Programming Paradigms
CISC 280

Problem Set 4

Henderson's Picture Language

Reading: Section 2.2.4

In this assignment, you will work with Peter Henderson's "square-limit" graphics design language, which is described in Section 2.2.4 of our textbook. Before beginning work on this programming assignment, you should review that section. 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—what many call today a "domain-specific language".[1]

Section 1 of this handout reviews the language, mostly similar to the text. You will need to study this in order to complete the exercises. Section 2 gives the actual assignment.

DrScheme note: use the "Pretty Big (includes MrEd and Advanced Student)" language.

1. The Square-limit language

Recall 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. To do this, we will need some basic building blocks.

Basic data structures

Vectors consist of a pair of numbers.

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

Then we need a set of operations on vectors:

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

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

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

---

[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. Adaptation to DrScheme by Michael Sperber and Allyn Dimrock. Problems based on material from Sussman, Abelson, Holly Yanco, and Allyn Dimrock

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 2.15 on page 135 of the text 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 $(1, 1)$.

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

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

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-relative-frame (make-vect 0.5 0.5) (make-vect 1 0.5) (make-vect 0.5 1))
```

returns the procedure that transforms a frame into the upper-right quarter of the frame.[2]

Painters

As described in Section 2.2.4 of the book, 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

---

[2]Actually into the corner of a frame farthest from the frames origin: thus to the upper-right if the frames origin is in the lower-left.

the frame. The image will be scaled and stretched to fit the frame.[3]

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 2.15 in the text) Thus, the frame is filled out by the set of points (x, y) such that $0 \le x \le 1$ and $0 \le y \le 1$. The DrScheme version has some rounding error that causes the range of displayable points (x, y) to be in the range $0 \le x < 1$ and $0 \le y < 1$ in the outermost frame. Any point that is mapped to within a pixel of 1 in the outermost frame is not displayed...)

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 1 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 you specify to create a painter. Images must be 128×128 pixels, stored in greyscale .gif format.

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

will paint an image of Albert Einstein. (See figure 1.)

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
```
(p ((make-relative-frame (make-vect 0.5 0.5) (make-vect 1 0.5) (make-vect 0.5 1))
    f))
```
will paint in the corner of the frame farthest from the origin.

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))))
```

---

[3]Actually, painting a picture requires some extra administrative work. Always use paint to paint a picture.
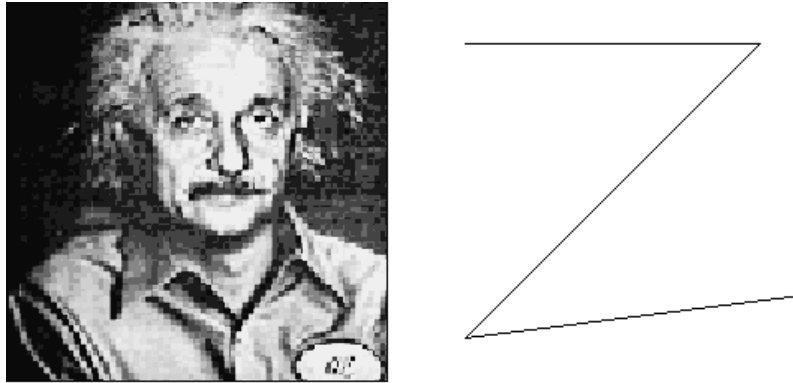
Figure 1: Examples of primitive painters: `mark-of-zorro` and `einstein`.

Note: the definition of `transform-painter` differs from the one in the book on page 138. Look at the definition of the procedure in the book and the one listed above. Convince yourself that they do the same thing.

Calling `transform-painter` 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-right 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-right
  (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)))
```

another rotates images counterclockwise by 90 degrees:

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

and similar rotations, `rotate180` and `rotate270`, can be created.

We can combine the results of two painters in 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. Exercises

In DrScheme, set the language to `Pretty Big`.

Copy all files for the assignment into a single directory.

Loading `ps4-ans.ss` in Dr. Scheme loads files containing the procedures described in Section 1. You will not need to modify any of these. Fill out your code and (commented out) written answers in `ps4-ans.ss`.

To paint a picture, use the procedure `paint`. For example, to paint a graphic image, `(paint einstein)` will show a picture of Albert Einstein in the interaction window.

To print a picture: You can print out the entire contents of your interaction window using File -¿ Print Interactions. If you have made lots of small versions of an image, `paint-hi-res` will paint in a frame of twice the size (looks bad for a single 128 by 128 image, but not so bad for multiplied and shrunken images (see the example pic below).

Exercise 1:   Answer the following exercise on paper. Do not actually make these changes to the system.

In the square-limit language, a frame is represented as a list of four thingsthe symbol frame followed by the origin and the two edge vectors.

- Pick some values for the coordinates of origin and edge vectors and draw the box-and-pointer structure for the resulting frame.

- 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?

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

Exercise 2:   Describe the patterns drawn by

```
(procedure->painter (lambda (x y) (* x y)))
(procedure->painter (lambda (x y) (* 255 x y)))
(procedure->painter (lambda (x y) (* 255 y)))
```
Try answering this problem without typing in the code. To check your answer, you could type

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

Exercise 3:   Describe the effect of

```
(transform-painter (make-vect  0 0.5)
                   (make-vect .5 1)
                   (make-vect .1 0))
```
Try this on paper first, before checking what the system does.

Exercise 4: Make a collection of primitive painters to use in the rest of this lab. In addition to the ones pre-defined 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 either loaded from the examples or created by you. Turn in a list of your definitions: : `my-number-painter`, `my-procedure-painter`, `my-segments-painter`, `my-image-painter`. If you are creating your own image, name it `my-pic.gif` and include it in your electronic submission.

You can create you own image using any image editing tool; for example xv on Unix. It must be a 128 x 128 greyscale GIF.

Exercise 5: 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.

Exercise 6: 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 2. 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 2. Try some examples, and turn in a listing of your procedure.

Figure 2: The "diamond" of a frame is formed by joining the midpoints of the sides. This is illustrated with a painting created by `(diamond einstein)`.

Exercise 7: The "diamond" transformation has the property that, if you start with a square frame, the diamond frame is still square (although rotated). Define a procedure `make-non-square-diamond` which takes 3 parameters: the first is the y coordinate of the origin (the x coordinate of the origin will always be 0); the second is the x coordinate of edge1 (y will always be 0); and the third is the x coordinate of edge2 (y will always be 1).

Use your procedure `make-non-square-diamond` to define `non-square-diamond-1` with the first parameter equal to 0.2, the second equal to 0.1 and the third equal to 0.9. Write at least one other transform that uses `make-non-square-diamond` to define it. Try your transformation(s) on some images to get some nice effects. Turn in a listing of your procedures.

Exercise 8: The following recursive `right-split` procedure was demonstrated in the text:

```
(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 3. 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`.)
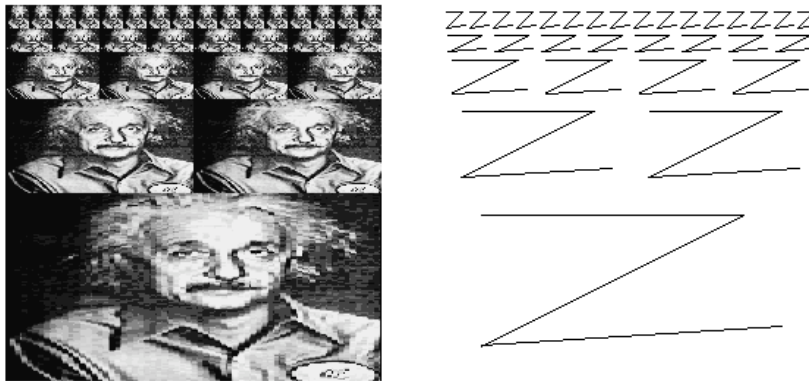

Figure 3: The `up-split` procedure places a picture below two (recursively) up-split copies of itself. This was created from `(up-split einstein 4)` and `(up-split mark-of-zorro 4)`

Exercise 9: `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)))
```

where `repeated` is given by:

```
(define (repeated f n)
  (cond ((= n 0) identity)
        ((= n 1) f)
        (else (compose f (repeated f (- n 1))))))
```

Show that you can indeed define `right-split` using your procedure, and give an analogous definition of `up-split`. Name your new procedures `right-split2` and `up-split2`.

Exercise 10   Once you have `keep-combining`, you can use it to define lots of recursive means of combination. Here are three examples:

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

(nest-diamonds diagonal-shading 4)

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

(new-comb mark-of-zorro 1)

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

(mix-with-einstein logo 3)
```

Invent some variations of your own. Turn in the code and one or two sample pictures.

Exercise 11:   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: procedure `painter-11a`, optionally `painter-11b`,...