General Instructions:

- Several problems required using particular other functions. In many cases, the auto-grader already checked for this, so you should not re-penalize for the same thing. In other cases, it was not feasible to check this automatically. The per-problem instructions will make clear what to look for and what has already been auto-graded.
- Mutation, such as `set!` or `set-mcar!` is generally poor style, except in problem 10, where it is essential to the solution. In other problems, give at most a 3 to solutions using mutation.

Problem 1:

Here is a sample solution:

```
1  (define (sequence low high stride)
2    (if (> low high)
3        null
4        (cons low (sequence (+ low stride) high stride))))
```

On this and all problems, do not penalize using the longer form (`define sequence` (`lambda` (`low high stride`) ... instead of using the syntactic sugar for function definitions as in the sample above.

There is little need for a solution more complicated than the one above, but it is okay to give a 5 to a solution that uses a local helper function to avoid passing `high` and `stride` recursively. It is also okay to use `cond` instead of `if` although `if` is usually better style when there are only two cases.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 2

Here is a sample solution:

```
1  (define (string-append-map xs suffix)
2    (map (lambda (s) (string-append s suffix)) xs))
```

The auto-grader already penalized solutions that did not use `string-append` and `map` as helper functions, so we do not need peer assessment to judge this same issue.

There is little benefit to a longer solution here, so probably give at most a 4 to solutions that use some form of `let`-expression.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 3

Here is a sample solution:

```
(define (list-nth-mod xs n)
  (cond [(< n 0) (error "list-nth-mod: n must be non-negative")]
        [(null? xs) (error "list-nth-mod: list must be non-empty")]
        [#t (let* ([len (length xs)]
                   [posn (remainder n len)])
              (car (list-tail xs posn)))]))
```

There are many reasonable style choices here, so mostly follow general guidelines for readable code.

A `cond` is probably better here than if because there are 3 clear cases, but a fine alternative is checking `(< n 0)` once outside of a local recursive helper function. It is also fine (probably even a little better than the sample above) to compute the length of `xs` once outside a local helper function.

It is not necessary to use a `let`-expression or `let*`-expression like in the sample above.

The auto-grader did not require using `length`, `remainder`, and `list-tail` because these were listed as "hints." You also do not *need* to check for these functions, but you should penalize solutions that are significantly more complicated because they are not using them.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 4

Here is a sample solution:

```
(define (stream-for-n-steps s n)
  (if (= n 0)
      null
      (let ([next (s)])
            (cons (car next) (stream-for-n-steps (cdr next) (- n 1))))))

```

The most important thing to check for is that each thunk in the stream is evaluated at most once, which is why the code above has a `let`-expression rather than `(car (s))` and `(cdr (s))`. Give at most a 4 if `s` is called twice. While such a solution is not horrible, it does matter in cases where stream elements have side effects or are expensive to compute. We expect this deduction to be common.

It is poor style to have extra *unnecessary* function wrapping related to thunks. For example, we would penalize the sample solution if `(cdr next)` were instead `(lambda () ((cdr next))`. Give at most a 4 for such a thing, or 3 if combined with the previous style issue (e.g., `(lambda () ((cdr (s))))`).

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 5

Here is a sample solution:

```
(define funny-number-stream
  (letrec ([f (lambda (n) (cons (if (= (remainder n 5) 0) (- n) n)
                                (lambda () (f (+ n 1)))))])
    (lambda () (f 1))))

```

While a local helper function (with `letrec` or a local `define`) is better style than a helper function at top-level, be lenient here and allow a 5 even with a top-level helper function.

There are various fine-style ways to implement the logic where the sample solution has `(if (= (remainder n 5) 0) (- n) n)`.

There should definitely not be any call to code that uses `stream-for-n-steps` or similar stream-processing code. The purpose here is to create a stream other code can use. Give at most a 2 for such code. However, there are elegant solutions that define `funny-number-stream` in terms of other streams using higher-order functions (the student would have to define) such as a map function over streams. Such code can receive a 5. If you see such code, it is probably used for problems 6-8 as well and you can continue to give high scores for an approach different from the sample solutions.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 6

Here are three different sample solutions:

```
(define dan-then-dog
  (letrec ([dan-st (lambda () (cons "dan.jpg" dog-st))]
           [dog-st (lambda () (cons "dog.jpg" dan-st))])
    dan-st))
```

```
(define (dan-then-dog)
  (cons "dan.jpg"
        (lambda () (cons "dog.jpg" dan-then-dog))))

```

```
(define dan-then-dog
  (letrec ([f (lambda (b)
                (if b
                    (cons "dan.jpg" (lambda () (f #f)))
                    (cons "dog.jpg" (lambda () (f #t)))))])
    (lambda () (f #t))))
```

As in the previous problem, be lenient and do not penalize solutions that put helper functions at top-level.

Give at most a 4 for code that has *unnecessary* function wrapping, such as replacing `dog-st` or `dan-st` in the first solution above with (`lambda` () (`dog-st`)) or (`lambda` () (`dan-st`)).

There should definitely not be any call to `stream-for-n-steps` or similar functions. Give at most a 2 for such code (but as necessary see comments in Problem 5 for solutions using higher-order stream functions).

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 7

Here is a sample solution:

```
(define (stream-add-zero s)
  (lambda ()
    (let ([next (s)])
      (cons (cons 0 (car next)) (stream-add-zero (cdr next))))))
```

As in Problem 4, the most important thing to check for is that each thunk in the stream is evaluated at most once, which is why the code above has a `let`-expression rather than (`car` (`s`)) and (`cdr` (`s`)). Give at most a 4 if `s` is called twice. While such a solution is not horrible, it does matter in cases where stream elements have side effects or are expensive to compute. We expect this deduction to be common.

As in Problem 5 and Problem 6, give at most a 4 for *unnecessary* function wrapping, such as (`lambda` () ((`cdr next`))) instead of (`cdr next`).

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 8

Here is a sample solution:

```
(define (cycle-lists xs ys)
  (letrec ([loop (lambda (n)
                   (cons (cons (list-nth-mod xs n)
                               (list-nth-mod ys n))
                         (lambda () (loop (+ n 1)))))])
    (lambda () (loop 0))))
```

As in previous problems, be lenient and do not penalize using a top-level helper function.

There are elegant solutions that do not use `list-nth-mod`, but such solutions should not be significantly longer than the sample solution.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 9

Here is a sample solution:

```
(define (vector-assoc v vec)
  (letrec ([loop (lambda (i)
                   (if (= i (vector-length vec))
                       #f
                       (let ([x (vector-ref vec i)])
                         (if (and (cons? x) (equal? (car x) v))
                             x
                             (loop (+ i 1))))))])
    (loop 0)))
```

The auto-grader already penalized solutions that did not use `vector-length`, `equal?`, and `vector-ref` as helper functions, so we do not need peer assessment to judge this same issue.

While it is slightly better style to call `vector-ref` and `vector-length` only once per iteration of the helper function as in the sample solution above, this is not required as long as the code is readable and elegant.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 10

Here is a sample solution:

```
(define (cached-assoc lst n)
  (let ([cache (make-vector n #f)]
        [next-to-replace 0])
    (lambda (v)
      (or (vector-assoc v cache)
          (let ([ans (assoc v lst)])
            (and ans
                 (begin (vector-set! cache next-to-replace ans)
                        (set! next-to-replace
                              (if (= (+ next-to-replace 1) n)
                                  0
                                  (+ next-to-replace 1)))
                        ans)))))))
```

The auto-grader already penalized solutions that did not use `vector`, `assoc`, and `vector-set!` as helper functions, so we do not need peer assessment to judge this same issue.

The essential thing to check (since it is too difficult for the auto-grader to do so) is that the cache and `next-to-replace` logic is in a `let`-expression (or `let*`-expression) outside the function bound to `cached-assoc` as otherwise the cache is not actually being used. Give at most a 2 if every call to the function returned by `cached-assoc` makes a new empty cache.

While the sample solution uses `or` and `and`, other logic, such as using `if`, is fine and in fact may be easier to read.

Similarly, the logic for the updated value of `next-to-replace` can be performed in various ways, such as using `remainder`.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Challenge problem

You do not need to assess the challenge problem, but you are welcome to provide text feedback for it if you wish. Here is a sample solution:

```
(define-syntax while-less
  (syntax-rules (do)
    ((while-less x do y)
      (let ([z x])
        (letrec ([loop (lambda ()
                         (let ([w y])
                           (if (or (not (number? w)) (>= w z))
                               #t
                               (loop))))])
          (loop))))))
```

Mark as completed