After Albert lost all of his top secret data, he decided that he should at least obfuscate his functions. Help him write a higher-order-function **mystify** that takes a target function (of one number argument) as an argument and returns a new function that obfuscates the return values by the following process:

1. On the first call, the new function will return what the target function usually returns with the argument.
2. On the next call, the new function will return what the target function would normally return, plus 3.
3. On the third call, the new function will return "i'm diamond 1"
4. Repeats this process from then on.

```
def mystify(fn):
    """
    You can scroll in this window to see the rest of the doctest ------->
    >>> square = mystify(lambda x: x * x)
    >>> square(3)
    9
    >>> square(4) #16 + 3
```

```
counter = 0
def new_func(x)
   nonlocal counter
   if counter == 0:
      counter = 1
      return fn(x)
   elif counter == 1:
      counter = 2
      return fn(x) + 3
   else:
      counter = 0
      return "i'm diamond 1"
return new_func
```

Reset

Next

Note that fn does not need to be declared nonlocal, since reassignment does not occur for fn.

Question 2

When interpreting the following lines of scheme code, how many calls to scheme_eval, and scheme_apply will we make?

1)

(+ 1 (* (+ 2 3 4) (/ 3 2)) (+ 1 2))

scheme_eval : 18
scheme_apply : 5

2)

(+ 3 4 5)

scheme_eval : 5 scheme_apply : 1

1) The first call to `scheme-eval` evalutes the entire expression. `scheme-eval` realizes this expression is a call expression, so we `scheme-eval` the operator and all of the operands. The second and third operands are themselves expressions, so those are recursively evaluted as call expressions.

Thus, we need 5 calls to s `scheme-eval` to evaluate each call expression (including the outer most call expression), and we need 13 calls to `scheme-eval` for each operator and number element, for a total of 5 + 13 = 18 calls.

There are 5 calls to `scheme-apply` in this case, because we are making 5 function applications.

Note that the functions in this problem, (+ - * /), are primitive procedures, which means there are not any recursive calls to `scheme-eval` from scheme-apply (look at the code for handling primitive procedures in `scheme-apply`). Remember that `scheme-apply` will need to call `scheme-eval` when evaluating the body of non-primitive functions. So, in this specific problem, scheme-apply does not recursively call `scheme-eval` at any point, since every function is a primitive function.

Question 3

Convert the following Scheme expressions into calls to the Pair constructor. For the first 2 blanks in this problem, the only time you should have a space is directly after a comma.

(define y 5)

Pair('define', Pair('y', Pair(5, nil))) #we don't put 5 in quotes, since scheme_read converts '5' into 5. The directions for this problem didn't mention that, so we would accept both '5' and 5 for this specific question.

(define (square x) (* x x))

Pair('define', Pair(Pair('square', Pair('x', nil)), Pair(Pair('*', Pair('x', Pair('x', nil))), nil)))

Convert the following Python representation of a Scheme expression into the proper Scheme representation. Be sure to space the expression correctly.

Pair('square', Pair(Pair('/', Pair(1, Pair(0, nil))), nil))

(square (/ 1 0))

Tip: It might be helpful to draw the box-and-pointer diagram when approaching these problems.

Question 4

> Fill in the blanks for a function *in_order*, which takes in a *Binary search tree* and returns a list of its elements in sorted order (smallest to largest).

```
def in_order(tree):
    if tree is None:
        return []
    left = in_order(tree.left)
    right = in_order(tree.right)
    return left + [tree.entry] + right
```

Comments: Notice that `in_order` always needs to return a list. Some common mistakes:

Not returning the empty list

Not returning tree.entry inside a list

We want to define a Bird class. By definition, all birds have two wings, feathers, and a vertebrate.
The class header would be:

```
class Bird(object):
    ...
```

For each of the bird parts below, choose which OOP keyword best describes that part.

Parts (drag into boxes on right):

Class Attribute	Instance Attribute
Number of wings	Feather Color Age

Should be a subclass	Should be another class, but not a subclass
Hummingbird	Dog Moth

Question 6

Given the definitions provided on the left, fill in the returned values in the interactive session on the right.

```
class Fib(object):
    a = 0
    b = 1
    def compute(self, n):
        a, b = self.a, self.b
        while n > 0:
            a, b = b, a + b
            n -= 1
        return a


class Fact(object):
    a = 1
    def compute(self, n):
        if n <= self.a:
            return self.a
        return n * self.compute(n - 1)


fibber1 = Fib()
fibber2 = Fib()
facter1 = Fact()
facter2 = Fact()
```

```
>>> fibber1.compute(3)
2
>>> facter1.compute(3)
6
>>> facter1.a = 3
>>> facter1.compute(4)
12
>>> facter2.compute(4)
24
>>> Fib.a = 2
>>> fibber1.compute(3)
4
>>> fibber2.compute(3)
4
>>> fibber1.compute = facter1.compute
>>> fibber1.compute(4)
12
```

Question 7

Give a running time for g(n) using Big Theta notation.

```
def f(x, y):
    if x < 1 or y < 1:
        return 1
    return f(x - 1, y - 1) + f(x - 1, y - 1)

def g(n):
    return f(n, n)
```

- (A) $\theta(1)$
- (B) $\theta(\log(n))$
- (C) $\theta\left(2^{log(n)}\right) = \theta(n)$
- (D) $\theta\left(n^2\right)$
- (E) $\theta\left(n^3\right)$
- (F) $\theta\left(n^4\right)$
- (G) $\theta\left(2^n\right)$

Question 8

Fill in the environment diagram for the following Python interpreter session.

```
def game(of, thrones):
    def stark(thrones):
        return of(thrones + 2) + thrones
    return thrones + stark(thrones)

thrones = 5

def stab(bones):
    return thrones * bones

game(stab, thrones + 1)
```

Global Frame

game |
thrones |5
stab |

f1: game

of |
thrones |6
stark |
Return Value |52

f2: stark [p: f1]

thrones |6
|
|
Return Value |46

f3: stab

bones |8
|
|
Return Value |40

func game(of, thrones)

func stab(bones)

func stark(thrones) [p: f1]

style:

Question 9 ▼

What are the possible final values of the variable x given these two parallel threads? Select all **possible** answers.

Non-parallel section:
x = 3

Thread 1	Thread 2
x = x + 1	x = x + 2 * x

Ⓐ 3 Ⓑ 4 Ⓒ 5 Ⓓ 6 Ⓔ 7 Ⓕ 8 Ⓖ 9 Ⓗ 10 Ⓘ 11 Ⓙ 12 Ⓚ 13 Ⓛ 14 Ⓜ 15

Ⓝ 16 Ⓞ 17

The two "correct" solutions are 10 and 12. The solutions 4 and 9 occur when the two threads have interleaved execution. Note that answers that included 11 were marked as correct, since we haven't discussed order of operations when it comes to threading. For the Summer 2013 final, we won't ask about parallelism when there's an ambiguous order of operations, like this question has.

Question **10**

Write logic rules for **every-other**, a relation between two lists that is satisfied if and only if the second list is the same as the first list, but with every other element removed.

logic> (query (every-other (frodo merry sam pippin) ?x))
Success!
x: (frodo sam)

logic> (query (every-other (gandalf) ?x))
Success!
x: (gandalf)

```
(fact (every-other () ()))
(fact (every-other (?a) (?a)))
(fact (every-other (?a ?b . ?r) (?a . ?z))
      (every-other ?r ?z))
```

The two base cases handle 0 and 1 element lists. The compound fact handles that case where we have 2 or more elements by not including the second element of the first list, and then recursively using every-other on the rest of the first list.

Question 11

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

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

Note:
Be exact in your spacing.
Match the spacing in the two provided 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 (moo 7 9 (7 9)))	Failed.
(query (moo (5) (6) (5 6)))	Failed.
(query (moo (5 7) (7 8) ?what))	Success! what: ((5 7) (7 8))
(query (fizz () ()))	Failed.
(query (fizz (8 7 6 5) ?what))	Success! what: (7 6 5 8)
(query (baz fizz ((4 3 2) (1 2) (9)) ?what))	Success! what: ((3 2 4) (2 1) (9))

The key to this question is to figure out what each relation does, since the names don't give off clues. `moo` "zips" (just like Python zip) elements together, `fizz` takes the first element and puts it at the end, and `baz` is simply map.

Question 12

Fill in the tail-recursive implementation of remove-all, which takes a list and an item, and removes *all occurrences* of that item in the list. If the item does not appear in the list, just return the original list.

Note: make sure all syntax is correct (including parentheses)!

```
(define (remove-all lst item)
  (define (helper lst so-far)
    (cond ((null? lst) so-far )
          ((eq? (car lst) item) (helper (cdr lst) so-far) )
          (else (helper (cdr lst) (append so-far (list (car lst))) )))
  (helper lst nil ))

; Tests

STk> (remove-all '(1 2 3 4) 3)
(1 2 4)
STk> (remove-all '(1 2 3) 5)
(1 2 3)
STk> (remove-all '(1 1 14 1 51) 1)
(14 51)
```

Question 13

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

```
(define x (lambda (y)
            (if (= y 0)
                3
                (+ (x (- y 1)) y))))
(define y (mu (x)
            (if (= x 0)
                3
                (- (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 evaluation causes an error, write ERROR. Otherwise, write the resulting value as the interactive intepreter would display it.

Expression	Evaluates to
(cons 3 (cons 4 nil))	(3 4)
(cdr (cons 3 (cons 5 (cons 6 7))))	(5 6 . 7)
(z x 3)	9
(r x y 3)	9
(r y x 3)	3
(z car (list 7 (6) 5))	ERROR
(append '(3) '((4 5 6) 7))	(3 (4 5 6) 7)

This is a question from the Spring 2013 final. When tracing through dynamic scope questions, it's incredibly helpful to draw the environment diagram. For mu calls, remember to extend the calling frame rather than the frame in which the mu was defined.

Question 14

Given the following definition of my_stream:

```
my_stream = Stream(3, lambda: Stream(4, lambda: add_streams(my_stream, my_stream.rest)))
```

What are the first 5 elements of my_stream? Write Error if any term generates an error.

3	4	7	11	18

Note that this is essentially the stream of Fibonacci numbers, except the base case values are 3 and 4 instead of 0 and 1.

Question 15

The first 15 elements of some stream are shown below.

1 1 2 1 2 3 1 2 3 4 1 2 3 4 5 ...

(Note: the underlines are just to clarify the problem for you; this is a single stream of numbers.)

Define a stream that represents the stream above (note that the stream continues on forever).

```
def helper(i, k):
    def compute_rest():
        if i == k:
            return helper(1, k + 1)
        return helper(i + 1, k)
    return Stream(i, compute_rest)

s = helper(1, 1) #represents the stream in the directions
```

This mirrors the structure of the problem pretty well: count up to a certain number, then when you get there, start at the beginning of the next bit. In this solution, k is the number we're counting to before starting over, while i is the number we're on.