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

Yuck!!

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 .65 .85 6 1)
  (draw-line rect .4 1 .35 .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

\[
\begin{align*}
&\text{(define } p1 \text{ (make-vect } \ 2 \ 3)) \\
&(\text{xcor } p1 \text{) 2} \\
&(\text{ycor } p1 \text{) 3} \\
&\text{(define } p2 \text{ (make-vect } \ 5 \ 4)) \\
&(\text{xcor } (\text{start-segment } s1) \text{) 2} \\
&(\text{ycor } (\text{end-segment } s1) \text{) 4}
\end{align*}
\]

A better George

\[
\begin{align*}
&\text{(define } p1 \text{ (make-vect } \ .25 \ 0)) \\
&\text{(define } p2 \text{ (make-vect } \ .35 \ .5)) \\
&\text{(define } p3 \text{ (make-vect } \ .3 \ .6)) \\
&\text{(define } p4 \text{ (make-vect } \ .15 \ .4)) \\
&\text{(define } p5 \text{ (make-vect } \ 0 \ .65)) \\
&\text{(define } p6 \text{ (make-vect } \ .4 \ 0)) \\
&\text{(define } p7 \text{ (make-vect } \ .5 \ .3)) \\
&\text{(define } p8 \text{ (make-vect } \ .6 \ .45)) \\
&\text{(define } p9 \text{ (make-vect } \ .75 \ 0)) \\
&\text{(define } p10 \text{ (make-vect } \ .6 \ .45)) \\
&\text{(define } p11 \text{ (make-vect } \ 1 \ .15)) \\
&\text{(define } p12 \text{ (make-vect } \ 1 \ .35)) \\
&\text{(define } p13 \text{ (make-vect } \ .75 \ .65)) \\
&\text{(define } p14 \text{ (make-vect } \ .6 \ .65)) \\
&\text{(define } p15 \text{ (make-vect } \ .65 \ .85)) \\
&\text{(define } p16 \text{ (make-vect } \ .6 \ 1)) \\
&\text{(define } p17 \text{ (make-vect } \ .4 \ 1)) \\
&\text{(define } p18 \text{ (make-vect } \ .8 \ .45)) \\
&\text{(define } p19 \text{ (make-vect } \ .35 \ .85)) \\
&\text{(define } p20 \text{ (make-vect } \ .3 \ .65)) \\
&\text{(define } p21 \text{ (make-vect } \ .15 \ .6)) \\
&\text{(define } p22 \text{ (make-vect } \ 0 \ .85))
\end{align*}
\]

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

Gluing things together

For pairs, use a \texttt{cons}:

\[
\begin{align*}
&\text{(cons 1 (cons 2 (cons 3 (cons 4 nil)))))
\end{align*}
\]

For larger structures, use a \texttt{list}:

\[
\begin{align*}
&\text{(list 1 2 3 4)}
\end{align*}
\]

Properties of data structures

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

Completing our abstraction

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

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

Drawing in a rectangle or frame
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 happens if we change an abstraction?

(define make-vect list)
(define xcor car)
(define ycor cadr)

Note that this still satisfies the contract

What else needs to change in our system? BUPKIS, NADA, 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

Manipulating vectors

+-vect

Select parts

Compute more primitive operation

Reassemble new parts

Creating a picture
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)))
```

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.

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:

\[
  o + xu + yv
\]

A better George

Remember we have george-lines from before

So here is George!

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

Operations on pictures

```
(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)))
```

A Georgian mess!
Operations on pictures

PictA:  \[ \begin{array}{c}
+ \\
- \\
\end{array} \]  \[ \rightarrow \]  \[ \begin{array}{c}
A \\
\end{array} \]

PictB:  \[ \begin{array}{c}
+ \\
- \\
\end{array} \]  \[ \rightarrow \]  \[ \begin{array}{c}
H \\
\end{array} \]

Creating a picture

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)))
    (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)))

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 acrobats
  (beside g
    (rotate180 (flip g))
    .5))
Recursive combinations of pictures

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

Pushing George around

(define (right-push pict n)
  (if (= n 0)
      pict
      (beside pict (right-push pict (- n 1)) .75)))

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)))

(corner-push 4bats 2)
Putting copies together

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

(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>Nil</td>
<td>Half-line, George, other picture</td>
</tr>
<tr>
<td>Primitive</td>
<td>+, map,…</td>
<td>Rotate90, …</td>
</tr>
<tr>
<td>procedures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combinations</td>
<td>(p a b)</td>
<td>Cons, car, cdr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Together, beside, …</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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>