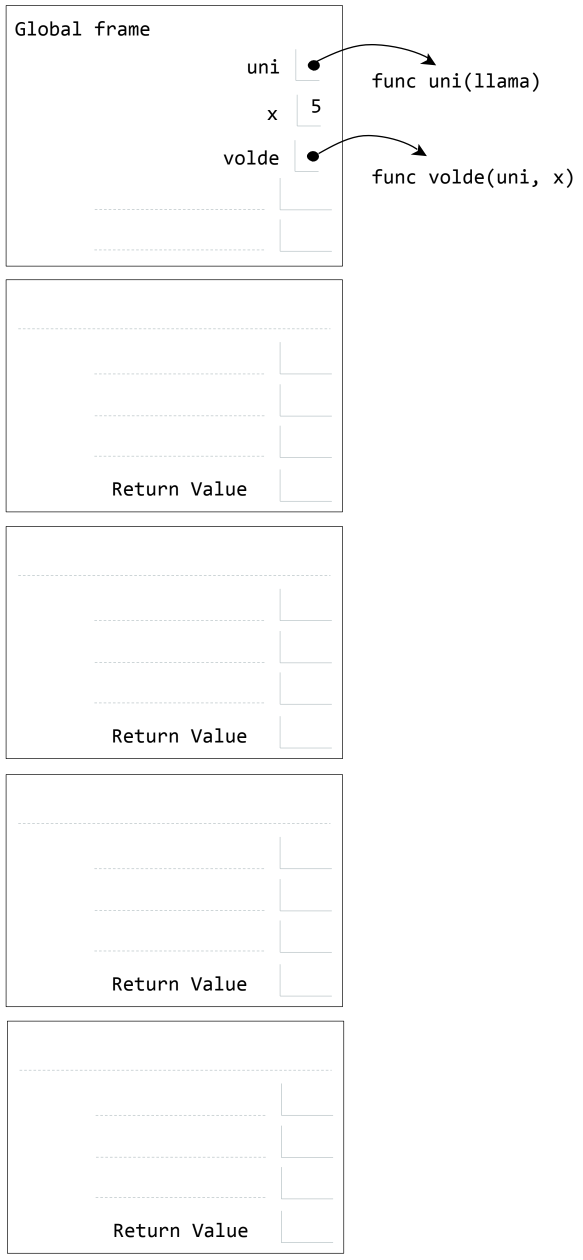
A complete answer will:

- 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 uni(llama):
    x = 10
    def volde(uni, x):
        return uni > llama(11, 7)
    return volde

x = 5
def volde(uni, x):
    if uni(6, x):
        return "pi"
    return "i"

llama = uni(lambda llama, uni: llama + uni + x)
volde(llama, 20)
```



http://goo.gl/Cfo9qP

4. (4 points) Cupcakes cupcakes cupcakes cupcakes cupcakes

Recall the `type_tag` function discussed in lecture:

```
def type_tag(generic_object):
    return type_tag.tags[type(generic_object)]
```

We will use this function to create a generic function. Because everyone likes pastries, you have created five different classes to represent five different kinds of pastries:

- `SugarCookie`
- `SnickerdoodleCookie`
- `RedVelvetCookie`
- `VanillaCake`
- `CheeseCake`

You have two functions, `eat_cookie` and `eat_cake`, which you call on the appropriate pastry to consume it. However, each function only works on pastries of a particular type – `eat_cookie` only works on cookies, and `eat_cake` only works on cake. Tired of having to manually select the correct function, you attempt to define a generic `eat` function:

```
def eat(baked_good):
    return eat.implementations[type_tag(baked_good)](baked_good)
```

This function takes a baked good and calls the appropriate eat function on it, regardless of its type. However, in your haste to consume delicious baked goods, you forgot to fill in the appropriate dictionaries to make this work! Complete the following two dictionaries so that the `eat` function works as expected. **Use as few tags as possible.**

```
type_tag.tags = {
  SugarCookie : 'cookie',
  SnickerdoodleCookie: 'cookie',
  RedVelvetCookie : 'cookie',
  VanillaCake : 'cake',
  CheeseCake : 'cake'
}
```

```
eat.implementations = {
  'cake' : eat_cake,
  'cookie' : eat_cookie
}
```

5. (10 points) Interpretive dance

(a) (4 pt) Assume the following definition has been loaded into the Scheme interpreter from Project 4:

```
(define (sum-of-squares x y z)
  (+ (* x x) (* y y) (* z z)))
```

Given the following Scheme expressions, circle the correct number of calls to `scheme_eval` and `scheme_apply`:

`(+ 5 (* 3 7 3))`

`scheme_eval` 3 4 5 6 7 [✓ 8] 9

`scheme_apply` 1 [✓ 2] 3 4 5 6

`(sum-of-squares 3 4 5)`

`scheme_eval` 4 5 8 10 14 [✓ 19] 24 25

`scheme_apply` 1 2 3 4 [✓ 5] 6

(b) (6 pt) For each of the following Scheme expressions, place an X on the line next to the correct `Pair` representation that `scheme_read` from Project 4 would create.

`(func (4 5) 3)`

______ `Pair('func', Pair(4, Pair(5, Pair(3))))`

______ `Pair('func', Pair(4, Pair(5, Pair(3, nil))))`

______ `Pair('func', Pair(Pair(4, 5), Pair(3, nil)))`

__X__ `Pair('func', Pair(Pair(4, Pair(5, nil)), Pair(3, nil)))`

______ Attempting to `scheme_read` the above expression would result in a syntax error.

`'(1 2 (3))`

______ `Pair(1, Pair(2, Pair(3, nil)))`

______ `Pair(1, Pair(2, Pair(Pair(3, nil), nil)))`

______ `Pair('quote', Pair(1, Pair(2, Pair(Pair(3, nil), nil))))`

__X__ `Pair('quote', Pair(Pair(1, Pair(2, Pair(Pair(3, nil), nil))), nil))`

______ Attempting to `scheme_read` the above expression would result in a syntax error.

`(cdr () 'cdr)`

______ `Pair('cdr', Pair(nil, Pair('cdr', nil)))`

______ `Pair('cdr', Pair(Pair(nil, Pair('cdr', nil))))`

__X__ `Pair('cdr', Pair(nil, Pair(Pair('quote', Pair('cdr', nil)), nil)))`

______ `Pair('cdr', Pair(Pair(nil, Pair(Pair('quote', Pair('cdr', nil))))))`

______ Attempting to `scheme_read` the above expression would result in a syntax error.

6. (9 points) You Oughta Like Objects

(a) (6 pt) Assume the definitions on the left have been loaded into the Python interpreter. In each of the blanks on the right, write what would be displayed by the Python interpreter, or write Error if that line would cause an error.

```
class A(object):

    def f(self):
        return 2

    def g(self, obj, x):
        if x == 0:
            return A.f(obj)
        return obj.f() + self.g(self, x - 1)


class B(A):

    def f(self):
        return 4
```

```
>>> x = A()
>>> y = B()
>>> x.f()
2
>>> B.f()
Error
>>> x.g(x, 1)
4
>>> y.g(x, 2)
8
>>> x.f = lambda self: 4
>>> y.g(x, 1)
Error
```

(b) (3 pt) Consider the following interpreter session:

```
>>> x = Yolo(1)
>>> x.g(3)
4
>>> x.g(5)
6
>>> x.motto = 5
>>> x.g(5)
10
```

Provide the definition of the `Yolo` class so that the above interpreter session works as expected.

```
class Yolo(object):
    "*** YOUR CODE HERE ***"

    def __init__(self, motto):
        self.motto = motto

    def g(self, n):
        return self.motto + n
```

7. (7 points) Let's get #basedmark trending on Twitter

(a) (3 pt) The following two statements are executed in parallel in a shared environment in which x is initially bound to 2:

Thread 1	Thread 2
`>>> x = x + 4 + x`	`>>> x = x * 3`

In the box below, list all possible values of x after both threads terminate.

6, 8, 12, 16, 24

Now, in the box below, list only the values that are possible when the threads are correctly synchronized.

16, 24

(b) (4 pt) Fill in the blanks to complete the output of this MapReduce job. This job is run on the key-value pairs contained within the DATA list, and its behavior depends on the value of the KEYWORD variable:

```
DATA = [
  ('8/11/13', "Studying for finals! Mark is so helpful. #basedmark"),
  ('8/12/13', "Mappers and reducers? Whaaat...? #basedmark"),
  ('8/12/13', "OMG WHAT ARE LOGIC?!"),
  ('8/12/13', "Eric and Steven aren't cool"),
  ('8/12/13', "Logic is so cool! :) #hw13yo #logic #clubsoda"),
  ('8/14/13', "OMG WHAT IS MAPREDUCE?!"),
  ('8/15/13', "Just kidding, MAPREDUCE IS SO COOL! #basedmark"),
  ('8/15/13', "Just kidding, the final was cool too. #coolcoolcool")
]

def map():
    for date, status in DATA:
        for word in [w.lower() for w in status.split() if KEYWORD in w]:
            emit(date, 1)

def reduce():
    for date, count_iterator in values_by_key(sys.stdin):
        emit(date, sum(count_iterator))
```

For each of the values of KEYWORD below, fill in the resulting output in the provided tables:

KEYWORD = '#basedmark'	
Date	**Count**
8/11/13	1
8/12/13	1
8/15/13	1

KEYWORD = 'cool'	
Date	**Count**
8/12/13	2
8/15/13	2 (3 was also accepted)

8. (7 points) Don't let this question get the BST of you

Consider the following binary tree abstract data type, implemented in Scheme:

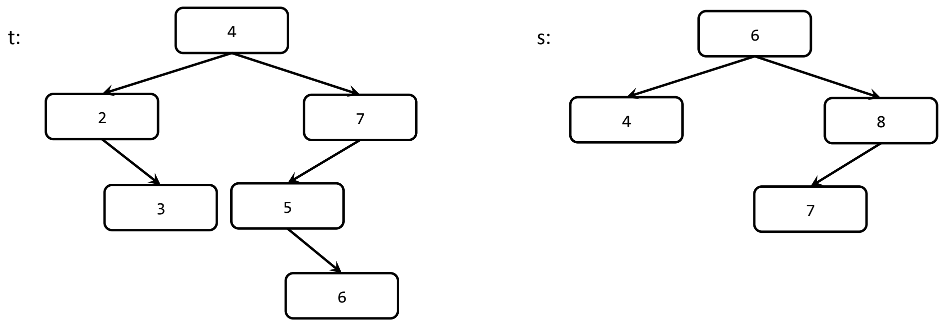
```
(define (tree entry left right)
  (cons entry (cons left right)))

(define (entry tree)
  (car tree))

(define (left tree)
  (car (cdr tree)))

(define (right tree)
  (cdr (cdr tree)))
```

Recall that a binary search tree is a tree in which all entries are greater than the elements in their left branches and less than the elements in their right branches. Use this abstract data type to define a *tail-recursive* procedure `bst-path`. This procedure takes a binary search tree, as well as an item contained in the binary search tree. It returns a list of the values encountered along the path from the root to the node containing that item. For example, assume we have defined the following binary search trees in the Scheme interpreter:



Then `bst-path` would work as follows:

```
STk> (bst-path t 2)
(4 2)
STk> (bst-path t 6)
(4 7 5 6)
STk> (bst-path s 6)
(6)
STk> (bst-path s 4)
(6 4)
```

You may assume that the item is always in the tree, and the tree does not contain duplicate elements. Complete the definition of `bst-path` **on the next page**.

Make sure your procedure is tail-recursive! A solution that is not tail-recursive is limited to 5 points possible.

```
(define (bst-path bst item)
   (define (bst-helper tree item acc)
     (cond ((= (entry tree) item) (append acc (list item)))
           ((> (entry tree) item)
                 (bst-helper (left tree) item (append acc (list (entry tree)))))
           (else (bst-helper (right tree) item (append acc (list (entry tree)))))))
   (bst-helper tree item nil))
```

9. (9 points) In space, no one can hear you stream

(a) (4 pt) Consider the following definitions:

```
def make_integer_stream(first=1):
    def compute_rest():
        return make_integer_stream(first + 1)
    return Stream(first, compute_rest)

def add_stream(s1, s2):
    def compute_rest():
        return add_stream(s1.rest, s2.rest)
    return Stream(s1.first + s2.first, compute_rest)

def my_stream():
    def rest():
        return add_stream(make_integer_stream(0),
                          add_stream(my_stream(), my_stream()))
    return Stream(-1, rest)
```

In the blanks below, write the first five elements of the stream returned by a call to `my_stream`:

__-1__ __-2__ __-3__ __-4__ __-5__

(b) (5 pt) Write a function `group_iterator` that takes another iterator of key-value tuples as its argument. It should return a new iterator that yields key-value tuples: one tuple per unique key in the original iterator. The value for each tuple should be a list containing all values corresponding to that key in the original iterator.

You may assume that the original iterator has been sorted such that all pairs with the same key are next to each other. You may *not* assume anything about the length of the provided iterator.

```
def group_iterator(orig):
    """Groups elements from the provided iterator by keys.

    >>> x = [('steven', 1), ('steven', 2), ('eric', 3), ('eric', 5), ('eric', 4)]
    >>> grouped = group_iterator(iter(x))
    >>> next(grouped)
    ('steven', [1, 2])
    >>> next(grouped)
    ('eric', [3, 5, 4])
    >>> next(grouped)
    Traceback
        ...
    StopIteration
    """
    key, val = next(orig)
    so_far = [val]
    for k, v in orig:
        if k == key:
            so_far.append(v)
        else:
            yield key, so_far
            key, so_far = k, [v]
    yield key, so_far
```

10. (8 points) Flat 10

Write a set of facts for the `flatten` relationship between two relations. The only atom present in either of these relations is the letter `a`. The `flatten` relationship is satisfied when its second relation is the flattened version of the first:

```
logic> (query (flatten (a a a) (a a a)))
Success!
logic> (query (flatten ((a (a)) a) ?what))
Success!
what: (a a a)
logic> (query (flatten (((a)) (a a)) ((a) a a)))
Failed.
```

You may assume that each relation's elements are either other relations, or the letter `a`. You can assume none of the nested relations are empty. You may find `append` useful in solving this problem.

```
(fact (append () ?x ?x))

(fact (append (?a . ?r) ?b (?a . ?z))
      (append ?r ?b ?z))

"*** YOUR CODE HERE ***"
(fact (flatten () ()))
(fact (flatten ((?a . ?b) . ?cdr1) ?s)
      (flatten (?a . ?b) ?flattened)
      (append ?flattened ?cdr2 ?s)
      (flatten ?cdr1 ?cdr2))
(fact (flatten (?a . ?cdr1) (?a . ?cdr2))
      (flatten ?cdr1 ?cdr2))
```

11. (5 points) The exciting life of Albert Wu

Albert worries that students find calling functions to be too boring. To make things more interesting, he decides to write a higher order function `liven` that converts boring functions into more exciting functions.

Help him complete his definition of `liven`. It should take three arguments:

1. `boring_fn`, a function that takes one argument.
2. `fun_fn`, a function that takes two arguments, the second of which is always an integer.
3. An integer `n`.

Once called with these three arguments, `liven` should return a *lively* function, which takes one argument and does the following:

- Every `n`-th time the *lively* function is called, it calls `fun_fn` with the provided argument and the number of times the *lively* function has ever been called. It returns the result of this call.
- Otherwise, calling the *lively* function should return the value of calling `boring_fn` with the provided argument.

Complete the definition of `liven` in the provided space below.

```
def liven(boring_fn, fun_fn, n):
    """Returns a lively function based on two provided functions and an integer.

    >>> great_deal = liven(lambda name: name + ' walked the dog.',
    ...                    lambda name, i: name + ' won ' + str(i) + ' new cars!',
    ...                    2)
    ...
    >>> great_deal('Sandy')
    'Sandy walked the dog.'
    >>> great_deal('Sandy')
    'Sandy won 2 new cars!'
    >>> great_deal('Sandy')
    'Sandy walked the dog.'
    >>> great_deal('Sandy')
    'Sandy won 4 new cars!'
    """
    count = 0
    def lively(arg):
        nonlocal count
        count += 1
        if count % n == 0:
            return fun_fn(arg, count)
        return boring_fn(arg)
    return lively
```

12. (1 points) Extra credit

The 61A staff members have used this test to encode a message for you:

```
2[3]  11[1]  1[2]  5[1]  4[5]  3[6]  10[0]  8[1]  2[4]  10[2]

1[4]  5[4]  7[1]  4[4]  7[2]  2[2]  6[2]  9[-1]  3[2]  11[2]  1[1]
```

Write the decoded message in the box below to receive a point of extra credit: