

# Graphing with Higher-order Procedures

**Date of Submission: 2nd February 11:59:59pm**

Acknowledgement: This assignment is based on a similar assignment in the MIT SICP site.

One of the things that makes Scheme different from other common programming languages is the ability to operate with *higher-order procedures*, namely, procedures that manipulate and generate other procedures. This problem set will give you extensive practice with higher-order procedures, in the context of a language for graphing two-dimensional curves and other shapes.

## 1 Higher Order Procedures

In this assignment we use many procedures which may be applied to many different types of arguments and may return different types of values.

If `f` and `g` are procedures, then we may *compose* them:

```
(define (compose f g)
  (lambda (x)
    (f (g x))))
```

Thus, for example `(compose square log)` is the procedure that returns the square of the logarithm of its argument, while `(compose log square)` returns the logarithm of the square of its argument:

```
(log 2)
;Value: .6931471805599453
```

```
((compose square log) 2)
;Value: .4804530139182014
```

```
((compose log square) 2)
;Value: 1.3862943611198906
```

Just as squaring a number multiplies the number by itself, `thrice` of a function composes the function three times. That is, `((thrice f) n)` will return the same number as `(f(f(f n)))`:

```

(define (thrice f)
  (compose (compose f f) f))

((thrice square) 3)
;Value: 6561

(square (square (square 3)))
;Value: 6561

```

As used above, `thrice` takes as input a function from numbers to numbers and returns the same kind of function. But `thrice` will actually work for other kinds of input functions. It is enough for the input function to have a type of the form  $T$  to  $T$ , where  $T$  may be any type.

Composition, like multiplication, may be iterated. Consider the following:

```

(define (identity x) x)

(define (repeated f n)
  (if (= n 0)
      identity
      (compose f (repeated f (- n 1)))))

((repeated sin 5) 3.1)
;Value: 4.1532801333692235e-2

(sin(sin(sin(sin(sin 3.1)))))
;Value: 4.1532801333692235e-2

```

**Exercise 1.** The type of `thrice` is of the form  $T'$  to  $T'$  (where  $T'$  happens to equal  $(T \text{ to } T)$ ), so we can legitimately use `thrice` as an input to `thrice`!

For what value of `n` will `((thrice thrice) f) 0` return the same value<sup>1</sup> as `((repeated f n) 0)`?

See if you can now predict what will happen when the following expressions are evaluated. Briefly explain what goes on in each case. If you cannot figure out what is happening or confirm your predictions using the stepper facility of DrRacket illustrated in the class.

---

<sup>1</sup>“Sameness” of procedure values is a sticky issue which we don’t want to get into here. We can avoid it by assuming that `f` is bound to a value of type  $F$ , so evaluation of `((thrice thrice) f) 0` will return a number.

1. (((thrice thrice) oneplus) 6)
2. (((thrice thrice) identity) compose)
3. (((thrice thrice) square) 1)
4. (((thrice thrice) square) 2).

You do not have to submit Exercise 1.

## 2 Curves as Procedures and Data

We’re going to develop a language for defining and drawing planar curves. We’d like to plot points, construct graphs of functions, transform curves by scaling and rotating, and so on. One of the key ideas that is that a well-designed language has parts that combine to make new parts that themselves can be combined. This property is called *closure*.

A planar curve in “parametric form” can be described mathematically as a function from parameter values to points in the plane. For example, we could describe the *unit-circle* as the function taking  $t$  to  $(\cos 2\pi t, \sin 2\pi t)$  where  $t$  ranges over the unit interval  $[0, 1]$ . We shall now describe how a point is represented.

To work with points, we need a *constructor*, `make-point`, which constructs points from numbers:

```
(define (make-point x y)
  (lambda (bit)
    (if (zero? bit) x y)))
```

WE also define *selectors*, `x-of` and `y-of`, for getting the  $x$  and  $y$  co-ordinates of a point. Here is one way to do this without `cons` `car` and `cdr`.

```
(define (x-of point)
  (point 0))

(define (y-of point)
  (point 1))
```

For example, we can define the curve `unit-circle` and the curve `unit-line` (along the  $x$ -axis):

```

(define (unit-circle t)
  (make-point (sin (* 2pi t))
              (cos (* 2pi t))))

(define (unit-line-at y)
  (lambda (t) (make-point t y)))

(define unit-line (unit-line-at 0))

```

### Exercise 2:

1. Define a procedure (**vertical-line** *p* *l*) with two arguments, a point and a length, that returns a vertical line of length *l* beginning at the point *p*. Thus you should be able to use this as ((**vertical-line** (make-point 2 3) 5) .25), which means “make a vertical line of length 5 starting at point (2,3), and return the point lying at the parameter value .25”.

In addition to the direct construction of Curve’s such as **unit-circle** or **unit-line**, we can use elementary Cartesian geometry in designing Scheme procedures which *operate* on Curve’s. For example, the mapping  $(x, y) \rightarrow (-y, x)$  rotates the plane by  $\pi/2$ , so

```

(define (rotate-pi/2 curve)
  (lambda (t)
    (let ((ct (curve t)))
      (make-point
        (- (y-of ct))
        (x-of ct)))))

```

defines a procedure which takes a curve and transforms it into another, rotated, curve.

**Exercise 3:** You can define several useful curve-transformations. Define

- (**reflect-through-y-axis** *curve*), which turns a curve into its mirror image. Thus you should be able to say (**reflect-through-y-axis** **unit-line**).

- `(translate x y)`, which rigidly moves a curve given distances along the  $x$  and  $y$  axes. Again, you should be able to use this function as `((translate 2 3) unit-line)`.
- `(scale x y)`, which stretches a curve along the  $x$  and  $y$  coordinates by given scale factors, and
- `(rotate-around-origin radians)` returns a Curve-Transform which rotates a curve by a given number of radians.
- A convenient Curve-Transform is `(put-in-standard-position curve)`. We'll say a curve is in *standard position* if its start and end points are the same as the unit-line, namely it starts at the origin,  $(0,0)$ , and ends at the point  $(1,0)$ . We can put any curve whose start and end-points are not the same into standard position by rigidly translating it so its starting point is at the origin, then rotating it about the origin to put its endpoint on the  $x$  axis, then scaling it to put the endpoint at  $(1,0)$ .

It is useful to have operations which combine curves into new ones. We let Binary-Transform be the type of binary operations on curves. The procedure `connect-rigidly` is a simple Binary-Transform. Evaluation of `(connect-rigidly curve1 curve2)` returns a curve consisting of `curve1` followed by `curve2`; the starting point of the curve `(connect-rigidly curve1 curve2)` is the same as that of `curve1` and the end point is the same as that of `curve2`.

```
(define (connect-rigidly curve1 curve2)
  (lambda (t)
    (if (< t (/ 1 2))
        (curve1 (* 2 t))
        (curve2 (- (* 2 t) 1)))))
```

**Exercise 4:** There is another, possibly more natural, way of connecting curves. The curve returned by `(connect-ends curve1 curve2)` consists of a copy of `curve1` followed by a copy of `curve2` after it has been rigidly translated so its starting point coincides with the end point of `curve1`.

Write a definition of the Binary-Transform `connect-ends`.

### 3 Drawing Curves

To draw curves in DrRacket, copy the lines given below at the beginning of the definition area:

```
;The facility to plot a curve.

(require plot)
(plot-new-window? #t)
(plot-width 900)
(plot-height 900)

(define (draw curve)
  (plot (parametric
        (lambda (t) (vector (x-of (curve t))
                             (y-of (curve t))))
        0 1 #:width 1 #:samples 20000
        #:x-min -0.5 #:x-max 1.5
        #:y-min -0.5 #:y-max 1.5)))

; plot-width and plot-height give the size of the window
; #:width refers to the thickness of the line
; #:samples is the number of sample points. decrease it if
; your program takes too much time
; #:x-min -1 #:x-max 2 says that the graph is plotted in
; the x-axis range -1 to 2
```

And to draw a curve, `unit-line`, say:

```
(draw unit-line)
```

Try to generate different curves using the functions given above, and draw them. Don't forget to say "WOW" if you are able to draw a particularly beautiful curve.

#### 3.1 Fractal Curves

To show off the power of our drawing language, let's use it to explore fractal curves. Fractals have striking mathematical properties.<sup>2</sup> Fractals have

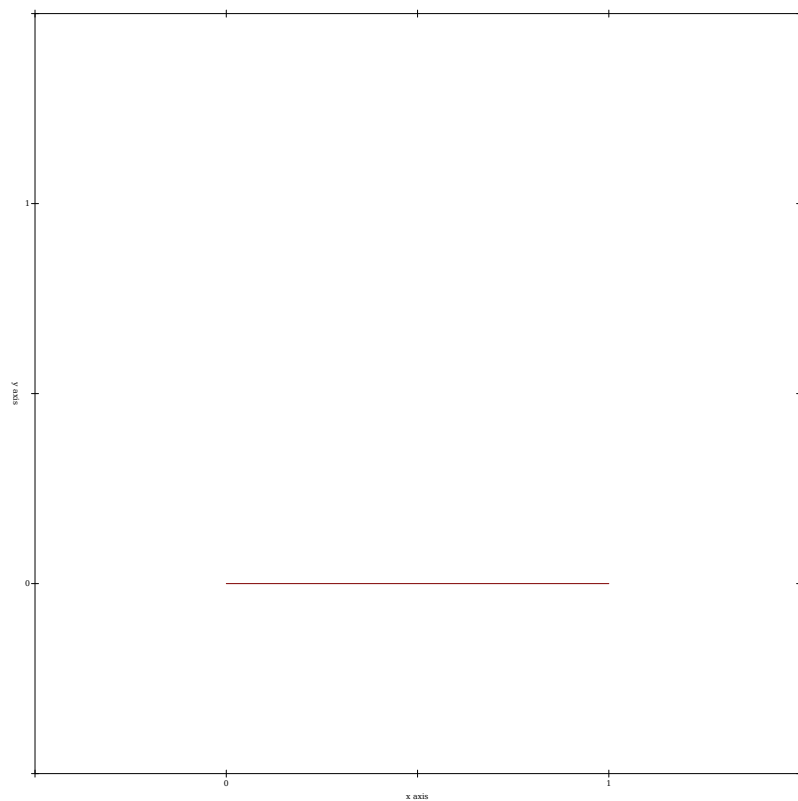
---

<sup>2</sup>If you expand a small part of a fractal curve, you get something similar to the original. The Gosper curve, for example, is neither a true 1-dimensional curve, nor a 2-dimensional region of the plane, but rather something in between.

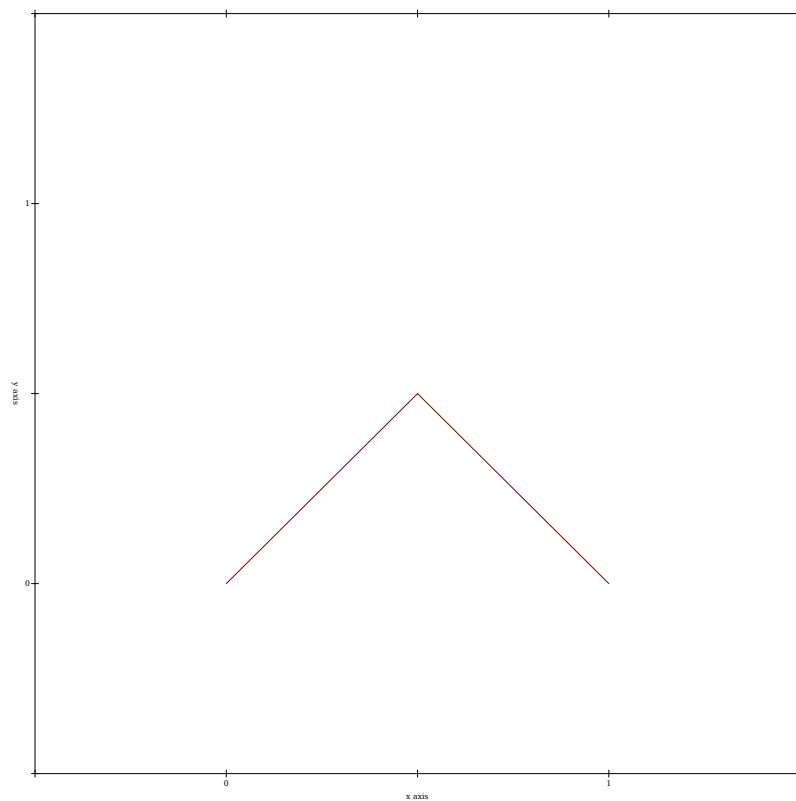
received a lot of attention over the past few years, partly because they tend to arise in the theory of nonlinear differential equations, but also because they are pretty, and their finite approximations can be easily generated with recursive computer programs.

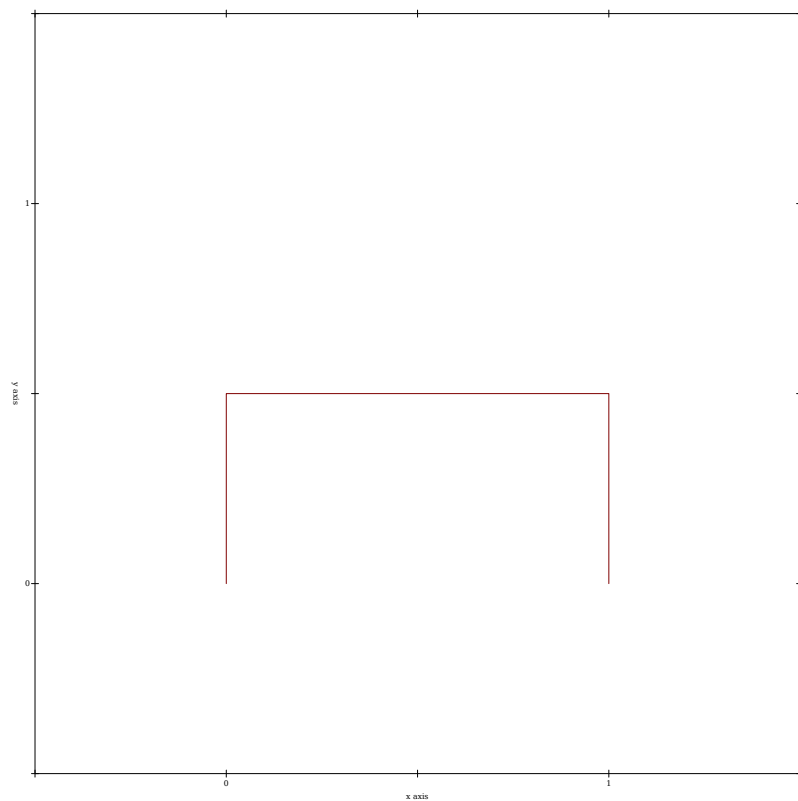
For example, Bill Gosper discovered that the infinite repetition of a very simple process creates a rather beautiful image, now called the *Gosper C Curve*. At each step of this process there is an approximation to the Gosper curve. The next approximation is obtained by adjoining two scaled copies of the current approximation, each rotated by 45 degrees.

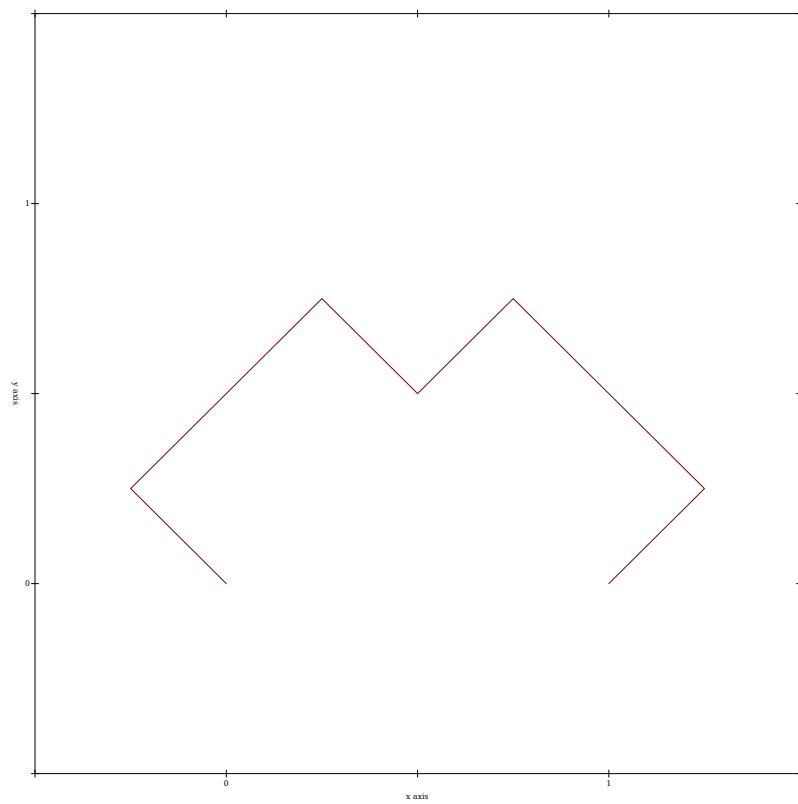
The figures below show the first, second, third, fourth and eighth approximations to the Gosper curve, where we stop after a certain number of levels: a level-0 curve is simply a straight line; a level-1 curve consists of two level-0 curves; a level 2 curve consists of two level-1 curves, and so on. The figure also illustrates a recursive strategy for making the next level of approximation: a level- $n$  curve is made from two level- $(n - 1)$  curves, each scaled to be  $\sqrt{2}/2$  times the length of the original curve. One of the component curves is rotated by  $\pi/4$  (45 degrees) and the other is rotated by  $-\pi/4$ . After each piece is scaled and rotated, it must be translated so that the ending point of the first piece is continuous with the starting point of the second piece.

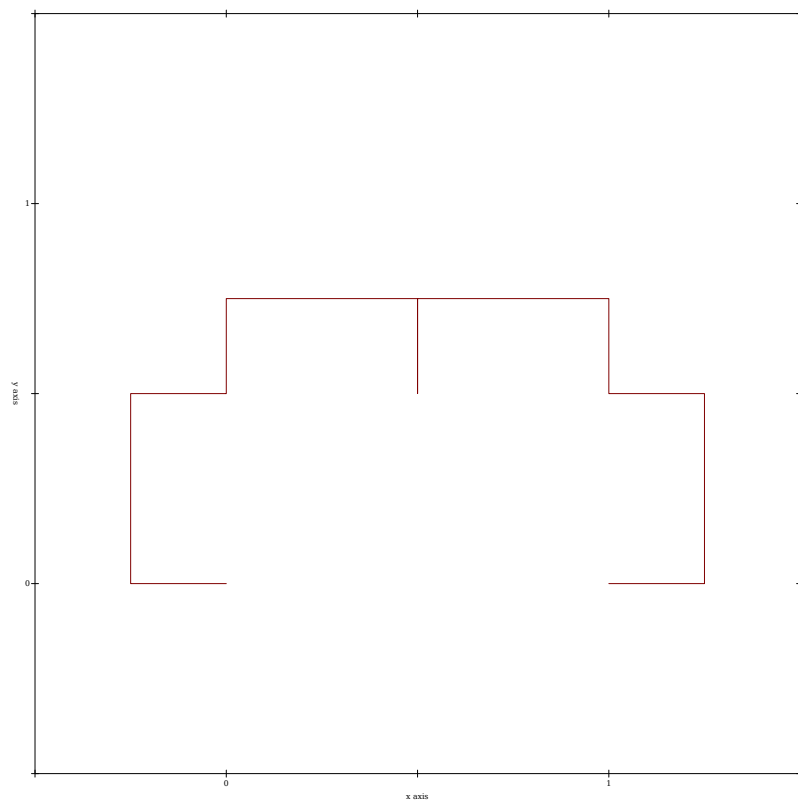


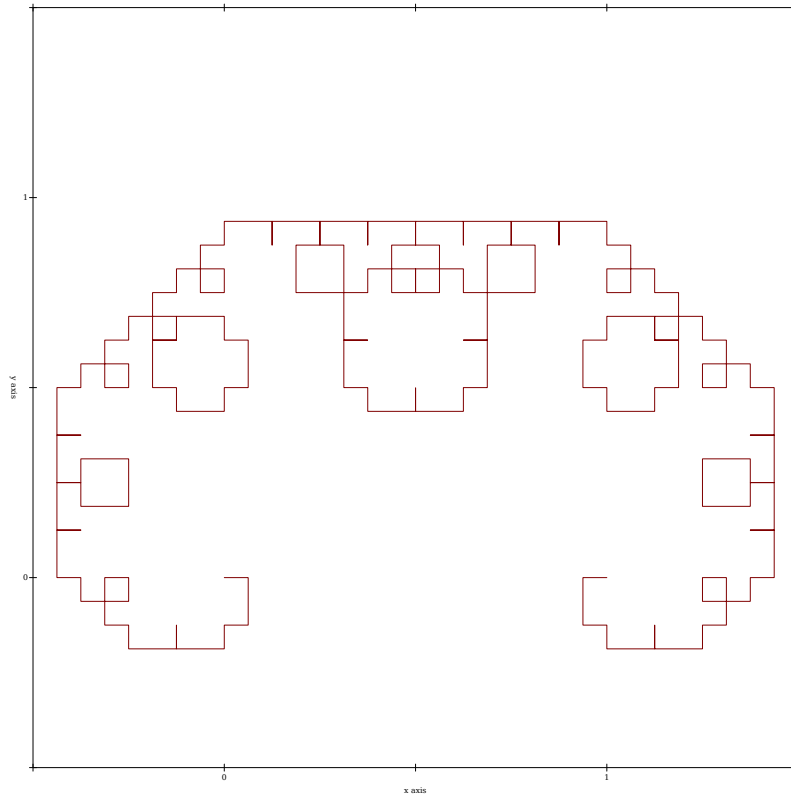












We assume that the approximation we are given to improve (named `curve` in the procedure) is in standard position. By doing some geometry, you can figure out that the second curve, after being scaled and rotated, must be translated right by `.5` and up by `.5`, so its beginning coincides with the end-point of the rotated, scaled first curve. This leads to the Curve-Transform `gosperize`:

```
(define (gosperize curve)
  (let ((scaled-curve ((scale (/ (sqrt 2) 2)) curve)))
    (connect-rigidly
      ((rotate-around-origin (/ pi 4)) scaled-curve)
      ((translate .5 .5)
        ((rotate-around-origin (/ -pi 4)) scaled-curve))))))
```

Now we can generate approximations at any level to the Gosper curve by repeatedly gosperizing the unit line,

```
(define (gosper-curve level)
```

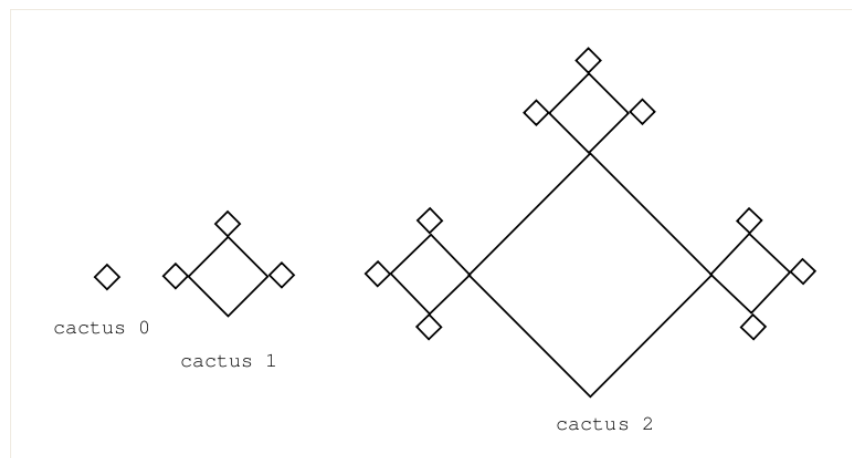
```
((repeated gosperize level) unit-line))
```

Plot Gosper curves for higher levels.

We can now invent other schemes like the Gosper process, and use them to generate fractal curves.

**Exercise 6** A sequence of 3 drawings, we shall call them cactuses, of order 0, 1 and 2, is given below. In these drawings the squares are of lengths 1, 3 and 9. Using the curve drawing primitives that we have supplied, write a function ( `cactus n`) that will draw a cactus of order `n`.

If you go above  $n = 3$ , then the curves may get distorted due to some problems with the plotting routine. Do not worry about this. Also keep the number of sample points to 70000 to get good plots. (12 Marks)



**Exercise 7** The *Koch curve* is produced by a process similar to the Gosper curve. We show once again the zeroth, first, second, third, fourth and eighth approximations to the Koch curve. Write a procedure (`koch-flakes level`) that generates Koch curves at different levels.

