

Today's Lecture

- Programming as the process of **creating a new task-specific language**
 - data abstractions
 - procedure abstractions
 - higher-order procedures

3/6/2007

1/43

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

3/6/2007

2/43



```

step1: (forward 1)
       (no-opening-on-right?)
       (goto step1)
       (turn right 90)
step2: (forward 1)
       (no-opening-on-left?)
       (goto step2)
step3: (forward 1)
       (no-opening on left?)
       (goto step3)
       (turn left 90)
       (forward 1)
       . . .
    
```

(while no-opening-on-right (forward 1))

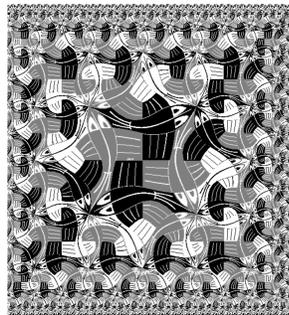
Go to *first right*
Go to *second left*

Go get me coffee, please.

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

3/6/2007

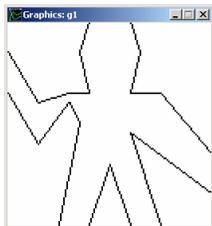
Our target – the art of M. C. Escher



3/6/2007

ESCHER on ESCHER: Exploring the Infinite, p. 41
Harry Abrams, Inc., New York, 1989

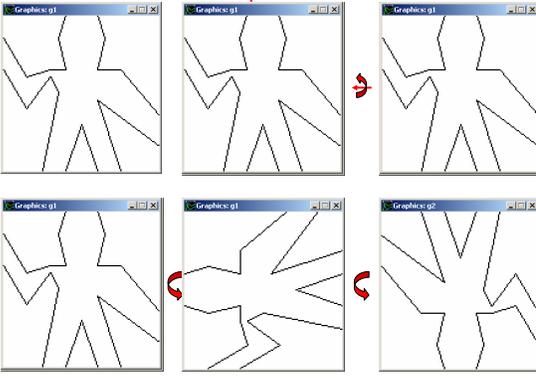
4/43



My buddy George

3/6/2007

5/43

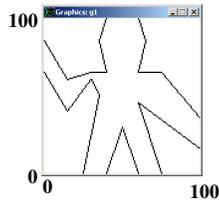


3/6/2007

6/43

A procedural definition of George

```
(define (george)
  (draw-line 25 0 35 50)
  (draw-line 35 50 30 60)
  (draw-line 30 60 15 40)
  (draw-line 15 40 0 65)
  (draw-line 40 0 50 30)
  (draw-line 50 30 60 0)
  (draw-line 75 0 60 45)
  (draw-line 60 45 100 15)
  (draw-line 100 35 75 65)
  (draw-line 75 65 60 65)
  (draw-line 60 65 65 85)
  (draw-line 65 85 60 100)
  (draw-line 40 100 35 85)
  (draw-line 35 85 40 65)
  (draw-line 40 65 30 65)
  (draw-line 30 65 15 60)
  (draw-line 15 60 0 85))
```



Yuck!!
Where's the abstraction?

3/6/2007

7/43

Need a data abstraction for lines

```
(define p1 (make-vect 2 3))
(xcor p1) → 2
(ycor p1) → 3
(define p2 (make-vect 5 4))
(define s1 (make-segment p1 p2))
(xcor (start-segment s1)) → 2
(ycor (end-segment s1)) → 4
```

3/6/2007

8/43

A better George

```
(define p1 (make-vect .25 0))
(define p2 (make-vect .35 .5))
(define p3 (make-vect .3 .6))
(define p4 (make-vect .15 .4))
(define p5 (make-vect 0 .65))
(define p6 (make-vect .4 0))
(define p7 (make-vect .5 .3))
(define p8 (make-vect .6 0))
(define p9 (make-vect .75 0))
(define p10 (make-vect .6 .45))
(define p11 (make-vect 1 .15))
(define p12 (make-vect 1 .35))
(define p13 (make-vect .75 .65))
(define p14 (make-vect .6 .65))
(define p15 (make-vect .65 .85))
(define p16 (make-vect .6 1))
(define p17 (make-vect .4 1))
(define p18 (make-vect .35 .85))
(define p19 (make-vect .4 .65))
(define p20 (make-vect .3 .65))
(define p21 (make-vect .15 .6))
(define p22 (make-vect 0 .85))

(define george-lines
  (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 p9 p10)
        (make-segment p10 p11)
        (make-segment p12 p13)
        (make-segment p13 p14)
        (make-segment p14 p15)
        (make-segment p15 p16)
        (make-segment p17 p18)
        (make-segment p18 p19)
        (make-segment p19 p20)
        (make-segment p20 p21)
        (make-segment p21 p22)))
```

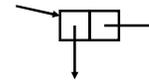
• Have isolated elements of abstraction
• Could change a point without having to redefine segments that use it
• Have separated data abstraction from its use

3/6/2007

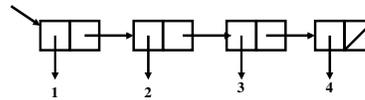
9/43

Gluing things together

For pairs, use a cons:



For larger structures, use a list:



```
(list 1 2 3 4)
(cons 1 (cons 2 (cons 3 (cons 4 '()) )))
```

3/6/2007

10/43

Properties of data structures

- Contract between constructor and selectors
- Property of closure:
 - consing anything onto a list produces a list
 - Taking the cdr of a list produces a list (except perhaps for the empty list)

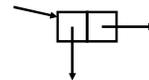
3/6/2007

11/43

Completing our abstraction

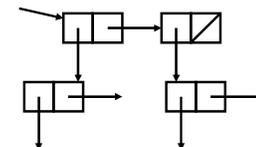
Points or vectors:

```
(define make-vect cons)
(define xcor car)
(define ycor cdr)
```



Line segments:

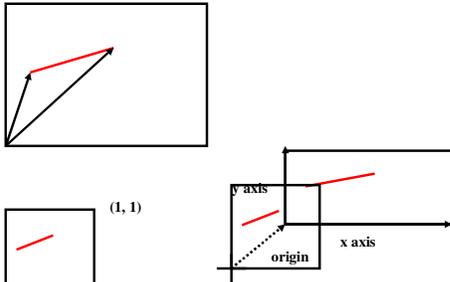
```
(define make-segment list)
(define start-segment first)
(define end-segment second)
```



3/6/2007

12/43

Drawing in a rectangle or frame

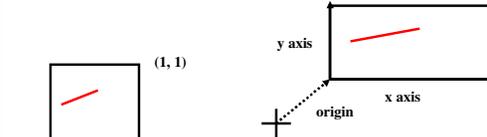


(0, 0)
3/6/2007

13/43

Drawing lines is just algebra

- Drawing a line relative to a frame is just some algebra.
- Suppose frame has origin \mathbf{o} , horizontal axis \mathbf{u} and vertical axis \mathbf{v}
- Then a point \mathbf{p} , with components x and y , gets mapped to the point: $\mathbf{o} + x\mathbf{u} + y\mathbf{v}$

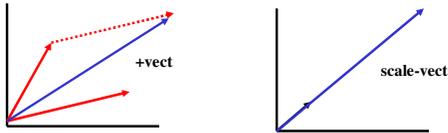


(0, 0)
3/6/2007

14/43

Manipulating vectors

$$\mathbf{o} + x\mathbf{u} + y\mathbf{v}$$



3/6/2007

15/43

```
(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)))))
```

Select parts

Compute more primitive operation

Reassemble new parts

What is the underlying representation of a point? Of a segment?

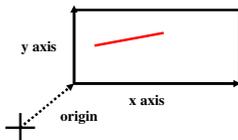
3/6/2007

16/43

Generating the abstraction of a frame

Rectangle:

```
(define make-rectangle list)
(define origin first)
(define x-axis second)
(define y-axis third)
```



Determining where to draw a point p : $\mathbf{o} + x\mathbf{u} + y\mathbf{v}$

```
(define (coord-map rect)
  (lambda (p)
    (+vect (origin rect)
          (+vect (scale-vect (x-axis rect) (xcor p))
                (scale-vect (y-axis rect) (ycor p)))))
```

3/6/2007

17/43

What happens if we change how an abstraction is represented?

```
(define make-vect list)
(define xcor first)
(define ycor second)
```

Note that this still satisfies the contract for vectors

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

NADA,

NIL,

NOTHING

3/6/2007

18/43

What is a picture?

- Maybe a collection of line segments?
 - That would work for George:

```
(define george-lines
  (list (make-segment p1 p2)
        (make-segment p2 p3)
        ...))
```



- ...but not for Mona
- We want to have flexibility of what we draw and how we draw it in the frame



- SO - we make a picture be a **procedure**

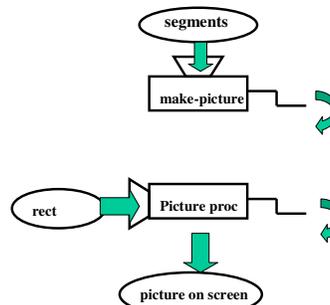
```
(define some-primitive-picture
  (lambda (rect)
    draw some stuff in rect ))
```

- Captures the procedural abstraction of drawing data within a frame

3/6/2007

19/43

Creating a picture



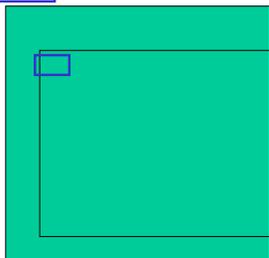
3/6/2007

20/43

The picture abstraction

```
(define (make-picture seglist)
  (lambda (rect)
```

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 its effect

let* is like let, except the names are defined in sequence, so m can be used in the expressions for b2 and e2

3/6/2007

21/43

The picture abstraction

```
(define (make-picture seglist)
  (lambda (rect)
    (for-each
     (lambda (segment)
       (let* ((b (start-segment segment))
              (e (end-segment segment))
              (m (coord-map rect)
                    (b2 (m b))
                    (e2 (m e)))
              (draw-line (xcor b2) (ycor b2)
                          (xcor e2) (ycor e2)))
         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 its effect

let* is like let, except the names are defined in strict sequence, so m is available in the expressions for b2 and e2

3/6/2007

22/43

A better George

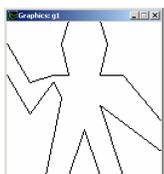
Remember we have george-lines from before

So here is George!

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

```
(define origin1 (make-vect 0 0))
(define x-axis1 (make-vect 100 0))
(define y-axis1 (make-vect 0 100))
(define frame1
  (make-rectangle origin1
                  x-axis1
                  y-axis1))
```

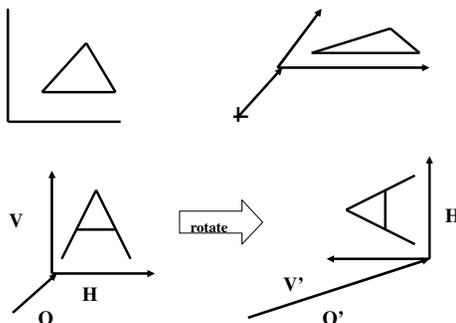
```
(george frame1)
```



3/6/2007

23/43

Operations on pictures



3/6/2007

24/43

Operations on pictures

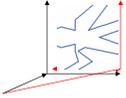
```

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

(define (rotate90 pict)
  (lambda (rect)
    (pict (make-rectangle
           (+vect (origin rect)
                  (x-axis rect))
               (y-axis rect)
               (scale-vect (x-axis rect) -1))))))

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

```

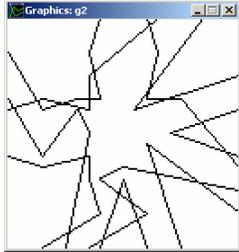
3/6/2007 25/43

A Georgian mess!

```

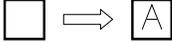
((together george (rotate90 george))
 frame1)

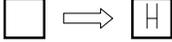
```



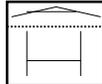
3/6/2007 26/43

Operations on pictures

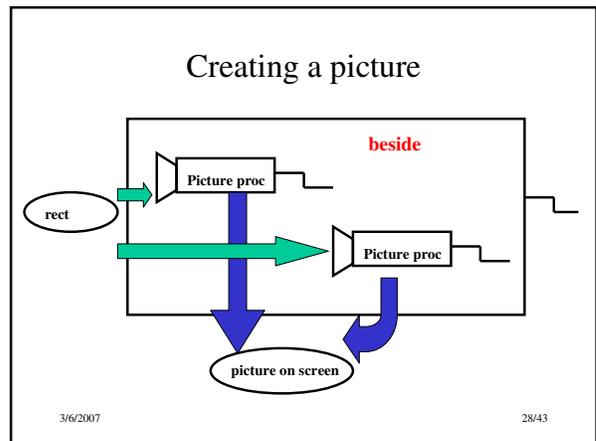
PictA:  → 

PictB:  → 

 **beside** → 

 **above** → 

3/6/2007 27/43



More procedures to combine pictures:

```

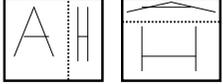
(define (beside pict1 pict2 a)
  (lambda (rect)
    (pict1
     (make-rectangle
      (origin rect)
      (scale-vect (x-axis rect) a)
      (y-axis rect)))
     (pict2
      (make-rectangle
       (+vect
        (origin rect)
        (scale-vect (x-axis rect) a)
        (scale-vect (x-axis rect) (- 1 a))
        (y-axis rect)))))))

(define (above pict1 pict2 a)
  ((repeated rotate90 3)
   (beside (rotate90 pict1)
            (rotate90 pict2)
            a))))

(define (repeated f n)
  (if (= n 1)
      f
      (compose
       f (repeated f (- n 1))))))

```

Picture operators have a closure property!



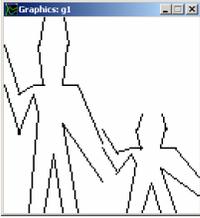
3/6/2007 29/43

Big brother

```

(define big-brother
  (beside george
    (above empty-picture george .5)
    .5))

```



3/6/2007 30/43

A left-right flip

```

(define (flip pict)
  (lambda (rect)
    (pict (make-rectangle
          (+vect (origin rect) (x-axis rect))
          (scale-vect (x-axis rect) -1)
          (y-axis rect))))))

```

3/6/2007 31/43

```

(define acrobats
  (beside george
    (rotatel80 (flip george))
    .5))

(define rotatel80 (repeated rotate90 2))

```

3/6/2007 32/43

```

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

```

3/6/2007 33/43

Recursive combinations of pictures

```

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

```

3/6/2007 34/43

Pushing George around

3/6/2007 35/43

Pushing George around

```

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

```

3/6/2007 36/43

Pushing George into the corner

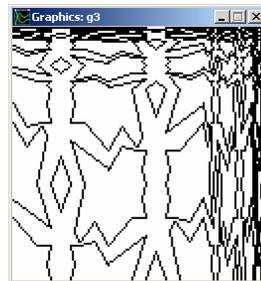
```
(define (corner-push pict n)
  (if (= n 0)
      pict
      (above
        (beside
          (up-push pict n)
          (corner-push pict (- n 1))
          .75)
        (beside
          pict
          (right-push pict (- n 1))
          .75)
        .25)))
```

3/6/2007

37/43

Pushing George into a corner

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



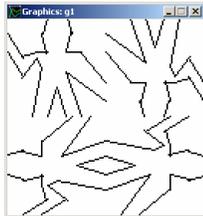
3/6/2007

38/43

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))
```

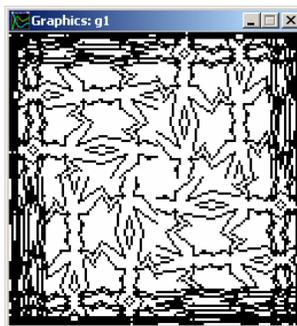


3/6/2007

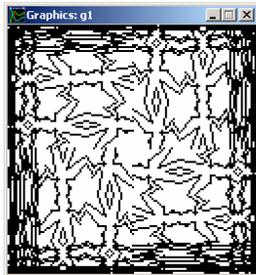
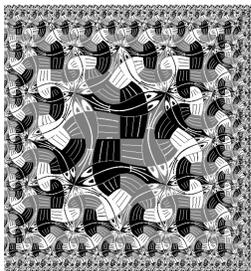
39/43

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

```
(square-limit 4bats 2)
```



3/6/2007



3/6/2007

41/43



3/6/2007

“Escher” is an embedded language

	Scheme	Scheme data	Picture language
Primitive data	3, #f, george	nil	george, mona, escher
Primitive procedures	+, map, ...		rotate90, flip, ...
Combinations	(p a b)	cons, car, cdr	together, beside, ..., and Scheme mechanisms
Abstraction			
Naming	(define ...)	(define ...)	(define ...)
Creation	(lambda ...)	(lambda ...)	(lambda ...)

3/6/2007

43/43