Version of September 28, 1997

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1997

Problem Set 3

Graphing with Higher-order Procedures

Issued: Tuesday, 23 September
Due: Wednesday, 1 October
Tutorial preparation for: Week of September 29
Reading:

- Reading for Lectures: *SICP* Chapter 2 through 2.2.3.
- Reading for this Problem Set: *SICP* Chapter 1.
- code files `utils.scm`, `curves.scm` (attached to this handout)

One of the things that makes Scheme different from other common programming languages is the ability to operate with *higher-order procedures*, namely, procedures that manipulate and generate other procedures. This problem set will give you extensive practice with higher-order procedures, in the context of a language for graphing two-dimensional curves and other shapes. Sections 1 and 2 give some background on the essential ideas, with Tutorial, Pre-Lab and Lab exercises included. Section 3, the actual lab assignment, describes the graphics language and gives applications to generating fractal designs.

1 Tutorial Preparation: Procedure Types and Procedure Constructors

Tutorial exercise 1: Do Exercise 1.32 in the course notes.

In this assignment we use many procedures which may be applied to many different types of arguments and may return different types of values. To keep track of this, it will be helpful to have some simple notation to describe types of Scheme values.

Two basic types of values are Sch-Num, the Scheme numbers such as 3, -4.2, 6.931479453e89, and Sch-Bool, the truth values `#t,#f`. The procedure `square` may be applied to a Sch-Num and will return another Sch-Num. We indicate this with the type notation:

$$\texttt{square} : \text{Sch-Num} \rightarrow \text{Sch-Num}$$

```
(define (square x) (* x x))
```

In trigonometry it is common to talk about $\cos^2$, the square of the cosine function. In Scheme, the procedure corresponding to the function $\cos^2$ is

```
(define (cos-square x) (square (cos x)))
```

Tutorial exercise 2: Explain why the following Scheme definition does not work. (How would Scheme respond if we evaluated this definition?)

```
(define cos-square (square cos))
```

Of course it really does make sense to square a numerical function, if we get the types right

```
(define square-a-procedure
 (lambda (f) (lambda (x) (square (f x)))))
```

Now the procedure `square-a-procedure` may be applied to a procedure of type Sch-Num $\rightarrow$ Sch-Num and returns another procedure of that type:

$$\texttt{square-a-procedure} : (\text{Sch-Num} \rightarrow \text{Sch-Num}) \rightarrow (\text{Sch-Num} \rightarrow \text{Sch-Num})$$

or, abbreviating Sch-Num $\rightarrow$ Sch-Num as F

$$\texttt{square-a-procedure} : F \rightarrow F$$

and now we can say correctly

```
(define cos-square (square-a-procedure cos))
```

There are many other operations like squaring which can usefully be made into operations on procedures. The method for making such operations is captured by

```
(define (make-procedure-op number-op)
  (lambda (f) (lambda (x) (number-op (f x)))))
```

so now we can define `square-a-procedure` another way:

```
(define square-a-procedure (make-procedure-op square))
```

That is, `make-procedure-op` may be applied to an "operator" of type F and returns the corresponding operator on procedures of type F:

$$\texttt{make-procedure-op} : F \rightarrow (F \rightarrow F)$$

We can also look at operations which can take two arguments, for example, subtraction. The procedure - may be applied to two Sch-Num's and will return a Sch-Num. We indicate this with the notation:

$$- : (\text{Sch-Num} \times \text{Sch-Num}) \rightarrow \text{Sch-Num}$$

We can also make binary operations on numbers into operations on procedures:

```
(define (make-procedure-binop binop)
  (lambda (f g) (lambda (x) (binop (f x) (g x)))))
(define subtract-procedures (make-procedure-binop -))
```

We can use this to define the `square-difference` procedure

```
(define (square-difference f g)
   (subtract-procedures
    (square-a-procedure f)
    (square-a-procedure g)))
```

$$\texttt{square-difference} : (F \times F) \rightarrow F$$

Tutorial exercise 3: What is the type of `make-procedure-binop` as it was used in the expression

```
(make-procedure-binop -)
```

So far we have made operations on numbers into operations on procedures, but it is actually common to make them into operations on operations on procedures! For example, the procedure `deriv` of *SICP*, Section 1.3.4, has type $F \rightarrow F$. With it, we can define another procedure `deriv-squared` of the same type:

```
(define (deriv-squared f)
  (square-a-procedure (deriv f)))
```

Tutorial exercise 4: Explain why the following Scheme definition does not work. (How would Scheme respond if we evaluated this definition?)

```
(define deriv-squared (square-a-procedure deriv))
```

How about a "squaring" operation which applies to an operation like `deriv`? No problem:

```
(define (square-an-operator-on-proc op-on-proc)
  (lambda (f) (square-a-procedure (op-on-proc f))))

(define deriv-squared (square-an-operator-on-proc deriv))
```

Tutorial exercise 5: What is the type of `square-an-operator-on-proc`?

2 Curves as Procedures and Data

We're going to develop a language for defining and drawing planar curves. We'd like to plot points, construct graphs of functions, transform curves by scaling and rotating, and so on. One of the key ideas that we'll stress throughout 6.001 is that a well-designed language has parts that combine to make new parts that themselves can be combined. This property is called *closure*.

A planar curve in "parametric form" can be described mathematically as a function from parameter values to points in the plane. For example, we could describe the *unit-circle* as the function taking t to $(\cos 2\pi t, \sin 2\pi t)$ where t ranges over the unit interval $[0, 1]$. In Scheme, we let Unit-Interval be the type of Scheme-numbers between 0 and 1, and we represent curves by procedures of Scheme type Curve, where

$$\text{Curve} = \text{Unit-Interval} \to \text{Point}$$

and Point is some representation of pairs of Sch-Num's.

To work with Point, we need a *constructor*, `make-point`, which constructs Point's from Sch-Num's, and *selectors*, `x-of` and `y-of`, for getting the x and y coordinates of a Point.

$$\begin{aligned} \texttt{make-point} &: (\text{Sch-Num} \times \text{Sch-Num}) \to \text{Point}, \\ \texttt{x-of}, \texttt{y-of} &: \text{Point} \to \text{Sch-Num}. \end{aligned}$$

We require only that the constructors and selectors obey the *pairing contract*

$$(\texttt{x-of}\,(\texttt{make-point}\,n\,m)) = n$$
$$(\texttt{y-of}\,(\texttt{make-point}\,n\,m)) = m$$

for all Sch-Num's m, n. This abstraction of points is maintained throughout this problem set, so it makes absolutely no difference how abstract points are implemented.[1]

For example, we can define the Curve `unit-circle` and the Curve `unit-line` (along the x-axis):

```
(define (unit-circle t)
  (make-point (sin (* 2pi t))
              (cos (* 2pi t))))


(define (unit-line-at y)
  (lambda (t) (make-point t y)))


(define unit-line (unit-line-at 0))
```

Note that we are already making a conceptual shift in our thinking. Here, a curve is not a geometric set of points, rather it is a procedure for taking number values in the unit interval, and generating the corresponding point. The key is that a curve is a procedure!

Pre-Lab exercise 1:

1. What is the type of `unit-line-at`?
2. Define a procedure `diagonal-line` with two arguments, a point and a length, and returns a line of that length beginning at the point and with tangent $\pi/4$ (that is, pointing upward at a 45 degree angle). By what we have discussed above, `diagonal-line` should return a curve (hence a procedure) that takes as input a parameter along the unit line (as do all curves) and returns the appropriate point along the line.
3. What is the type of `diagonal-line`?

[1] Just to emphasize the abstract treatment of points, a nonstandard implementation of points as procedures is used (cf., the attached file `curves.scm`) rather than the typical one using `cons`, `car`, `cdr`.

In addition to the direct construction of Curve's such as `unit-circle` or `unit-line`, we can use elementary Cartesian geometry in designing Scheme procedures which *operate* on Curve's. For example, the mapping $(x, y) \longrightarrow (-y, x)$ rotates the plane by $\pi/2$, so

```
(define (rotate-pi/2 curve)
  (lambda (t)
    (let ((ct (curve t)))
      (make-point
       (- (y-of ct))
       (x-of ct)))))
```

defines a procedure which takes a curve and transforms it into another, rotated, curve. The type of `rotate-pi/2` is

$$\text{Curve-Transform} = \text{Curve} \rightarrow \text{Curve}.$$

Thus, a curve transform is a procedure that converts an input procedure to a new procedure, i.e., a curve transform is a higher order procedure. Note in general that having a framework for describing types enables us to reason about procedures, e.g., knowing the type of an argument and the desired type of the output provides guidance on what must be done within a procedure to execute the desired transformation.

We have provided a variety of other procedure Curve-Transform's and procedures which construct Curve-Transform's in the file `curves.scm`. For example,

- `translate` returns a Curve-Transform which rigidly moves a curve given distances along the x and y axes,
- `scale-x-y` returns a Curve-Transform which stretches a curve along the x and y coordinates by given scale factors, and
- `rotate-around-origin` returns a Curve-Transform which rotates a curve by a given number of radians.

Pre-Lab exercise 2: Write a definition of a Curve-Transform `upside-down`, which flips a curve over a horizontal line through its start point.

A convenient, if somewhat more complicated, Curve-Transform is `put-in-standard-position`. We'll say a curve is in *standard position* if its start and end points are the same as those of the unit-line, namely, it starts at the origin, $(0, 0)$, and ends at the point $(1, 0)$. We can put any curve whose start and endpoints are not the same into standard position by (1) rigidly translating it so its starting point is at the origin, then (2) rotating it about the origin to put its endpoint on the x axis, and (3) finally scaling it to put the endpoint at $(1, 0)$:

```
(define (put-in-standard-position curve)
  (let* ((start-point (curve 0))
         (curve-started-at-origin
          ((translate (- (x-of start-point))
                      (- (y-of start-point)))
           curve))
         (new-end-point (curve-started-at-origin 1))
         (theta (atan (y-of new-end-point) (x-of new-end-point)))
         (curve-ended-at-x-axis
          ((rotate-around-origin (- theta)) curve-started-at-origin))
         (end-point-on-x-axis (x-of (curve-ended-at-x-axis 1))))
    ((scale (/ 1 end-point-on-x-axis)) curve-ended-at-x-axis)))
```

It is useful to have operations which combine curves into new ones. We let Binary-Transform be the type of binary operations on curves,

$$\text{Binary-Transform} = (\text{Curve} \times \text{Curve}) \to \text{Curve}.$$

The procedure `connect-rigidly` is a simple Binary-Transform. Evaluation of `(connect-rigidly curve1 curve2)` returns a curve consisting of `curve1` followed by `curve2`; the starting point of the curve returned by `(connect-rigidly curve1 curve2)` is the same as that of `curve1` and the end point is the same as that of `curve2`.

```
(define (connect-rigidly curve1 curve2)
  (lambda (t)
    (if (< t (/ 1 2))
        (curve1 (* 2 t))
        (curve2 (- (* 2 t) 1)))))
```

Pre-Lab exercise 3: There is another, possibly more natural, way of connecting curves. The curve returned by `(connect-ends curve1 curve2)` consists of a copy of `curve1` followed by a copy of `curve2` after it has been rigidly translated so its starting point coincides with the end point of `curve1`.

Write a definition of the Binary-Transform `connect-ends`. Think carefully about the types of the different procedures you use, as this will help you decide how to stitch together the different kinds of operations you need.

3 To do in Lab: Drawing Curves

Use `M-x load-problem-set` to load the code for Problem Set 3. This will create three graphics windows called `g1`, `g2` and `g3`. The window coordinates go from 0 to 1 in both x and y with $(0, 0)$ at the lower left.

A *drawing procedure* takes a curve argument and automagically displays points on the curve in a window[2]. We've provided several procedures that take a window (for example `g1`) and a number of points, and return a drawing procedure, namely,

- `draw-points-on`,
- `draw-connected`,
- `draw-points-squeezed-to-window`, and
- `draw-connected-squeezed-to-window`.

Lab exercise 4: The procedure `alternative-unit-circle` in the file `curves.scm` also defines a unit-circle curve. Apply `(draw-connected g1 200)` to `unit-circle`, and `(draw-connected g2 200)` to `alternative-unit-circle`. Can you see a difference? Now try using `draw-points-on` instead of `draw-connected`. Also try `draw-points-squeezed-to-window`. Print out the resulting figures. Briefly explain why the unit circles printed using `unit-circle` and `alternative-unit-circle` differ.

3.1 Fractal Curves

To show off the power of our drawing language, let's use it to explore fractal curves. Fractals have striking mathematical properties.[3] Fractals have received a lot of attention over the past few years, partly because they tend to arise in the theory of nonlinear differential equations, but also because they are pretty, and their finite approximations can be easily generated with recursive computer programs.

For example, Bill Gosper[4] discovered that the infinite repetition of a very simple process creates a rather beautiful image, now called the *Gosper C Curve.* At

[2] *The Hacker's Dictionary* (see the Jargon file) defines "automagically" as "automatically, but in a way which, for some reason (typically because it is too complicated, or too ugly, or perhaps even too trivial), the speaker doesn't feel like explaining." In this case, we don't want to explain the bletcherous (see Jargon) details of how points are plotted.

[3] A fractal curve is a "curve" which, if you expand any small piece of it, you get something similar to the original. The Gosper curve, for example, is neither a true 1-dimensional curve, nor a 2-dimensional region of the plane, but rather something in between.

[4] Bill Gosper is a mathematician now living in California. He was one of the original hackers who worked for Marvin Minsky in the MIT Artificial Intelligence Laboratory during the '60s. He is perhaps best known for his work on the Conway Game of Life—a set of rules for evolving cellular automata. Gosper invented the "glider gun", resolving Conway's question as to whether it is possible to produce a finite pattern that evolves into an unlimited number of live cells. He used this result to prove that the Game of Life is Turing universal, in that it can be used to simulate any other computational process!

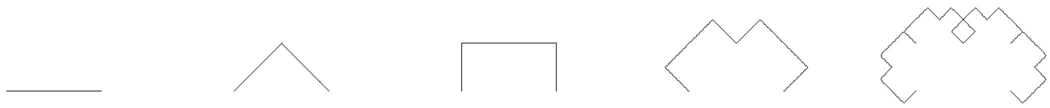

Figure 1: Examples of the Gosper C curve at levels 0, 1, 2, 3, and 5.

each step of this process there is an approximation to the Gosper curve. The next approximation is obtained by adjoining two scaled copies of the current approximation, each rotated by 45 degrees.

Figure 1 shows the first few approximations to the Gosper curve, where we stop after a certain number of levels: a level-0 curve is simply a straight line; a level-1 curve consists of two level-0 curves; a level 2 curve consists of two level-1 curves, and so on. You can also see from the figure how we use a recursive strategy for making the next level of approximation: a level-n curve is made from two level-$(n-1)$ curves, each scaled to be $\sqrt{2}/2$ times the length of the original curve. One of the component curves is rotated by $\pi/4$ (45 degrees) and the other is rotated by $-\pi/4$. After each piece is scaled and rotated, it must be translated so that the ending point of the first piece is continuous with the starting point of the second piece.

We assume that the approximation we are given to improve (named `curve` in the procedure) is in standard position. By doing some geometry, you can figure out that the second curve, after being scaled and rotated, must be translated right by .5 and up by .5, so its beginning coincides with the endpoint of the rotated, scaled first curve. This leads to the Curve-Transform `gosperize`:

```
(define (gosperize curve)
  (let ((scaled-curve ((scale (/ (sqrt 2) 2)) curve)))
    (connect-rigidly ((rotate-around-origin (/ pi 4)) scaled-curve)
                     ((translate .5 .5)
                      ((rotate-around-origin (/ -pi 4)) scaled-curve)))))
```

Now we can generate approximations at any level to the Gosper curve by repeatedly gosperizing the unit line, using the `repeated` procedure of Problem Set 2.

```
(define (gosper-curve level)
    ((repeated gosperize level) unit-line))
```

To look at the level `level` gosper curve, evaluate `(show-connected-gosper level)`:

```
(define (show-connected-gosper level)
  ((draw-connected g1 200)
   ((squeeze-rectangular-portion -.5 1.5 -.5 1.5)
    (gosper-curve level))))
```

Pre-Lab exercise 5.A Define a procedure `show-points-gosper` such that evaluation of

```
(show-points-gosper window level number-of-points initial-curve)
```

will plot `number-of-points` unconnected points of the level `level` gosper curve in `window`, but starting the gosper-curve approximation with an arbitrary `initial-curve` rather than the unit line. For instance,

```
(show-points-gosper g1 level 200 unit-line)
```

should display the same points as `(show-connected-gosper level)`, but without connecting them. But you should also be able to use your procedure with arbitrary curves. (You can find the description of procedure `squeeze-rectangular-portion` in the file `curves.scm`; you don't need to understand it in detail to do this exercise.)

Lab exercise 5.B Try gosperizing the arc of the unit circle running from 0 to π. You will probably want to put the curve in standard position, as well as ensuring that the result is a curve (which means it should take as input a parameter from the unit line). Find some examples that produce interesting designs. (You may also want to change the scale in the plotting window and the density of points plotted.)[5]

The Gosper fractals we have been playing with have had the angle of rotation fixed at 45 degrees. This angle need not be fixed. It need not even be the same for every step of the process. Many interesting shapes can be created by changing the angle from step to step.

We can define a procedure `param-gosper` that generates Gosper curves with changing angles. `Param-gosper` takes a level number (the number of levels to repeat the process) and a second argument called `angle-at`. The procedure

[5] One of the things you should notice is that, for larger values of n, all of these curves look pretty much the same. As with many fractal curves, the shape of the Gosper curve is determined by the Gosper process itself, rather than the particular shape we use as a starting point. In a sense that can be made mathematically precise, the "infinite level" Gosper curve is a fixed point of the Gosper process, and repeated applications of the process will converge to this fixed point.

`angle-at` should take one argument, the level number, and return an angle (measured in radians) as its answer

$$\texttt{angle-at} : \text{Sch-Nonneg-Int} \to \text{Sch-Num}.$$

Procedure `param-gosper` can use this to calculate the angle to be used at each step of the recursion.

```
(define (param-gosper level angle-at)
  (if (= level 0)
      unit-line
      ((param-gosperize (angle-at level))
       (param-gosper (- level 1) angle-at))))
```

The procedure `param-gosperize` is almost like `gosperize`, except that it takes an another argument, the angle of rotation to use at each level:

```
(define (param-gosperize theta)
  (lambda (curve)
    (let ((scale-factor (/ (/ 1 (cos theta)) 2)))
      (let ((scaled-curve ((scale scale-factor) curve)))
        (connect-rigidly ((rotate-around-origin theta) scaled-curve)
                         ((translate .5 (* (sin theta) scale-factor))
                          ((rotate-around-origin (- theta)) scaled-curve)))))))
```

For example, the ordinary Gosper curve at level `level` is returned by

```
(param-gosper level (lambda (level) pi/4))
```

Pre-Lab exercise 6.A Designing `param-gosperize` required using some elementary trigonometry to figure out how to shift the pieces around so that they fit together after scaling and rotating. It's easier to program if we let the computer figure out how to do the shifting. Show how to redefine `param-gosperize` using the procedures `put-in-standard-position` and `connect-ends` from Exercise 4 to handle the trigonometry. Your definition should be of the form

```
(define (param-gosperize theta)
  (lambda (curve)
    (put-in-standard-position
      (connect-ends
        ...
        ...))))
```

Lab exercise 6.B Generate some parameterized Gosper curves where the angle changes with the level n. We suggest starting with $\pi/(n+2)$ and $\pi/(1.3^n)$. Submit sample printouts with your problem solutions.

Lab exercise 6.C We now have three procedures to compute gosper curves: `gosper-curve`, and `param-gosper` with argument `(lambda (level) (/ pi 4)))` using the "hand-crafted" definition of `param-gosperize` above, or using your version of `param-gosperize` in 7.A based on `put-in-standard-position`. Compare the speed of these procedures for computing selected points on the curve at a few levels. Is there a speed advantage for the more customized procedures?

The procedure `show-time` will report the time in milliseconds required to evaluate a procedure of no arguments (a "thunk"). For example, evaluating

```
(show-time (lambda () ((gosper-curve 10) .1)))
```

will print out the time to compute the point at .1 on the level 10 gosper-curve.

Pre-Lab exercise 7.A Ben Bitdiddle isn't entirely happy with the style of several of the Scheme definitions on this problem set. In particular, he feels the code goes overboard in inventing names for values that are used infrequently, and this lengthens the code and burdens someone reading the code with remembering the invented names. For example, Ben thinks the definition

```
(define (rotate-around-origin theta)
  (let ((cth (cos theta))
        (sth (sin theta)))
    (lambda (curve)
      (lambda (t)
        (let ((ct (curve t)))      ;Ben eliminates the declaration of ct
          (let ((x (x-of ct))
                (y (y-of ct)))
            (make-point
             (- (* cth x) (* sth y))
             (+ (* sth x) (* cth y)))))))))
```

would be a bit more readable if the name `ct` for the value of `(curve t)` was dropped. He proposes instead:

```
(define (bens-rotate theta)
  (let ((cth (cos theta))
        (sth (sin theta)))
    (lambda (curve)
      (lambda (t)
        (let ((x (x-of (curve t)))      ;Ben writes (curve t)
              (y (y-of (curve t))))     ;twice
          (make-point
           (- (* cth x) (* sth y))
           (+ (* sth x) (* cth y))))))))
```

Is Ben's definition correct?

Alyssa P. Hacker warns Ben that the `let` declarations are more significant computationally than mere abbreviations. Briefly explain why using `bens-rotate` as a subprocedure in place of the original `rotate-around-origin` in the definition of `gosper-curve` will turn a process whose time is linear in the level into one which is exponential in the level.

Lab exercise 7.B Look up the online documentation of the `trace-entry` procedure in the Scheme Users Manual. Trace `x-of` to show how dramatically Alyssa's warning is confirmed when computing points on the gosper curve using `bens-rotate` as a subprocedure in place of the original `rotate-around-origin`. Turn in a table summarizing the number of calls to `x-of` by `gosper-curve` using the two different rotating procedures at four or five illustrative levels. (A simple way to switch to use of `bens-rotate` in place of `rotate-around-origin` is to evaluate

```
(define rotate-around-origin bens-rotate)
```

Of course, you had better save the procedure `rotate-around-origin` under some other name so you can restore it. Otherwise, you may have to reload the problem set.)

We can now invent other schemes like the Gosper process, and use them to generate fractal curves.

Pre-Lab exercise 8.A The *Koch curve* is produced by a process similar to the Gosper curve, as shown in figure 2, and figure 3. Given a basic curve, we split the extent of the curve into three equal pieces, then draw scaled copies of the curve in the first and third piece. In the middle piece, we draw two scale copies of the initial curve, one rotated by 60 degrees, and the other rotated by -60 degrees, where all of the curves are drawn so that their endpoints meet.

Write a procedure `kochize` that generates Koch curves.

(Teaser: You can generate the Koch curve by using `param-gosper` with an appropriate argument. Can you find this?)

Lab exercise 8.B Print some pictures of your Koch curve at various levels.

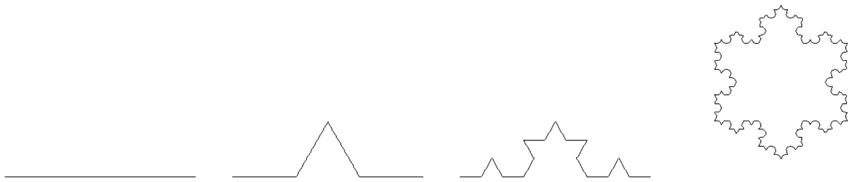

Figure 2: The Koch at levels 0, 1 and 2. Also, at the right is a "snowflake" curve formed by connecting together three rotated versions of a Koch curve.

Figure 3: To generate the next level of curve, split the current level into three equal pieces. The first and third contain scaled copies of the current curve. In the middle piece, place two scaled and rotated (by 60 and -60 degrees) copies of the current level curve.

Lab Exercise 9 You now have a lot of elements to work with: scaling, rotation, translation, curve plotting, Gosper processes, Koch processes, and generalizations. For example, you can easily generalize the parameterized Gosper process to start with something other than an horizontal line. The Gosper curve is continuous but *nowhere differentiable*, so it may be interesting to display its derivatives at various levels and numbers of points (see the procedure `deriv-t` in the file `curves.scm`). Or you can create new fractal processes. Or you can combine the results of different processes into one picture. Spend some time playing with these ideas to see what you can come up with.

We hope you will have generated some interesting designs in doing this assignment. Please turn in at least one example of a new design that you have created, and show the expression you used to generate it. Feel free to create entirely new kinds of fractals if you wish. In order to minimize demand on the graphics printers *do not turn in more than four pictures.*

See Next Page

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

How much time did you spend on this homework assignment?

Indicate all of the following facilities you in doing this problem set:

- The 6.001 lab in building 36.
- MIT Scheme on Project Athena.
- MIT Scheme on a Windows PC.
- EdScheme on a PC.
- Some other Scheme system (which one?).

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 course text (such as some of the course archives), please indicate your consultants' names and your references in the space below. 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 course reference materials." and sign your statement.