MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996

Lecture Notes – September 26, 1996

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 4, and much of the background for the language is discussed in Section 2.2.4 of the text.

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

We layer the implementation of the picture language much as in the code we have used in Problem Sets 1, 2 and 3.

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))))
```

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

```
(define (rotate-vect v angle)
  (let ((c (cos angle))
        (s (sin angle)))
    (make-vect (- (* c (xcor v)) (* s (ycor v)))
               (+ (* c (ycor v)) (* s (xcor v))))))
```

We make lines out of vectors:

```
(define make-line list)
(define beg car)
(define end cadr)
```

We also make rectangles out of vectors:

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

Finally, we define pictures and some primitives on them:

```
(define (make-picture seglist)
  (lambda (rect)
    (for-each
       (lambda (segment)
         (let ((b (beg segment))
               (e (end segment)))
           (draw-line rect
                      (xcor b)
                      (ycor b)
                      (xcor e)
                      (ycor e))))
       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.

We haven't specified `draw-line`, but for our purposes here, we assume that the list of segments specifies points in the unit square, and that `draw-line` draws those segments scaled to fit within the specified rectangle.

Now we can use these ideas:

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

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

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

Now we can make a particular picture:

```
(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))))
(define big-bro
  (beside
    g
    (above empty-picture
           g
           .5)
    .5))

(define acrobats
   (beside g
           (rotate180 (flip g))
           .5))

(define 4bats
   (above acrobats
          (flip acrobats)
          .5))
```

Here are some operations on pictures, others are very similarly defined:

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

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

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

(define (beside pict1 pict2 a)
  (lambda (rect)
    (pict1 (make-rectangle
             (origin rect)
             (scale-vect (horiz rect) a)
             (vert rect)))
    (pict2 (make-rectangle
             (+vect (origin rect)
                    (scale-vect (horiz rect) a))
             (scale-vect (horiz rect) (- 1 a))
             (vert rect)))))
(define (above pict1 pict2 a)
  (rotate270 (beside (rotate90 pict1)
                     (rotate90 pict2)
                     a)))
```

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

(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 (rightp-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))
```