

1. (3 points) Pin the Tail

Identify whether or not each of the following procedures uses a constant amount of space in a tail-recursive Scheme implementation (i.e. whether **every** recursive call is a tail call).

```
(define (copy lst result)
  (if (null? lst) result
      ((lambda (copy) copy) (copy (cdr lst)
                                   (append result (list (car lst)))))))
```

(As a reminder, `append` takes zero or more lists and constructs a new list with all of the lists' elements.)

copy is not tail-recursive. After the recursive call returns, we still have to apply a lambda procedure.

```
(define (broken lst) (broken (broken lst)))
```

broken is not tail-recursive. One of the recursive calls is not a tail call.

```
(define (is-ascending lst last-num)
  (if (null? lst) #t
      (and (is-ascending (cdr lst) (car lst)) (> (car lst) last-num))))
```

(Assume that this procedure is always called with a `last-num` that is less than all of the elements in the list.)

is-ascending is not tail-recursive. The recursive call isn't even in a tail context!

2. (4 points) Hail Recursion

Write a *tail-recursive* version of `hailstone`. This procedure accepts a positive integer `n` and an empty list `lst`, and returns a list containing the hailstone sequence that starts at `n`.

As an example, `(hailstone 5 '())` would return `(5 16 8 4 2 1)`.

```
(define (hailstone n lst)
  (cond ((= n 1) (append lst (list 1)))
        ((even? n) (hailstone (/ n 2) (append lst (list n))))
        (else (hailstone (+ 1 (* 3 n)) (append lst (list n))))))
```

3. (3 points) Humans Need Not Apply

What does `eval` do (in the context of an interpreter)? What does `apply` do?

eval parses expressions (all kinds of expressions; **eval** doesn't discriminate!), *evaluating* an expression's form to determine its meaning. On the other hand, **apply** handles function calls (it *applies* operators to arguments).

eval and **apply** are mutually recursive. Whenever **eval** encounters a function call, it sends the expression to **apply** to do the actual calling. In turn, **apply** uses **eval** to evaluate its arguments and to parse the body of user-defined functions.