6.001: Structure and Interpretation of Computer Programs

• Today
  – Building a new language using data and procedure abstractions

Themes to be integrated

• Data abstraction
  – Separate use of data structure from details of data structure

• Procedural abstraction
  – Capture common patterns of behavior and treat as black box for generating new patterns

• Means of combination
  – Create complex combinations, then treat as primitives to support new combinations

• Use modularity of components to create new language for particular problem domain

Our target – the art of M. C. Escher

ESCHER on ESCHER: Exploring the Infinite, p. 41

My buddy George

A procedural definition of George

(define (george rect)
  (draw-line rect .25 0 .35 .5)
  (draw-line rect .35 .5 .3 .6)
  (draw-line rect .3 .6 .15 .4)
  (draw-line rect .15 .4 0 .65)
  (draw-line rect .4 0 .5 .3)
  (draw-line rect .5 .3 .6 0)
  (draw-line rect .75 0 .6 .45)
  (draw-line rect .6 .45 1 .15)
  (draw-line rect 1 .35 .75 .65)
  (draw-line rect .75 .65 .6 .65)
  (draw-line rect .6 .65 .65 .85)
  (draw-line rect .65 .85 .6 1)
  (draw-line rect .4 1 .15 .85)
  (draw-line rect .35 .85 .4 .65)
  (draw-line rect .4 .65 .3 .65)
  (draw-line rect .3 .65 .15 .6)
  (draw-line rect .15 .6 0 .85))

Yuck!!
Data abstractions for lines

\[(\text{define p1 (make-vect 2 3))}\]
\[(\text{xcor p1}) \rightarrow 2\]
\[(\text{ycor p1}) \rightarrow 3\]
\[(\text{define s1 (make-segment p1 p2))}\]
\[(\text{xcor (start-segment s1)}) \rightarrow 2\]
\[(\text{ycor (end-segment s1)}) \rightarrow 4\]

Gluing things together

For pairs, use a cons:

For larger structures, use a list:

\[(\text{list 1 2 3 4})\]
\[(\text{cons 1 (cons 2 (cons 3 (cons 4 nil))})]\]

Completing our abstraction

Points or vectors:
\[(\text{define make-vect cons})\]
\[(\text{define xcor car})\]
\[(\text{define ycor cdr})\]

Line segments:
\[(\text{define make-segment list})\]
\[(\text{define start-segment car})\]
\[(\text{define end-segment cdr})\]

Properties of data structures

- Contract between constructor and selectors
- Property of closure:
  - A list is a sequence of pairs, ending in the empty list, nil.
  - Consing anything onto a list results in a list (by definition)
  - Taking the cdr of a list results in a list (except perhaps for the empty list)

Drawing in a rectangle or frame

\[(\text{define p1 (make-vect .25 0})\]
\[(\text{define p2 (make-vect .35 .5})\]
\[(\text{define p3 (make-vect .3 .6})\]
\[(\text{define p4 (make-vect .15 .4})\]
\[(\text{define p5 (make-vect 0 .65})\]
\[(\text{define p6 (make-vect .4 0})\]
\[(\text{define p7 (make-vect .6 .45})\]
\[(\text{define p8 (make-vect 1 .15})\]
\[(\text{define p9 (make-vect 1 .35})\]
\[(\text{define p10 (make-vect .75 .65})\]
\[(\text{define p11 (make-vect .6 .65})\]
\[(\text{define p12 (make-vect .65 .85})\]
\[(\text{define p13 (make-vect .6 1})\]
\[(\text{define p14 (make-vect .4 1})\]
\[(\text{define p15 (make-vect .6 .45})\]
\[(\text{define p16 (make-vect .8 .85})\]
\[(\text{define george-lines (list (make-segment p1 p2)\]
\[(\text{(make-segment p2 p3})\]
\[(\text{(make-segment p3 p4})\]
\[(\text{(make-segment p4 p5})\]
\[(\text{(make-segment p5 p6})\]
\[(\text{(make-segment p6 p7})\]
\[(\text{(make-segment p7 p8})\]
\[(\text{(make-segment p8 p9})\]
\[(\text{(make-segment p9 p10})\]
\[(\text{(make-segment p10 p11})\]
\[(\text{(make-segment p11 p12})\]
\[(\text{(make-segment p12 p13})\]
\[(\text{(make-segment p13 p14})\]
\[(\text{(make-segment p14 p15})\]
\[(\text{(make-segment p15 p16})\]
\[(\text{(make-segment p16 p17})\]
\[(\text{(make-segment p17 p18})\]
\[(\text{(make-segment p18 p19})\]
\[(\text{(make-segment p19 p20})\]
\[(\text{(make-segment p20 p21})\]
\[(\text{(make-segment p21 p22})\])\]
Drawing lines is just algebra

- Drawing a line is just some algebra. If a rectangle has an origin \( o \), a horizontal axis \( u \) and a vertical axis \( v \) then a point \( p \), with components \( x \) and \( y \) gets mapped to the point:

\[
o + xu + yv
\]

Manipulating vectors

[Diagram showing vectors and their manipulation]

Generating the abstraction of a frame

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

Picture:
- (define some-primitive-picture
  (lambda (rect)
    (draw some stuff in rect))

What is a picture?

- Could just create a general procedure to draw collections of line segments
- But want to have flexibility of using any frame to draw in frame
- SO – we make a picture be a procedure!!
- Captures the procedural abstraction of drawing data within a frame
The picture abstraction

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

For-each is like map, except it doesn’t collect a list of results, but simply applies procedure to each element of list for effect.

A better George

Remember we have george-lines from before

So here is George!

(define g (make-picture george-lines))

(define frame1 (make-rectangle origin1 horiz1 vert1))

(g frame1)

Operations on pictures

(define g (make-picture george-lines))

(define frame1 (make-rectangle origin1 horiz1 vert1))

Operations on pictures

(define (rotate90 pict)
  (lambda (rect)
    (pict (make-rectangle
      (+vect (origin rect) (horiz rect) –1))))

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

(define g (make-picture george-lines))

((together g (rotate90 g)) frame1)

A Georgian mess!

((together g (rotate90 g)) frame1)

Operations on pictures

PictA:  A
PictB:  H

beside
above
Creating a picture

More procedures to combine pictures:

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

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

Pictures have a closure property!

Big brother

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

A left-right flip

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

(define 4bats
  (above acrobats
       (flip acrobats)
       .5))
Recursive combinations of pictures

(\text{define} \ (\text{up-push pict} \ n) \n \quad (\text{if} \ (= \ n \ 0) \n \quad \quad \text{pict} \n \quad \quad (\text{above} \ (\text{up-push pict} \ (- \ n \ 1)) \n \quad \quad \quad \text{pict} \n \quad \quad .25)))

Pushing George around

(\text{define} \ (\text{right-push pict} \ n) \n \quad (\text{if} \ (= \ n \ 0) \n \quad \quad \text{pict} \n \quad \quad (\text{beside} \ \text{pict} \n \quad \quad \quad (\text{right-push pict} \ (- \ n \ 1)) \n \quad \quad \quad .75))]

Pushing George into the corner

(\text{define} \ (\text{corner-push pict} \ n) \n \quad (\text{if} \ (= \ n \ 0) \n \quad \quad \text{pict} \n \quad \quad (\text{above} \n \quad \quad \quad (\text{beside} \n \quad \quad \quad \quad (\text{up-push pict} \ n) \n \quad \quad \quad \quad (\text{corner-push pict} \ (- \ n \ 1)) \n \quad \quad \quad \quad .75) \n \quad \quad \quad \quad \text{pict} \n \quad \quad \quad \quad (\text{right-push pict} \ (- \ n \ 1)) \n \quad \quad \quad \quad .75) \n \quad \quad \quad .25)))

Putting copies together

(\text{define} \ (\text{4pict} \ p1 \ r1 \ p2 \ r2 \ p3 \ r3 \ p4 \ r4) \n \quad (\text{beside} \n \quad \quad (\text{above} \n \quad \quad \quad ((\text{repeated rotate90} \ r1) \ p1) \n \quad \quad \quad ((\text{repeated rotate90} \ /2) \ p2) \n \quad \quad \quad .5) \n \quad \quad \quad (\text{above} \n \quad \quad \quad ((\text{repeated rotate90} \ r3) \ p3) \n \quad \quad \quad ((\text{repeated rotate90} \ r4) \ p4) \n \quad \quad \quad .5)) \n \quad \quad (\text{4same} \ p1 \ 1 \ 2 \ 3))

(\text{define} \ (\text{4same} \ p1 \ r1 \ r2 \ r3 \ r4) \n \quad (\text{4pict} \ p1 \ r1 \ p2 \ r2 \ p3 \ r3 \ p4 \ r4))

(\text{4same} \ g 0 \ 1 \ 2 \ 3)
(define (square-limit pict n)
  (4same (corner-push pict n) 1 2 0 3))

(square-limit 4bats 2)

"Escher" is an embedded language

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Scheme data</th>
<th>Picture language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive data</td>
<td>3, #f, george</td>
<td>nil, george, other pictures</td>
</tr>
<tr>
<td>Primitive procedures</td>
<td>+, map, _</td>
<td>Rotate90, _</td>
</tr>
<tr>
<td>Combinations</td>
<td>(p a b)</td>
<td>cons, car, cdr, together, beside, _, And Scheme mechanisms</td>
</tr>
<tr>
<td>Abstraction</td>
<td>(define ...)</td>
<td>(define ...)</td>
</tr>
<tr>
<td>Naming</td>
<td>(lambda ...)</td>
<td>(lambda ...)</td>
</tr>
</tbody>
</table>