1. Write a SCHEME function that builds an object representing a number; initially the object should represent the number 0. It should expose the following methods:

   increment should increase x by one;

   decrement should decrease x by one;

   power-up should multiply x by ten;

   power-down should divide the number by ten, ignoring any remainder (which is to say that it initially sets the least significant bit to zero and then divides by ten);

   ones-digit should return the ones digit of x;

   zero? should return `#t` if x is zero, and `#f` otherwise.

   positive? should return `#t` if x is greater than 0, and `#f` otherwise.

   Your object should maintain the number as a list of digits (along with a Boolean value that reflects the sign of the number).

   This solution maintains the number as a list of digits, starting with the least significant, along with a single numeric value `sign`, which maintains the sign. (It takes the value 1 for positive numbers, 0 for zero, and −1 for negative numbers.)

```
(define (number)
  (let ((digits '())
        (sign 0))
    (define (positive?) (> sign 0))
    (define (negative?) (< sign 0))
    (define (zero?) (eq? sign 0))
    (define (ones-digit)
      (if (zero?) 0 (car digits)))
    (define (power-up)
      (if (zero?)
          (set! digits '())
          (set! digits (cons 0 digits))))
    (define (power-down)
      (cond ((zero?) (set! digits '()))
            ((null?  (cdr digits)) (begin (set! digits (list 0))
                                          (set! sign 0)))
            (else (set! digits (cdr digits)))))
    (define (inc-digs elements)
      (cond ((null? elements) (list 1))
            ((< (car elements) 9) (cons (+ 1 (car elements))
                                        (cdr elements)))
            (else (cons 0 (inc-digs (cdr elements))))))
    (define (dec-digs elements)
      (cond ((> (car elements) 1) (cons (- (car elements) 1)
                                         (cdr elements)))
            ((null? (cdr elements)) '())
            (else (cons 9 (dec-digs (cdr elements))))))
    (define (increment)
      (cond ((positive?) (set! digits (inc-digs digits)))
            ((zero?) (begin (set! digits '(1))
                            (set! sign 1)))
            (else (begin (set! digits (dec-digs digits))
                         (if (null? digits) (set! sign 0))))))
    (define (decrement)
      (cond ((positive?) (begin (set! digits (dec-digs digits))
                                (if (null? digits)
                                    (set! sign 0))))
            ((zero?) (begin (set! digits '(1))
                            (set! sign -1)))
            (else (set! digits (inc-digs digits)))))
    (define (dispatch method)
      (cond ((eq? method 'zero) zero?)
            ((eq? method 'positive) positive?)
            ((eq? method 'power-up) power-up)
            ((eq? method 'power-down) power-down)
            ((eq? method 'increment) increment)
            ((eq? method 'decrement) decrement)
            ((eq? method 'ones-digit) ones-digit)
            ((eq? method 'show) (lambda () digits))))
    dispatch))
```

STREAMS

Recall that a SCHEME stream is a pair of the form

```
(a . (lambda () ...)).
```

The `car` of the pair is the first element of the stream. The job of the cdr of the pair (the lambda expression) is to return the rest of the stream (which is another stream, and must have exactly the same structure) when it is called with no arguments.

Using this convention, one can define

```
(define (first str) (car str))
(define (rest str) ((cdr str)))
```

As an example, the function `int-stream` returns the stream of positive integers $a, a+1, a+2, \ldots$:

```
(define (int-stream a)
  (cons a (lambda () (int-stream (+ 1 a)))))
```

Then, to continue our example

```
> (define a (int-stream 5))
> a
'(5 . #<procedure>)
> (rest a)
'(6 . #<procedure>)
> (rest (rest a))
```

```
'(7 . #<procedure>)
> (first (rest (rest a)))
7
```

There are natural ways to *operate* on streams. For example, the following function takes two streams and returns the stream containing the "elementwise product."

```
(define (str-product s t)
  (cons (* (first s) (first t))
        (lambda () (str-product (rest s) (rest t)))))
```

Then, for example

```
> (define a (str-product (int-stream 5) (int-stream 2)))
> a
'(10 . #<procedure>)
> (rest a)
'(18 . #<procedure>)
> (rest (rest a))
'(28 . #<procedure>)
```

You will notice that in the example above, we created a function that returned the stream; when such a stream is created, it will have embedded in the `cdr` of the stream a call to this function. We mention that there are circumstances where it may be helpful for the definition of the stream to carry additional information (beyond simply the first element of the stream). As an example, consider the stream of Fibonacci numbers.

```
(define (fib-stream current next)
  (cons current (lambda ()
                  (fib-stream next (+ current next)))))
```

To generate the stream of regular Fibonacci numbers, you would use the call `(fib-stream 0 1)`.

A final comment: In class, we defined the special functions

```
  (define (delay x) (lambda () x))
  (define (force x) (x))
```

and then used these in our definitions of streams. This is a well-established convention and it would be great to use it in your code. Note, however, that SCHEME has built-in functions (called `delay` and `force`) that behave almost exactly the as the functions above do; one quirk, however, is that if an expression has been delayed (via `(delay E)`), the only way you can get SCHEME to evaluate it is via (force ...). (You cannot simply call it on no arguments.) There is actually a good reason for this.

---

2. Write a function `merge` that *merges* two streams. Specifically, if s and t are two streams of positive numbers in increasing order, `(merge s t)` should be the stream containing all elements appearing in either s or t, also in increasing (sorted) order. For example, if s is the stream of all perfect squares

$$1, 4, 9, 16, 25, 36, \ldots$$

and t is the stream of all perfect cubes

$$1, 8, 27, 64, 125, 216, \ldots$$

then `(merge s t)` would be the stream

$$1, 1, 4, 8, 9, 16, 25, 27, 36, \ldots .$$

Note that there are numbers, such as 1 and $(2^3)^2 = (2^2)^3 = 2^6 = 64$ that are *both* perfect squares and perfect cubes. In this case, your merge function should output the numbers twice.

```
(define (merge-streams s t)
  (cond ((null? s) t)
        ((null? t) s)
        ((< (first s) (first t))
         (cons (first s)
               (delay (merge-streams (rest s) t))))
        (else
         (cons (first t)
               (delay (merge-streams s (rest t)))))))
```

3. Picking up from the previous problem, define a merge function that avoids repeated elements (which is to say that if an element appears in both streams, it should appear in the output stream only once).

   Use this to build a stream that generates the ordered list of all positive numbers that are divisible by 2, 3, or 5. To do this, merge together individual streams for the multiples of 2, 3, and 5. Your stream should output these numbers in sorted order, and should output any given integer only once.

```
(define (merge-clean-s s t)
  (cond ((null? s) t)
        ((null? t) s)
        ((< (first s) (first t))
         (cons (first s)
               (delay (merge-clean-s (rest s) t))))
        ((< (first t) (first s))
         (cons (first t)
               (delay (merge-clean-s s (rest t)))))
        (else (merge-clean-s (rest t) s))))
```

4. **(Stream zip.)** Define a `stream-zip` function that takes two streams

$$s = s_1, \quad s_2, \quad \ldots \qquad \text{and} \qquad t = t_1, \quad t_2, \quad \ldots$$

   as input and returns the stream of pairs

$$(s_1 \, . \, t_1), \quad (s_2 \, . \, t_2), \quad \ldots$$

```
(define (zip-s s t)
  (cond ((null? s) '())
        ((null? t) '())
        (else (cons (cons (first s) (first t))
                    (delay (zip-s (rest s) (rest t)))))))
```

5. **(The Mandelbrot Stream.)** The Mandelbrot set is a remarkable subset of the complex numbers. It's nice to look at, and is related to the mathematical study of "chaos."

The set contains all complex numbers z with the property that the sequence

$$0, \quad z, \quad z^2+z, \quad (z^2+z)^2+z, \quad \ldots$$

contains only "short" complex numbers (that have length ≤ 2). There is a standard way that mathematicians "picture" the complex numbers (by placing the number $a+b\mathbf{i}$ at position (a,b) in the plane); in this problem, you will use streams to create an image of the Mandelbrot set in the plane. (See below for an example of the kind of image you can produce with this code.)

You won't need to know anything special about the complex numbers in order to solve the problem.

(a) Recall that a complex number can be written $a+b\mathbf{i}$, where a and b are real numbers. Adding two complex numbers is straightforward:

$$(a+b\mathbf{i})+(c+d\mathbf{i})=(a+c)+(b+d)\mathbf{i}\,.$$

Multiplication of complex numbers is defined

$$(a+b\mathbf{i})\cdot(c+d\mathbf{i})=(ac-bd)+(ad+bc)\mathbf{i}\,.$$

Finally, the *length* of a complex number $a+b\mathbf{i}$, denoted $|a+b\mathbf{i}|$, is defined to be $\sqrt{a^2+b^2}$. For this problem, I recommend that you simply represent complex numbers as pairs, so the complex number $a+b\mathbf{i}$ would be represented as the pair `(a . b)`.

Write the functions `c-add`, `c-multiply`, and `c-length`, which add, multiply, and compute the length of complex numbers. (Your addition and multiplication functions should take two complexes (pairs) and return a complex with the result. Your length function should take a complex and return its length, a real number.)

```
(define (re a) (car a))
(define (im a) (cdr a))

(define (c-add a b)
  (cons (+ (re a) (re b))
        (+ (im a) (im b))))

(define (c-multiply a b)
  (cons (- (* (re a) (re b))
           (* (im a) (im b)))
        (+ (* (re a) (im b))
           (* (im a) (re b)))))

(define (c-square a)
  (c-multiply a a))

(define (c-length a)
  (sqrt (+ (* (re a) (re a))
           (* (im a) (im a)))))
```

(b) In this problem you will implement a famous stream of complex numbers that define a remarkable fractal—the Mandelbrot set. An image of the Mandelbrot set appears in Figure 1.

Let $z=\alpha+\beta i$ be a fixed complex number and let f_z be the function $f_z(y)=y^2+z$. (Here y is also meant to be a complex number.)

The *Mandelbrot-z stream* consists of the infinite sequence of complex numbers

$$0, \quad f_z(0), \quad f_z(f_z(0)), \quad f_z(f_z(f_z(0))), \quad \ldots.$$

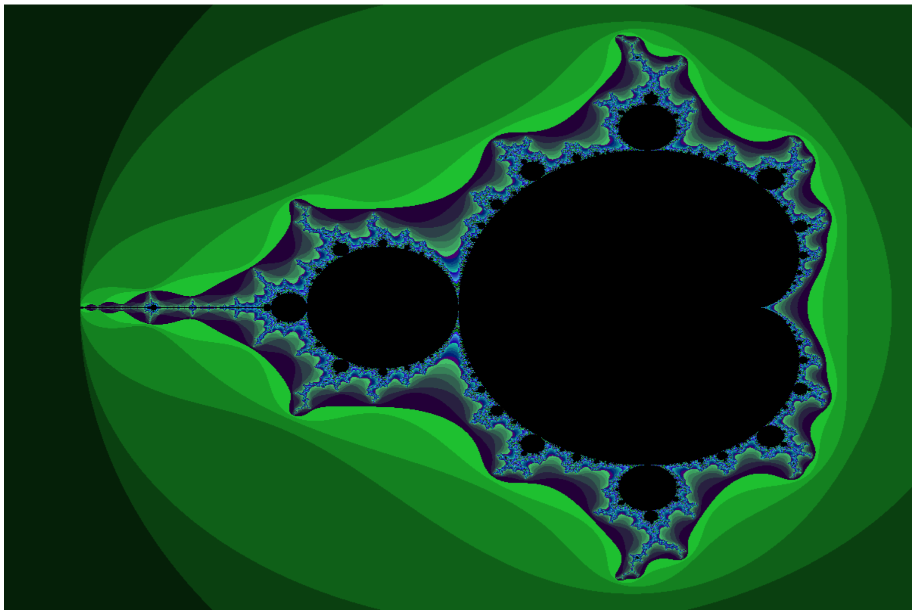

Figure 1: The Mandelbrot set.

Define a function (`mandelbrot z`) that returns the Mandelbrot-z stream. (Along the way, it may help you to define a function (`mandelbrot z a`), which returns the stream

$$a, \quad f_z(a), \quad f_z(f_z(a)), \quad f_z(f_z(f_z(a))), \quad \ldots .)$$

```
(define (mandelbrot-stream-seed z a)
  (cons a
        (delay (mandelbrot-stream-seed z (c-add
                                           z
                                           (c-square a))))))

(define (mandelbrot-stream z)
        (mandelbrot-stream-seed z '(0 . 0)))
```

(c) Recall the `stream-map` function, which takes a function f and a stream $s = s_1, s_2, \ldots$ and returns the stream $f(s_1), f(s_2), \ldots$, which we defined in class. Correctly define `stream-map`.
Using (`mandelbrot z`) and `stream-map`, construct the stream (`mandelbrot-l z`) whose elements are the *lengths* of the elements of the stream (`mandelbrot z`).

```
(define (s-map f s)
  (cons (f (car s))
        (delay (s-map f (rest s)))))

(define (mandelbrot-length-stream z)
  (s-map c-length (mandelbrot-stream z)))
```

(d) Next, define a function (`depth z max`) which computes the *index* of the first element of the stream (`mandelbrot-l z`) that is larger than 2. For example, the stream (`mandelbrot-l '(1/2 . 1/2)`) contains the numbers

$$0, \quad .707, \quad 1.118, \quad 1.521, \quad 1.706, \quad 3.550, \quad \ldots$$

and thus the index of the first entry that is larger than 2 is 5 (because the 5th element of the stream is the first one that is larger than 2).

Now, there are Mandelbrot streams for which this never happens (that is, the lengths are always less than 2). To protect your function from this, it may give up after `max` steps; specifically, if it does not find an element of the stream that is at least 2 in the first `max` elements, it should just return `max`.

Now, finally, you are ready to generate the Mandelbrot set. One you have `depth` defined, use the code below, which will write the image to a file called `mandelbrot.png`. (You'll have to be in Racket mode for this to work.)

```
(define (depth z max)
  (define (depth-iterator s iterate)
    (if (or (>= (first s) 2)
            (>= iterate max))
        iterate
        (depth-iterator (rest s) (+ iterate 1))))
  (depth-iterator (mandelbrot-length-stream z) 0))

(define (iterations a z i)
  (define z-prime (+ (* z z) a))
  (if (or (= i 255) (> (magnitude z-prime) 2))
      i
      (iterations a z-prime (add1 i))))
```

```
(define (iter->color i)
  (if (= i 255)
      (make-object color% "black")
      (make-object color% (* 5 (modulo i 15))
                          (* 32 (modulo i 7))
                          (* 8 (modulo i 31)))))

(define (mandelbrot width height)
  (define target (make-bitmap width height))
  (define dc (new bitmap-dc% [bitmap target]))
  (for* ([x width] [y height])
    (define real-x (- (* 3.0 (/ x width)) 2.25))
    (define real-y (- (* 2.5 (/ y height)) 1.25))
    (send dc set-pen
      (iter->color (depth (cons real-x real-y) 255)) 1 'solid)
    (send dc draw-point x y))
  (send target save-file "mandelbrot.png" 'png))
```