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

Lecture Notes—September 24th.

Henderson Picture Language

Today's lecture will use the Henderson Picture Language as an example of how we can merge together the themes of data abstraction, higher order procedures and procedural abstractions to create a new language for describing pictures. You will be dealing with this language in Problem Set 3, and much of the background for the language is discussed in Section 2.2.4 of the text.

One of the key features of the implementation is that it represents figures as procedures.

In particular, a "picture" is defined to eb a procedure that takes a rectangle as input and causes something to be drawn, scaled to fit in the rectangle. A rectangle will be represented by three vectors: an origin, a "horizontal" vector, and a "vertical" vector. The origin **o** will be represented as a vector whose coordinates give the coordinates of the origin with respect to some external coordinate system, such as the graphics screen coordinates. The "horizontal" vector, (call it **h**) and the "vertical" vector (call it **v**) give the offsets of the sides of the rectangle from the origin. As a consequence, any rectangle defines a linear transformation coordinate map by mapping the point $(1, 0)$ in its coordinate frame into the point specified by the end of the **horizontal** vector, offset from the origin, and by mapping the point $(0, 1)$ in its coordinate frame into the point specified by the end of the **vertical** vector, offset from the origin.

In algebraic terms, a general point (x, y) gets mapped to the vector

$$\mathbf{o} + x \times \mathbf{h} + y \times \mathbf{v}$$

where **o**, **h**, and **v** are vectors, and x and y are scalars.

Below is the code we will be using during this lecture to discuss this language.

Basic abstractions

First, we define an abstraction for vectors, and primitive operations on vectors:

```
(define make-vect cons)
(define xcor car)
(define ycor cdr)
```

Here is one simple procedure for manipulating vectors

```
(define (+vect v1 v2)
  (make-vect (+ (xcor v1) (xcor v2))
             (+ (ycor v1) (ycor v2))))
```

Related procedures include `-vect` for subtracting vectors, and `scale` for scaling a vector by some amount.

```
(define (scale s vect)
  (make-vect (* s (xcor vect))
             (* s (ycor vect))))
```

We make line segments out of vectors:

```
(define make-segment cons)
(define seg-start car)
(define seg-end cdr)
```

We also make rectangles out of vectors, consisting of an origin, and a vector from that origing to the end of the "horizontal" axis, and one to the end of the "vertical" axis.

```
(define make-rectangle list)
(define origin car)
(define horiz cadr)
(define vert caddr)
```

Using this map, points with coordinates between 0 and 1 end up inside the rectangle, essentially in a new coordinate system. Here is a procedure which takes a rectangle as input and returns the corresponding coordinate transform which is itself a procedure:

```
(define (coord-map rect)
  (lambda (point)
    (+vect
      (+vect (scale (xcor point)
                    (horiz rect))
             (scale (ycor point)
                    (vert rect)))
      (origin rect))))
```

Finally, we define pictures and some primitives on them:

```
(define (make-picture seglist)
  (lambda (rect)
    (for-each
       (lambda (segment)
         (drawline ((coord-map rect) (seg-start segment))
                   ((coord-map rect) (seg-end segment))))
       seglist)))
```

Note the form of this procedure – it takes a list of segments as input, and returns a procedure – when this procedure is applied to a rectangle, it will draw the segments inside that rectangle. Thus a picture is actually a procedure.

`drawline` draws a line in screen coordinates from one point to another.

Now we can use these ideas:

```
(define empty-picture (make-picture '()))

(define outline-picture
  (make-picture
    (list (make-segment
            (make-vect 0 0)
            (make-vect 0 1))
          (make-segment
            (make-vect 0 1)
            (make-vect 1 1))
          (make-segment
            (make-vect 1 1)
            (make-vect 1 0))
          (make-segment
            (make-vect 1 0)
            (make-vect 0 0)))))

(define (prim-pict list-of-lines)
  (make-picture
   (map
     (lambda (line)
        (make-segment
           (make-vect (car line) (cadr line))
           (make-vect (caddr line) (cadddr line))))
     list-of-lines)))
```

Now we can make a particular picture (george):

```
(define g (prim-pict (list (list .25 0 .35 .5)
                           (list .35 .5 .3 .6)
                           (list .3 .6 .15 .4)
                           (list .15 .4 0 .65)
                           (list .4 0 .5 .3)
                           (list .5 .3 .6 0)
                           (list .75 0 .6 .45)
                           (list .6 .45 1 .15)
                           (list 1 .35 .75 .65)
                           (list .75 .65 .6 .65)
                           (list .6 .65 .65 .85)
                           (list .65 .85 .6 1)
                           (list .4 1 .35 .85)
                           (list .35 .85 .4 .65)
                           (list .4 .65 .3 .65)
                           (list .3 .65 .15 .6)
                           (list .15 .6 0 .85))))
```

Rotations

To rotate a picture 90 degrees counterclockwise, we need only draw the picture with respect to the new rectangle:

```
(define (rotate90 pict)
  (lambda (rect)
    (pict (make-rectangle
            (+vect (origin rect)
                   (horiz rect))
            (vert rect)
            (scale -1 (horiz rect))))))

(define (repeated function n)
  (lambda (thing)
    (if (= n 0)
        thing
        ((repeated function (- n 1)) (function thing)))))

(define rotate180 (repeated rotate90 2))
(define rotate270 (repeated rotate90 3))
```

Horizontal flip is also drawing with respect to a new rectangle:

```
(define (flip pict)
  (lambda (rect)
    (pict (make-rectangle (+vect (origin rect) (horiz rect))
                          (scale -1 (horiz rect))
                          (vert rect)))))
```

Means of combining pictures

Since pictures are specified by procedures, we can create higher order procedures that combine simple pictures together in various ways. For example, the `together` operation takes two procedures and combines them into a single picture, lying superimposed.

```
(define (together pict1 pict2)
   (lambda (rect)
      (pict1 rect)
      (pict2 rect)))
```

The `beside` operation takes two pictures and scales them according to some relative width to fit in a single rectangle. `beside` takes as input the two pictures plus a number, `a`, which specifies the proportion (between 0 and 1) of horizontal devoted to the first picture.

```
(define (beside pict1 pict2 a)
  (lambda (rect)
    (pict1 (make-rectangle
             (origin rect)
             (scale a (horiz rect))
             (vert rect)))
    (pict2 (make-rectangle
             (+vect (origin rect)
                    (scale a (horiz rect)))
             (scale (- 1 a) (horiz rect))
             (vert rect)))))
```

`above` is defined in terms of `beside`.

```
(define (above pict1 pict2 a)
  (rotate270 (beside (rotate90 pict1)
                     (rotate90 pict2)
                     a)))
```

Here are some operations defined in terms of the basic ones above:

```
(define (4pict pict1 rot1 pict2 rot2 pict3 rot3 pict4 rot4)
  (beside (above ((repeated rotate90 rot1) pict1)
                 ((repeated rotate90 rot2) pict2)
                 .5)
          (above ((repeated rotate90 rot3) pict3)
                 ((repeated rotate90 rot4) pict4)
                 .5)
          .5))

(define (4same pict rot1 rot2 rot3 rot4)
  (4pict pict rot1 pict rot2 pict rot3 pict rot4))

(define (up-push pict n)
  (if (= n 0)
      pict
      (above (up-push pict (- n 1))
             pict
             .25)))

(define (right-push pict n)
  (if (= n 0)
      pict
      (beside (right-push pict (- n 1))
              pict
              .25)))

(define (corner-push pict n)
  (if (= n 0)
      pict
      (above (beside (up-push pict n)
                     (corner-push pict (- n 1))
                     .75)
             (beside pict
                     (right-push pict (- n 1))
                     .75)
             .25)))

(define (square-limit pict n)
  (4same (corner-push pict n) 1 2 0 3))
```