MP 11 – Combinator-based drawing library

CS 421 – Spring 2013
Revision 1.0

Assigned Thursday, April 11, 2013
Due Tuesday, April 16, 09:30
Extension 48 hours (20% penalty)

1 Change Log

0.1 Preliminary Release.

2 Objective

You will fill in the missing parts of a set of combinators for creating line drawings. Using these combinators, you will be able to write functions that will produce a set of drawing commands. Examples are given below. This assignment was inspired by a paper entitled "A special-purpose language for picture-drawing," by David Hyatt and Sam Kamin, which you can find at `loome.cs.illinois.edu/pubs.html` (about two-thirds down the page); the version here is much simpler, but you might find that interesting.

After completing this MP, you should have a better idea both of how to use higher-order functions, and why they might be useful.

In terms of lines of code, this will be the shortest MP since MP1. However, that count may be deceiving; higher-order functions can allow for programs that are concise but difficult to understand. In this case, we have included in the skeleton file a count of lines in our solution *of each part* of the problem, some as short as a single line.

If you find this MP interesting, we suggest in the last section of the assignment various ways you might want to build on it.

3 Collaboration

Collaboration is NOT allowed in this assignment.

4 What to submit

The handin process is not complete at this time. We assume it will be essentially the same as in the past. For now, we are distributing only the skeleton file, which includes some examples with their correct output. We believe this is enough information to do the MP, but we will release an actual tarball when it is ready.

5 Overall concept

The idea of this MP is to define functions allowing you to build values of type `picture`. The set of functions we have defined in the solution — some of these are included in the skeleton, some are omitted, and some are given only partially — is as follows:

- Point operations. Points are just pairs of floats. There are just three operations: component-wise addition (`++`), subtraction (`--`) and multiplication (`***`). These are infix operators.
- Primitive picture-building functions:

  - `emptypic`: `picture`
  - `line`: `point` → `point` → `picture`
  - `oval`: `point` → `point` → `point` → `picture`

- Picture-combining operations:
  - `&&`: `picture` → `picture` → `picture` (Draw both pictures. Infix.)
  - `join_list`: `picture list` → `picture` (Draw all the pictures.)

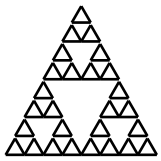
- Transformations. A transformation is just a function of type `point` → `point`. (These operations are not normally called by users of the library, but rather are used in the implementation of picture-transforming operations; you will be acting as both implementers and users, so you need to know these.)
  - `id_trans`: `transformation`. (identity transformation.)
  - `compose`: `transformation` → `transformation` → `transformation`
  - `matrix_transform`: `matrix` → `transformation` (Defines a transformation from a 2×3 matrix, in a standard way; see `http://en.wikipedia.org/wiki/ Homogeneous_coordinates #-Use_in_computer_graphics`; you don't need to know the details anyway.)

- Picture-transforming operations:
  - `scaleWithPoint`: `picture` → `point` → `point` → `picture` (Scales the picture using scaling factors given by the first point, while leaving the second point at its original location.)
  - `rotateAroundPoint`: `picture` → `float` → `point` → `picture` (Rotates picture by amount given in `float` argument, in degrees.)
  - `translate`: `picture` → `point` → `picture` (Moves the picture by the amount given in the second argument.)

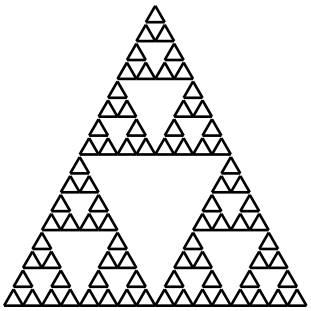
- Output operations. (There is just the function `draw`, which produces output in the style of LaTeX `picture` environment, allowing the pictures to be included in this assignment; feel free to add your own. Also, we have implemented only line-drawing; this is because we're tired.)
  - `draw`: `draw_cmd list` → `unit` (Just prints the list of drawing commands to the console, so they can be copied and pasted into a LaTeX document.)

With these operations, you can write functions to construct complex line drawings. These two examples are given in the skeleton file:

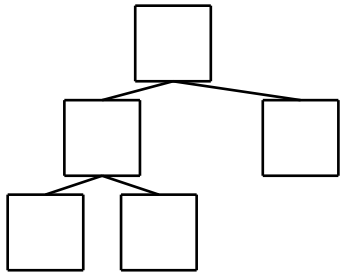
```
let unit_triangle = triangle origin (1.0,0.0) (0.5,0.866)

let rec sierpinski n : picture =
    if n=0
    then unit_triangle
    else let s = scaleWithPoint (sierpinski (n-1)) (0.5,0.5) origin
         in join_list [s; translate s (0.5,0.0); translate s (0.25,0.5)]

sierpinski 2
```



`sierpinski 3`



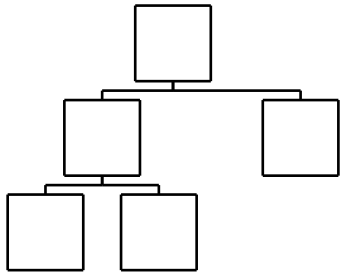
```
let node (pic:picture) (w0:float) (h0:float)
         ((p1,w1,h1):tree) ((p2,w2,h2):tree)
         (lf:linefun)
   = ...
```



```
let box1 = box (0.0,0.0) (1.0,0.0) (1.0,1.0);;
let leaf1 = (box1, 1.0, 1.0);;
let t1 = node box1 1.0 1.0 leaf1 leaf1 line;;
let t2 = node box1 1.0 1.0 t1 leaf1 line;;
let (p,_,_) = t2 in draw p;;
```

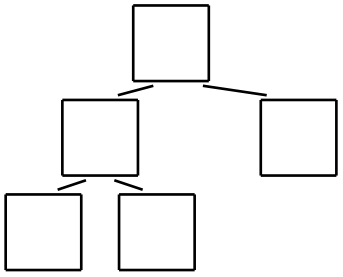
```
let manline ((x0,y0) as pt0:point) ((x1,y1) as pt1:point) : picture
   = let ymid = (y0 +. y1) /. 2.0
     in let corner1 = (x0, ymid)
        and corner2 = (x1, ymid)
        in line pt0 corner1 && line corner1 corner2 && line corner2 pt1
```

```
let box1 = box (0.0,0.0) (1.0,0.0) (1.0,1.0);;
let leaf1 = (box1, 1.0, 1.0);;
let t1 = node box1 1.0 1.0 leaf1 leaf1 manline;;
let t2 = node box1 1.0 1.0 t1 leaf1 manline;;
let (p,_,_) = t2 in draw p;;
```

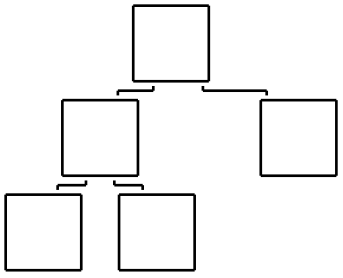


```
let shorten (lf:linefun) : linefun
  = ...
```

above code with `(shorten line)` *instead of* `line`



above code with `(shorten manline)`



6 Problems

The problems are simply the areas in the skeleton file that are marked for you to fill in. The test cases create lists of drawing commands which will be compared to the lists of drawing commands that we produce. If you wish to actually see the pictures (which can be a big help in debugging), you can use the `draw` command and the LaTeX `picture` environment, as follows:

```
\usepackage{pict2e}
\usepackage{picture}
 ...
\begin{picture}(50,50)(0,70)
\setlength{\unitlength}{1cm}
\linethickness{1pt}
 ...  copy and paste commands from draw function ...
\end{picture}
```

You will have to play with the numbers here — (50,50) is the size of the "canvas," (0,70) the offset (actually, the reverse of the offset) of the origin, and 1cm is the unit for drawing — if you want the drawing to appear at a particular place and a particular size — or, sometimes, just to have it show up on the output.

We give an explanation here of what you are to do; you will also want to read the code in the skeleton file.

The main definitions for this assignment are:

```
type point = float * float

type transformation = point -> point

type draw_cmd = Pixel of point
              | Line of point * point
              | Oval of point * point * point

type picture = transformation -> draw_cmd list
```

(We are only using the `Line` command in this assignment.)

The idea is that `draw_cmd` gives the primitive drawing operations supported by some output device. However, putting pictures together involves transforming them — moving, resizing, etc. — so that they fit. That is why we consider a picture to be a function from transformations to drawing commands.

Users will not want to manipulate transformations and drawing commands directly. Instead, they will use this line-drawing function:

```
let line (pt1:point) (pt2:point) : picture
     = fun phi -> [Line(phi pt1, phi pt2)]
```

The individual problems should be clear, and mostly only require one or two lines of code. The hardest one is problem 6, where we ask you to rewrite a function we have defined, using higher-order functions in the List module. The idea is that the function we have provided has a number of recursively-defined auxiliary functions; you are to define those functions non-recursively, turning every "let rec" into a "let." The most difficult cases are functions `xlocs` and `toplocs`; we provide two hints: For `xlocs`, you will need to reverse the argument list (function `rev` is in the List module, so you are allowed to use it), and then reverse the resulting list back; for `toplocs`, use the result of the call to `xlocs`.

7 Enhancements

The paper referred to above has a number of enhancements, including things like adding circles and arrows and color. It also included a point-naming facility that simplifies drawing functions substantially. Most usefully, it had text,

which could be transformed the same way as other pictures (using a package called `pstricks` and some fancy TEX hacking).

Having other output functions than `draw` would be nice. It would also be nice to have a more convenient way to produce pictures than by copying and pasting.

Finally, it might be fun to play with non-linear transformations. To give one example, this code:

```
open Random;;
let random_trans (d:float) (pic:picture) =
   let pttrans ((x,y):point) : point =
            let r = Random.float (2.0*.d) in (x+.(d-.r), y+.(d-.r))
   in fun phi -> pic (compose phi pttrans);;
draw (random_trans 0.01 (sierpinski 3));;
```

produces this picture: