MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1998

Problem Set 3

- Issued: Tuesday, February 17 (and copies distributed in recitation on Wednesday, February 18)
- Tutorial preparation for: Week of February 23.
- Written solutions due: Friday, February 27 in recitation
- Reading: Read sections 2.1 and 2.2.1 before lecture on February 19. Finish section 2.2 before lecture on February 24. Read sections 2.3.1 and 2.3.2 before lecture on February 26. Read section 2.2.4 for this problem set.

A Graphics Design Language[1]

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. We'll do this by working with Peter Henderson's "square-limit" graphics design language, which is described in section 2.2.4 of the textbook. You should study that section before beginning work on this assignment.[2]

1. Tutorial exercises

Tutorial exercise 1: Do exercises 2.46, 2.47, and 2.48 of the textbook, which ask you to define selectors and constructors that implement data structures for vectors (`make-vect`, `xcor-vect`, `ycor-vect`), for frames (`make-frame`, `origin-frame`, `edge1-frame`, `edge2-frame`), and for line segments (`make-segment`, `start-segment`, `end-segment`), as well as some basic vector operations (`add-vect`, `sub-vect`, `scale-vect`). You need not try these on the computer now (although you'll need to do this as part of the programming assignment). Note that there are different possible answers for these: the choice of representation is up to you. In tutorial, expect that your tutor will ask you to draw box-and-pointer diagrams to describe some of these data structures, and also to discuss how these structures are printed by the Scheme interpreter.

Tutorial exercise 2: Do exercise 2.18 of the textbook, which asks you to define a procedure that returns the reverse of a given list.

[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.

[2] Section 2.2.4 does not depend very strongly on section 2.2.3, so you can start working on this problem set without reading 2.2.3. Be sure, however, to read all of section 2.2 before lecture on February 24.

Tutorial exercise 3: Do exercise 2.23 of the textbook.

Tutorial exercise 4: Do exercise 2.24 of the textbook.

Tutorial exercise 5: Do exercise 2.25 of the textbook.

2. Programming assignment

We won't provide any general explanation of the square-limit language here, since this is covered in section 2.2.4 of the textbook. One thing that is not explained in the book, however, is how primitive painters are implemented (see the book, pages 136–137) and how to actually use a painter to draw something on the screen.

Primitive painters

The code for this assignment 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 vertical-shading
  (procedure->painter (lambda (x y) (* 255 y))))
```

The x and y arguments run from 0 to 1 and specify the fraction that each point is offset from the frame's origin along the frame's edges. Thus, the frame is filled out by the set of points (x, y) such that $0 \leq x, y \leq 1$.[3]

A third kind of painter is created by `segments->painter`, as described in the textbook. This takes a list of line segments as argument. This paints the line drawing specified by the list segments. For example, the `wave` painter shown in figure 2.10 of the book is generated by

[3] This way of constructing a painter is a lot like the `procedure->picture` procedure you used in problem set 1. The difference is that here the x and y arguments range between 0 and 1, rather than from 0 to 127.

```
(define wave
  (segments->painter
   (list (make-segment (make-vect .25 0) (make-vect .35 .5))
         (make-segment (make-vect .35 .5 ) (make-vect .3 .6))
         (make-segment (make-vect .3 .6) (make-vect .15 .4))
         (make-segment (make-vect .15 .4) (make-vect 0 .65))
         (make-segment (make-vect .4 0) (make-vect .5 .3))
         (make-segment (make-vect .5 .3) (make-vect .6 0))
         (make-segment (make-vect .75 0) (make-vect .6 .45))
         (make-segment (make-vect .6 .45) (make-vect 1 .15))
         (make-segment (make-vect 1 .35) (make-vect .75 .65))
         (make-segment (make-vect .75 .65) (make-vect .6 .65))
         (make-segment (make-vect .6 .65) (make-vect .65 .85))
         (make-segment (make-vect .65 .85) (make-vect .6 1))
         (make-segment (make-vect .4 1) (make-vect .35 .85))
         (make-segment (make-vect .35 .85) (make-vect .4 .65))
         (make-segment (make-vect .4 .65) (make-vect .3 .65))
         (make-segment (make-vect .3 .65) (make-vect .15 .6))
         (make-segment (make-vect .15 .6) (make-vect 0 .85))
         )))
```

The final way to create a primitive painter is from a stored image. The procedure `pgm-file->painter` uses an image from the 6001 image collection to create a painter.[4] For instance:

```
(define rogers (pgm-file->painter "fovnder"))
```

will create the William Barton Rogers painter shown on page 130 of the textbook and give it the name `rogers`.

Drawing on the screen

When the problem set code is loaded (don't load it yet!), it will create three graphics windows, named `g1`, `g2`, and `g3`. To paint a picture in a window, use the procedure `paint`. `Paint` takes a graphics window and a painter, determines the frame for the graphics window, and gives the frame to the painter. For example,

```
(paint g1 rogers)
```

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

[4]The images are kept in the directory specified by the variable `6001-image-directory`. These images are accessible in a shared directory in the lab, and they are loaded as part of the PS3 problem set code if you are using your own computer. Use the Edwin command `M-x list-directory` to see the entire contents of the image directory. Each image is 128×128, stored in "pgm" format.

takes four times as long. Depending on how fast your computer is, you may want to work on this problem viewing images using `paint`, and reserve `paint-hi-res` to see the details of images that you find interesting.[5] When you print images, we suggest that you print only images created with `paint-hi-res`, not `paint`.[6]

Computer exercise 1: Load the code for problem set 3 using `M-x load-problem-set`. Before you can do anything else, you'll need to define the data representations and operations that you designed for tutorial exercise 1. Type in these definitions now, in a file that will hold all your answers for this problem set, and evaluate them.

If these are correct, you should be able to evaluate the expression `(setup)`. This will create the three graphics windows and load the rest of the problem set code, which includes all of the code from section 2.2.4 of the textbook and the primitive painters `black`, `white`, `gray`, `vertical-shading`, and `rogers` described above. If `setup` works, you should be able to use `paint` and `paint-hi-res` to view images of the primitive painters.[7] If you work on the problem set in multiple sessions, be sure that you reload your data abstraction definitions each time, before doing `setup`. You need not turn in anything for this exercise.

Computer exercise 2: Make a collection of primitive painters to use in the rest of this lab. In addition to the ones predefined for you, 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 `pgm-file->painter`. Turn in a list of your definitions.

Computer exercise 3: Do exercise 2.50 of the textbook. The way to think about these transformations is to keep in mind where the new origin and edges of the frame should be. It will help to make a sketch. If you are confused by this, study the definition of `rotate90` on page 139. Turn in a listing of your three procedures.

Computer exercise 4: Do exercise 2.51 of the textbook, which asks for two different definitions of the procedure `below`. The first definition can be tricky—make sure you understand how `beside` works. Turn in listings of both definitions.

Computer exercise 5: Do exercise 2.44 of the textbook, which asks you to define the procedure `up-split`. Turn in a listing of your `up-split` procedure. If you do this correctly (and also exercise 4), then `corner-split` and `square-limit` (both of which have been pre-defined for you) should work. You should now be able to duplicate the designs in figures 2.9 and 2.14 of the textbook.

[5] Painting a primitive image like `rogers` 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.

[6] You may have already learned how to print graphics when you did problem set 1. If not, consult the "Don't Panic" manual.

[7] If `setup` (or painting) does not work, there are several things that could be wrong. Your data abstraction definitions might be incorrect. Or the system might not be able to locate the image files or the compiled code files need for this problem set. Whatever the problem is **fix it now**, getting help if necessary, before going on.

Computer exercise 6: Examine the procedure `squash-inwards` (and also the diamond-shaped images in figures 2.10 and 2.11). You should be able to duplicate these, since `squash-inwards` is predefined in the problem set code. Define a couple of procedures that, like `squash-inwards`, draws in a non-rectangular frame. It's also interesting to make the corners of the diamond go *outside* the original square. Turn in a listing of your procedure.

Computer exercise 7: Spend some time creating some images using your primitive painters, together with the operations you've defined so far in this problem set such as `beside`, squashes, flips, and rotations. You needn't turn in anything for this exercise.

Computer exercise 8: Do exercise 2.45 of the textbook, which defines the general splitting operation `split`. In order not to overwrite the existing definitions of `right-split` and `up-split`, test your procedure by defining

```
(define new-right-split (split beside below))
(define new-up-split (split below beside))
```

Turn in a listing of `split`. Hint: This exercise will really test your understanding of higher-order procedures. The thing to keep in mind is that the result returned by `split` is a procedure that takes as arguments a painter and a number.

Computer exercise 9: `Beside` takes two painters as arguments and returns a painter. `Below` does likewise. Here's another example that is predefined in the problem-set code:

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

`Superpose` simply draws both painters in the same frame.[8] Define one or two other (interesting) means of combination that takes two painters as arguments and produces a painter. You can use these (together with `beside`, `below`, and `superpose`) in conjunction with `split`, to produce new recursive designs. Explore some of these. Turn in a listing of your procedures, together with a printout of some interesting recursive design you've made, and the code that produced it.

Computer exercise 10: Invent a new higher-order combiner (like `split`) and see what interesting images you can create. Turn in the procedure listing and a printout of some interesting design made with it.

[8] This isn't interesting if both painters are images that fill the entire frame—you'll just see `painter2`—but it becomes interesting if the second painter is a line drawing.

PS3 Design contest (Optional): Hopefully, you generated some appealing designs in doing this problem set. You are invited to 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 *two* entries per student. Make sure to turn in not only the pictures, but also the procedure(s) that generated them.

Turn in answers to the following questions along with your answers to the questions in the problem set:

1. About how much time did you spend on this homework assignment? (Reading and preparing the assignment plus computer work.)
2. Which scheme system(s) did you use to do this assignment (for example: 6.001 lab, your own NT machine, your own Win95 machine, your own Linux machine)?
3. We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout).
   - If you cooperated with other students, LA's, or others, or found portions of your answers for this problem set in references other than the text (such as some of the archives), please indicate your consultants' names and your references. Also, explicitly label all text and code you are submitting which is the same as that being submitted by one of your collaborators.
   - Otherwise, write "I worked alone using only the reference materials," and sign your statement.