MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996-97

Problem Set 4

The Square-limit language

Issued: Tuesday, September 24, 1996

Written solutions due: in recitation on Friday, October 4, 1996

Tutorial preparation for: week of September 30, 1996

Reading: Section 2.2; code files `hend.scm` and `hutils.scm` (attached)

In this assignment, you will work with Peter Henderson's "square-limit" graphics design language, which will be described in lecture on September 26, and which appears in Section 2.2.4 of the notes. 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.[1]

Section 1 of this handout reviews the language, similar to what appears in the notes. 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.

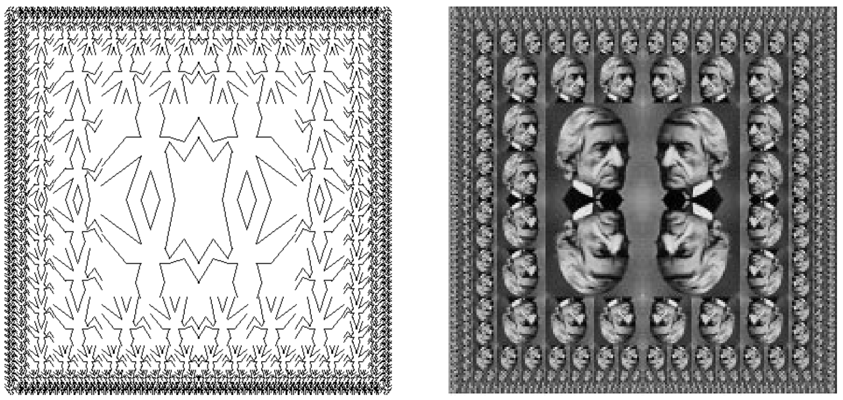
Figure 1: Designs generated with the picture language.

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, glued together in some appropriate manner. The contract we enforce for this data abstraction is the following:

- the constructor `make-vect` takes two arguments and associates them into a vector, and
- the selectors `xcor-vect` and `ycor-vect` return the components of the vector constructed by `make-vect`. (You'll actually implement this abstraction later on, but of course for purposes of discussion, we only need to know the abstraction contract.)

We need a set of operations on vectors, and in particular assume three:

- `add-vect` takes two vectors as input, and returns as output another vector, whose components are the sums of the components of the input vectors;
- `sub-vect` takes two vectors as input, and returns as output another vector, whose components are the differences of the components of the input vectors (e.g. the first vector minus the second);
- `scale-vect` takes as input a number and a vector and returns a vector whose components are the components of the input vector, multiplied by the input number.

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. Again, we just need a contract:

- constructor is `make-segment` and
- selectors are `start-segment` and `end-segment`.

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

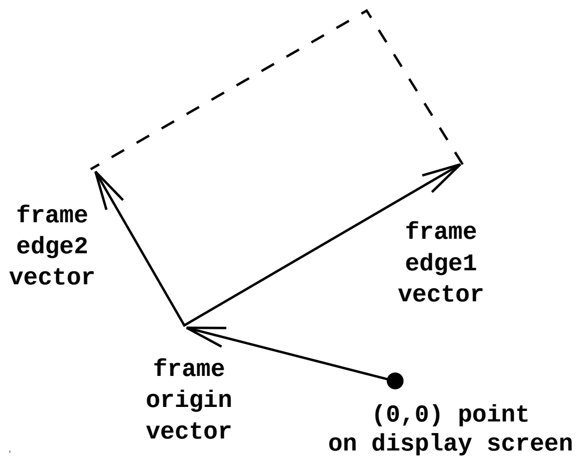

Figure 2: A frame is described by three vectors—an origin and two edges.

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 $(1, 1)$.

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)
    (add-vect
     (origin-frame frame)
     (add-vect (scale-vect (xcor-vect point-in-frame-coords)
                           (edge1-frame frame))
               (scale-vect (ycor-vect point-in-frame-coords)
                           (edge2-frame frame))))))
```

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

The procedure `make-relative-frame` provides a convenient way to transform frames. Given three points `origin`, `corner1`, and `corner2` (expressed in frame coordinates), it returns a procedure which for any frame f returns a new frame g which uses those points in f coordinates to define the corners of the g frame:

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

For example,

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

returns the procedure that takes a frame f and returns the upper right quarter of f as a new frame g.

Painters

As described in the notes, 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

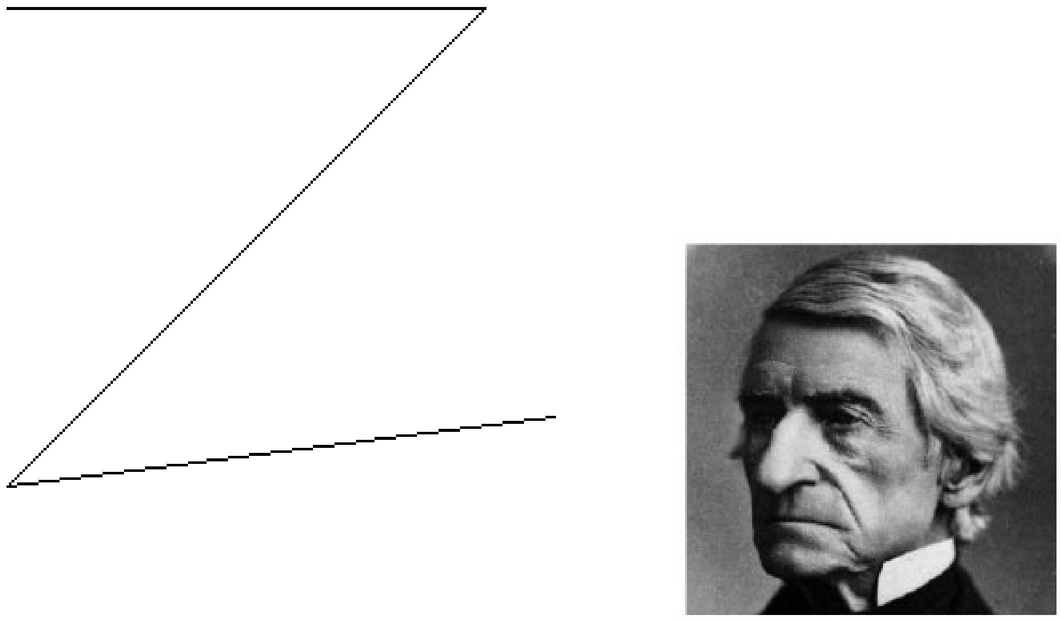

Figure 3: Examples of primitive painters: `mark-of-zorro` and `fovnder`.

```
(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]. For 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

Given a painter, we can produce a new painter that transforms its frame before painting in it. For example, if `p` is a painter and `f` is a frame, then

[2]The images are kept in the directory `6001-images`. The `load-painter` procedure transforms them into painters, so that they can be scaled and deformed by the operations in the square-limit language. Use the Edwin command `M-x list-directory` to see entire contents of the directory. Each image is 128×128, stored in "pgm" format.

```
(p ((make-relative-frame (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 `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-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)))
```

Another transformed frame, called `flip-horiz` should flip images horizontally (we'll ask you to implement this later), 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. 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 this week's lab in. 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: Do exercises 2.21 and 2.22 from the notes.

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 – using some definition that you choose for vectors as well.
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`, `origin-frame`, `edge1-frame`, and `edge2-frame` 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: Section 1 defines `below` in terms of `beside` and `rotate`. Give an alternative definition of `below` that does not use `beside`.

Tutorial exercise 5: 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 3, 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` (once you have successfully completed lab exercise 1 and `(load-rest)` successfully evaluates.

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]

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: Complete the implementation of the data abstractions, by choosing a representation for vectors `(make-vect, xcor-vect, ycor-vect)` and for segments `(make-segment, start-segment, end-segment)`. Use those abstractions to implement the three operations on vectors `(add-vect, sub-vect, scale-vect)`. You may find it convenient to do this in the copy

[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.

of `hutils.scm` that was loaded into your editor. Once you are done adding your definitions, save the file and then load it into your Scheme buffer by evaluating:

```
(load `` /work/hutils.scm'')
```

If you work on the problem set in multiple sessions, be sure that you reload this file after you have loaded up the problem set, so that your new definitions will override the ones in the problem set file.

Turn in a listing of your code, and some examples of using it.

Once you have created your data abstractions, evaluate

```
(load-rest)
```

to have the rest of the code for the problem set loaded into your Scheme environment.

Lab exercise 2: 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 3: Earlier we referred to examples of transforming painters, i.e. procedures that take a painter as input and create a new painter that will draw relative to some new frame. Increase the repetoire of such methods by implementing a transformation, `flip-horiz`, which takes as input a painter, and returns a new painter that draws its input flipped about the vertical axis. Also implement `rotate180` and `rotate270` in analogy to `rotate90`. Turn in a listing of your procedures.

Lab exercise 4: 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 5: 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 4. 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 4. Try some examples, and turn in a listing of your procedure.

Lab exercise 6: 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.

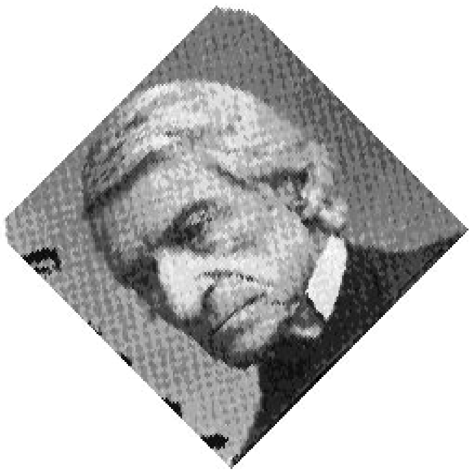

Figure 4: The "diamond" of a frame is formed by joining the midpoints of the sides. This is illustrated with a painting created by `(diamond fovnder)`.

Lab exercise 7: 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 8: `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`:

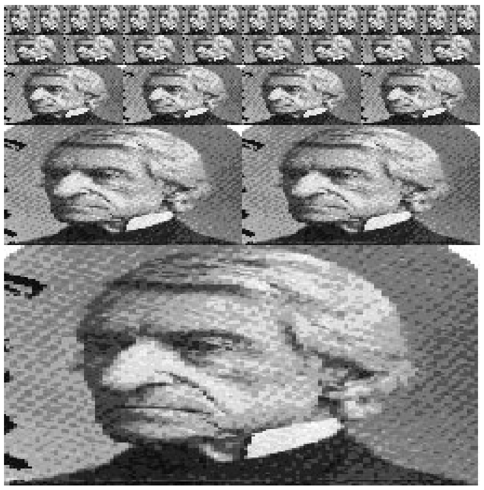

Figure 5: The `up-split` procedure places a picture below two (recursively) up-split copies of itself. This was created from `(up-split fovnder 4)`

```
(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`.

Lab exercise 9 Once you have `keep-combining`, you can use it to define lots of recursive means of combination. Figure 6 shows an example, which comes from:

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

(nest-diamonds fovnder 4)
```

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

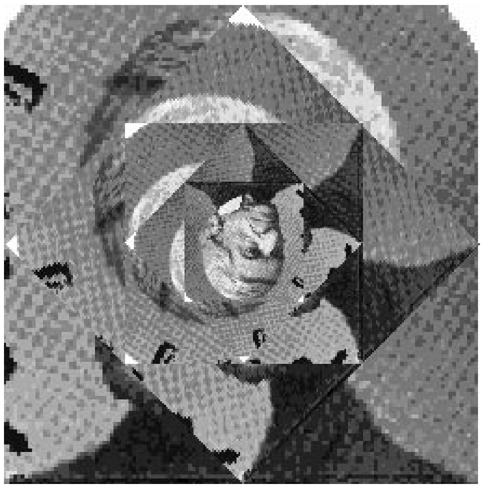

Figure 6: Some recursive combination schemes, defined with `keep-combining`.

Lab exercise 10: 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 PS3 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 internationally famous art critics, 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.

How much time did you spend on this homework assignment? Report separately time spent before going to lab and time spent in the 6.001 lab.

If you cooperated with other students working this problem set please indicate their names on your solutions. As you should know, as long as the guidelines described in the *6.001 Policy on Cooperation* handout are followed, such cooperation is allowed and encouraged.