A picture language

Demonstrates

- power of procedures to represent data objects
- power of data abstraction
- power of higher-order procedures

Primitive elements

- **frame** - a parallelogram -shaped region in DrScheme's graphics window
- **painter** - a procedure that draws a particular pattern in a frame

```
(flag square-frame)
(flag rectangle-frame)
(flag rhomboid-frame)
```

Compound expressions

We'll provide some higher-order procedures that combine painters to make new painters

```
(flip-vert painter1)
(beside painter1 painter2)
(below painter1 painter2)
```

Abstraction

We'll use Scheme's **`define`** to give names to the painters created by compound expressions

```
(define flag2 (beside flag
                      (flip-vert
                        flag)))
(define flag4 (below flag2 flag2))
```

More operators for making compound expressions

```
(define (flipped-pairs painter)
  (let ((painter2 (beside painter
                          (flipped-vert
                            painter))))
    (below painter2 painter2)))
```

`(flipped-pairs flag)` paints the same as `flag4`

Right split

```
(define (right-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (right-split painter
                                  (- n 1))))
        (beside painter
                (below smaller smaller)))))
```

Corner split

```
(define (corner-split painter n)
  (if (= n 0)
      painter
      (let ((up (up-split painter (- n 1)))
            (right (right-split painter
                                (- n 1))))
```

(continued)

```
        (let ((top-left (beside up up))
              (bottom-right (below right
                                   right))
              (corner (corner-split
                        painter
                       (- n 1))))
          (beside (below painter top-left)
                  (below bottom-right
                         corner))))))
```

Square limit

```
(define (square-limit painter n)
  (let ((quarter (corner-split painter n)))
    (let ((half (beside (flip-horiz quarter)
                        quarter)))
      (below (flip-vert half) half))))
```

Meta-operators

- In the picture language, operators are procedures that take painters as arguments and return painters as values
- Meta-operators are procedures that take operators as arguments and return operators as values

Meta-operator example

```
(define (square-of-four top-left top-right
            bottom-left bottom-right)
  (lambda (painter)
    (let ((top (beside (top-left painter)
                       (top-right painter)))
          (bottom (beside (bottom-left
                           painter)
                          (bottom-right
                           painter))))
      (below bottom top))))
```

Using **`square-of-four`**

```
(define (flipped-pairs painter)
  ((square-of-four identity
                   flip-vert
                   identity
                   flip-vert)  painter))
```

or

```
(define flipped-pairs
  (square-of-four identity
                  flip-vert
                  identity
                  flip-vert))
```

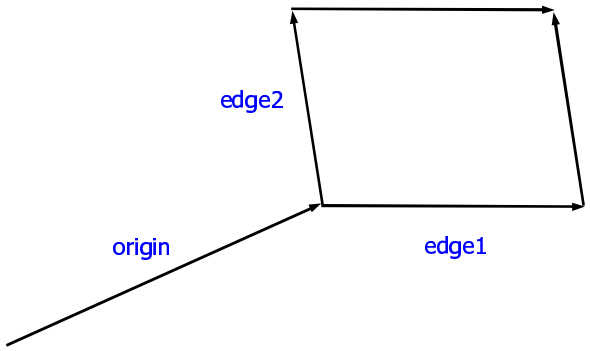
Implementation of frames, painters, basic operations

- We'll assume that the graphics window has a coordinate system with the origin at the lower left-hand corner
- The unit of measurement is the pixel; the location (10,20) is 10 pixels to the right and 20 pixels up from the origin
- The x-coordinate axis is horizontal, y-coordinate axis is vertical

Frames

- represent four-sided regions in the graphics window
- described by three vectors:
  - origin - location of one corner of the frame
  - edge1 - vector representing one edge of frame
  - edge2 - vector representing other edge of frame not parallel to edge1

Corners of the frame



Basic implementation of frames

- Constructor:

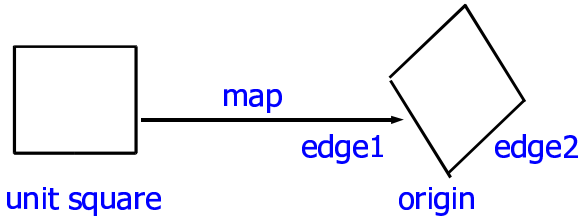
```
(make-frame origin edge1 edge2)
```

- Selectors:

```
(origin-frame frame)
(edge1-frame frame)
(edge2-frame frame)
```

- (You will implement these)

Map from unit square to frame



(x,y) --> origin + (x * edge1) + (y * edge2)

Unit frame mapping procedure

```
(define (frame-coord-map frame)
  (lambda (v)
    (add-vect
      (origin-frame frame)
      (add-vect
        (scale-vect (xcor-vect v)
                    (edge1-frame frame))
        (scale-vect (ycor-vect v)
                    (edge2-frame frame))))))
```

Drawing lines in DrScheme version 103 (and 205)

```
; for version 205,
; remove semi-colon from next line
;(require (lib "graphics.ss" "graphics"))
(open-graphics)
(define size 400)
(define w
 (open-viewport "graphics window" size size))
(define (clear) ((clear-viewport w)))
(define line-drawer (draw-line w))
```

(version 103 continued)

```
(define (drawline p-start p-end)
  (line-drawer
    (make-posn (xcor-vect p-start)
               (- size
                  1
                  (ycor-vect p-start)))
    (make-posn (xcor-vect p-end)
               (- size
                  1
                  (ycor-vect p-end)))))
```

Drawing lines in DrScheme version 205

```
(define size 400)
(define graphics-window
  (instantiate frame%
               ("GRAPHICS WINDOW")
               (width size)
               (height size)))
(define canvas
  (instantiate canvas% (graphics-window)))
```

(version 205 continued)

```
(define dc (send canvas get-dc))
(define (clear) (send dc clear))
(define pen (instantiate pen%
                         ("BLACK" 1 'solid)))
(send dc set-pen pen)
(send graphics-window show #t)
(sleep/yield 1)  ; wait 1 second for window
```

(version 205 continued again)

```
(define (drawline p-start p-end)
  (send dc
        draw-line
        (xcor-vect p-start)
        (- size 28 (ycor-vect p-start))
        (xcor-vect p-end)
        (- size 28 (ycor-vect p-end))))
```

Painters draw sequences of line segments (constructor)

```
(define (segments->painter segment-list)
  (lambda (frame)
    (for-each
      (lambda (segment)
        (drawline
          ((frame-coord-map frame)
           (start-segment segment))
          ((frame-coord-map frame)
           (end-segment segment))))
      segment-list)))
```

Flag painter

```
(define flag-top (make-vect 0 1))
(define flag-tip (make-vect 1 0.75))
(define flag-middle (make-vect 0 0.5))
(define flag-bottom (make-vect 0 0))
(define flag
  (segments->painter
   (list
     (make-segment flag-top flat-bottom)
     (make-segment flag-top flag-tip)
     (make-segment flag-tip flag-middle))))
```

Drawing flags (demo 1)

```
(define square-frame
  (make-frame (make-vect 10 10)
              (make-vect 30 0)
              (make-vect 0 30)))
(define rectangle-frame
  (make-frame (make-vect 60 10)
              (make-vect 20 0)
              (make-vect 0 30)))
```

(demo 1 continued)

```
(define rhomboid-frame
  (make-frame (make-vect 50 50)
              (make-vect 20 5)
              (make-vect 10 30)))
(flag square-frame)
(flag rectangle-frame)
(flag rhomboid-frame)
```

Drawing flags (demo 2)

```
(define full-frame
  (make-frame (make-vect 0 0)
              (make-vect (- size 1) 0)
              (make-vect 0 (- size 1))))
```

(Replace the last two vectors with

```
(make-vect (- size 10) 0)
(make-vect 0 (- size 28))
```

in the version 205 only implementation.)

```
((square-limit flag 6) full-frame)
```

Some operators on painters

One way to make an operator on painters is to make a painter's input (a frame) into another frame, then apply the painter to that frame

Painter transformer

```
(define (transform-painter painter origin
                corner1 corner2)
  (lambda (frame)
    (let ((m (frame-coord-map frame)))
      (let ((new-origin (m origin)))
        (painter
          (make-frame
            new-origin
            (sub-vect (m corner1) new-origin)
            (sub-vect (m corner2)
                      new-origin)))))))
```

Flip-vert operator

```
(define (flip-vert painter)
  (transform-painter painter
                     (make-vect 0.0 1.0)
                     (make-vect 1.0 1.0)
                     (make-vect 0.0 0.0)))
```

Shrink operator

```
(define (shrink-to-upper-right painter)
  (transform-painter painter
                     (make-vect 0.5 0.5)
                     (make-vect 1.0 0.5)
                     (make-vect 0.5 1.0)))
```

Rotate operator

```
(define (rotate90 painter)
  (transform-painter painter
                     (make-vect 1.0 0.0)
                     (make-vect 1.0.1.0)
                     (make-vect 0.0 0.0)))
```

Squash operator

```
(define (squash-inwards painter)
  (transform-painter painter
                     (make-vect 0.0 0.0)
                     (make-vect 0.65 0.35)
                     (make-vect 0.35 0.65)))
```

Beside operator

```
(define (beside painter1 painter2)
  (let ((split-point (make-vect 0.5 0.0)))
    (let ((paint-left
           (transform-painter
              painter1
              (make-vect 0.0 0.0)
              split-point
              (make-vect 0.0 1.0)))
```

beside (continued)

```
          (paint-right
            (transform-painter
              painter2
              split-point
              (make-vect 1.0 0.0)
              (make-vect 0.5 1.0))))
      (lambda(frame)
        (paint-left frame)
        (paint-right frame)))))
```

Picture language has stratified design

Hierarchy of different data objects:

- meta-operators
- operators
- painters, frames
- line-segments
- vectors