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, higher level language for particular problem domain

Our target – the art of M. C. Escher

Level of language matters.
Programming is a process of inventing task-specific languages.

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 .36 .15 .4)
  (draw-line rect .4 0 .5 .3)
  (draw-line rect .5 .3 .6 0)
  (draw-line rect .75 .6 .45 1 .15)
  (draw-line rect 1 .35 .75 .65)
  (draw-line rect .75 .65 .65 .85)
  (draw-line rect .65 .85 .6 1)
  (draw-line rect 4 .1 .35 .85)
  (draw-line rect .36 0 .65 1)
  (draw-line rect .15 .6 0 .85))

Yuck!!

Need a data abstraction for lines

(define p1 (make-vect 2 3))
(xcor p1) → 2
(ycor p1) → 3

(define p2 (make-vect 5 4))
(xcor (start-segment p1)) → 2
(ycor (end-segment p1)) → 4

A better George

(define (george-line)
  (list (make-segment p1 p2)
         (make-segment p2 p3)
         (make-segment p3 p4)
         (make-segment p4 p5)
         (make-segment p6 p7)
         (make-segment p7 p8)
         (make-segment p8 p10)
         (make-segment p10 p11)
         (make-segment p11 p13)
         (make-segment p13 p14)
         (make-segment p14 p15)
         (make-segment p15 p16)
         (make-segment p16 p17)
         (make-segment p17 p18)
         (make-segment p18 p20)
         (make-segment p20 p21)
         (make-segment p21 p23))))

• Have isolated elements of abstraction
• Could change a point without having to redefine segments that use it
• Have separated data abstraction from its use

Gluing things together

For pairs, use a cons:

For larger structures, use a list:

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)

Completing our abstraction

Points or vectors:
(define make-vect cons)
(define xcor car)
(define ycor cdr)

Line segments:
(define make-segment-list)
(define start-segment car)
(define end-segment cdr)
Drawing in a rectangle or frame

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

Select parts

Compute more primitive operation

Reassemble new parts

Generating the abstraction of a frame

What happens if we change an abstraction?

Note that this still satisfies the contract

What else needs to change in our system? BUPKIS, NADA, NIL, NOTHING
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

Creating a picture

- Higher order procedure
- 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

The picture abstraction

```scheme
(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)))
```

A better George

Remember we have george-lines from before

So here is George!

```scheme
(define g (make-picture george-lines))
(define frame1 (make-rectangle origin1 horiz1 vert1))
(g frame1)
```

Operations on pictures

- V -> H'
Operations on pictures

(define g (make-picture george-lines))
(g frame1)

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

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

(define g (make-picture george-lines))
(g frame1)

A Georgian mess!

((together g (rotate90 g))
  frame1)

Operations on pictures

PictA: ------------------------ A

PictB: ------------------------ H

Creating a picture

Beside

More procedures to combine pictures:

(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))
        (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

(\texttt{define} (flip pict)
  (\texttt{lambda} (rect)
    (pict (make-rectangle
        (+vect (origin rect) (horiz rect))
        (scale-vect (horiz rect) \texttt{-1})
        (vert rect))))))

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

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

Recursive combinations of pictures

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

Pushing George around

(\texttt{define} (right-push pict n)
  (if (= n 0)
    pict
    (beside pict
      (right-push pict (- n 1))
      .70))))
Pushing George into the corner

```
(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)))
```

Pushing George into a corner

```
(corner-push 4bats 2)
```

Putting copies together

```
(define (4pict p1 r1 p2 r2 p3 r3 p4 r4)
  (beside
   (above
    ((repeated rotate90 r1) p1)
    ((repeated rotate90 r2) p2)
    .5)
   (above
    ((repeated rotate90 r3) p3)
    ((repeated rotate90 r4) p4)
    .5)
   .5))

(define (4same p r1 r2 r3 r4)
  (4pict p r1 p r2 p r3 p r4))

(4same g 0 1 2 3)
```

(square-limit pict n)

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

(square-limit 4bats 2)
```

(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>$, $f, george</td>
<td>nil, george, other pictures</td>
</tr>
<tr>
<td>Primitive procedures</td>
<td>*s, 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>
<tr>
<td>Creation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>