KOÇ UNIVERSITY
Department of Computer Engineering
Comp200 Structure and Interpretation of Computer Programs
Graphing with Higher-order Procedures

Introduction

- Read this handout file very carefully.
- code files to read: curve-utils.scm, curves.scm, points_segs.scm, curves_setup_racket.scm
- code file simply to load: drawing_racket.scm
- Write your solutions to only project2.scm file, don't touch other source code files.
- Don't ever change procedure prototypes.
- Please use DrRacket with R5RS language. In order to use R5RS language, click *Languages* on the top menu, click on *Choose Language* and then select R5RS language from *Other Languages* section. After that, click on *Show Details*, uncheck *Disallow redefinition of initial bindings* and you will be finally able to complete this assignment in DrRacket environment.
- Only the error-free projects will be graded.
- Upload your project to DriveF@VOL/COURSES/UGRADS/COMP200/HOMEWORK/[username]/project2/
- Name your submission file as project2.scm. If you want to submit a new version of your work, submit it as project2_X.scm where X is the submission number. For instance, if you want to submit your work for the second time, the name of the file should be project2_2.scm
- Only the projects uploaded to the correct F-drive path with the correct filename will be graded.

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 project will give you extensive practice with higher-order procedures, in the context of a language for graphing two-dimensional curves and other shapes. The project comes in three parts. The first gives you some experience in writing procedures and manipulating simple data structures. We recommend that you get started on this part as soon as you can. The second part deals with types of procedures, and is intended to help you understand different kinds of higher order procedures. This leads directly to the third part, where you will write some higher order procedures to graph interesting curves.

Part 1: Simple Procedures for Planar Geometry

Let's get some practice in writing procedures and manipulating data structures by considering some simple procedures for dealing with polygons in the plane. A polygon is an ordered sequence

of line segments, where each line segment is represented by a start and end point (and implicitly all the points along the line between these two points). Of course, it is not just a random set of line segments, but rather a continuous one, in which the end point of one segment becomes the start point for the next, and in which the end point of the last segment is the start point of the first one.

The file points_segs.scm contains some simple definitions for constructing points (with an x and y coordinate) and segments, using Scheme's basic constructor list. You should load this file into your Scheme so you can use these definitions and read it before attempting the exercises below.

Exercise 1:

Given a segment, we might want to know its length. Write a procedure segment-length that computes the length of a segment. Be sure to use the correct selectors for the data abstractions for segments and for points. Show some examples of your procedure in operation.

Exercise 2:

Now suppose you have a set of segments representing a polygon, and you want to compute the perimeter of the polygon. Write a recursive procedure called perimeter, using segment-length as a component, that takes as input a list of segments, and returns the sum of the lengths of the segments. Demonstrate this procedure working on test-segments.

Exercise 3:

In the previous exercise, we provided you with a pre-made list of segments. More generally, however, you will want to create a polygon as a list of segments, by connecting points together. Suppose you are given an ordered list of points, e.g., test-points, and you want to convert this to an ordered list of segments. To do so, you want to make a segment out of the first and second point, a segment out of the second and third point and so on. But you also need to make a segment out of the last and first point, to completely connect the polygon. One easy way to do this is to create a new list, in which the first point of the original list is added at the end of the list (think about using append), and then recursively creating segments out of pairs of points. Write a procedure to do this. Compare its performance when applied to test-points, to the predefined test-segments.

Exercise 4:

A useful characteristic of a polygon is its centroid, which is defined as the average of the points of the polygon. We can compute this in a variety of ways. First, let's think about adding two points together (that is, treating them as vectors). Write a procedure add-two-points that takes as input two points, and returns a new point, whose coordinates are the sums of the coordinates of the input points. Be sure to use appropriate constructors and selectors. Show your procedure working on some test cases.

Next, generalize this to add up a list of points, by writing a procedure add-all-points that uses add-two-points recursively to add up all the points.

Finally, use this procedure to write a procedure for computing the centroid of a list of points. Be sure to use the right constructor in generating the return value. Demonstrate this procedure applied to test-points.

Exercise 5:

An alternative way of computing the centroid is to first construct a list of all the x-coordinates, then add them up and scale by the length of the list; do the same for the y-coordinates; and then construct the centroid. Implement this variation on computing the centroid, and demonstrate that it finds the same answer as the previous version.

Part 2. 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 Number, the Scheme numbers such as 3, -4.2, 6.931479453e89, and Boolean, the truth values #t, #f. The procedure square may be applied to a Number and will return another Number. We indicate this with the type notation:

square: Number → Number

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

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

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

Exercise 6:

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 Number→Number and returns another procedure of that type:

square-a-procedure: (Number→Number) → (Number→Number)

or, abbreviating Number→Number as F:

square-a-procedure: F → 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)))))
```

Note the syntactic sugar being used here. This is equivalent to

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

so that there is a hidden lambda that is creating a procedure to associate with the name "make-procedure-op" (see the textbook for more discussion on this).

Thus, we now can say

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

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

make-procedure-op: F → (F → F)

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

− : Number, Number → Number

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)))
```

square-difference: F, F $\rightarrow$ F

Exercise 7:

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)))
```

Exercise 8:

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))
```

Exercise 9:

What is the type of `square-an-operator-on-proc`?

Of course, we can generalize these ideas in other ways. For example, if f and g are procedures of type: Number$\rightarrow$ Number, we can compose them:

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

Thus, for example, (compose square log) is the procedure of type Number→Number 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: .693147805599453

((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=Number→Number, and returns another such procedure. We indicate this with the notation:

compose:(F,F)→F

Just as square 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→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→T, where T may be any type. So more generally, we can write

thrice: (T→T) → (T→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
```

Exercise 10:

For what value of n will `(((thrice thrice) f) 0)` return the same value as `((repeated f n) 0)` (ignore issues of equality by assuming that f has a value of type F so that the whole expression will return a number)?

Part 3. 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 this course 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 the point $(\sin 2\pi t, \cos 2\pi t)$ where t ranges over the unit interval [0,1]. To visualize this representation, you can think of a runner going around the unit-circle in one hour, and our function returns his position when we specify the time t between 0 and 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

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

and Point is some representation of pairs of Scheme-Numbers.

To work with Point, we need a constructor, make-point, which constructs Point's from Number'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 for all Number's m,n. In this problem set, we'll use:

```
(define (make-point x y)
  (list x y))

(define (x-of point)
```

```
  (car point))

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

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!

Exercise 11:

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 (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?
4. Show some examples that test your procedure, and that show it returns the correct value.

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) $\rightarrow$ (-y, x) rotates the plane by 90 degrees, 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

Curve-Transform = Curve → Curve

where Curve = Unit-Interval → Point.

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.

Exercise 12:

Write a definition of a Curve-Transform upside-down, which flips a curve over a horizontal line through its start point.

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 (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,

Binary-Transform = (Curve, Curve) → 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 13:

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.

Part 4. Drawing Curves

We are now ready to start drawing curves and building complex curves. To do this, you should load the following files:

curves-utils.scm
curves.scm
drawing.scm
curves-setup.scm

and then evaluate (setup-windows). 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.

What this means is that each drawing procedure will take as argument a drawing window (one of g1, g2 and g3) and a specified number of points. It will return a procedure that can then be applied to a curve to draw that curve in the window. We have 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 14:

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. Describe the difference in the two displays. Also try using draw-points-squeezed-to-window. Briefly explain why the unit circles printed using unit-circle and alternative-unit-circle differ.

(Note: when one of your graphics windows gets covered, its contents may get erased. If that happens you need to bring the window forward and issue the drawing command once more.)

You may want to save example curves. To do this, if you are using a Windows machine, then hit Alt PrtSc (or whatever key is labeled Print Screen). This will grab a copy of the window. Now open a new page in Paint and Ctrl-V. This will insert a copy of the image into page, then save that page with an appropriate name (for instance "Exercise14Curve") as png or jpg file.

Alternatively, you can use Snipping Tool to capture image of a window.

For other operating systems, you can find similar short-cuts and tools. If you need help, do not hesitate to ask help from Tutors.

Upload curve image files when you are submitting your solution.

Part 5. Fractal Curves

To show off the power of our drawing language, let's use it to explore fractal curves. Fractals have striking mathematical properties. 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. 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 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.

The 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. 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 45 degrees and the other is rotated by 45 degrees. 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.

```
(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 15:

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 16:

Try gosperizing the arc of the unit circle running from 0 to pi. 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.) (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.)

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.

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))
```

Exercise 17:

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
        ...
        ...))))
```

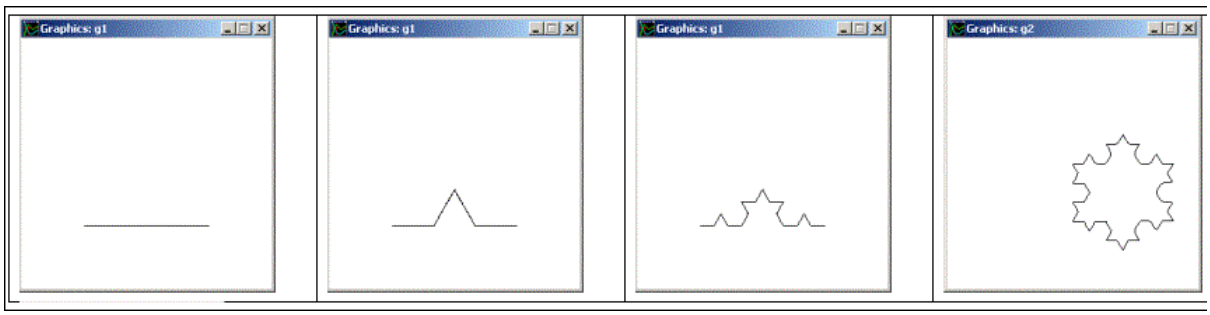
Exercise 18:

Generate some parameterized Gosper curves where the angle changes with the level n. We suggest starting with pi/(n+2) and pi/$(1.3)^n$. Save sample images and upload with your solutions.

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

Exercise 19:

The Koch curve is produced by a process similar to the Gosper curve. The first few levels of a Koch curve are shown in the figure, illustrating level 0, 1 and 2, and a "snowflake" curve formed by connecting together three rotated versions of a Koch curve.



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?)

Exercise 20:

Save some pictures of your Koch curve at various levels (at least 3). Upload your Koch curve images. You can give name to Koch images with suffixes that define level and angle, for example "Koch_1_-30".

Exercise 21:

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. Save graphics that you have created and upload them with your submission.

We encourage you to work with others on problem sets as long as you acknowledge it (nothing written gets exchanged!).

If you cooperated with other students, TA's, or others, please indicate your consultants' names. Otherwise, write "I worked alone using only the course reference materials."