

Drawing in Script-Fu, Revisited

Summary: We consider some new techniques for making and modifying simple drawings.

Code: You can find most of the procedures discussed herein in `drawing.scm`.

Contents:

- Introduction: Simple Drawings
- Simple Drawings
- Translating Drawings
- Scaling Drawings
- Varying Drawings
- Algorithmic Drawings
- Drawings: Sequences of Sequences
- Detour: Design Patterns
 - Checking List Types
 - Applying Procedures to List Values

Introduction: Simple Drawings

As you may have noted, it is a bit difficult to make drawings in DrScheme that we can replicate, scale, and place at different positions in the image. We have experimented with *parameterized drawings*, in which we can set certain parameters and get different versions of the drawings. However, such drawings required significant effort on our part to compute the various points.

Since Scheme is so good at calculation, we should be able to use Scheme to help make different versions of images. How? We'll work in two steps: First, we'll think of a way to represent simple drawings and then we'll look at ways to manipulate those simple drawings.

Simple Drawings

If you think about how we normally draw on paper, we typically put the pen down, draw from place on the paper to place on the paper to place on the appear, pick the pen up, put it down somewhere else, draw from place to place to place again, and so on and so forth.

Let's focus on one line in that drawing. We might represent that line as a sequence of (x,y) points. How do we represent a point? One easy way is as a pair. For example, we might represent (10,15) by `(cons 10 15)`. We might then write some helper procedures to go with that description.

```
;;; Procedure:  
;;; point  
;;; Parameters:  
;;; x, a real number
```

```

;;; y, a real number
;;; Purpose:
;;; Build the point (x,y)
;;; Produces:
;;; pt, a point
;;; Postconditions:
;;; (xcoord pt) = x
;;; (ycoord pt) = y
(define point cons)

;;; Procedure:
;;; xcoord
;;; Parameters:
;;; pt, a point
;;; Purpose:
;;; Extracts the x coordinate of a point.
;;; Produces:
;;; x, a real number
;;; Preconditions:
;;; pt was created with point.
;;; Postconditions:
;;; (xcoord (point x y)) = x
(define xcoord car)

;;; Procedure:
;;; ycoord
;;; Parameters:
;;; pt, a point
;;; Purpose:
;;; Extracts the y coordinate of a point.
;;; Produces:
;;; y, a real number
;;; Preconditions:
;;; pt was created with point.
;;; Postconditions:
;;; (ycoord (point x y)) = y
(define ycoord cdr)

;;; Procedure:
;;; point?
;;; Parameters:
;;; val, a Scheme value
;;; Purpose:
;;; Determine if val is a point, using our current representation.
;;; Produces:
;;; is-point, a Boolean
;;; Postconditions:
;;; If val seems to have been produced with point, then is-point is
;;; true.
;;; Otherwise, is-point is false.
(define point?
  (lambda (val)
    (and (pair? val)
         (real? (car val))
         (real? (cdr val)))))

```

Why do we bother to define `point`, `xcoord`, and `ycoord`? So that if we decide to change the representation (e.g., to use a two-element list rather than a pair), the only code we need to change is these three procedures. (Of course, that assumes that we only use these procedures elsewhere, and don't rely on the underlying representation.)

Now that we can represent individual points, we need to figure out how to represent sequences of points. The natural representation of a sequence is a list.

For example, the following defines the sequence of points for a simple, somewhat inelegant, tree.

```
(define tree (list (point 15 30) (point 18 14) (point 5 18) (point 8 8) (point 20 0) (point 30 2) (point 35 19) (point 23 16) (point 23 30)))
```

How did I decide on those points? I sketched the tree on grid paper and then found the places on the grid. You could also sketch something in the GIMP, zoom in, pick key points, and list those points. Some very talented people can just visualize points.

We won't write something that builds such lists, but we will write a predicate for such lists.

```
;;; Procedure:
;;; list-of-points?
;;; Parameters:
;;; val, a Scheme value
;;; Purpose:
;;; Determine if val is a list of points.
;;; Produces:
;;; ok, a Boolean
(define list-of-points?
  (lambda (val)
    (or (null? val)
        (and (pair? val)
              (point? (car val))
              (list-of-points? (cdr val))))))
```

Now, what can we do with a list of points? We could write a procedure that takes each neighboring pair and draws a line between them. However, that isn't necessary, as `gimp.scm` contains a procedure, (`connect-the-dots image points`) that does something similar. That is, it draws a line between the points, using the current brush and color.

Translating Drawings

Now that we know how to make one drawing, what can we do with it? One possibility is to make another copy of the drawing somewhere else. It turns out that it's fairly straightforward to move a drawing: We simply move each point by the same delta-x and delta-y. Let's start with a procedure that translates points.

```
;;; Procedure:
;;; translate-point
;;; Parameters:
;;; delta-x, an integer
;;; delta-y, an integer
;;; pt, a point
;;; Purpose:
;;; Translates the point by the specified amount.
```

```

;;; Produces:
;;;   translated, a point
;;; Postconditions:
;;;   (xcoord translated) = (+ (xcoord pt) delta-x)
;;;   (ycoord translated) = (+ (ycoord pt) delta-y)
(define translate-point
  (lambda (delta-x delta-y pt)
    (point (+ (xcoord pt) delta-x) (+ (ycoord pt) delta-y))))

```

Now, it's fairly simple to translate a sequence of points: We simply recurse through the list, applying `translate-point` at each position.

```

;;; Procedure:
;;;   translate-points
;;; Parameters:
;;;   delta-x, a real number
;;;   delta-y, a real number
;;;   points, a list of points
;;; Purpose:
;;;   Translate all of the points by the specified amount.
;;; Produces:
;;;   translated, a list of points
;;; Postconditions:
;;;   For each i, 0 <= i < (length points)
;;;     (list-ref translated i) =
;;;       (translate-point delta-x delta-y (list-ref points i))
(define translate-points
  (lambda (delta-x delta-y points)
    (if (null? points)
        null
        (cons (translate-point delta-x delta-y (car points))
              (translate-points delta-x delta-y (cdr points))))))

```

Once we've defined this procedure, we can draw a sequence of trees.

```

(define draw-forest
  (lambda (image)
    (set-brush "Circle (05)")
    (set-fgcolor DARK_GREEN)
    (connect-the-dots image tree)
    (connect-the-dots image (translate-points 20 5 tree))
    (set-fgcolor GREEN)
    (connect-the-dots image (translate-points 10 3 tree))
    (connect-the-dots image (translate-points 30 4 tree))))

```

Scaling Drawings

Scaling a drawing is not much different than translating. We first define a procedure, `scale-point`, that “scales” a point by a certain amount. We then apply that procedure to each point in the drawing. (Yes, it makes little sense to scale a point. However, if we change the x and y coordinate of each point in a drawing by multiplying those coordinates by a scale, we effectively scale the whole drawing. We also end up scaling the distance between the drawing and the origin.)

```

;;; Procedure:
;;;   scale-point
;;; Parameters:
;;;   scale-x, an integer
;;;   scale-y, an integer
;;;   pt, a point
;;; Purpose:
;;;   "Scales" the point by the specified amount. Think of a point as
;;;   representing a vector from the origin to the point; that vector
;;;   has been scaled.
;;; Produces:
;;;   scaled, a point
;;; Postconditions:
;;;   (xcoord scaled) = (* (xcoord pt) scale-x)
;;;   (ycoord scaled) = (* (ycoord pt) scale-y)
(define scale-point
  (lambda (scale-x scale-y pt)
    (point (* (xcoord pt) scale-x) (* (ycoord pt) scale-y))))

;;; Procedure:
;;;   scale-points
;;; Parameters:
;;;   scale-x, a real number
;;;   scale-y, a real number
;;;   points, a list of points
;;; Purpose:
;;;   Scale all of the points by the specified amount.
;;; Produces:
;;;   scaled , a list of points
;;; Postconditions:
;;;   For each i, 0 <= i < (length points)
;;;   (list-ref scaled i) = (scale-point delta-x delta-y (list-ref points i))
(define scale-points
  (lambda (delta-x delta-y points)
    (if (null? points)
        null
        (cons (scale-point delta-x delta-y (car points))
              (scale-points delta-x delta-y (cdr points))))))

```

Once we've defined this procedure, we might define a variant tree with

```
(define tall-thin-tree (scale-points 2 5 tree))
```

Varying Drawings

Of course, the images made by scale and translate are nearly identical to the original images, differing only in size or position. Can we make more interesting changes, so that the copies are not quite the same? Certainly, we can change the x and y coordinate of each point by a "random" amount, giving a similar but different drawing. Here's a procedure that does just that for one point.

```

;;; Procedure:
;;;   vary-point
;;; Parameters:
;;;   amt, the amount of variance
;;;   pt, a point

```

```

;;; Purpose:
;;; Vary the x and y coordinate of pt by a random amount.
;;; Produces:
;;; varied, a point
;;; Postconditions:
;;; (<= (- (xcoord pt) amt) (xcoord varied) (+ (xcoord pt) amt))
;;; (<= (- (ycoord pt) amt) (ycoord varied) (+ (ycoord pt) amt))
;;; Each point that meets those requirements is equally likely.
(define vary-point
  (lambda (amt pt)
    (point (+ (- (xcoord pt) amt) (random (+ 1 (* 2 amt))))
           (+ (- (ycoord pt) amt) (random (+ 1 (* 2 amt)))))))

;;; Procedure:
;;; vary-points
;;; Parameters:
;;; amt, the amount of variance
;;; points, a list of points
;;; Purpose:
;;; Vary each point
;;; Produces:
;;; varied, a sequence of points.
(define vary-points
  (lambda (amt points)
    (if (null? points)
        null
        (cons (vary-point amt (car points))
              (vary-points amt (cdr points))))))

```

We can now define a few variants of our simple tree.

```

(define trees (list (vary-points 2 tree)
                   (translate-points 20 10 (vary-points 2 tree))
                   (translate-points 40 0 (vary-points 2 tree))))

```

Algorithmic Drawings

We can also generate some of our drawings (or lists of points) algorithmically. For example, here's a simple procedure that draws n points on a simple spiral.

```

;;; Procedure:
;;; spiral
;;; Parameters:
;;; n, the number of points to draw in the spiral (90 points are needed
;;; for one loop).
;;; Purpose:
;;; Generate a list of points for a spiral
;;; Produces:
;;; spiral-points, a list of points
(define spiral
  (let ((scale (/ 3.14 45)))
    (lambda (n)
      (if (<= n 0)
          null
          (cons (point (+ (- (xcoord pt) amt) (random (+ 1 (* 2 amt))))
                     (+ (- (ycoord pt) amt) (random (+ 1 (* 2 amt))))))
                (spiral (+ n 1)))))))

```

```

null
(cons (point (* n (sin (* scale n)))
           (* n (cos (* scale n))))
      (spiral (- n 1))))))

```

We can use a similar process to draw an expanding zig-zag.

```

;;; Procedure:
;;; zig-zag
;;; Parameters:
;;; n, the number of lines in the zig zag (at least 2)
;;; Purpose:
;;; Generate a series of lines for an "interesting" zig-zag figure of
;;; width approximately (20 + 2n) and height approximately 4n.
;;; Produces:
;;; zig-zag-points, a list of points
(define zig-zag
  (lambda (n)
    (if (<= n 0)
        null
        (cons (point (if (even? n) (- 20 n) (+ 20 n))
                (* 4 n))
              (zig-zag (- n 1))))))

```

We don't worry too much about the size or placement of our zig-zags and spirals, since we can always translate them or scale them.

Drawings: Sequences of Sequences

In the sections above, we've focused on one sequence of points in a drawing. However, as we noted in the beginning, a drawing is really a sequence of sequences. That is, you draw the first sequence, move the pen somewhere else, draw another sequence, move the pen somewhere else, draw another sequence, and so on and so forth. Hence, we should really represent each drawing as a list of lists of points. We'll begin with the predicate.

```

;;; Procedure:
;;; drawing?
;;; Parameters:
;;; val, a Scheme value
;;; Purpose:
;;; Determine if val is a drawing.
;;; Produces:
;;; is-drawing, a Boolean
(define drawing?
  (lambda (val)
    (or (null? val)
        (and (pair? val)
              (list-of-points? (car val))
              (drawing? (cdr val))))))

```

We can draw a drawing by connecting-the-dots for each line.

```

;;; Procedure:
;;; draw
;;; Parameters:
;;; image, an image
;;; drawing, a drawing
;;; Purpose:
;;; Draws the drawing.
;;; Produces:
;;; (nothing)
;;; Preconditions:
;;; (none)
;;; Postconditions:
;;; The drawing has been done in the current brush and color
(define draw
  (lambda (image drawing)
    (if (not (null? drawing))
        (begin
          (connect-the-dots image (car drawing))
          (draw image (cdr drawing)))))))

```

Now, we can do the same things with a drawing that we do with a sequence of points (or with a point): translate it or scale it (or both).

```

;;; Procedure:
;;; translate
;;; Parameters:
;;; delta-x, a real
;;; delta-y, a real
;;; drawing, a drawing (a list of lists of points)
;;; Purpose:
;;; Translate the drawing by the specified amount.
;;; Produces:
;;; translated, a translated version of the drawing
(define translate
  (lambda (delta-x delta-y drawing)
    (if (null? drawing) null
        (cons (translate-points delta-x delta-y (car drawing))
              (translate delta-x delta-y (cdr drawing))))))

```

```

;;; Procedure:
;;; scale
;;; Parameters:
;;; scale-x, a real
;;; scale-y, a real
;;; drawing, a drawing (a list of lists of points)
;;; Purpose:
;;; Scale the drawing by the specified amount.
;;; Produces:
;;; scaled, a scaled version of the drawing
(define scale
  (lambda (scale-x scale-y drawing)
    (if (null? drawing) null
        (cons (scale-points scale-x scale-y (car drawing))
              (scale scale-x scale-y (cdr drawing))))))

```

```

;;; Procedure:
;;; vary

```

```

;;; Parameters:
;;;   amt, an integer
;;;   drawing, a drawing (a list of lists of points)
;;; Purpose:
;;;   Vary the drawing by the specified amount.
;;; Produces:
;;;   varied, a randomly modified version of the drawing.
(define vary
  (lambda (amt drawing)
    (if (null? drawing) null
        (cons (vary-points amt (car drawing))
              (vary amt (cdr drawing))))))

```

Detour: Design Patterns

One mark of successful programmers is that they identify and remember common techniques for solving problems. Such abstractions of common structures for solving problems are often called *patterns* or *design patterns*. You should already have begun to identify some patterns. For example, you know that procedures almost always have the form

```

(define procname
  (lambda (parameters)
    body))

```

You may also have a pattern in mind for the typical recursive procedure over lists:

```

(define procname
  (lambda (lst)
    (if (null? lst)
        base-case
        (do-something (car lst) (procname (cdr lst))))))

```

In some languages, these patterns are simply guides to programmers as they design new solutions. In other languages, such as Scheme, you can often *encode* a design pattern in a separate procedure.

Checking List Types

Here's one common pattern. Often, we want to check a list to make sure that all of the elements of the list are of a certain type (or otherwise meet a particular predicate). For example, both `list-of-points?` and `drawing?` above have the same basic structure:

```

(define list-pred?
  (lambda (val)
    (or (null? val)
        (and (pair? val)
             (PRED? (car val))
             (list-pred? (cdr val))))))

```

Rather than typing the same code each time, we can write a more generic `list-of?` procedure that takes both the predicate and the list as parameters.

```

;;; Procedure:
;;; list-of?
;;; Parameters:
;;; type?, a predicate
;;; val, a Scheme value
;;; Purpose:
;;; Verify that val is a list of values and that type? holds for each
;;; value in the list.
;;; Produces:
;;; is-list-of, a Boolean
;;; Postconditions:
;;; If val is not a list, is-list-of is #f.
;;; If (type? (list-ref val i)) is #f for some valid i, is-list-of is #f.
;;; Otherwise, is-list-of is true.
(define list-of?
  (lambda (type? val)
    (or (null? val)
        (and (pair? val)
              (type? (car val))
              (list-of? type? (cdr val))))))

```

Now, we can define `list-of-points?` and `drawing?` in terms of this procedure.

```

(define list-of-points?
  (lambda (val)
    (list-of? point? val)))
(define drawing?
  (lambda (val)
    (list-of? list-of-points? val)))

```

In fact, if we remember our friendly `left-section` procedure (also called `l-s`, we can even do without the `lambda`.

```

(define list-of-points? (l-s list-of? point?))
(define drawing? (l-s list-of? list-of-points?))

```

Applying Procedures to List Values

Here's another common pattern. Let's consider the procedures we've defined above. Do some of them have a common form? Yes. Most of the procedures (except the point procedures) recurse over a list, building a new list by applying a procedure to each value in the original list. If we ignore the other parameters, we can write this as

```

(define procname
  (lambda (lst)
    (if (null? lst)
        null
        (cons (modify (car lst))
              (procname (cdr lst))))))

```

Now, if we make `modify` a parameter to this pattern, we can define the pattern as a procedure. This pattern is typically called `map`

```

;;; Procedure:
;;; map
;;; Parameters:
;;; proc, a procedure that takes one parameter and returns one value
;;; lst, a list of values
;;; Purpose:
;;; Builds a list by applying proc to each value in lst.
;;; Produces:
;;; newlst, a list of values.
;;; Preconditions:
;;; proc can be applied to each value in lst.
;;; Postconditions:
;;; (length newlst) = (length lst)
;;; For each i, 0 <= i < (length lst)
;;; (list-ref newlst i) = (proc (list-ref lst i))
(define map
  (lambda (proc lst)
    (if (null? lst) null
        (cons (proc (car lst))
              (map proc (cdr lst))))))

```

Once we've defined map, we can define each of the preceding recursive procedures using map

```

(define translate-points
  (lambda (delta-x delta-y points)
    (map (lambda (pt) (translate-point delta-x delta-y pt)) points)))
(define scale-points
  (lambda (scale-x scale-y points)
    (map (lambda (pt) (scale-point scale-x scale-y pt)) points)))
(define vary-points
  (lambda (amt points)
    (map (lambda (pt) (vary-point amt pt)) points)))
(define translate
  (lambda (delta-x delta-y drawing)
    (map (lambda (points) (translate-points delta-x delta-y points) drawing))))
(define scale
  (lambda (scale-x scale-y drawing)
    (map (lambda (points) (scale-points scale-x scale-y points) drawing))))
(define vary
  (lambda (amt drawing)
    (map (lambda (points) (vary-points amt points) drawing))))

```

Since the last anonymous function simply fills in the first parameter to vary-points, we can rewrite it with l-s.

```

(define vary
  (lambda (amt drawing)
    (map (l-s vary-points amt) drawing)))

```

Copyright © 2007 Samuel A. Rebersky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.