1. What is a tail context/tail call? What is a tail recursive function?

**Solution:** A tail call is a call expression in a tail context. A tail context is usually the final action of a procedure/function.

A tail recursive function is where all the recursive calls of the function are in tail contexts.

An ordinary recursive function is like building up a long chain of domino pieces, then knocking down the last one. A tail recursive function is like putting a domino piece up, knocking it down, putting a domino piece up again, knocking it down again, and so on. This metaphor helps explain why tail calls can be done in constant space, whereas ordinary recursive calls need space linear to the number of frames (in the metaphor, domino pieces are equivalent to frames).

2. Why are tail calls useful for recursive functions?

**Solution:** When a function is tail recursive, it can effectively discard all the past recursive frames and only keep the current frame in memory. This means we can use a constant amount of memory with recursion, and that we can deal with an unbounded number of tail calls with our Scheme interpreter.

Answer the following questions with respect to the following function:

```
(define (sum-list lst)
```

3. Why is sum-list not a tail call? Optional: draw out the environment diagram of this sum-list with list: (1 2 3). When do you add 2 and 3?

Solution: Sum list is not the last call we make, its actually the other addition which we do after we evaluate sum-list. Sum list is not the last expression we evaluate.

4. Rewrite sum-list in a tail recursive context.

Solution:
(define (sum-list-tail lst)
  (define (sum-list-helper lst sofar)
    (if (null? lst)
        sofar
        (sum-list-helper (cdr lst) (+ sofar (car lst))))
  )
  (sum-list-helper lst 0)
)

2 Interpreters

5. Circle the number of calls to scheme_eval and scheme_apply for the code below.

(define (square x) (* x x))
(+ (square 3) (- 3 2))

scheme_eval  2  5  14  24
scheme_apply  1  2  3  4

Solution: 14 scheme_eval, 4 scheme_apply.
6. Circle the number of calls to scheme_eval and scheme_apply for the code below.

```
scm> (+ 1 2)
3

scheme_eval  1 3 4 6
scheme_apply  1 2 3 4

Solution: 4 scheme_eval, 1 scheme_apply.
```

```
scm> (if 1 (+ 2 3) (/ 1 0))
5

scheme_eval  1 3 4 6
scheme_apply  1 2 3 4

Solution: 6 scheme_eval, 1 scheme_apply.
```

```
scm> (or #f (+ 1 2) 'apple) (- 5 2))
apple

scheme_eval  6 8 9 10
scheme_apply  1 2 3 4

Solution: 8 scheme_eval, 1 scheme_apply.
```

```
scm> (define (add x y) (+ x y))
add
scm> (add (- 5 3) (or 0 2))
2

scheme_eval 12 13 14 15
scheme_apply 1 2 3 4

Solution: 13 scheme_eval, 3 scheme_apply.
```
7. Identify the number of calls to scheme_eval and the number of calls to scheme_apply.
   (a) scm> (define pi 3.14)
      pi
   scm> (define (hack x)
         (cond
          ((= x pi) 'pwned)
          ((< x 0) (hack pi))
          (else (hack (- x 1))))
       hack
   Solution: 3 scheme_eval, 0 scheme_apply

   (b) scm> (hack 3.14)
       pwned
   Solution: 9 scheme_eval, 2 scheme_apply

   (c) scm> ((lambda (x) (hack x)) 0)
       pwned
   Solution: 39 scheme_eval, 10 scheme_apply

3. Iterators

8. What is difference between an iterator and an iterable?

   Solution: Iterator: Mutable object that tracks a position in a sequence, advancing on each call to next
   Iterable: Represents a sequence and returns a new iterator on each call to iter

   To use in an English sentence: Lists are "iterable". To go through a list, you make an object called an "iterator" to scan through the list.