Version of October 4, 1997

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1997

Solutions to Problem Set 3

Graphing with Higher-order Procedures

1 1. Curves as Procedures and Data

Pre-Lab exercise 1:

1. What is the type of `unit-line-at`?

   From just examining the code for `unit-line` itself, a reasonable type would be

   $$\texttt{unit-line-at} : \text{Sch-Num} \rightarrow (\text{Sch-Num} \rightarrow \text{Point})$$

   But how are we going to use `unit-line-at`? The idea is to produce a Curve, so we should be able to assume that the values of `t` will be of type Unit-Interval, since that's how Curves were defined earlier. So another answer would be

   $$\text{unit-line-at} : \text{Sch-Num} \rightarrow (\text{Unit-Interval} \rightarrow \text{Point})$$

   But, really, wasn't the goal to write a function that takes a point on the `y` axis and return a particular curve (the unit line). So perhaps the most informative answer would be

   $$\text{unit-line-at} : \text{Sch-Num} \rightarrow \text{Curve}.$$

2. Define a procedure `diagonal-line` with two arguments, a point and a length, and returns a diagonal line of that length beginning at the point.

```
(define (diagonal-line point lngth)
  ;; As an oddity (or feature ... your choice) if length is
  ;; negative the line will run DOWN from the starting point,
  ;; rather than UP.
  (let ((x (x-of point))
        (y (y-of point)))
    (lambda (t)                          ; T: Unit-Interval
      (make-point
       (+ x (* t (/ lngth (sqrt 2))))
       (+ y (* t (/ lngth (sqrt 2))))))))
```

3. What is the type of `diagonal-line`?

   vertical-line : (Point, Sch-Num) $\rightarrow$ Curve (see point 1).

Pre-Lab exercise 2: Write a definition of a Curve-Transform `upside-down`, which turns a curve into its mirror image.

First the curve is translated to the x-axis. Reflecting about the x axis maps $(x, y) \longrightarrow (x, -y)$. Then the curve is translated back to its original position.

```
(define (upside-down curve)
  (let ((start-pt (curve 0)))
    (let ((moved-curve ((translate 0 (- (y-of start-pt)))
                        curve)))
      ((translate 0 (- (y-of start-pt)))
       (lambda (t)
        (let ((ct (moved-curve t)))
          (make-point
            (x-of ct)
            (- (y-of ct))))))))))
```

But translating to the x axis, then flipping, then translating back takes

$$(x, y) \longrightarrow (x, y - y_0) \longrightarrow (x, -(y - y_0)) \longrightarrow (x, -(y - y_0) + y_0) = (x, 2y_0 - y)$$

so another solution is

```
(define (upside-down curve)
  (let ((start-pt (curve 0)))
    (let ((y0 (y-of start-pt)))
      (lambda (t)
        (let ((ct (curve t)))
          (make-point
           (x-of ct)
           (- (* 2 y0) (y-of ct)))))))))
```

Pre-Lab exercise 3: There is another, possibly more natural, way of connecting curves. The curve returned by `(connect-ends curve1 curve2)` consists of a copy of `curve1` follwed 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`.

We're going to start with the `connect-rigidly` routine, but rewrite it a bit. The basic problem is that `connect-rigidly` uses the original `curve2`, but we need to translate `curve2` so that its origin is the same as `curve1`'s end point. In the code below, we start by calculating this "shared point" and the actual origin of `curve2`. We then calculate the translated version of `curve2` by subtracting its origin (this translates the curve back to the true origin) and adding in the shared point. The returned value (i.e., the value of the `lambda` expression) is pretty much the one in `connect-rigidly`, but we've replaced `curve2` with the translated version.

```
(define (connect-ends curve1 curve2)
  (let ((shared-point (curve1 1))      ; End of 1 == new start of 2
        (curve2-start (curve2 0)))     ; Original start of 2
    (let ((translated-curve2
           ((translate (- (x-of shared-point)
                          (x-of curve2-start))
                       (- (y-of shared-point)
                          (y-of curve2-start)))
            curve2)))
      (lambda (t)                      ; T: Unit-Interval
        (if (< t (/ 1 2))
            (curve1 (* 2 t))
            (translated-curve2 (- (* 2 t) 1)))))))
```

Here is another solution that uses `connect-rigidly`:

```
(define (connect-ends curve1 curve2)
  (connect-rigidly
    curve1
    (let ((end-point1 (curve1 1))
          (start-point2 (curve2 0)))
      (let ((x-diff (- (x-of end-point1) (x-of start-point2)))
            (y-diff (- (y-of end-point1) (y-of start-point2))))
        ((translate x-diff y-diff) curve2)))))
```

2 3. To do in Lab: Drawing Curves

Lab exercise 4: Apply `(draw-connected g1 200)` to `unit-circle`, and `(draw-connected g2 200)` to `alternative-unit-circle`. Can you see a difference? Now try using `draw-points-on` instead of `draw-connected`. Also try `draw-points-squeezed-to-window`. Print out the resulting figures.

The expressions I evaluated were:

a. `((draw-connected g1 200) unit-circle)`

b. `((draw-connected g2 200) alternative-unit-circle)`

The above curves are very, very similar, although on close inspection small differences are visible.

c. `((draw-points-on g1 200) unit-circle)`

d. `((draw-points-on g2 200) alternative-unit-circle)`

These two are clearly different. Unit-circle is written in a way which generates evenly spaced points, while alternative-unit-circle produces very closely spaced points at the start of the curve, and they become increasingly farther apart.

e. `((draw-points-squeezed-to-window g1 200) unit-circle)`

f. `((draw-points-squeezed-to-window g2 200) alternative-unit-circle)`

These two are even more dramatically different.

See attached pictures.

2.1 Fractal Curves

Pre-Lab exercise 5.A Define a procedure `show-points-gosper` such that evaluation of

```
(show-points-gosper window level number-of-points initial-curve)
```

will plot `number-of-points` unconnected points of the level `level` gosper curve in `window`, but starting the gosper-curve approximation with an arbitrary `initial-curve` rather than the unit line.

Here's how I thought about this. I decided to just copy the code for `show-connected-gosper`, but with the changes needed by the problem. So instead of calling `draw-connected`, I call `draw-points-on`. And instead of calling `(gosper-curve level)` I had to copy (and modify) the code for `gosper-curve` itself, so that we can specify the initial curve to use (instead of using `unit-line` like `gosper-curve` does):

```
(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))))
```

Lab exercise 5.B Try gosperizing the arc of the unit circle running from 0 to π. Find some examples that produce interesting designs.

The first trick is to generate the arc of the circle from 0 to π. Here's how I generated it. The idea is to start with a half unit circle centered on (0, 0). Then scale it by 1/2 and translate the origin to (1/2, 1/2).

```
(define arc
  (let ((half-unit-circle
         (lambda (t)
           (make-point (cos (* t pi))
                       (sin (* t pi))))))
    ((translate .5 .5)
     ((scale .5)
      half-unit-circle))))
```

Another way to approach this is to notice that the `half-unit-circle` runs from (-1, -1) to (1, 1) and we already have a procedure which will translate and scale an image to fit in the square from (0, 0) to (1, 1) – the procedure `squeeze-rectangular-portion`:

```
(define arc
  (let ((half-unit-circle
         (lambda (t)
           (make-point (cos (* t pi))
                       (sin (* t pi))))))
    ((squeeze-rectangular-portion -1 1 -1 1)
     half-unit-circle)))
```

But perhaps the most elegant solution comes from noticing that we can generate a half circle from the existing unit-circle and then relocate it and stretch it automatically by using `put-in-standardposition`:

```
(define arc
  (let ((half-circle
         (lambda (t)
           (unit-circle (/ t 2)))))
    (put-in-standard-position half-circle)))
```

Once `arc` is defined, the rest is very simple. Just try expressions like

```
(show-points-gosper g1 7 1000 arc)
```

Pre-Lab exercise 6.A Designing `param-gosperize` required using some elementary trigonometry to figure out how to shift the pieces around so that they fit together after scaling and rotating. It's easier to program if we let the computer figure out how to do the shifting. Show how to redefine `param-gosperize` using the procedures `put-in-standard-position` and `connect-ends` from Exercise 4 to handle the trigonometry.

The real trick to answering this question is to realize that `param-gosper` expects the result of `param-gosperize` to be a curve in standard position (i.e. `(curve 0)` should be (0, 0) and `(curve 1)` should be (1, 0)). That's precisely what `put-in-standard-position` arranges for an arbitrary curve. So all we need to do is arrange for two copies of the original curve rotated by `theta` and `-theta` to be connected end to end:

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

Lab exercise 6.B Generate some parameterized Gosper curves where the angle changes with the level n. We suggest starting with $\pi/(n+2)$ and $\pi/(1.3^n)$.

See attached printouts.

Lab exercise 6.C We now have three procedures to compute gosper curves: `gosper-curve`, and `param-gosper` with argument `(lambda (level) (/ pi 4)))` using the "hand-crafted" definition of `param-gosperize` above, or using your version of `param-gosperize` in 7.A based on `put-in-standard-position`. Compare the speed of these procedures for computing selected points on the curve at a few levels. Is there a speed advantage for the more customized procedures?

Here are the numbers I get:

Levels	Using GOSPER-CURVE	Using PARAM-GOSPER	Using 7.A
10	16	18	472
20	33	36	2554
30	50	53	4167
40	67	72	8437
50	83	88	12293
60	99	210	17742
70	117	121	

Obviously, the customized `param-gosperize` does *much* better.

Pre-Lab exercise 7.A Is Ben's definition correct? Briefly explain why using `bens-rotate` as a subprocedure in place of the original `rotate-around-origin` in the definition of `gosper-curve` will turn a process whose time is linear in the level into one which is exponential in the level.

Ben's version is correct, just slow. Until we introduce side-effects later in the class, we can rely on a simple substitution model. Using this model it is perfectly legal to replace `ct` with the expression that is used to define it, `(curve t)`. That's what Ben did.

Unfortunately, after Ben rewrote the program there are two calls to `curve` every time `bens-rotate` is called, while there was originally only one such call. Since `curve` is the procedure which generates the next simpler version of the curve (with one smaller level number), the extra call is catastrophic. Previously, each attempt to draw a curve at one level results in one call to `curve` at the smaller level: a linear process. After Ben's change, however, a single call at one level results in *two* calls at the next level, and this in turn requires *two* calls at the level below that and so on: a tree of calls, or an exponential process.

Lab exercise 7.B Turn in a table summarizing the number of calls to `x-of` by `gosper-curve` using the two different rotating procedures at four or five illustrative levels.

```
(trace x-of)
;Value: #[undefined-value]

((gosper-curve 4) .1)
[Entering #[compound-procedure 117 x-of]
    Args: #[compound-procedure 118]]
[.6000000000000001
. . .

(define saved-rotate rotate-around-origin)
;Value: "saved-rotate --> #[compound-procedure 143 rotate-around-origin]"

(define rotate-around-origin bens-rotate)
;Value: "rotate-around-origin --> #[compound-procedure 144 bens-rotate]"

((gosper-curve 1) .1)
[Entering #[compound-procedure 117 x-of]
    Args: #[compound-procedure 145]]
. . .

(define rotate-around-origin saved-rotate)
;Value: "rotate-around-origin --> #[compound-procedure 143 rotate-around-origin]"
```

Table of number of calls versus level:

Levels	Regular rotate	Ben's rotate
1	2	3
2	4	9
3	6	21
4	9	53

It does indeed look like Ben's rotate is exponential (more than doubling for each new level).

Pre-Lab exercise 8.A The *Koch curve* is produced by a process similar to the Gosper curve, as shown in figure 3. Write a procedure `kochize` that generates Koch curves. (Teaser: You can generate the Koch curve by using `param-gosper` with an appropriate argument. Can you find this?)

Here's a first try that just uses the direct information from the picture. It works in the usual recursive manner, with the "drawing procedure" taking an initial curve. It draws this curve at 1/3 size both at the origin and translated over by 2/3, and adds in a 1/3 size version of the curve angled up by 60 degrees connected to a 1/3 size version of the curve angled down by 60 degrees.

```
(define kochize
  (let ((scale/3 (scale (/ 1 3)))
        (up-60 (rotate-around-origin (/ pi 3)))
        (down-60 (rotate-around-origin (- (/ pi 3))))
        (over-1/3 (translate (/ 1 3) 0))
        (over-2/3 (translate (/ 2 3) 0)))
    (lambda (level)
      (define (koch curve)
        (let ((curve/3 (scale/3 curve))
              (up (scale/3 (up-60 curve)))
              (down (scale/3 (down-60 curve))))
          (connect-rigidly curve/3
           (connect-rigidly
            (over-1/3 (connect-ends up down))
            (over-2/3 curve/3)))))
      (if (= level 0)
          unit-line
          (koch (kochize (- level 1)))))))
```

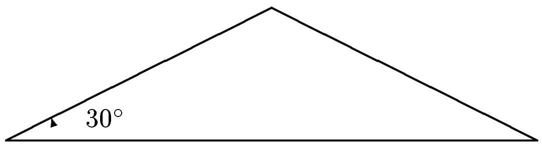
Here's the same idea, but using `connect-ends` throughout to avoid having to do much trigonometry:

```
(define kochize
  (let ((scale/3 (scale (/ 1 3)))
        (up-60 (rotate-around-origin (/ pi 3)))
        (down-60 (rotate-around-origin (- (/ pi 3)))))
    (lambda (level)
      (define (koch curve)
        (let ((curve/3 (scale/3 curve))
              (up (scale/3 (up-60 curve)))
              (down (scale/3 (down-60 curve))))
          (connect-ends curve/3
           (connect-ends up
            (connect-ends down curve/3)))))
      (if (= level 0)
          unit-line
          (koch (kochize (- level 1)))))))
```

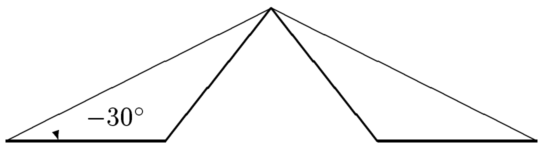
Here's one using even less trigonometry by using `put-in-standard-position` instead:

```
(define kochize
  (let ((up-60 (rotate-around-origin (/ pi 3)))
        (down-60 (rotate-around-origin (- (/ pi 3)))))
    (lambda (level)
      (define (koch curve)
        (put-in-standard-position
          (connect-ends curve
           (connect-ends (up-60 curve)
            (connect-ends (down-60 curve) curve)))))
      (if (= level 0)
          unit-line
          (koch (kochize (- level 1)))))))
```

Now let's go for the full prize. How can we use `param-gosper` to make a Koch curve? The thing to notice is that a level 1 Koch curve can be generated by a level 2 Param-Gosper curve. At the first `param-gosper` level, we bend up by 30 degrees:



At the second level, we replace each of these lines from level 1 with a pair that bends -30 degrees:



So, for a given level of Koch curve, we just double the number of levels and have the sign of the angle switch:

```
(define (kochize level)
  (let ((pi/6 (/ pi 6))
        (-pi/6 (- (/ pi 6))))
    (param-gosper (* level 2)
     (lambda (level)
       (if (even? level) pi/6 -pi/6)))))
```

All of these procedures produce the correct results, but they do so in slightly different ways with slightly different performance and quality. This raises an interesting question: which of these is "best"? Is it better to be fast at a given number of points, but produce a slightly lower quality picture (the quality gets better as the number of points gets larger)? Is it better to give a good picture with a few points, but take a long time to do it? There is no obvious answer.

Lab exercise 8.B Print some pictures of your Koch curve at various levels.

See attached figures.

Lab Exercise 9

Just for fun, I tried two ideas for neat graphics. The first one is called the "Sierpinski Triangle": a nested set of triangles, which leaves "holes" behind (kind of like Swiss cheese).

```scheme
(define (connect-rigidly-3 curve1 curve2 curve3)
  (lambda (t)
    (let ((s (* t 3)))
      (cond ((< s 1) (curve1 s))
            ((< s 2) (curve2 (- s 1)))
            (else    (curve3 (- s 2)))))))

(define (unit-triangle t)
  ; draw a unit side triangle in a clockwise direction
  ;   this version explicitly computes the coordinates
  (let ((s (* t 3))
        (tan60 (/ (sqrt 3) 2)))
    (cond ((< s 1) (make-point (* 0.5 s) (* tan60 s)))
          ((< s 2) (make-point (* 0.5 s) (* tan60 (- 2 s))))
          (else    (make-point (- 3 s) 0)))))

(define unit-triangle
  ; draw a unit side triangle in a clockwise direction
  ;   this version uses curve manipulation and connect-rigidly-3
  (let ((left-side ((rotate-around-origin (/ pi 3)) unit-line))
        (right-side ((translate 0.5 (* 0.5 (sqrt 3)))
                     ((rotate-around-origin (/ pi -3)) unit-line)))
        (bottom (lambda (t) (unit-line (- 1 t)))))
    (connect-rigidly-3 left-side right-side bottom)))

(define (sierpinskize curve)
  ;  A Sierpinski triangle is a nested set of triangle with holes in it
  ;     Use 3 smaller versions of the triangle, translated
  (let ((t1 ((scale 0.5) curve))
        (trans-2 (translate 0.25 (* 0.25 (sqrt 3))))
        (trans-3 (translate 0.5 0)))
    (connect-rigidly-3 t1 (trans-2 t1) (trans-3 t1))))

(define (sierpinski level)
  ((repeated sierpinskize level) unit-triangle))

;;; Example of use:
((draw-points-on g1 (* 2 (expt 3 7))) (sierpinski 4))
```

The second idea is to implement in software an old children's game I had called the *SpiroGraph*, which was a toothed plastic wheel inside a toothed ring, with holes for pens to draw "twirly" designs.

```
(define (spirograph r1 r2 r3)
  ;; r1 is outer ring radius, r2 is inner disk radius, r3 is pen radius
  ;; r1 > r2 > r3
  (lambda (t)
    (make-point (+ (* (- r2 r1) (sin t))
                   (* r3 (sin (* (/ r1 r2) t))))
                (+ (* (- r1 r2) (cos t))
                   (* r3 (cos (* (/ r1 r2) t)))))))

(define (spirodraw r1 r2 r3 window n-points)
  (let ((n-theta (* 2 pi (/ r1 (gcd r1 r2)))))
    ((draw-connected-squeezed-to-window window n-points)
     (lambda (t) ((spirograph r1 r2 r3) (* t n-theta))))))

;;; Example of use:
(spirodraw 128 33 32 g1 2048)
```