CS 61A Structure and Interpretation of Computer Programs
Fall 2013 Final

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.
- Fill in the information on this page using **PERMANENT INK**. You may use pencil for the rest of the exam.

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
/24	/10	/20	/26	/80

THIS PAGE INTENTIONALLY LEFT BLANK

1. (24 points) What a Value!

(a) (10 pt) For each of the following Python expressions, write the value to which it evaluates, assuming that expressions are evaluated in order in a single interactive session. **Answers may depend on previous lines; be careful to keep track of bindings from names to values!** The first two rows have been provided as examples. If evaluation causes an error, write Error. If evaluation never completes, write Forever. If the value is a function, write Function.

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

```
def outer(f):
    x = 0
    def inner(g):
        nonlocal x
        x, y = g(x), f(x)
        return f(x, y)
    return inner

def add(a, b=2):
    return a+b

def grow(c):
    return sum(range(c, c+2))

h1, h2 = outer(add), outer(add)
```

Expression	Evaluates to
`5*5`	25
`1/0`	Error
`grow(5)`	
`list(map(str, map(grow, range(3, add(3)))))`	
`h1(grow)`	
`outer(grow)(add)`	
`h1(grow) + h2(add) + h2(grow)`	

(b) **(8 pt)** Each of the following expressions evaluates to a `Stream` instance. For each one, write the values of the three elements in the stream. The first value of the first stream is filled in for you.

Assume that you have started Python 3 and executed the following statements, in addition to the `Stream` class statement on your final study guide.

```
def m(t):
    def compute_rest():
        return m(t.rest.rest)
    return Stream(t.rest.first+1, compute_rest)

s = lambda t: Stream(t, lambda: s(t+2))
t = s(1)
u = m(t)
```

Stream	Has the first three elements
t	1, ____________, ____________
u	____________, ____________, ____________
m(u)	____________, ____________, ____________

(c) **(6 pt)** For each of the following Scheme expressions, write the Scheme value to which it evaluates. The first three rows are completed for you. If evaluation causes an error, write Error. If evaluation never completes, write Forever. **Hint**: No dot should appear in a well-formed list.

Assume that you have started the Project 4 Scheme interpreter and evaluated the following definitions.

```
(define f (lambda (x y) (g (cons x y)) ))
(define g (mu (z) (list (h x) y z)))
(define h (mu (y) (if (> y 0)
                      (+ x (h (- y 1)))
                      1)))
```

Expression	Evaluates to
(* 5 5)	25
'(1 2 3)	(1 2 3)
(/ 1 0)	Error
'((1 . (2 3 . (4))) . 5)	
(cons 1 (list 2 3 '(car (4 5))))	
(f 2 3)	

2. (10 points) Good Game

(a) **(8 pt)** In the box below, write the **final** value to which each numbered blank would have an arrow in the environment diagram. Blank 0 is completed for you. **In addition**, fill in the parent annotations for all functions and frames that have non-global parents.

You may wish to fill in the diagram, but only the numbered values and parent annotations will be scored.

Hint: The `pop` method removes and returns the last element of a list.

Note: There is another question at the bottom of this page.

```
def g(g):
    def h(s):
        nonlocal g
        g = [s.pop(), g[0]]
        s = s[1:]
        return [s[1:], g]
    return h
def s(g):
    return [8, 7] + g
c = g([3, 4, 5])
h = [1, 2, [3, 4]]
c(s([6, 1])) # A
```

Write each value as it would be displayed by the Python interpreter

0 [1, 2, [3, 4]]

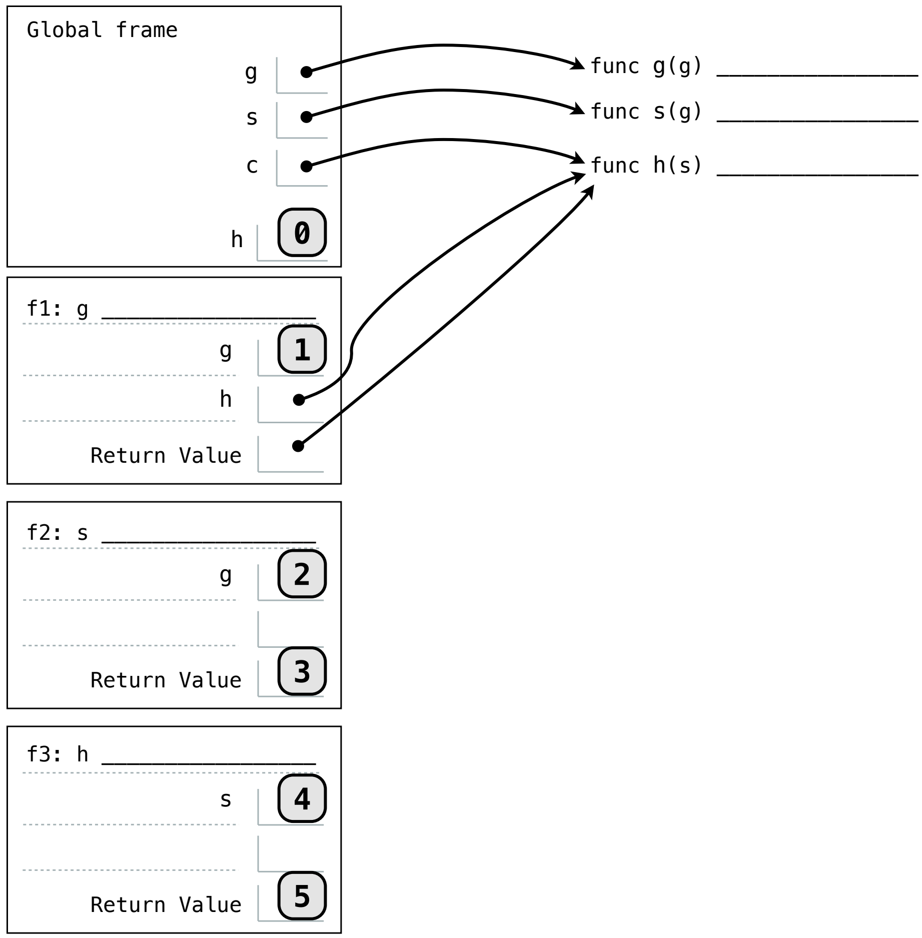
1 ____________________

2 ____________________

3 ____________________

4 ____________________

5 ____________________



(b) **(2 pt)** What value would result from evaluating `c(s([6, 1]))` another time after executing the code above? If evaluation causes an error, write Error. If evaluation never completes, write Forever.

3. (20 points) Equality

(a) **(4 pt)** Fill in the blanks in the implementation of `paths`, which takes as input two positive integers `x` and `y`. It returns the number of ways of reaching `y` from `x` by repeatedly incrementing or doubling. For instance, we can reach 9 from 3 by incrementing to 4, doubling to 8, then incrementing again to 9.

```
def inc(x):
    return x+1
def double(x):
    return x*2
def paths(x, y):
    """Return the number of ways to reach y from x by repeated
    incrementing or doubling.

    >>> paths(3, 5) # inc(inc(3))
    1
    >>> paths(3, 6) # double(3), inc(inc(inc(3)))
    2
    >>> paths(3, 9) # E.g., inc(double(inc(3)))
    3
    >>> paths(3, 12) # E.g., double(double(3))
    6
    >>> paths(3, 16) # E.g., double(double(inc(3)))
    11
    >>> paths(1, 8) # E.g., double(inc(inc(double(1))))
    10
    >>> paths(3, 3) # No calls is a valid path
    1
    """

    if x > y:

        return ___________________________________________________________

    elif x == y:

        return ___________________________________________________________

    else:

        return ___________________________________________________________
```

(b) **(2 pt)** Write one of `<=`, `>=`, or `!=` in each blank below such that the following statements are true for all positive integers `x`, `y`, and `z`. If it is not possible to do so, write `X` in the blank.

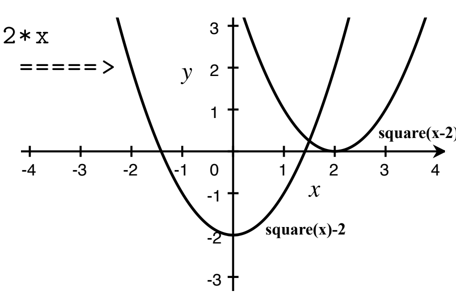
```
paths(min(x, y), z) ______________ paths(max(x, y), z)

paths(x, z) ________________ paths(x, y) * paths(y, z)
```

(c) **(4 pt)** Fill in the blanks in the implementation of `pathfinder`, a higher-order function that takes an increasing function `f` and a positive integer `y`. It returns a function that takes a positive integer `x` and returns whether it is possible to reach `y` by applying `f` to `x` zero or more times. For example, 8 can be reached from 2 by applying `double` twice. A function f is *increasing* if $f(x) > x$ for all positive integers x.

```
def pathfinder(f, y):
    """Return a function find_from that takes x and returns whether
    repeatedly applying the increasing function f to x can reach y.

    >>> f = pathfinder(double, 8)
    >>> {k: f(k) for k in (1, 2, 3, 4, 5)}
    {1: True, 2: True, 3: False, 4: True, 5: False}
    >>> g = pathfinder(inc, 3)
    >>> {k: g(k) for k in (1, 2, 3, 4, 5)}
    {1: True, 2: True, 3: True, 4: False, 5: False}
    """

    def find_from(x):

        while ____________________________________________________________:

            ____________________________________________________________

        return __________________________________________________________

    ____________________________________________________________________
```

(d) **(4 pt)** A shifted-k point of a differentiable function f is a number x such that $f(x) - k = f(x - k)$ for constant k. Fill in the blanks in the implementation of `shifted`, which takes a differentiable function `f`, its derivative `df`, and a constant `k`. It returns a shifted-k point of `f` using the `find_zero` function defined on your study guide. **Hint**: The derivative of $f(x) - k$ is the same as the derivative of $f(x)$.

```
def shifted(f, df, k):
    """Return a shifted-k point of the function f with derivative df.

    >>> square, dsquare = lambda x: x*x, lambda x: 2*x
    >>> shifted(square, dsquare, 2)  # Graphed here =====>
    1.5
    >>> shifted(square, dsquare, 6)
    3.5
    """
```




```
    def g(x):

        return _______________________________________________________

    def dg(x):

        return _______________________________________________________

    return find_zero(g, dg)
```

(e) **(6 pt)** Fill in the blanks in the implementation of `zero`, which returns whether it is possible to create a Calculator expression that evaluates to 0 using the input numbers as leaves of the expression tree. The only Calculator operations allowed are two-argument `+`, such as in `(+ 1 2)`; two-argument `-`, such as in `(- 2 1)`; and two-argument `*`, such as in `(* 2 2)`.

```
from operator import add, sub, mul

def zero(*s):
    """Return whether s can be the ordered leaves of a Calculator expression
    that evaluates to 0. The argument s is a tuple of positive integers.

    Values can only be combined with two-argument +, -, and *.
    No division or single-argument negation is allowed.

    >>> zero(1, 1, 2) # (- (+ 1 1) 2)
    True
    >>> zero(1, 1, 3) # (* (- 1 1) 3)
    True
    >>> zero(12, 4, 3) # (- 12 (* 4 3))
    True
    >>> zero(9, 6, 3, 5) # (- (+ 9 6) (* 3 5))
    True
    >>> zero(1, 3, 2, 5) # (- (+ (* 1 3) 2) 5)
    True
    >>> zero(5, 3)
    False
    >>> zero(7, 5, 3)
    False
    >>> zero(8, 4, 2) # (- 8 (* 4 2))
    True
    >>> zero(4, 8, 2) # Order matters.
    False
    """

    if len(s) == 1:

        return s[0] == 0

    for i in range(len(s)-1):


        for f in __________________________________________________________:


            a = f(___________________________________________________)


            if zero(______________________________________________________):

                return True

    return False
```

4. (26 points) Python Crossing

(a) (8 pt) The `alphabetical` function takes as input an iterable over lowercase letters, such as a string. It returns whether those letters are in alphabetical order. Cross out whole lines so that `alphabetical` is implemented correctly with the fewest lines of code possible. **Hint**: `'e'<'f'` evaluates to `True`.

```
def alphabetical(w):
    """Return whether the letters in w are in alphabetical order.

    >>> alphabetical('')
    True
    >>> alphabetical('why')
    False
    >>> alphabetical('how')
    True
    >>> alphabetical('above')
    False
    >>> alphabetical('below')
    True
    >>> alphabetical('full')
    False
    >>> alphabetical('empty')
    True
    >>> alphabetical('matte')
    False
    >>> alphabetical('glossy')
    True
    """
    previous = 'a'
    previous = w[0]
    previous = None
    i = 0
    next = lambda x: x[i]
    w = iter(w)
    try:
        while True:
        while i < len(w):
            nonlocal i
            nonlocal previous
            nonlocal w
            w = iter(w)
            letter = next(w)
            letter = previous
            if letter < previous:
            if letter <= previous:
                return False
                return True
            return letter < previous
            previous = letter
            previous = w[i-1]
            previous = next(w)
    except StopIteration:
        return False
        return True
    return w.sort() is w
```

Below is a complete `Rlist` class that is identical to the one from lecture, but without any assert statements.

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0
    empty = EmptyList()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
    def __getitem__(self, index):
        if index == 0:
            return self.first
        else:
            return self.rest[index-1]
    def __len__(self):
        return 1 + len(self.rest)
```

(b) (6 pt) The `Alist` class, which subclasses `Rlist` above, represents a recursive list, but instead of storing the first element and the rest, it stores a recursive list of elements at the beginning of the list, then a recursive list of elements at the end. The first and rest of an `Alist` instance can each be either an `Alist` instance or an `Rlist` instance. Cross out whole lines so that `Alist` is implemented correctly with the fewest lines of code possible.

```
class Alist(Rlist):
    """An appended list consists of a first list followed by the rest.

    >>> s = Rlist(1, Rlist(2, Rlist(3)))
    >>> t = Rlist(4, Rlist(5, Rlist(6)))
    >>> st = Alist(s, t)
    >>> len(st), st[1], st[4]
    (6, 2, 5)
    >>> stst = Alist(st, st)
    >>> len(stst), stst[8]
    (12, 3)
    """
    def __init__(self, prefix, rest):
        Rlist.__init__(self, prefix, rest)
        Rlist.__init__(self, prefix.first, rest)

    def __getitem__(self, index):
        if index == 0:
        if index < len(self.first):
            return self.first
            return self.first[index]
        else:
            r = self.rest
            return r[index-len(self.first)]
            return rest[index-len(self.first)]
            return r[index-1]
            return rest[index-1]

    def __len__(self):
        return len(self.first) + len(self.rest)
        return 1 + len(self.rest)
```

Below is a complete `Tree` class that is identical to the one from lecture, but without a `__repr__` method.

```
class Tree:
    """A binary tree with internal entries."""
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right
```

(c) **(4 pt)** The `tree_to_list` function takes a binary search tree `t` as input. The `tree_to_list` function returns a recursive list, represented as either an `Rlist` or `Alist`, that contains all entries of `t` in sorted order. Cross out whole lines so that `tree_to_list` is implemented correctly in as few lines as possible.

Reminder: In a binary search tree, all entries in the left branch are smaller than the root entry and all entries in the right branch are larger. In addition, each branch is either a binary search tree or `None`.

```
def tree_to_list(t):
    """Return a list with the elements of binary search tree t in sorted order.

    >>> odds = Tree(7, Tree(3, Tree(1), Tree(5)), Tree(9, None, Tree(11)))
    >>> s = tree_to_list(odds)
    >>> len(s)
    6
    >>> [s[i] for i in range(len(s))]
    [1, 3, 5, 7, 9, 11]
    """
    if t.left is None and t.right is None:
    if t is None:
        return Rlist.empty
    rest = Rlist(t.entry, tree_to_list(t.right))
    rest = Alist(t.entry, tree_to_list(t.right))
    if t.entry:
    if t.left:
        return Rlist(tree_to_list(t.left), rest)
        return Alist(tree_to_list(t.left), rest)
    else:
        return rest
        return None
```

(d) **(2 pt)** Consider a tree `t` with maximum depth d and total number of entries n. Define a mathematical function $f(d, n)$ such that calling `tree_to_list(t)` makes $\Theta(f(d, n))$ recursive calls to `tree_to_list`.

$f(d, n) =$

(e) **(6 pt)** An `append-all` relation is true if it contains multiple **non-empty** lists and all but the first list append to form the first. Cross out whole lines in the facts below so that the queries at the end all give the correct expected results.

```
(fact (append-all () ()))

(fact (append-all (?a . ?r) (?a . ?s)))

(fact (append-all (?a) (?a)))

(fact (append-all ?all . ?parts))

(fact (append-all (?a . ?s) (?a) . ?t)

      (append-all ?s . ?t)

      (append-all ?s ?t)

      (append-all . ?s . ?t)

      (append-all (?a . ?s) (?a) . ?t)

      )

(fact (append-all (?a . ?s) (?a . ?r) . ?t)

      (append-all . (?s ?r ?t))

      (append-all (?s) . (?r ?t))

      (append-all (?s) (?r) . (?t))

      (append-all ?s ?r . ?t)

      )

(query (append-all (a b c d e) (a b) (c) (d e)))
; expect Success!

(query (append-all (a b c d e) (a b c) . ?r))
; expect Success! ; r: ((d) (e)) ; r: ((d e))

(query (append-all (a c) (a b) (c)))  ; (a b) & (c) append to form (a b c)
; expect Failed.

(query (append-all (a b c d) () (a b c) () (d)))  ; No empty lists
; expect Failed.
```

SCRATCH PAPER

SCRATCH PAPER

SCRATCH PAPER

SCRATCH PAPER