Graphing with Higher-order Procedures

Date of Submission: 3rd February 2017, Friday 11:59:59pm

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

1 Higher Order Functions

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

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

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

Thus, for example `(compose square log)` is the function 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
```

The type of `thrice` is of the form T' to T' (where T' happens to equal (T 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[1] 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.

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 ansswers to these questions.

2 Curves as Functions 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.

[1] "Sameness" of function 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.

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 coordinates 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)
  (lambda (t)
  (make-point (sin (* 2 pi t))
              (cos (* 2 pi t)))))

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

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

Q1. Define a function `(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 functions which *operate* on Curve's. For example, the mapping $(x, y) \longrightarrow (-y, x)$ 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 function which takes a curve and transforms it into another, rotated, curve.

Q2. You can define several useful curve-transformations. Define

(a) `(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)`.

(b) `(translate x y curve)`, 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)`.

(c) `(scale x y curve)`, which stretches a curve along the x and y coordinates by given scale factors, and

(d) `(rotate-around-origin radians curve)` returns a Curve-Transform which rotates a curve by a given number of radians.

(e) 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 endpoints 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 function `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)))))
```

Q3. 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 `connect-ends`.

3 Drawing Curves

To draw curves in DrRacket, copy the lines given below at the begining 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.

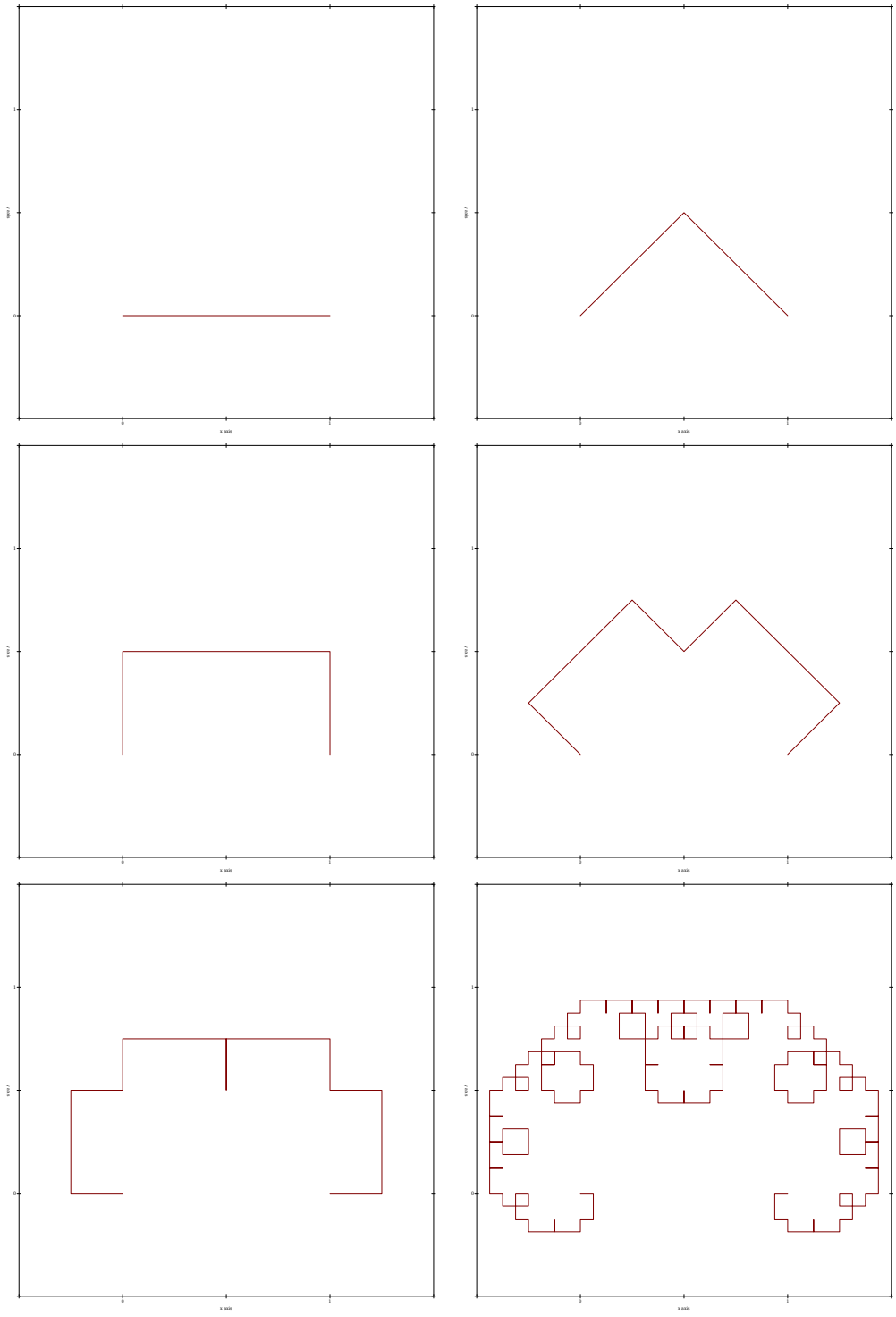
3.1 Fractal Curves

To demonstrate the power of our drawing language, we shall use it to explore the drawing of fractal cutves. Fractals have striking mathematical properties; If you expand a small part of a fractal curve, you get something similar to the original. Fractals have 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, the Gosper curve, is the infinite repetition of a very simple process. Each step of this process, which we shall call *gosper-step*, takes a curve that is the current approximation of the final curve, and joins two scaled copies of this 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. In general, 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 show the zeroth, first, second, third, fourth and eighth approximations to the Gosper curve.

4. Write a function `(gosper-step curve)`, which given a `curve`, would give the curve after a single gosper-step. You can assume that the input curve is in the standard position, and the curve returned by `gosper-step` should also be in the standard position.


y axis
x axis
0
1

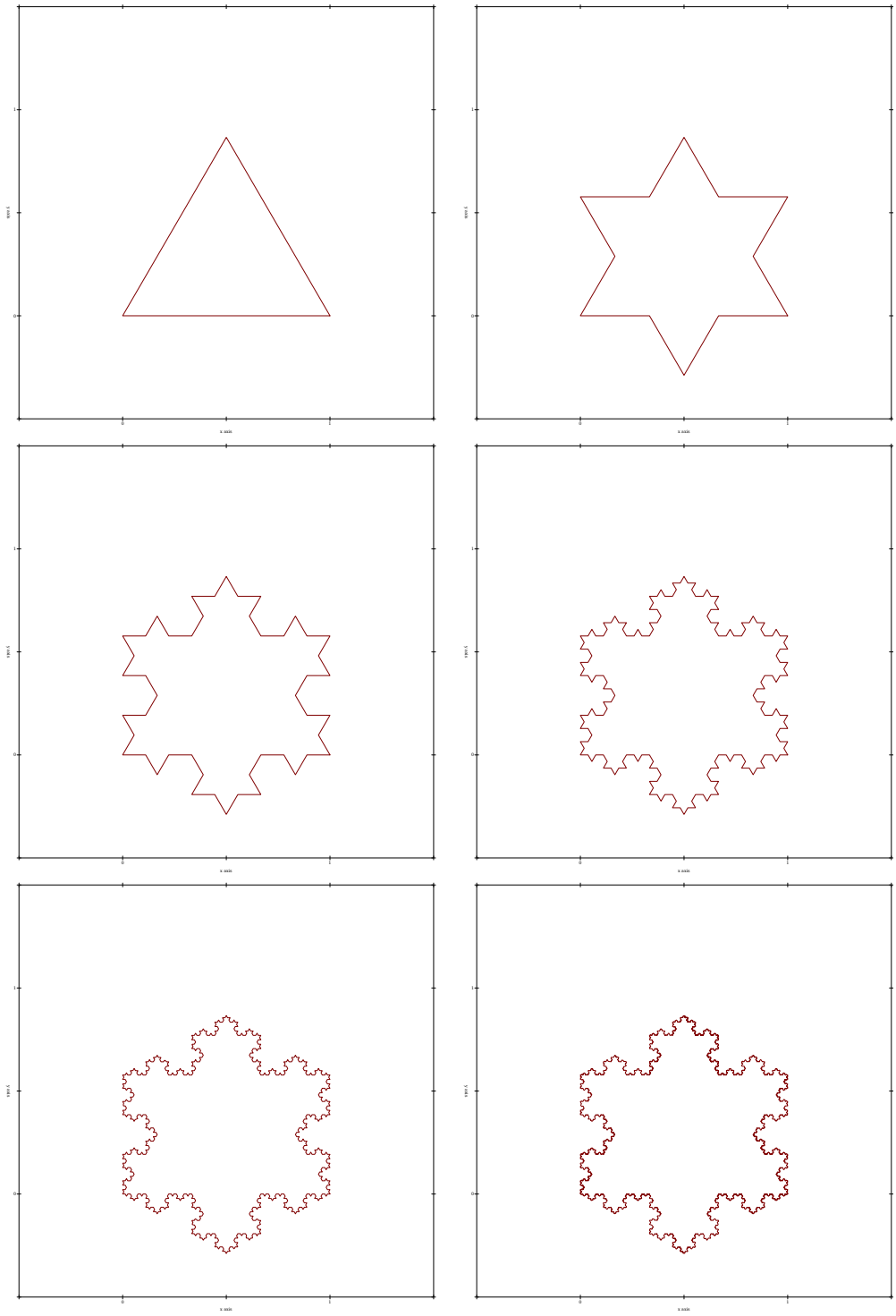
5. Now using `gosper-step`, write a function `(gosper-curve level)` which will draw the gosper curve for the given level. `(gosper-curve level)` should use the function `repeated` mentioned earlier.

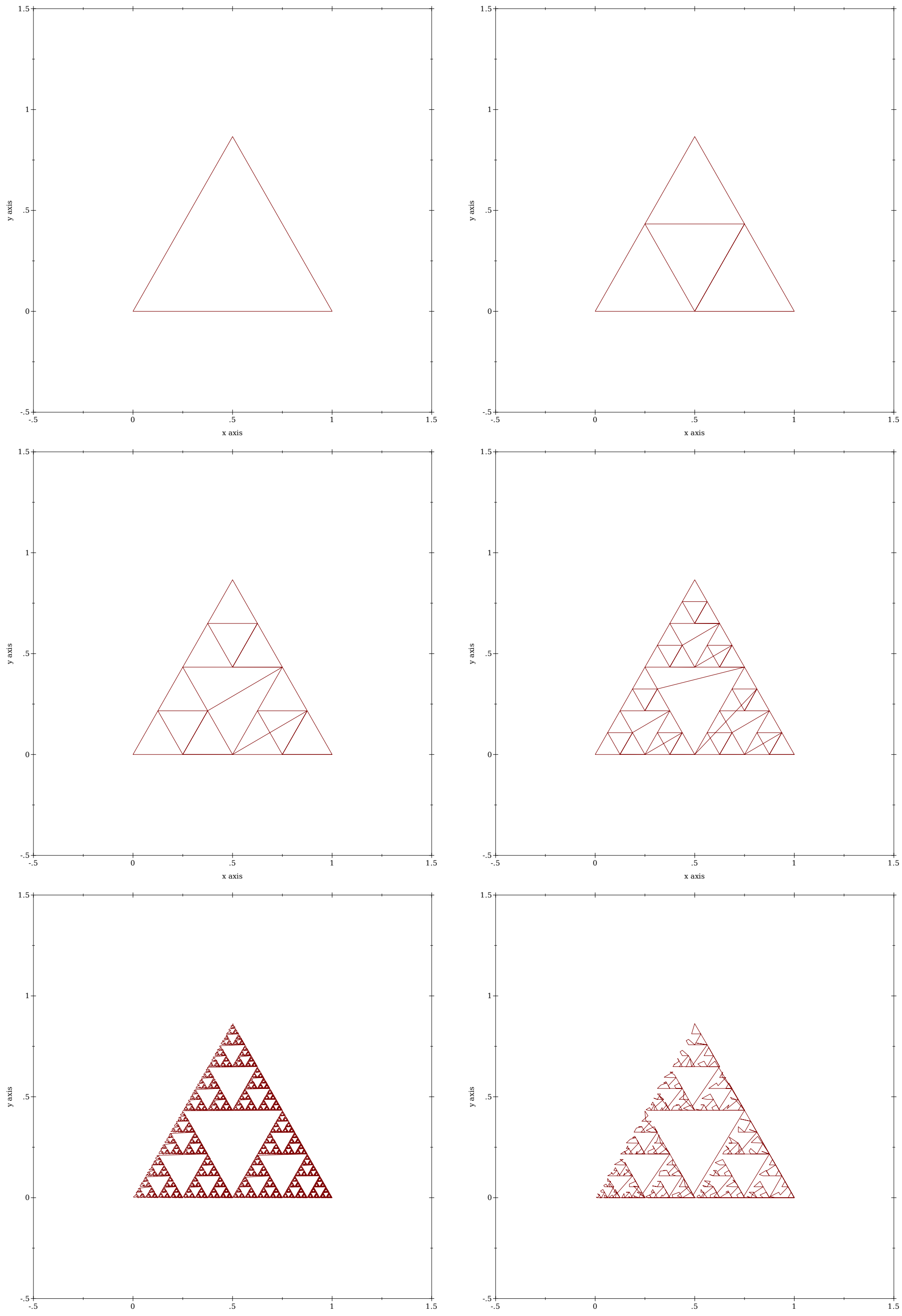
6. We can generalize the gosper curve in two ways. First, the starting curve need not always be a unit-line. If can be any curve in the standard position. Secondly, the angle of rotation also need not be fixed. It can change from one level to another. Write a function called `(gosper-general curve calc-angle l)`, which will generate a l-level gosper-curve with `curve` as the starting curve, and the angle at each level obtained by applying `calc-angle` to `level`. You should be able to produce the gosper curves shown in the example by calling `gosper-general` as `(gosper-general unit-line (lambda (theta) (/ 3.1415 4)) 8)`.

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

7. The *(koch-curve level)* 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 function `(koch-curve level)` that generates Koch curves at different levels.

8. Repeat the process for the Sierpinski triangles, whose zeroth, first, second, third, fourth and eighth approximations are shown in the figure below.




1.5
1
.5
0
-.5
y axis
-.5
0
.5
1
1.5
x axis