Note that \texttt{fn} does not need to be declared nonlocal, since reassignment does not occur for \texttt{fn}.
1) The first call to `scheme-eval` evaluates 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 evaluated as call expressions.

Thus, we need 5 calls to `scheme-eval` to evaluate each call expression (including the outermost 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.
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('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(1, Pair(0, nil))), nil)

(square (/ 1 0))

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

```python
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:

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

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

<table>
<thead>
<tr>
<th>Parts (drag into boxes on right):</th>
<th>Class Attribute</th>
<th>Instance Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of wings</td>
<td>Feather Color</td>
</tr>
<tr>
<td>Should be a subclass</td>
<td>Hummingbird</td>
<td>Age</td>
</tr>
<tr>
<td>Should be another class, but not a subclass</td>
<td>Dog</td>
<td>Moth</td>
</tr>
</tbody>
</table>
Given the definitions provided on the left, fill in the returned values in the interactive session on the right.

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

```python
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. \( \theta(2^{\log n}) = \theta(n) \)
- D. \( \theta(n^2) \)
- E. \( \theta(n^3) \)
- F. \( \theta(n^4) \)
- G. \( \theta(2^n) \)
Fill in the environment diagram for the following Python interpreter session.

```python
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]
```
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.
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.

\[
\text{logic> (query (every-other (frodo merry sam pippin) ?x))}
\]
\[
\text{Success!}
\]
\[
\text{x: (frodo sam)}
\]

\[
\text{logic> (query (every-other (gandalf) ?x))}
\]
\[
\text{Success!}
\]
\[
\text{x: (gandalf)}
\]

\[
\text{(fact (every-other () ()))}
\]
\[
\text{(fact (every-other (?a) (?a)))}
\]
\[
\text{(fact (every-other (?a ?b . ?r) (?a . ?z))}
\]
\[
\quad \text{(every-other ?r ?z))}
\]

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.
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.

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

```scheme
(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 interpreter would display it.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td>(cons 3 (cons 4 nil))</td>
<td>(3 4)</td>
</tr>
<tr>
<td>(cdr (cons 3 (cons 5 (cons 6 7)))))</td>
<td>(5 6 7)</td>
</tr>
<tr>
<td>(z x 3)</td>
<td>9</td>
</tr>
<tr>
<td>(r x y 3)</td>
<td>9</td>
</tr>
<tr>
<td>(r y x)</td>
<td>3</td>
</tr>
<tr>
<td>(z car (list 7 6 5))</td>
<td>ERROR</td>
</tr>
<tr>
<td>(append '3 '((4 5 6) 7))</td>
<td>(3 (4 5 6) 7)</td>
</tr>
</tbody>
</table>
Given the following definition of `my_stream`:

```python
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.
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.