Structure and Interpretation of Computer Programs
Second Edition
Fall Semester, 1993

Problem Set 4

Issued: Tuesday, September 28

Tutorial preparation for: Week of October 4

Written solutions due: Friday, October 8 in Recitation

Reading:

- Course notes: finish section 2.2
- code files `hend.scm` and `hutils.scm` (attached to this handout)

A Graphics Design Language

In this assignment, you will work with the Peter Henderson's "square-limit" graphics design language, which Hal described in lecture on September 28. Before beginning work on this programming assignment, you should review the notes for that lecture. The goal of this problem set is to reinforce ideas about data abstraction and higher-order procedures, and to emphasize the expressive power that derives from appropriate primitives, means of combination, and means of abstraction.[1]

Section 1 of this handout reviews the language, as presented in lecture. You will need to study this in order to prepare the tutorial presentations in section 2. Section 3 gives the lab assignment, which includes an optional design contest.

[1]This problem set was developed by Hal Abelson, based upon work by Peter Henderson ("Functional Geometry," in *Proc. ACM Conference on Lisp and Functional Programming*, 1982). The image display code was designed and implemented by Daniel Coore.

Figure 1: A picture of M.C. Escher, and the same picture transformed by the square-limit process.

1. The Square-limit language

Remember from lecture that the key idea in the square-limit language is to use *painter* procedures that take frames as inputs and paint images that are scaled to fit the frames.

Basic data structures

Vectors are represented as pairs of numbers.

```
(define make-vect cons)
(define vector-xcor car)
(define vector-ycor cdr)
```

Here are the operations of vector addition, subtraction, and scaling a vector by a number:

```
(define (vector-add v1 v2)
  (make-vect (+ (xcor v1) (xcor v2))
             (+ (ycor v1) (ycor v2))))

(define (vector-sub v1 v2)
  (vector-add v1 (vector-scale -1 v2)))

(define (vector-scale x v)
  (make-vect (* x (xcor v))
             (* x (ycor v))))
```

A pair of vectors determines a directed line segment—the segment running from the endpoint of the first vector to the endpoint of the second vector:

```
(define make-segment cons)
(define segment-start car)
(define segment-end cdr)
```

Frames

A frame is represented by three vectors: an origin and two edge vectors.

```
(define (make-frame origin edge1 edge2)
  (list 'frame origin edge1 edge2))

(define frame-origin cadr)
(define frame-edge1 caddr)
(define frame-edge2 cadddr)
```

The frame's origin is given as a vector with respect to the origin of the graphics-window coordinate system. The edge vectors specify the offsets of the corners of the frame from the origin of the frame. If the edges are perpendicular, the frame will be a rectangle; otherwise it will be a more general parallelogram. Figure 2 shows a frame and its associated vectors.

Each frame determines a system of "frame coordinates" (x, y) where $(0, 0)$ is the origin of the frame, x represents the displacement along the first edge (as a fraction of the length of the edge) and y is the displacement along the second edge. For example, the origin of the frame has frame coordinates $(0, 0)$ and the vertex diagonally opposite the origin has frame coordinates $(1, 1)$.

Figure 2: A frame is represented by an origin and two edge vectors.

Another way to express this idea is to say that each frame has an associated *frame coordinate map* that transforms the frame coordinates of a point into the Cartesian plane coordinates of the point. That is, (x, y) gets mapped onto the Cartesian coordinates of the point given by the vector sum

$$\text{Origin}(\text{Frame}) + x \cdot \text{Edge}_1(\text{Frame}) + y \cdot \text{Edge}_2(\text{Frame})$$

We can represent the frame coordinate map by the following procedure:

```
(define (frame-coord-map frame)
  (lambda (point-in-frame-coords)
    (vector-add
     (frame-origin frame)
     (vector-add (vector-scale (vector-xcor point-in-frame-coords)
                               (frame-edge1 frame))
                 (vector-scale (vector-ycor point-in-frame-coords)
                               (frame-edge2 frame))))))
```

For example, `((frame-coord-map a-frame) (make-vect 0 0))` will return the same value as `(frame-origin a-frame)`.

The procedure `make-relative-frame` provides a convenient way to transform frames. Given a frame and three points `origin`, `corner1`, and `corner2` (expressed in frame coordinates), it returns a new frame with those corners:

```
(define (make-relative-frame origin corner1 corner2)
  (lambda (frame)
    (let ((m (frame-coord-map frame)))
      (let ((new-origin (m origin)))
        (make-frame new-origin
                    (vector-sub (m corner1) new-origin)
                    (vector-sub (m corner2) new-origin))))))
```

For example,

```
(make-frame-relative (make-vect .5 .5) (make-vect 1 .5) (make-vect .5 1))
```

returns the procedure that transforms a frame into the upper-right quarter of the frame.

Painters

As described in lecture, a painter is a procedure that, given a frame as argument, "paints" a picture in the frame. That is to say, if `p` is a painter and `f` is a frame, then evaluating `(p f)` will cause an image to appear in the frame. The image will be scaled and stretched to fit the frame.

The language you will be working with includes four ways to create primitive painters.

The simplest painters are created with `number->painter`, which takes a number as argument. These painters fill a frame with a solid shade of gray. The number specifies a gray level: 0 is black, 255 is white, and numbers in between are increasingly lighter shades of gray. Here are some examples:

```
(define black (number->painter 0))
(define white (number->painter 255))
(define gray (number->painter 150))
```

You can also specify a painter using `procedure->painter`, which takes a procedure as argument. The procedure determines a gray level (0 to 255) as a function of (x, y) position, for example:

```
(define diagonal-shading
  (procedure->painter (lambda (x y) (* 100 (+ x y)))))
```

The x and y coordinates run from 0 to 1 and specify the fraction that each point is offset from the frame's origin along the frame's edges. (See figure 2.) Thus, the frame is filled out by the set of points (x, y) such that $0 \leq x, y \leq 1$.

A third kind of painter is created by `segments->painter`, which takes a list of line segments as argument. This paints the line drawing specified by the list segments. The (x, y) coordinates of the line segments are specified as above. For example, you can make the "Z" shape shown in figure 3 as

```
(define mark-of-zorro
  (let ((v1 (make-vect .1 .9))
        (v2 (make-vect .8 .9))
        (v3 (make-vect .1 .2))
        (v4 (make-vect .9 .3)))
    (segments->painter
     (list (make-segment v1 v2)
           (make-segment v2 v3)
           (make-segment v3 v4)))))
```

The final way to create a primitive painter is from a stored image. The procedure `load-painter` uses an image from the 6001 image collection to create a painter.[2] or instance:

```
(define fovnder (load-painter "fovnder"))
```

will paint an image of William Barton Rogers, the FOVNDER of MIT. (See figure 3.)

Transforming and combining painters

We can transform a painter to produce a new painter which, when given a frame, calls the original painter on the transformed frame. For example, if `p` is a painter and `f` is a frame, then

[2]The images are kept in the directory `6001-images`. Use the Edwin command `M-x list-directory` to see entire contents of the directory. Each image is 128×128, stored in "pgm" format.

Figure 3: Examples of primitive painters: `mark-of-zorro` and `fovnder`.

```
(p ((make-frame-relative (make-vect .5 .5) (make-vect 1 .5) (make-vect .5 1))
    f))
```

will paint in the upper-right-hand corner of the frame.

We can abstract this idea with the following procedure:

```
(define (transform-painter origin corner1 corner2)
  (lambda (painter)
    (compose painter
             (make-relative-frame origin corner1 corner2))))
```

Calling this with an origin and two corners, returns a procedure that transforms a painter into one that paints relative to a new frame with the specified origin and corners. For example, we could define:

```
(define (shrink-to-upper-left painter)
  ((transform-painter (make-vect .5 .5) (make-vect 1 .5) (make-vect .5 1))
   painter))
```

Note that this can be written equivalently as

```
(define shrink-to-upper-left
  (transform-painter (make-vect .5 .5) (make-vect 1 .5) (make-vect .5 1)))
```

Other transformed frames will flip images horizontally:

```
(define flip-horiz
  (transform-painter (make-vect 1 0)
                     (make-vect 0 0)
                     (make-vect 1 1)))
```

or rotate images counterclockwise by 90 degrees:

```
(define rotate90
  (transform-painter (make-vect 1 0)
                     (make-vect 1 1)
                     (make-vect 0 0)))
```

By repeating rotations, we can create painters whose images are rotated through 180 or 270 degrees:

```
(define rotate180 (repeated rotate90 2))
(define rotate270 (repeated rotate90 3))
```

We can combine the results of two painters a single frame by calling each painter on the frame:

```
(define (superpose painter1 painter2)
  (lambda (frame)
    (painter1 frame)
    (painter2 frame)))
```

To draw one image beside another, we combine one in the left half of the frame with one in the right half of the frame:

```
(define (beside painter1 painter2)
  (let ((split-point (make-vect .5 0)))
    (superpose
     ((transform-painter zero-vector
                         split-point
                         (make-vect 0 1))
      painter1)
     ((transform-painter split-point
                         (make-vect 1 0)
                         (make-vect .5 1))
      painter2))))
```

We can also define painters that combine painters vertically, by using `rotate` together with `beside`. The painter produced by `below` shows the image for `painter1` below the image for `painter2`:

```
(define (below painter1 painter2)
  (rotate270 (beside (rotate90 painter2)
                     (rotate90 painter1))))
```

2. Tutorial exercises

You should prepare the following exercises for oral presentation in tutorial. They cover material in sections 2.1 and 2.2 of the text, and they also test your understanding of the square-limit language described above, in preparation for doing the lab in section 1 of this handout. You may wish to use the computer to check your answers to these questions, but you should try to do them (at least initially) without the computer.

Tutorial exercise 1: Show how to define a procedure `butlast` that, given a list, returns a new list containing all but the last element of the original list. For example

```
(butlast '(1 2 3 4 5 6))
;Value: (1 2 3 4 5)
```

Tutorial exercise 2: In the square-limit language, a frame is represented as a list of four things—the symbol `frame` followed by the origin and the two edge vectors.

1. Pick some values for the coordinates of origin and edge vectors and draw the box-and-pointer structure for the resulting frame.
2. Suppose we change the representation of frames and represent them instead as a list of three vectors—the origin and the two edges—without including the symbol `frame`. Give the new definitions of `make-frame`, `frame-origin`, `frame-edge1`, and `frame-edge2` for this

representation. In addition to changing these constructors and selectors, what other changes to the implementation of the square-limit language are required in order to use this new representation?

3. Why might it be useful to include the symbol `frame` as part of the representation of frames?

Tutorial exercise 3: Describe the patterns drawn by

```
(procedure->painter (lambda (x y) (* x y)))
(procedure->painter (lambda (x y) (* 255 x y)))
```

Tutorial exercise 4: Captain Abstraction is insulted that the Mark of Zorro appears in this problem set, while his mark does not. Show how to use `segments->painter` to draw an "A" for Captain Abstraction. Don't worry about getting the coordinates exactly correct—just get the general shape right.

Tutorial exercise 5: Section 1 defines `below` in terms of `beside` and `rotate`. Give an alternative definition of `below` that does not use `beside`.

Tutorial exercise 6: Define the painter transformation `flip-vertically`.

Tutorial exercise 7: Describe the effect of

```
(transform-painter (make-vect .1 .9)
                   (make-vect 1.5 1)
                   (make-vect .2 0))
```

3. To do in lab

Load the code for problem set 4, which contains the procedures described in section 1. You will not need to modify any of these. We suggest that you define your new procedures in a separate (initially empty) editor buffer, to make it easy to reload the system if things get fouled up.

When the problem set code is loaded, it will create three graphics windows, named `g1`, `g2`, and `g3`. To paint a picture in a window, use the procedure `paint`. For example,

```
(paint g1 fovnder)
```

will show a picture of William Barton Rogers in window `g1`. There is also a procedure called `paint-hi-res`, which paints the images at higher resolution (256×256 rather than 128×128). Painting at a higher resolution produces better looking images, but takes four times as long. As you work on this problem set, you should look at the images using `paint`, and reserve `paint-hi-res` to see the details of images that you find interesting.[3]

[3]Painting a primitive image like `fovnder` won't look any different at high resolution, because the original picture is only 128×128. But as you start stretching and shrinking the image, you will see differences at higher resolution.

Figure 4: (a) The "diamond of a frame is formed by joining the midpoints of the sides. (b) Painting created by `(diamond fovnder)`.

Printing pictures You can print images on the laserjet printer with Edwin's `M-x print-graphics` command as described in the *Don't Panic* manual. The laserjet cannot print true greyscale pictures, so the pictures will not look as good as they do on the screen. Please print only a few images—only the ones that you really want—so as not to waste paper and clog the printer queues. We suggest that you print only images created with `paint-hi-res`, not `paint`.

Lab exercise 1: Make a collection of primitive painters to use in the rest of this lab. In addition to the ones predefined for you (and listed in section 1), define at least one new painter of each of the four primitive types: a uniform grey level made with `number->painter`, something defined with `procedure->painter`, a line-drawing made with `segments->painter`, and an image of your choice that is loaded from the 6001 image collection with `load-painter`. Turn in a list of your definitions.

Lab exercise 2: Experiment with some combinations of your primitive painters, using `beside`, `below`, `superpose`, flips, and rotations, to get a feel for how these means of combination work. You needn't turn in anything for this exercise.

Lab exercise 3 The "diamond" of a frame is defined to be the smaller frame created by joining the midpoints of the original frame's sides, as shown in figure 4a. Define a procedure `diamond` that transforms a painter into one that paints its image in the diamond of the specified frame, as shown in figure 4b. Try some examples, and turn in a listing of your procedure.

Lab exercise 4 The "diamond" transformation has the property that, if you start with a square frame, the diamond frame is still square (although rotated). Define a transformation similar to `diamond`, but which does not produce square frames. Try your transformation on some images to get some nice effects. Turn in a listing of your procedure.

`(up-split mark-of-zorro 4)` `(up-split fovnder 4)`

Figure 5: The `up-split` procedure places a picture below two (recursively) up-split copies of itself.

Lab exercise 5 The following recursive `right-split` procedure was demonstrated in lecture:

```
(define (right-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (right-split painter (- n 1))))
        (beside painter (below smaller smaller)))))
```

Try this with some of the painters you've previously defined, both primitives and combined ones. Now define an analogous `up-split` procedure as shown in figure 5. Make sure to test it on a variety of painters. Turn in a listing of your procedure. (In defining your procedure, remember that `(below painter1 painter2)` produces `painter1` below `painter2`.)

Lab exercise 6 `Right-split` and `up-split` are both examples of a common pattern that begins with a means of combining two painters and applies this over and over in a recursive pattern. We can capture this idea in a procedure called `keep-combining`, which takes as argument a `combiner` (which combines two painters). For instance, we should be able to give an alternative definition of `right-split` as

```
(define new-right-split
  (keep-combining
   (lambda (p1 p2)
     (beside p1 (below p2 p2)))))
```

Complete the following definition of `keep-combining`:

```
(define (keep-combining combine-2)
  ;; combine-2 = (lambda (painter1 painter2) ...)
  (lambda (painter n)
    ((repeated
       ⟨ fill in missing expression ⟩
      n)
     painter)))
```

Show that you can indeed define `right-split` using your procedure, and give an analogous definition of `up-split`.

Lab exercise 7 Once you have `keep-combining`, you can use it to define lots of recursive means of combination. Figure 6 shows three examples. Invent some variations of your own. Turn in the code and one or two sample pictures.

Lab exercise 8 Define the `square-limit` transformation illustrated in figure 1. The recursive plan for this transformation was described in lecture. Using `right-split` and `up-split`, define a `corner-split` transformation that returns a painter that will generate one-fourth of the desired image. Then apply a transformation `4times` to the `corner-split` painter. `4times` returns a painter that paints its image (suitably rotated) in each quarter of its argument frame. Turn in a listing of the procedures you define: `corner-split`, `4times`, and any others. Also turn in a clear explanation (a few sentences together with a diagram) of the recursive plan for `corner-split`.

Lab exercise 9 The procedures you have implemented give you a wide choice of things to experiment with. Invent some new means of combination, both simple ones like `beside` and complex higher-order ones like `keep-combining` and see what kinds of interesting images you can create. Turn in the code and one or two figures.

Contest (Optional) Hopefully, you generated some interesting designs in doing this assignment. If you wish, you can enter printouts of your best designs in the 6.001 PS4 design contest. Turn in your design collection together with your homework, but *stapled separately*, and make sure your name is on the work. For each design, show the expression you used to generate it. Designs will be judged by the 6.001 staff and other paragons of design expertise, and fabulous prizes will be awarded in lecture. There is a limit of five entries per student. Make sure to turn in not only the pictures, but also the procedure(s) that generated them.

```
(define nest-diamonds
  (keep-combining
   (lambda (p1 p2) (superpose p1 (diamond p2)))))

(nest-diamonds fovnder 4)
```

```
(define new-comb
  (keep-combining
   (lambda (p1 p2)
     (square-limit (below p1 p2) 2))))

(new-comb mark-of-zorro 2)
```

```
(define mix-with-fovnder
  (keep-combining
   (lambda (p1 p2)
     (below (beside p1
                    fovnder)
            (beside p2 p2)))))

(mix-with-fovnder romana 3)
```

Figure 6: Some recursive combination schemes, defined with `keep-combining`.