Q1 (Streams): Define a function zero-through-three that returns a stream which cycles through the values 0, 1, 2, 3 every time it’s called, starting with 0 (Racket has a function remainder that may be useful).

```scheme
(define zero-through-three
  (letrec ([f (lambda (x)
                (cons (remainder x 4)
                      (lambda () (f (+ x 1)))))]
           (lambda () (f 0)))))
```

Q2 (Streams): Define a function zero-through-n that takes a number n and returns a stream which cycles through the values 0, 1, 2, ..., n-1 every time it’s called, starting with 0. You may assume n is non-negative.

```scheme
(define (zero-through-n n)
  (letrec ([f (lambda (x)
                (cons (remainder x n)
                      (lambda () (f (+ x 1)))))]
           (lambda () (f 0)))))
```

Q3 (2019 Summer Final Q2 (a)): 
2. (Thunks and Streams – 18 points) As in class, we define a stream to be a thunk that when called returns a pair where the cdr of the pair is a stream. We assume all streams are pure (no printing, mutation, etc.). Assume the following streams are defined:

\[ \text{nats} = 1, 2, 3, 4, 5, \ldots \text{ (the natural numbers)} \]
\[ \text{evens} = 2, 4, 6, 8, 10, \ldots \text{ (the positive even integers)} \]
\[ \text{negs} = -1, -2, -3, -4, -5, \ldots \text{ (the negative integers)} \]

a) Write a Racket function `weave-streams` that takes two stream arguments, `s1` and `s2`, and returns a stream. The resulting stream should contain alternating elements from the two argument streams. That is, the odd-numbered elements of the result stream should be elements (in order) from `s1`, and the even-numbered elements of the result stream should be elements (in order) from `s2`.

For example, `(weave-streams nats negs)` would represent `1, -1, 2, -2, 3, -3, \ldots`.

```
(define (weave-streams s1 s2)
  (letrec ([loop (lambda (curr next)
                  (lambda () (cons (car (curr))
                                 (weave-streams next (cdr (curr)))))])
    (loop s1 s2)))
```