Structure and Interpretation of Computer Programs
Second Edition
SAMPLE

Graphing Problem Set

Graphing with Higher-order Procedures

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 exercises included. Section 3, the actual programming assignment, describes the graphics language and gives applications to generating fractal designs. There is also an optional design problem.

1. Procedure Types and Procedure Constructors

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 notation:

`square` : Sch-Num → Sch-Num

If `f` and `g` are procedures of type Sch-Num → Sch-Num, then we may *compose* them:

```
(define (compose f g)
  (lambda (x)
    (f (g x))))
```

Thus, for example `(compose square log)` is the procedure of type Sch-Num → Sch-Num that returns the square of the logarithm of its argument, while `(compose log square)` returns the logarithm of the square of its argument:

```
(log 2)
;Value: .6931471805599453

((compose square log) 2)
;Value: .4804530139182014

((compose log square) 2)
;Value: 1.3862943611198906
```

As we have used it above, the procedure `compose` takes as arguments two procedures of type F = Sch-Num $\rightarrow$ Sch-Num, and returns another such procedure. We indicate this with the notation:

$$\texttt{compose} : (F, F) \rightarrow F$$

Just as squaring a number multiplies the number by itself, `thrice` of a function composes the function three times. That is, `((thrice f) n)` will return the same number as `(f(f(f n)))`:

```
(define (thrice f)
  (compose (compose f f) f))

((thrice square) 3)
;Value: 6561

(square (square (square 3)))
;Value: 6561
```

As used above, `thrice` is of type $(F \rightarrow F)$. That is, it takes as input a function from numbers to numbers and returns the same kind of function. But `thrice` will actually work for other kinds of input functions. It is enough for the input function to have a type of the form $T \rightarrow T$, where T may be any type. So more generally, we can write

$$\texttt{thrice} : (T \rightarrow T) \rightarrow (T \rightarrow T)$$

Composition, like multiplication, may be iterated. Consider the following:

```
(define (identity x) x)

(define (repeated f n)
  (if (= n 0)
      identity
      (compose f (repeated f (- n 1)))))

((repeated sin 5) 3.1)
;Value: 4.1532801333692235e-2

(sin(sin(sin(sin(sin 3.1)))))
;Value: 4.1532801333692235e-2
```

$$\texttt{repeated} : ((T \rightarrow T), \text{Sch-Nonneg-Int}) \rightarrow (T \rightarrow T)$$

Exercise 1.A The type of `thrice` is of the form $(T' \rightarrow T')$ (where T' happens to equal $(T \rightarrow T)$), so we can legitimately use `thrice` as an input to `thrice`!

For what value of `n` will `(((thrice thrice) f) 0)` return the same value[1] as `((repeated f n) 0)`?

See if you can now predict what will happen when the following expressions are evaluated. Briefly explain what goes on in each case.

[1] "Sameness" of procedure values is a sticky issue which we don't want to get into here. We can avoid it by assuming that `f` is bound to a value of type F, so evaluation of `(((thrice thrice) f) 0)` will return a number.

1. `(((thrice thrice) 1+) 6)`
2. `(((thrice thrice) identity) compose)`
3. `(((thrice thrice) square) 1)`
4. `(((thrice thrice) square) 2)`.

Exercise 1.B Test your predictions. (**Warning**: Before you do this, make sure you understand how to *interrupt* a Scheme evaluation.)

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. We require only that the constructors and selectors obey the rules

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

for all Sch-Num's m, n. Here is one way to do this (we'll learn several other, better ways, in two weeks.)

```
(define (make-point x y)
  (lambda (bit)
    (if (zero? bit) x y)))

(define (x-of point)
  (point 0))

(define (y-of point)
  (point 1))
```

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

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))
```

Exercise 2:

1. What is the type of `unit-line-at`?
2. Define a procedure `vertical-line` with two arguments, a point and a length, and returns a vertical line of that length beginning at the point.
3. What is the type of `vertical-line`?

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

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

Exercise 3: Write a definition of a Curve-Transform `reflect-through-y-axis`, which turns a curve into its mirror image.

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.

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 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 rigidly translating it so its starting point is at the origin, then rotating it about the origin to put its endpoint on the x axis, then 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,

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

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)))))
```

Exercise 4: There is another, possibly more natural, way of connecting curves. The curve returned by `(connect-ends curve1 curve2)` consists of a copy of `curve1` follwed 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`.

3. Drawing Curves

Use `M-x load-problem-set` to load the code for problem set 2. 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`.

Exercise 5: 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.

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 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 **??** 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. The figure also illustrates 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

[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 various levels, and the recursive transformation that produces each level from the previous level.

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,

```
(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))))
```

Exercise 6.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.)

Exercise 6.B Try gosperizing the arc of the unit circle running from 0 to π. 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 `angle-at` should take one argument, the level number, and return an angle (measured in radians) as its answer

`angle-at` : Sch-Nonneg-Int → 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, and implements the process shown in figure **??**:

```
(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))
```

[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.

Figure 2: A parameterized version of the Gosper process, where the angle can vary. Note that the endpoints of the transformed figure should be the same as the endpoints of the original figure, and the interior endpoints should match.

Exercise 7.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
        ...
        ...))))
```

Exercise 7.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.

Exercise 7.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-positio`
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.

Exercise 8.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.

Exercise 8.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.

Exercise 9.A The *Koch curve* is produced by a process similar to the Gosper curve, as shown in figure **??**. 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?)

Figure 3: The Koch at various levels, and a "snowflake" curve formed from three Koch curves. As with the C curve, each level approximation to the Koch curve is obtained by applying the a transformation to the previous level.

Exercise 9.B Print some pictures of your Koch curve at various levels.

Exercise 10 (Optional) 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.