Exercise 1 Doing something thrice once is it do it three times. Doing something thrice twice is that for each thing we do thrice, we do that thrice, and this is nine times. Doing something thrice thrice is that for each thing we do nine times, we do that thrice, and this is 27 times.

1. 27 applications of increment to 6 is 33.
2. 27 applications of identity to compose is compose.
3. 27 applications of square to 1 is 1.
4. 27 applications of square to 2 is a big number.

Exercise 2

1.

$$\text{Sch-Num} \rightarrow \text{Curve}$$

2.

```
(define (vertical-line point length)
  (lambda (t)
    (make-point (x-of point) (* t length))))
```

3.

$$(\text{Point}, \text{Sch-Num}) \rightarrow \text{Curve}$$

Exercise 3 A procedure of type Curve-Transform takes a Curve as argument and returns a Curve. The returned Curve is a procedure taking a Unit-Interval and returning a Point. We apply the input Curve to `t` to get the Point of the Curve with that particular parameter. Then we transform the Point by manipulating the individual coordinates. In total, we transform a Curve at a particular Point by transforming that Point.

For `reflect-through-y-axis`, we map $(x, y) \rightarrow (-x, y)$.

```
(define (reflect-through-y-axis curve)
  (lambda (t)
    (let ((ct (curve t)))
      (make-point (- (x-of ct))
                  (y-of ct)))))
```

Exercise 4 After transforming `curve2` we need some way of having both Curves be "combined" into one. `connect-rigidly` does exactly that.

```
(define (connect-ends curve1 curve2)
  (let* ((curve1-end-point (curve1 1))
 (curve2-start-point (curve2 0))
 (new-curve2 ((translate (- (x-of curve1-end-point) (x-of curve2-start-point))
 (- (y-of curve1-end-point) (y-of curve2-start-point)))
      curve2)))
    (connect-rigidly curve1 new-curve2)))
```

Exercise 5 This was particularly brutal!

Ensure you are using Edwin. This ensures that you are using mit-scheme. Evaluate `(enumerate-graphics-types)` to find what device type you can use. For me it was `(x)` as a value. Thus I use the symbol `x` as argument in the next statement. Evaluate

```
(define g1 (make-graphics-device 'x))
(define g2 (make-graphics-device 'x))
(define g3 (make-graphics-device 'x))
```

Each expression creates a new window we can draw on. However, we need only define `g1` for the exercises.

Then load the programs given on the website.

```
(load ``curves.scm'')
(load ``drawing.scm'')
(load ``utils.scm'')
```

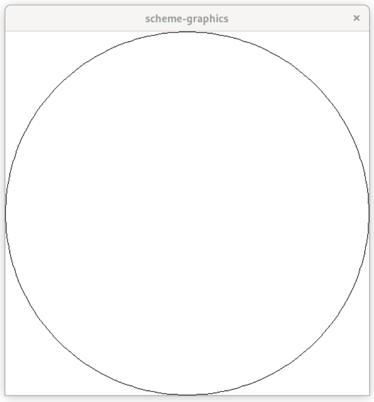

Figure 1: `((draw-connected g1 200) unit-circle)`

There is a minuscule difference.

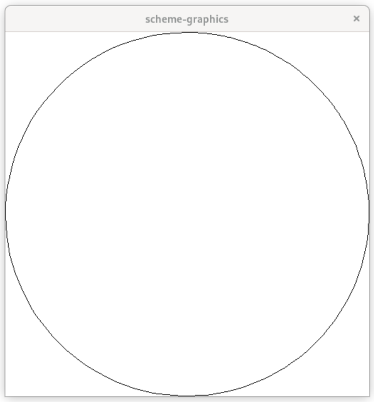

Figure 2: `((draw-connected g1 200) alternative-unit-circle)`

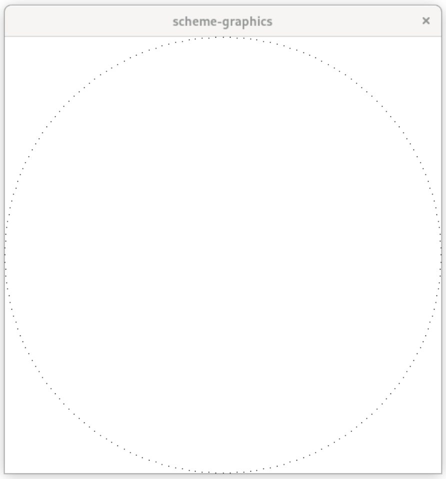

Figure 3: `((draw-points-on g1 200) unit-circle)`

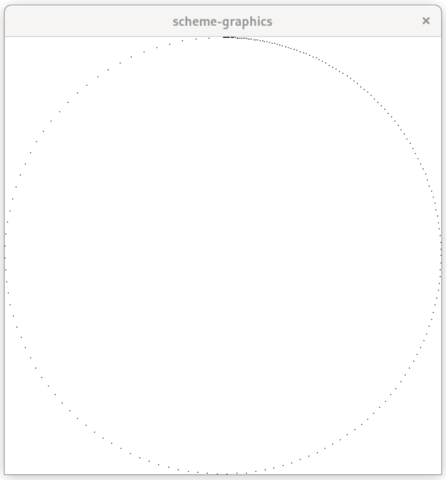

Figure 4: `((draw-points-on g1 200) alternative-unit-circle)`

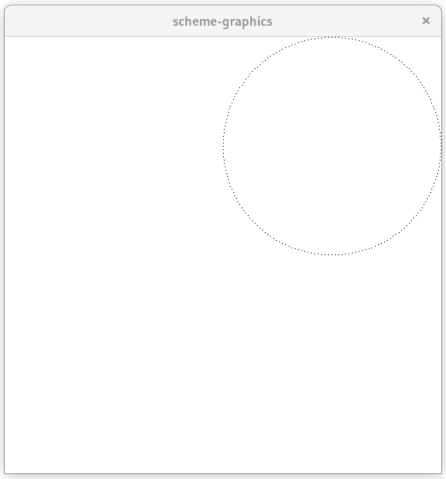

Figure 5: `((draw-points-squeezed-to-window g1 200) unit-circle)`

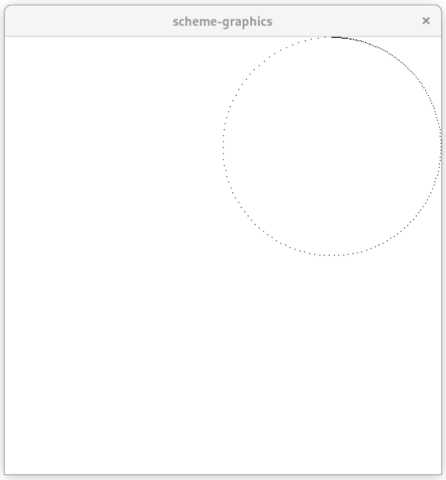

Figure 6: `((draw-points-squeezed-to-window g1 200) alternative-unit-circle)`

Exercise 6.A

```
(define (show-points-gosper window level number-of-points initial-curve)
  ((draw-points-on window number-of-points)
   ((squeeze-rectangular-portion -.5 1.5 -.5 1.5)
    ((repeated gosperize level)
     initial-curve))))
```

Exericse 6.B

```
(define (half-unit-circle t)
  (make-point (cos (* pi t))
              (sin (* pi t))))
```

Exercise 7.A

```
(define (param-gosperize theta)
  (lambda (curve)
    (put-in-standard-position
     (connect-ends ((rotate-around-origin theta) curve)
                   ((rotate-around-origin (- theta)) curve)))))
```

Exercise 7.B Sorry I did not print them.

```
((draw-connected g1 200) (param-gosper 2 (lambda (n) (/ pi (+ n 2)))))
((draw-connected g1 200) (param-gosper 2 (lambda (n) (/ pi (expt 1.3 n)))))
```

Exercise 7.C

```
(show-time (lambda () ((gosper-curve 10) .1)))                      ;       => real time: 0
(show-time (lambda () ((param-gosper 10 (lambda (level) pi/4)) .1))) ;normal => real time: 0
(show-time (lambda () ((param-gosper 10 (lambda (level) pi/4)) .1))) ;put-in => real time: 8
```

Exercise 8.A Since `gosper-curve` calls `repeated`, which is in terms of `compose`, calling `(curve t)` twice in `bens-rotate` leads to an exponential process. That is, the number of calls to `(curve t)` double with each composition.

Exercise 8.B

procedure	level	calls to `x-of`
`bens-rotate`	4	115
`bens-rotate`	5	218
`rotate-around-origin`	4	21
`rotate-around-origin`	5	26

Exercise 9.A Unfortunately, as the figures are lost and that the Gosper curves we generated are different than the Gosper curves we see online, it is hard to tell what our curve should look like.