CS 61A Structure and Interpretation of Computer Programs

Spring 2014

FINAL (WITH CORRECTIONS)

INSTRUCTIONS

- You have 3 hours to complete the exam.
- The exam is open book and open notes. You may not use a computer, calculator, or anything responsive.
- 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	Q. 5	Total
/24	/10	/20	/	/26	/80

1. (24 points) Evaluation

(a) (12 points)

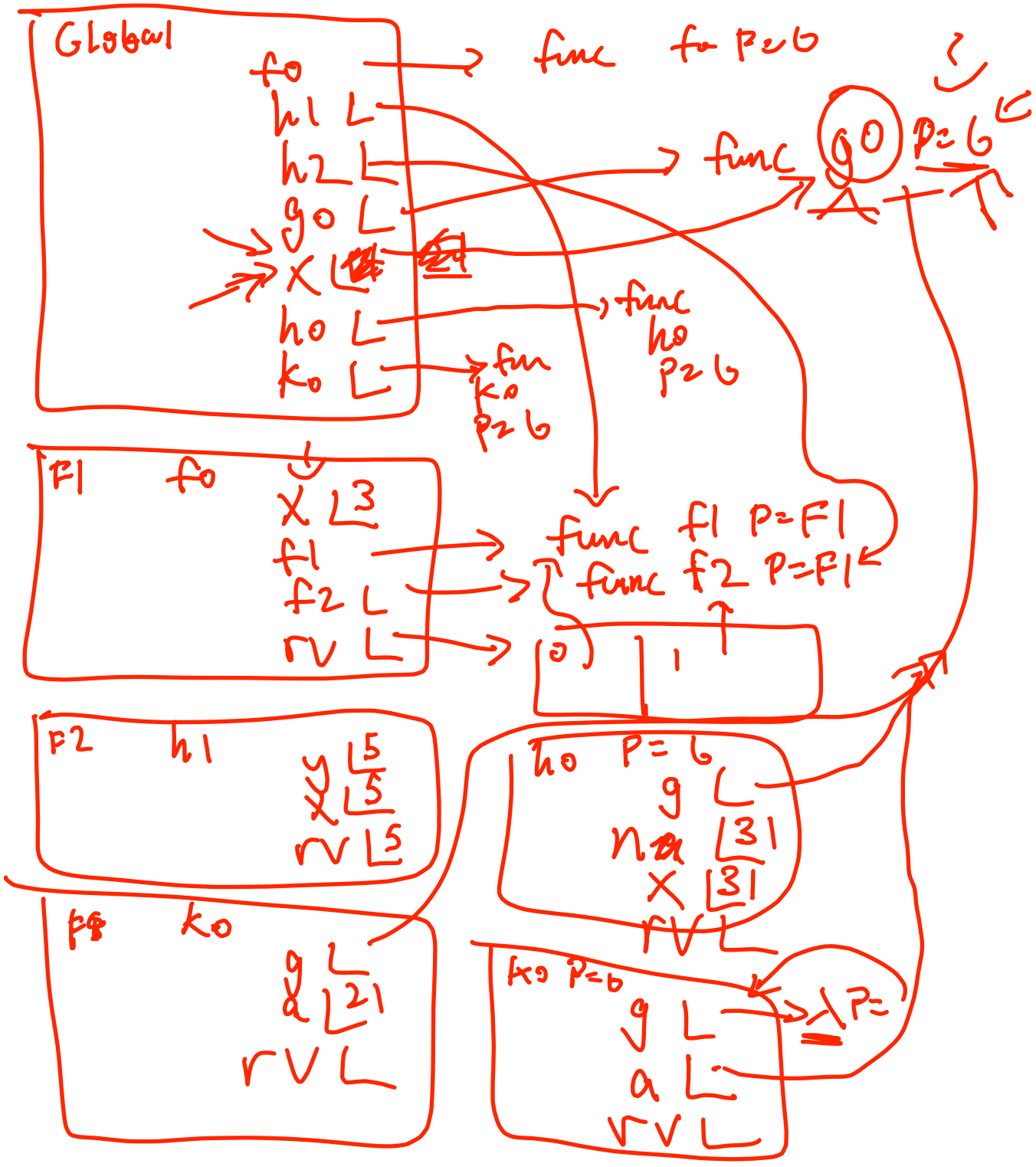
For each of the following Python expressions, write the value to which it evaluates, assuming that expressions are evaluated in order in a single interactive session. **Warning:** Answers may depend on previous lines; be careful to keep track of bindings from names to values. The first two rows have 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. [**Reminder:** The **global** keyword has the same function as **nonlocal**, but refers to the global frame rather than an enclosing function frame.]

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

```
def f0(x):
    def f1(y):
        x = y;
        return x
    def f2():
        return x
    return f1, f2

h1, h2 = f0(3)

def g0():
    return x

x = 14

def h0(g, n):
    x = n
    return g()

def k0(g, a):
    global x
    x = a
    return g()
```

Expression	Evaluates to
`abs(-3)`	3
`1/0`	ERROR
`h1(5)`	5
`h2()`	3
`g0()`	14
`k0(g0, 21)`	21
`h0(g0, 31)`	21
`k0(lambda: x(), g0)`	Function

THIS PAGE INTENTIONALLY LEFT BLANK



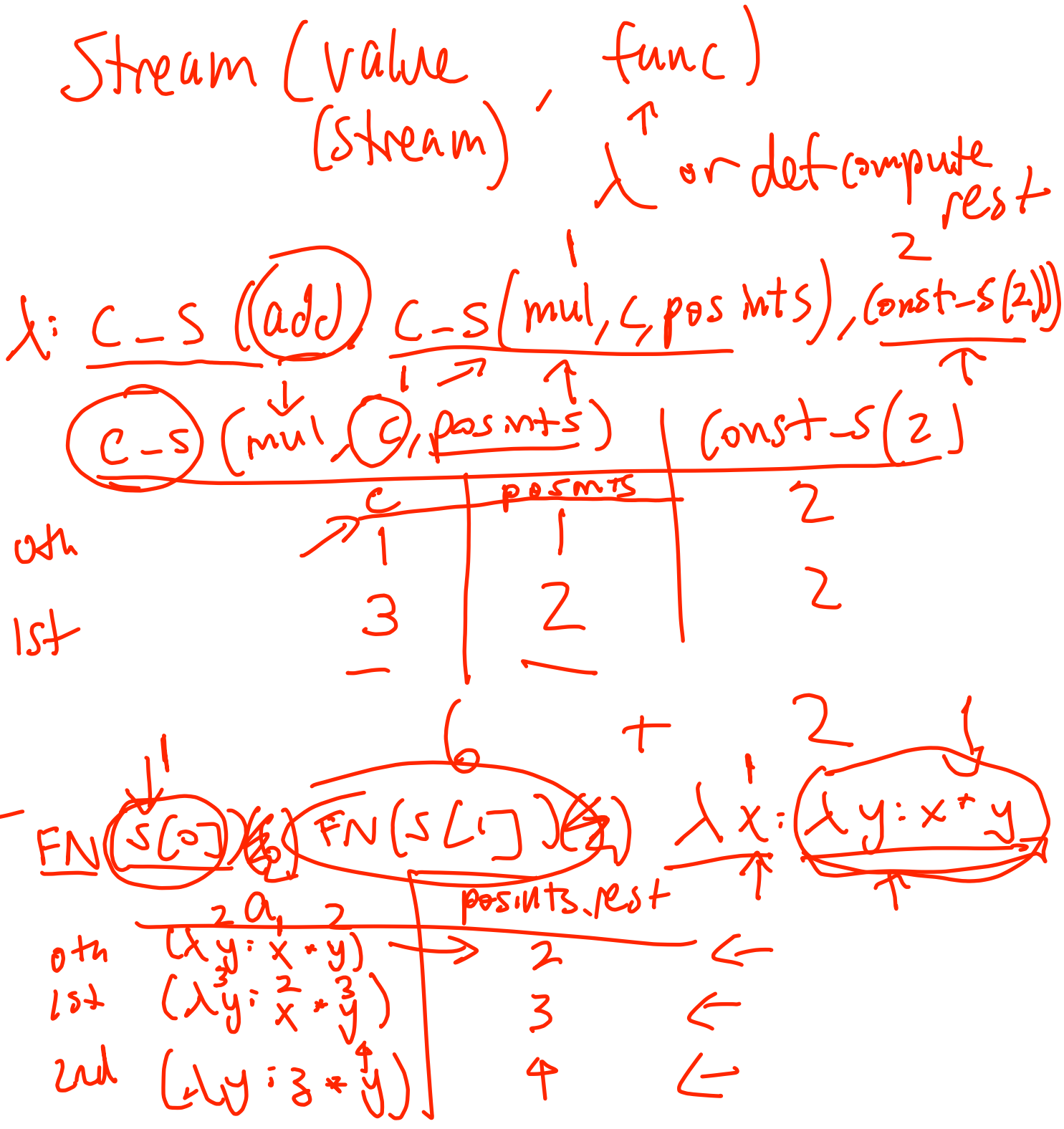
(b) (6 points)

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

```
from operator import add, mul

def const_stream(n):
    result = Stream(n, lambda: result)
    return result

posints = make_integer_stream(1)

c = Stream(1, lambda: combine_streams(add,
                                      combine_streams(mul, c, posints),
                                      const_stream(2)))
a = map_stream(lambda x: lambda y: x*y, posints)
d = combine_streams(lambda f, x: f(x),
                    a, posints.rest)
```

Stream	Has the first three elements
c	1, 3, 8
a.rest	Func, Func, Func
d	2, 6, 12

```
# (For reference): Headers from homework and lecture:

class Stream:
    def __init__(self, first, compute_rest):
        self.first = first;  self._compute_rest = compute_rest
    @property
    def rest(self): ...
    def __getitem__(self, k): ...
    empty = ...

# Functions from lecture (no bodies)

def map_stream(fn, s):            """The stream FN(S[0]), FN(S[1]), FN(S[2]), ..."""
def combine_streams(fn, s0, s1): """The stream FN(S0[0], S1[0]), FN(S0[1], S1[1]), ..."""
def make_integer_stream(k):       """The stream K, K+1, K+2, ..."""
```


Stream (value, func)
(stream)
λ or def compute rest
λ: c_s (add) c_s (mul, 4, posints), const_s(2))
c_s (mul, 4, posints) | const_s(2)
c
posints
2
0th
1
1
2
1st
3
2
2
6
+
FN (s[0]) (FN (s[1:]) (2))
λ x: λ y: x * y
posints.rest
0th (λ y: x * y)
1st (λ y: 2 * 3 * y)
2nd (λ y: 3 * 4 * y)
2
3
4

(c) (6 points)

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:** Scheme never uses dots to print well-formed lists.] Assume that you have started the Project 4 Scheme interpreter and evaluated the following definitions.

```
(define (f g x y)
        (g x))
(define h1 (lambda (a) (+ a x)))
(define h2 (mu (y) (+ y x)))
```

mu → dynamically scope, P of frame is frame that called it; λ: P is F where it's defined

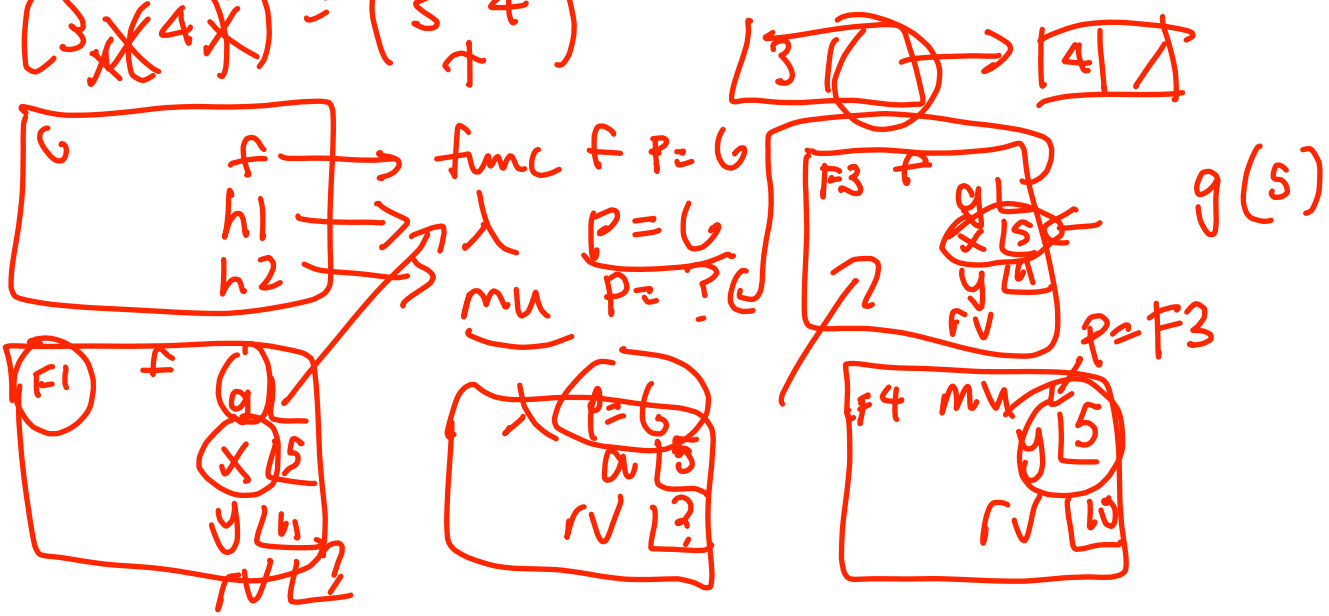
Expression	Evaluates to
(* 5 5)	25
'(1 2 3)	(1 2 3)
(/ 1 0)	Error
(cons 4 '(0 . ()) (3 . (4)) (+ 1 1))	(4 (0) (3 4) (+ 1 1))
(f h1 5 11)	ERROR
(f h2 5 11)	10

(cons 4 '(5)) = (4 5) = (cons 4 (cons 5 nil))

' = (quote) → converts well-formed ill-formed lists → well-formed

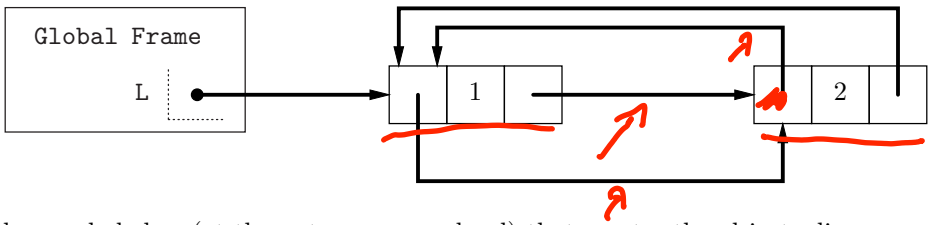
(0 . ()) = (0) → does not evaluate

(3 . (4)) = (3 4)



2. (10 points) Diagrams

(a) (4 points) The two anonymous objects in the diagram below represent 3-element Python lists (arrows point to the *whole* list object, not just its first element):



Write Python code below (at the outer program level) that creates the objects diagrammed above.

```
L = [None, 1, [None, 2, None]]
L[0] = L[2]
L[2][0] = L
L[2][2] = L
```

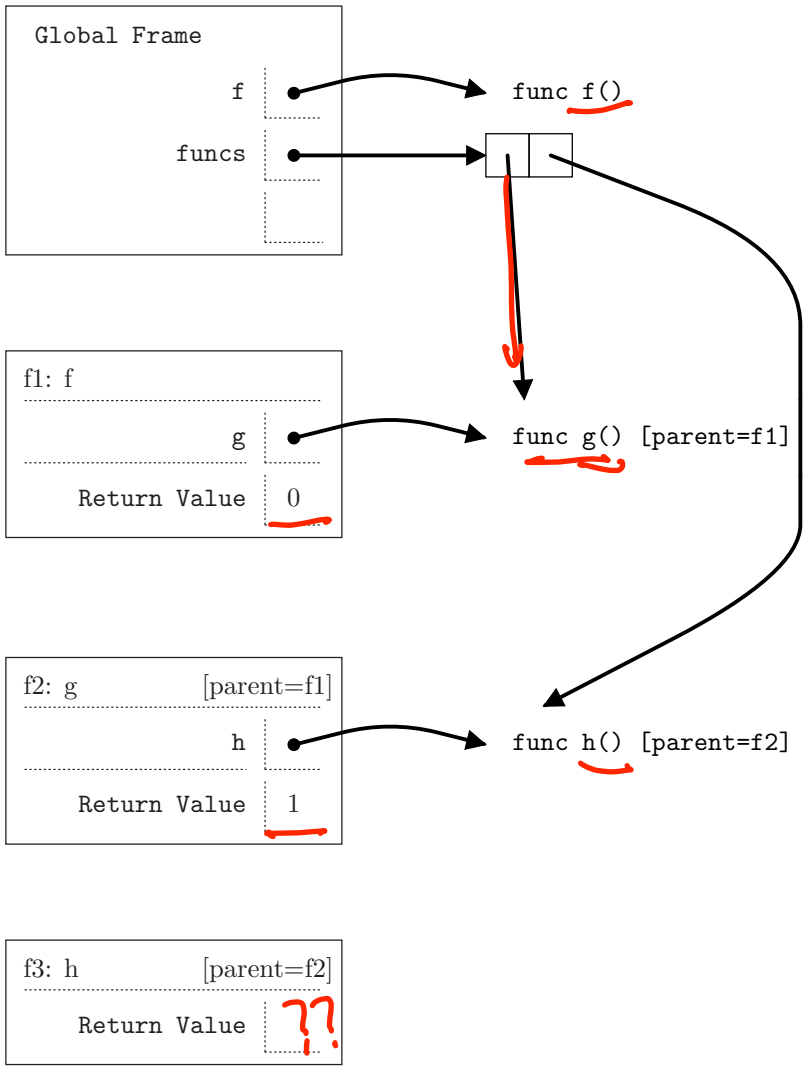
(b) (1 point) If the objects in the last problem were three-element Python tuples rather than Python lists, would the diagram represent a possible configuration of objects? If so, how would you create it, and if not, why not?

No

(c) (5 points)

Fill in Python code on the left that, when executed, will eventually result in the situation shown by the environment diagram on the right. The diagram shows a point during execution where the program is just about to call `print("Hello")`, but hasn't done so yet.

```
def f():
    def g():
        def h():
            print("Hello")
        func[1] = h
        return 1
    func[0] = g
    return 0

funcs = [None, None]
f()
funcs[0]()
# The diagram corresponds to
# a point reached in the
# next statement
funcs[1]()
```



3. (20 points) Programs

(a) (5 points) A *nested dictionary* is a dictionary in which some values are themselves dictionaries. For example,

```
>>> a = {'a': {'b': {'c': 1, 'd': 3}, 'c': 2}, 'd': {'b': [5]}}
```

To *flatten* a nested dictionary means to replace each $K : D$—where D is a dictionary—with multiple entries, one for each key in D, with $K\_$ concatenated in front of each key. The process continues recursively until the dictionary is not nested. So, for example,

```
>>> print(flatten_dict(a))
{'a_c': 2, 'a_b_c': 1, 'd_b': [5], 'a_b_d': 3}
```

Assume that all keys in the dictionaries are strings. Fill in the definition of `flatten_dict` below.

```
def flatten_dict(d):
    new_dict = {}
    for k, v in d.items():
        if isinstance(v, dict):
            flattened = flatten_dict(v)
            for k0, v0 in flattened.items():
                new_dict[k + "_" + k0] = v0
        else:
            new_dict[k] = v
    return new_dict
```

Hint: The `.items()` method on dictionaries converts entries to pairs:

```
>>> d = { 'a': 1, 'b': 2 }
>>> list(d.items())
[ ('a', 1), ('b', 2) ]
```

(b) (3 points) A polynomial,

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

can be represented as a list of coefficients:

$[a_n,\ a_{n-1},\ \cdots,\ a_2,\ a_1,\ a_0]$

One way to quickly compute a polynomial at a specific value x is to break it into n additions and multiplications using *Horner's Rule:*

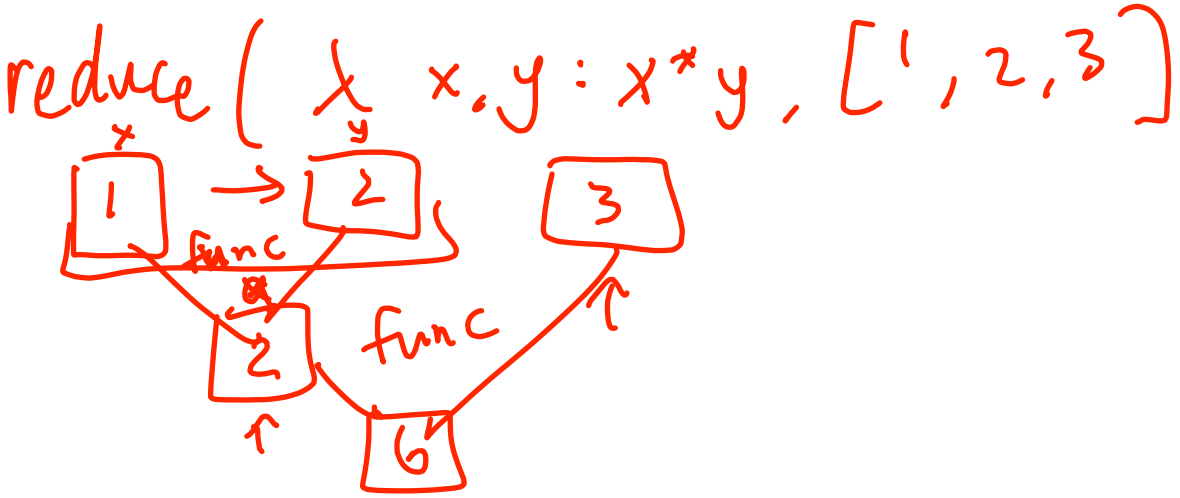
$$p(x) = (\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0.$$

So, for example,

$$3x^3 + 2x^2 + 4x + 1 = ((3x + 2)x + 4)x + 1.$$

Fill in the following function, which takes in a list of coefficients representing $p(x)$ and a value x, and returns the value $p(x)$:

```
from functools import reduce

def polyx(coeff, x):
    """ Assume len(coeff) > 1
    >>> polyx([1, 2, 3], 2)      # p(x) = x^2 + 2x + 3
    11
    >>> polyx([0, 3], 100)       # p(x) = 3
    3
    >>> polyx([1, 0, 0, 0], 3)   # p(x) = x^3
    27
    """
    return reduce(lambda a,b: a*x+b, coeff)
```



(c) (6 points)

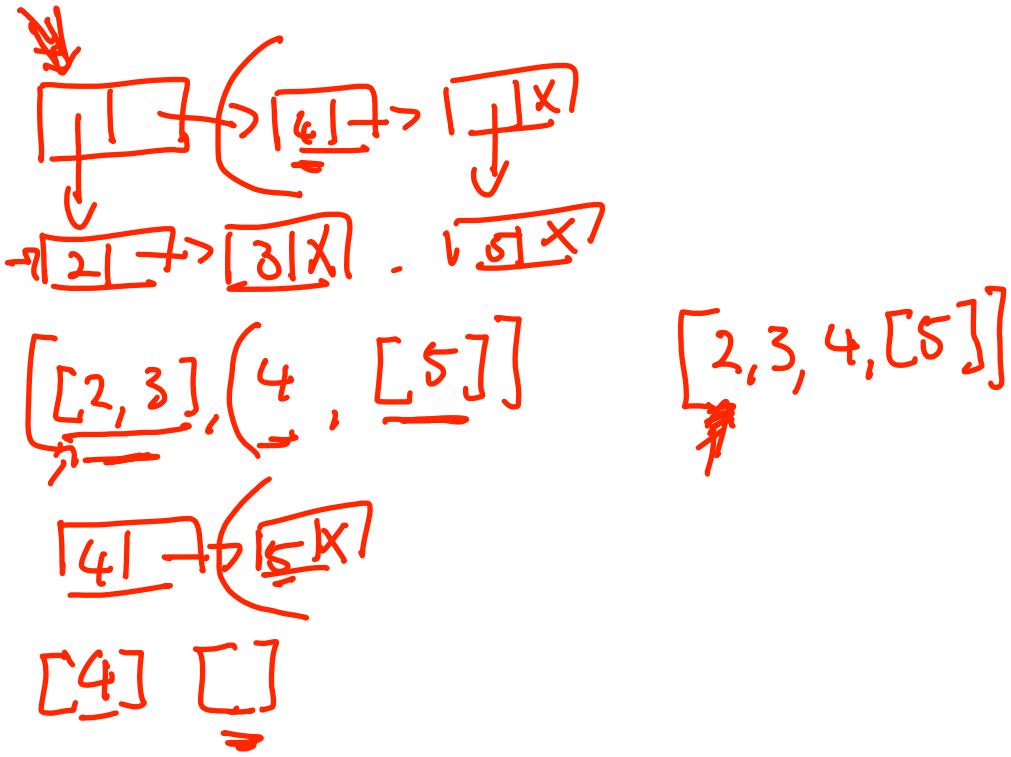
Here's a simplified definition of `Rlist` from lecture and elsewhere:

```
class Rlist:
    empty = ...
    def __init__(self, first, rest=empty): self.first, self.rest = first, rest
```

Use this definition in the problem below (in particular, do not assume that `Rlist` defines `__getitem__` (i.e., indexing).

Write a function named `all_equal` that takes in a (potentially deep) Rlist of integers and a (potentially deep) standard Python list of integers, and returns True if the Rlist's elements are equal to (and in the same position as) the list's elements, at the same depths. (The term "deep" here simply means that some of the Rlist or list values may themselves be Rlists or lists.)

```
def all_equal(r, l):
    """Assuming R is an Rlist (possibly containing other Rlists and numbers)
    and L is a list (possibly containing other lists), True iff R and L
    denote the same (possible deep) sequence of values.
    >>> all_equal(Rlist(Rlist(2, Rlist(3)), Rlist(4, Rlist(Rlist(5)))),
    ...           [[2, 3], 4, [5]]
    True
    >>> all_equal(Rlist(Rlist(2, Rlist(3)), Rlist(4, Rlist(Rlist(5)))),
    ...           [2, 3, 4, [5]]
    False
    >>> all_equal(Rlist(Rlist(Rlist(3)), [[[3]]])
    True
    >>> all_equal(Rlist.empty, [])
    True"""
    if r is Link.empty and l == []:
        return True
    if r is Link.empty or l == []:
        return False
    if type(r.first) is Rlist and type(l[0]) is list:
        return all_equal(r.first, l[0]) and all_equal(r.rest, l[1:])
    if type(r.first) is Rlist or type(l[0]) is list:
        return False
    return r.first == l[0] and all_equal(r.rest, l[1:])
```


1
6
1
2
3
5
[2, 3], (4, [5]]
[2, 3, 4, [5]]
4
5
[4] []

(d) (6 points) For reasons best known to himself, Ben Bitdiddle wants to generate powers of 2 from 2^1 up to and including some limit 2^n. He tries to write both an iterator and a generator version, but, being sleep-deprived, he gets both wrong. Correct his errors so that both versions work. That is, write in any lines that are missing and cross out any erroneous lines, writing in the correct lines next to them. For example, the doctest below should work for either version.

```
>>> for k in PowerGen(6):
...     print(k)
2
4
8
16
32
64
```

```
# Iterator version
class PowerGen:
    def __init__(self, n):

        self.i = 2

    def __next__(self):

        if self.n > 0:

            r = self.i


            self.i = self.i * 2


            return self.i

        else:
            raise StopIteration

    def __iter__(self):
        return self

# Generator version
def PowerGen(n):
    i = 2
    while n > 0:

        return i

        i = i * 2
```

4. (1 point) Sum of Human Knowledge

How old is the Sun, in galactic years (±5%)? 18.4

5. (26 points) Various Programs

(a) (6 points)

Consider a new sequence that is kind of like the Fibonacci sequence:

1, 1, 1, 3, 5, 9, 17, 31, ...

That is,

$$f(n) = \begin{cases} 1, & \text{for } n < 3 \\ f(n-1) + f(n-2) + f(n-3), & \text{otherwise.} \end{cases}$$

We can implement a function in Scheme that finds the nth item of our sequence (indexed from 0):

```
(define (f n)
    (if (< n 3) 1
        (+ (f (- n 1))
           (f (- n 2))
           (f (- n 3)))))
```

However, this implementation is not tail recursive (and not very efficient). Write a new version of the function that can calculate the nth item of our sequence tail recursively:

```
(define (f n)
    (define (ftail k ____n1____ ____n2____ ____n3____)
        (if ____(< k 0)____ ____n1____
            ____(ftail (- k 1) (+ n1 n2 n3) n1 n2)____))
    (if (< n 3) 1
        (ftail (- n 3) ____1____ ____1____ ____1____)))
```

(b) (6 points) A *stack* is a sequence in which items are added and removed from one end, which is known as the top of the stack. You can imagine this like a stack of pancakes. You may either add a pancake to the top of your pancake stack (the push operation) or you may remove a pancake from the top of the stack (the pop operation). In Python, a list may be used to implement a stack. This kind of data structure fits well with a particular kind of calculator—a *postfix calculator*—in which the operator appears after the operands. For example, the computation $5 + (1 + 2) \times 4 + 3$ could be written

```
5 1 2 + 4 * + 3 +
```

in postfix notation. No parentheses are ever needed; they are implicit in the order of the operands and operators. The relationship to stacks is simple: to process a number, one adds it to the stack. To process an operator, one pops off the last two numbers on the stack, applies the operator, and pushes the result back on the stack. After processing an entire expression, the final result is on the top of the stack.

Fill in the following program to evaluate postfix expressions (presented as sequences of strings). You need not use all the blanks. **DO NOT** use the `.pop` method on Python lists.

```
from operator import *

OPERATORS = { "+": add, "*": mul, "/": truediv, "-": sub }

def compute_postfix(tokens):
    """Assuming that TOKENS is a sequence of strings containing
    numerals and the operators +, -, *, and /, and together comprising
    a postfix expression, return the result of the indicated computation.
    >>> compute_postfix(['5', '1', '2', '+', '4', '*', '+', '3', '+'])
    20
    """
    stack = PostfixStack()
    for symbol in tokens:
        if ____symbol in OPERATORS____:
            stack.operate(symbol)
        else:
            stack.push(float(symbol))
    return stack.pop()

class PostfixStack:  # Do not use .pop on Python lists.
    def ____def __init__(self):____
        ____self.list = []____

    def ____def push(self, sym):____
        ____self.list.append(sym)____

    def ____def pop(self):____
        ____x = self.list[-1]____
        ____del self.list[-1]____
        ____return x____

    def ____def operate(self, sym):____
        ____x, y = self.pop(), self.pop()____
        ____self.push(OPERATORS[sym](x, y))____
```

(c) (2 points) Give worst-case asymptotic $\Theta(\cdot)$ bounds for the running time of the following code snippets. (Note: although we haven't explicitly talked about it, it is meaningful to write things with multiple arguments like $\Theta(a+b)$, which you can think of as "$\Theta(N)$ where $N = a + b$.")

```
def a(m, n):
    for i in range(m):
        for j in range(n // 100):
            print("hi")
```

Bound: $\Theta(mn)$

```
def b(m, n):
    for i in range(m // 3):
        print("hi")
    for j in range(n * 5):
        print(bye")
```

Bound: $\Theta(m+n)$

```
def d(m, n):
    for i in range(m):
        j = 0
        while j < i:
            print("hi")
            j = j + 100
```

Bound: $\Theta(m^2)$

```
def f(m):
    i = 1
    while i < m:
        i = i * 2
    return i
```

Bound: $\Theta(\lg m)$

j: 0 1 2 ... $n//100$

i: 0, 1, 2, ..., m — each row $\Theta(n)$

~~# of op~~ = $n//100$ → $\Theta(n)$

$\Theta(mn)$

i: 0, 1, 2, ..., $m//3$ j: 0, 1, 2, 3, ..., $n \cdot 5$

# of op = $m//3 + 5n$

$\Theta(m+n)$

i: 0, 1, ..., 100, 101, ..., 200, ..., 300, ..., m

j: 0, 100 / 0, 100 / ... / 0, 100 / 0, 100, 200 / 0, 100, 200, 300 / ... $m//100$

$\Theta(m^2)$

$\Theta(m)$

1, 2, 4, 8, 16, ..., m → $\lg(m)$

n, $n/2$, $n/4$, ..., 16, 8, 4, 2, 1 → $\lg(n)$

(d) (6 points) In a monotonically increasing subsequence, each element is greater than or equal to all the elements that come before it. For example, the following sequence is monotonically increasing:

```
2, 3, 3, 5, 5, 9, 10
```

but these are not:

```
1, 4, 2, 5
5, 4, 3, 2
```

Write a function called `increasing_subsequence` that takes in an infinitely long stream of numbers and returns another infinite stream that is a monotonically increasing subsequence of the input stream. There are many such subsequences; we want the one that would result from considering the input from left to right and always including an element in the result if it is greater than or equal to all previously chosen elements. So, given the input sequence

```
10, 2, 3, 5, 20, 6, 7, 8, 30, ...
```

we want a stream beginning `10, 20, 30,...`, rather than, say, `2, 3, 5, 6...`. You may assume that an infinite monotonically increasing subsequence always exists.

```
def increasing_subsequence(s):
    """Returns a monotonically increasing subsequence of s."""
    if s.first <= ____s.rest.first____:
        return Stream(____s.first____,
                      lambda: ____increasing_subsequence(s.rest)____)
    else:
        return increasing_subsequence(Stream(____s.first____,
                                             lambda: ____s.rest.rest____))
```

(e) (6 points)

We've already seen one way to write the `anagram` (or `permutation`) relation in Logic. Fill in two facts below to complete an alternate implementation of the anagram relation, using the `remove` relation. `remove` relates an element `elem` and two lists that are equal except that the second is missing *one occurrence* of `elem`

```
logic> (fact (remove ?elem (?elem . ?rest) ?rest))
logic> (fact (remove ?elem (?first . ?rest) (?first . ?rest2))

                    _____________________________________________________)
logic> (query (remove 3 (1 2 3 4 5) ?lst))
Success!
lst: (1 2 4 5)
logic> (query (remove 5 (1 2 3 4) ?lst))
Failed.
logic> (fact (anagram () ()))
logic> (fact (anagram (?first . ?rest) ?other)

             (remove _____________________________________________________)

             (anagram _____________________________________________________))
logic> (query (anagram () ()))
Success!
logic> (query (anagram ?what (c a t)))
Success!
what: (c a t)
what: (c t a)
what: (a c t)
what: (a t c)
what: (t c a)
what: (t a c)
```