Structure and Interpretation of Computer Programs

COMP200
Building a new language using data and procedure abstractions
• **Data abstraction**
  Separate use of data structure from details of data structure
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
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
**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
LET'S START SIMPLE

Wave
LET’S START SIMPLE

Waves
LET’S START SIMPLE

Waves
LET’S START SIMPLE
maybe not so simple after all...

(define (wave 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 .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))
(define (wave 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 .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))
(define p1 (make-vect 2 3))
(xcor p1) ; → 2
(ycor p1) ; → 3
DATA ABSTRACTIONS

Lines

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

(define p2 (make-vect 5 4))

(define s1 (make-seg p1 p2))
(xcor (start-seg s1)) ; → 2
(xcor (end-seg s1)) ; → 4
DATA ABSTRACTIONS

Lines

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

(define p2 (make-vect 5 4))

(define s1 (make-seg p1 p2))
(xcor (start-seg s1)) ; → 2
(xcor (end-seg s1)) ; → 4
DATA ABSTRACTIONS

A Better Wave

(define (wave 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 .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))

(define wave-lines
  (list (make-seg p1 p2)
        (make-seg p2 p3)
        (make-seg p3 p4)
        (make-seg p4 p5)
        (make-seg p6 p7)
        (make-seg p7 p8)
        (make-seg p9 p10)
        (make-seg p10 p11)
        (make-seg p12 p13)
        (make-seg p13 p14)
        (make-seg p14 p15)
        (make-seg p15 p16)
        (make-seg p17 p18)
        (make-seg p18 p19)
        (make-seg p19 p20)
        (make-seg p20 p21)
        (make-seg p21 p22)))
DATA ABSTRACTIONS

A Better Wave

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

(define (wave-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 .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))

(define wave-lines
  (list (make-seg p1 p2)
        (make-seg p2 p3)
        (make-seg p3 p4)
        (make-seg p4 p5)
        (make-seg p6 p7)
        (make-seg p7 p8)
        (make-seg p9 p10)
        (make-seg p10 p11)
        (make-seg p12 p13)
        (make-seg p13 p14)
        (make-seg p14 p15)
        (make-seg p15 p16)
        (make-seg p17 p18)
        (make-seg p18 p19)
        (make-seg p19 p20)
        (make-seg p20 p21)
        (make-seg p21 p22)))
DATA ABSTRACTIONS
Glueing Things Together

For pairs, use a **cons**: (cons <el1> <el2>)

For larger structures, use a **list**:
(list <el1> <el2> ... <eln>)

(list 1 2 3 4)
(cons 1 (cons 2 (cons 3 (cons 4 nil))))
DATA STRUCTURES

Properties

• Contract between constructor and selectors

• Property of closure:
  • A list is a sequence of pairs, ending in the empty list, nil.
  • cons’ing 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)

• Better to use adjoin, first and rest, instead of cons, car and cdr.
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 cadr)
DRAWING
in a rectangle or frame
DRAWING
in a rectangle or frame
DRAWING

in a rectangle or frame

Rectangle

(define make-rectangle list)
(define origin car)
(define horiz cadr)
(define vert caddr)
DRAWING
in a rectangle or 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 the abstraction?

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

still satisfies the contract
(define (+vect v1 v2)
  (make-vect
    (+ (xcor v1) (xcor v2))
    (+ (ycor v1) (ycor v2))))
(define (+vect v1 v2)
  (make-vect
   (+ (xcor v1) (xcor v2))
   (+ (ycor v1) (ycor v2))))

(define (scale-vect vect factor)
  (make-vect
   (* factor (xcor vect))
   (* factor (ycor vect))))
(define (+vect v1 v2)
  (make-vect
   (+ (xcor v1) (xcor v2))
   (+ (ycor v1) (ycor v2))))

(define (scale-vect vect factor)
  (make-vect
   (* factor (xcor vect))
   (* factor (ycor vect)))))

(define (-vect v1 v2)
  (+vect v1 (scale-vect v2 -1)))
(define (+vect v1 v2)
  (make-vect
   (+ (xcor v1) (xcor v2))
   (+ (ycor v1) (ycor v2))))

(define (scale-vect vect factor)
  (make-vect
   (* factor (xcor vect))
   (* factor (ycor vect))))

(define (-vect v1 v2)
  (+vect v1 (scale-vect v2 -1)))

(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))))))
(define (+vect v1 v2)
  (make-vect
   (+ (xcor v1) (xcor v2))
   (+ (ycor v1) (ycor v2))))

(define (scale-vect vect factor)
  (make-vect
   (* factor (xcor vect))
   (* factor (ycor vect))))

(define (-vect v1 v2)
  (+vect v1 (scale-vect v2 -1)))

(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))))))
(define (+v v1 v2)
  (make-vect
   (+ (xc v1) (xc v2))
   (+ (yc v1) (yc v2))))

(define (scale-v vect factor)
  (make-vect
   (* factor (xc vect))
   (* factor (yc vect))))

(define (-v v1 v2)
  (+v v1 (scale-v v2 -1)))

(define (rotate-v v angle)
  (let ((c (cos angle))
         (s (sin angle)))
    (make-vect
     (- (* c (xc v))
        (* s (yc v)))
     (+ (* c (yc v))
        (* s (xc v))))))

select parts

compute more primitive operations

reassemble new parts
DRAWING
Creating a Picture

segments

make-picture
DRAWING

Creating a Picture

segments

make-picture

Picture proc

draw

picture on screen
(define (make-picture seglist)
  (lambda (rect)
    (for-each
      (lambda (segment)
        (let ((b (start-segment segment)))
          (let ((e (end-segment segment)))
            (draw-line rect
              (icorn b)
              (ycorn b)
              (icorn e)
              (ycorn e)))))
      seglist))))
(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))))
If a rectangle has
- an origin \( \mathbf{o} \),
- a horizontal axis \( \mathbf{u} \)
- a vertical axis \( \mathbf{v} \)

then a point \( \mathbf{p} \), with components \( x \) and \( y \) gets mapped to the point: \( \mathbf{o} + x \mathbf{u} + y \mathbf{v} \)
(define w (make-picture wave-lines))
Operations on Pictures
DRAWING

Operations on Pictures

rotate

V

H

O

V'

H'

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

(draw (together
       g
       (rotate90 g)))
**DRAWING**

Operations on Pictures

PictA: □ ➔ A

PictB: □ ➔ H
DRAWING

Operations on Pictures

PictA:  

PictB:  

beside

above
DRAWING
Creating a Picture

rect → Picture proc → beside → Picture proc → picture on screen
(define (beside pict1 pict2 a)
   (lambda (rect)
(define (beside pict1 pict2 a)
  (lambda (rect)
    (pict1 (make-rectangle
             (origin rect)
             (scale-vect (horiz rect) a)
             (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 (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 (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)
(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)))
DRAWING
Combining Pictures

Pictures have closure property!
(define big-bro
  (beside g
    (above empty-picture g .5)
    .5))
(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))
(define 4bats
  (above acrobats
    (flip acrobats)
    .5))
(define (up-push pict n)
(define (up-push pict n)
  (if (= n 0)
      pict
      (above (up-push pict (- n 1))
            pict .25))))
DRAWING
Pushing Wave Around
(define (right-push pict n)
  (if (= n 0)
      pict (beside pict
           (right-push pict (- n 1))
            75))))
(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)
(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)
(define (square-limit pict n)
  (4same (corner-push pict n)
    1 2 0 3))

(square-limit 4bats 2)
### DRAWING

“Escher” is an Embedded Language

<table>
<thead>
<tr>
<th></th>
<th>Scheme</th>
<th>Scheme data</th>
<th>Picture language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primitive data</strong></td>
<td>3, #f, wave</td>
<td>nil</td>
<td>half-line, wave, other pictures</td>
</tr>
<tr>
<td><strong>Primitive procedures</strong></td>
<td>+, map, ...</td>
<td></td>
<td>rotate90, ...</td>
</tr>
<tr>
<td><strong>Combinations</strong></td>
<td>(p a b)</td>
<td>cons, car, cdr</td>
<td>together, beside, …, and Scheme mechanisms</td>
</tr>
<tr>
<td><strong>Abstraction Naming Creation</strong></td>
<td>(define ...) (lambda ...)</td>
<td>(define ...) (lambda ...)</td>
<td>(define ...) (lambda ...)</td>
</tr>
</tbody>
</table>