Slide 1

6.001: Structure and Interpretation of Computer Programs

- Capturing common patterns across procedures
- Capturing common patterns across data structures
- Application: using higher order procedures and common patterns to create a new language

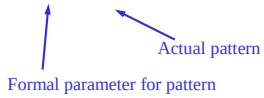
Slide 2

What is procedure abstraction?

Capture a common pattern

(* 2 2)
(* 57 57)
(* k k)

(lambda (x) (* x x))



Give it a name (define square (lambda (x) (* x x)))

Note the type: number → number

Slide 3

Other common patterns

- 1 + 2 + … + 100 = (100 * 101)/2 $\sum_{k=1}^{100} k$
- 1 + 4 + 9 + … + 100^2 = (100 * 101 * 201)/6 $\sum_{k=1}^{100} k^2$
- 1 + $1/3^2$ + $1/5^2$ + … + $1/101^2$ = $\pi^2/8$ $\sum_{k=1,odd}^{101} k^{-2}$

```
(define (sum-integers a b)
   (if (> a b)
      0
      (+ a (sum-integers (+ 1 a) b))))
(define (sum-squares a b)
   (if (> a b)
      0
      (+ (square a)
         (sum-squares (+ 1 a) b))))
(define (pi-sum a b)
   (if (> a b)
      0
      (+ (/ 1 (square a))
         (pi-sum (+ a 2) b))))
```

```
(define (sum term a next b)
  (if (> a b)
     0
     (+ (term a)
        (sum term (next a) next b))))
```

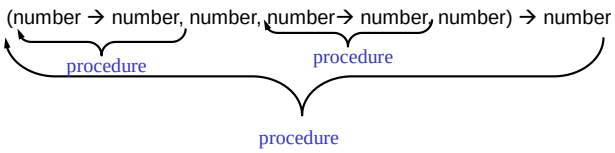
Slide 4

Let's check this new procedure out!

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

A higher order procedure!!

What is the type of this procedure?



Slide 5

Higher order procedures

- A higher order procedure:
  takes a procedure as an argument or returns one as a value

```
(define (sum-integers1 a b)
        (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))
(define (sum-squares1 a b)
        (sum square a (lambda (x) (+ x 1)) b))
(define (pi-sum1 a b)
     (sum (lambda (x) (/ 1 (square x))) a
          (lambda (x) (+ x 2)) b))
```

```
(define (sum term a next b)
 (if (> a b)
    0
    (+ (term a)
       (sum term (next a) next b))))
```

Slide 6

Computing derivatives

$f : x \to x^2$ $\qquad$ $f : x \to x^3$

$Df : x \to 2x$ $\qquad$ $Df : x \to 3x^2$

We can easily write f in either case:

(define f (lambda (x) (* x x x)))

But what is D??

Slide 7

Computing derivatives

$f : x \rightarrow x^2$ $\qquad$ $f : x \rightarrow x^3$

$Df : x \rightarrow 2x$ $\qquad$ $Df : x \rightarrow 3x^2$

But what is D??

- maps a function (or procedure) to a different function
- here is a good approximation:

$$Df(x) \approx \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

Slide 8

Computing derivatives

$$f : x \to x^2 \qquad Df : x \to 2x$$

$$Df(x) \approx \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

```
(define deriv
  (lambda (f)
    (lambda (x) (/ (- (f (+ x epsilon)) (f x))
                   epsilon))))
```

(number → number) → (number→ number)

Slide 9

Using “deriv”

```
(define square (lambda (y) (* y)))
(define epsilon 0.001)

((deriv square) 5)
```

```
(define deriv
  (lambda (f)
    (lambda (x) (/ (- (f (+ x epsilon)) (f x))
                   epsilon))))
```

```
( (lambda (x) (/ (- ((lambda (y) (* y y)) (+ x epsilon) )
                    ((lambda (y) (* y y)) x) ) )
                 epsilon))
  5 )

(/ (- ((lambda (y) (* y y)) (+ 5 epsilon) )
      ((lambda (y) (* y y)) 5) ) )
   epsilon))

10.001
```

Slide 10

Common Pattern #1: Transforming a List

```
(define (square-list lst)
  (if (null? lst)
      nil
      (cons (square (car lst))
            (square-list (cdr lst)))))

(define (double-list lst)
  (if (null? lst)
      nil
      (cons (* 2 (car lst))
            (double-list (cdr lst)))))

(define (MAP proc lst)
  (if (null? lst)
      nil
      (cons (proc (car lst))
            (map proc (cdr lst)))))

(define (square-list lst)
  (map square lst))

(define (double-list lst)
  (map (lambda (x) (* 2 x)) lst))
```

Slide 11

Common Pattern #2: Filtering a List

```
(define (keep-it-odd lst)
 (cond ((null? lst) nil)
       ((odd? (car lst))
        (cons (car lst) (keep-it-odd (cdr lst)))
       (else (keep-it-odd (cdr lst)))))

 (define (filter pred lst)
   (cond ((null? lst) nil)
         ((pred (car lst))
          (cons (car lst)
                (filter pred (cdr lst))))
         (else (filter pred (cdr lst)))))
```

Slide 12

Common Pattern #3: Accumulating Results

```
(define (add-up lst)
  (if (null? lst)
      0
      (+ (car lst)
         (add-up (cdr lst)))))

(define (mult-all lst)
  (if (null? lst)
      1
      (* (car lst)
         (mult-all (cdr lst)))))

(define (REDUCE op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (reduce op init (cdr lst)))))

(define (add-up lst)
  (reduce + 0 lst))
```

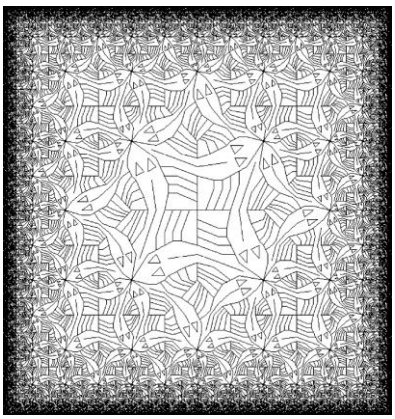
Slide 13

Themes to be integrated

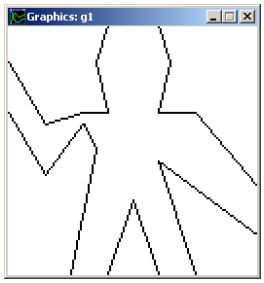
- Building a new language using data and procedure abstractions
- Data abstraction
  - Separate use of data structure from details of data structure
- Procedural abstraction
  - Capture common patterns of behavior and treat as black box for generating new patterns
- Means of combination
  - Create complex combinations, then treat as primitives to support new combinations
- Use modularity of components to create new language for particular problem domain

Slide 14

Our target – the art of M. C. Escher

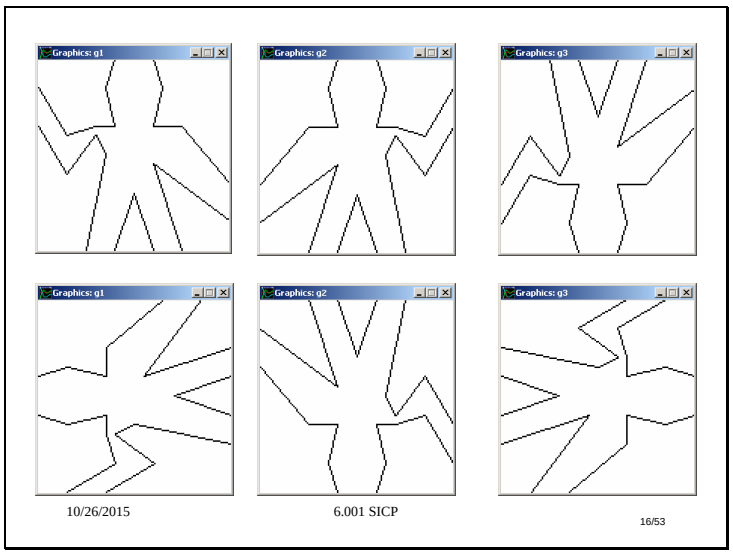


Slide 15



My buddy George

Slide 16


Graphics: g1
Graphics: g2
Graphics: g3
Graphics: g1
Graphics: g2
Graphics: g3
10/26/2015
6.001 SICP
16/53

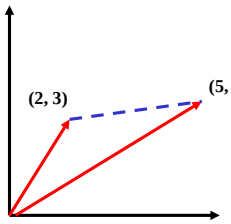
Slide 17

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 .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))
```

Slide 18

Data abstractions 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

Slide 19

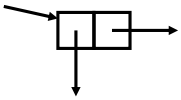
A better George

```
(define george
   (list (make-segment (make-vect .25 0) (make-vect .35 .5))
         (make-segment (make-vect .35 .5 ) (make-vect .3 .6))
         (make-segment (make-vect .3 .6) (make-vect .15 .4))
         (make-segment (make-vect .15 .4) (make-vect 0 .65))
         (make-segment (make-vect .4 0) (make-vect .5 .3))
         (make-segment (make-vect .5 .3) (make-vect .6 0))
         (make-segment (make-vect .75 0) (make-vect .6 .45))
         (make-segment (make-vect .6 .45) (make-vect 1 .15))
         (make-segment (make-vect 1 .35) (make-vect .75 .65))
         (make-segment (make-vect .75 .65) (make-vect .6 .65) )
         (make-segment (make-vect .6 .65) (make-vect .65 .85))
         (make-segment (make-vect .65 .85) (make-vect .6 1))
         (make-segment (make-vect .4 1) (make-vect .35 .85))
         (make-segment (make-vect .35 .85) (make-vect .4 .65))
         (make-segment (make-vect .4 .65) (make-vect .3 .65))
         (make-segment (make-vect .3 .65) (make-vect .15 .6))
         (make-segment (make-vect .15 .6) (make-vect 0 .85))))
```

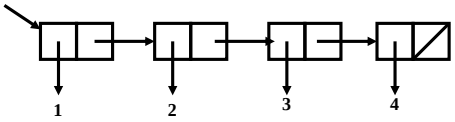
Slide 20

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 nil))))

Slide 21

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)
- Would be better to use **adjoin, first** and **rest,** instead of **cons, car** and **cdr.**

Slide 22

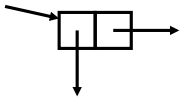
Completing our abstraction

Points or vectors:

(define make-vect cons)

(define xcor car)

(define ycor cdr)

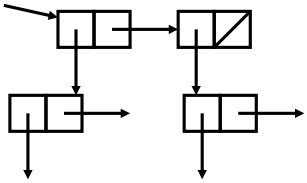


Line segments:

(define make-segment list)

(define start-segment car)

(define end-segment cadr)



Slide 23

What happens if we change an abstraction?

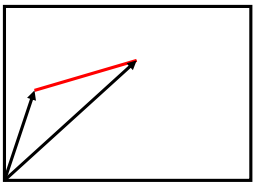
```
(define make-vect list)
(define xcor car)
(define ycor cadr)
```

Note that this still satisfies the contract

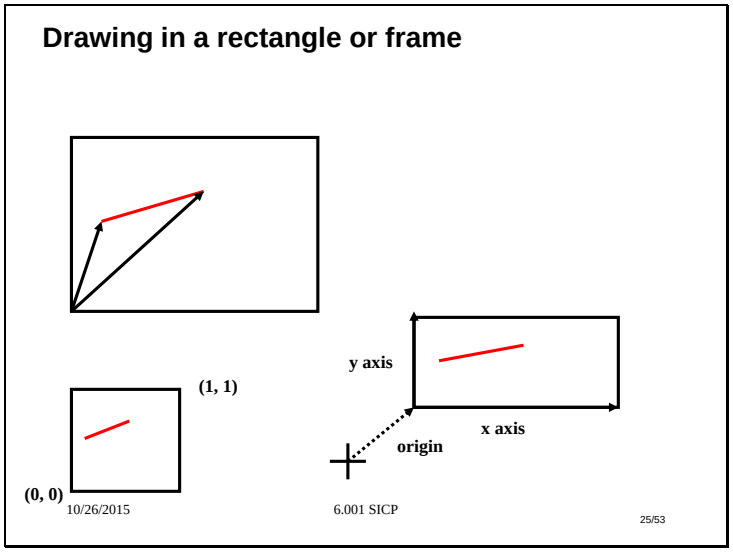
What else needs to change in our system? **BUPKIS, NADA, NOTHING**

Slide 24

Drawing in a rectangle or frame



Slide 25


Drawing in a rectangle or frame
(1, 1)
(0, 0)
y axis
x axis
origin
10/26/2015
6.001 SICP
25/53

Slide 26

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 >
  ))
```

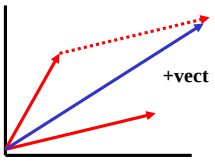
Slide 27

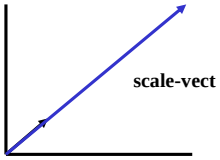
What is a picture?

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

Slide 28

Manipulating vectors





Slide 29

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

Slide 30

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

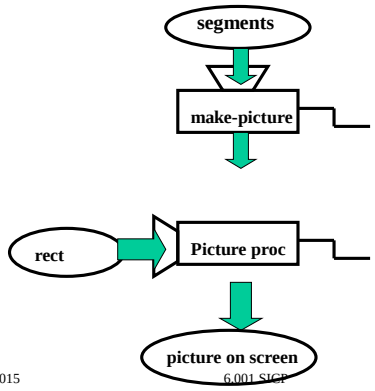
Select parts

Compute more primitive operation

Reassemble new parts

Slide 31

Creating a picture



Slide 32

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

Slide 33

A better George

```
(define g-lines
  (list (list .25 0 .35 .5) (list .35 .5 .3 .6) … (list .15 .6 0 .85)))

(define (prim-pict list-of-lines)
  (make-picture (create-lines list-of-lines)))

(define (create-lines list-of-lines)
  (map (lambda (line)
         (make-segment (make-vect (car line) (cadr line))
                       (make-vect (caddr line) (cadddr line))))
       list-of-lines))
(define george (prim-pict g-lines))
```

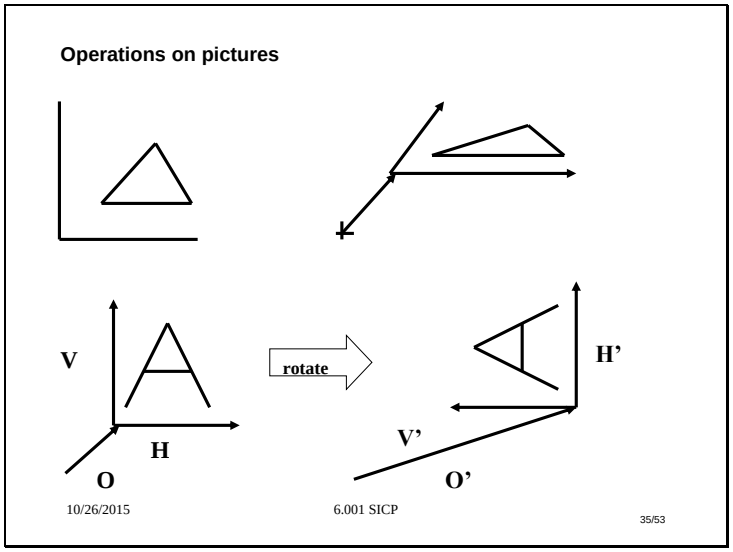
Slide 34

Defining George in terms of data lets us do other things with him, e.g., how much ink do we need to draw him?

```
(define (ink seglist)
  (define (add-em l)
    (reduce + 0 l))

 (define (seg-length seg)
   (magnitude (-vect (end-segment seg) (start-segment seg))))

 (add-em (map seg-length seglist)))

(define (magnitude v)
   (sqrt (+ (square (xcor v))
            (square (ycor v)))))
```

For George, it happens to be 4.59

Slide 35


Operations on pictures
V
H
O
rotate
H'
V'
O'
10/26/2015
6.001 SICP
35/53

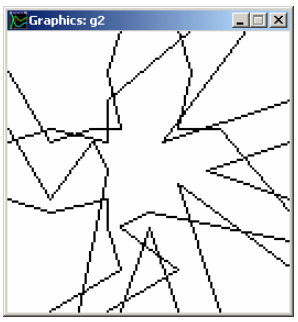
Slide 36

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)))
```

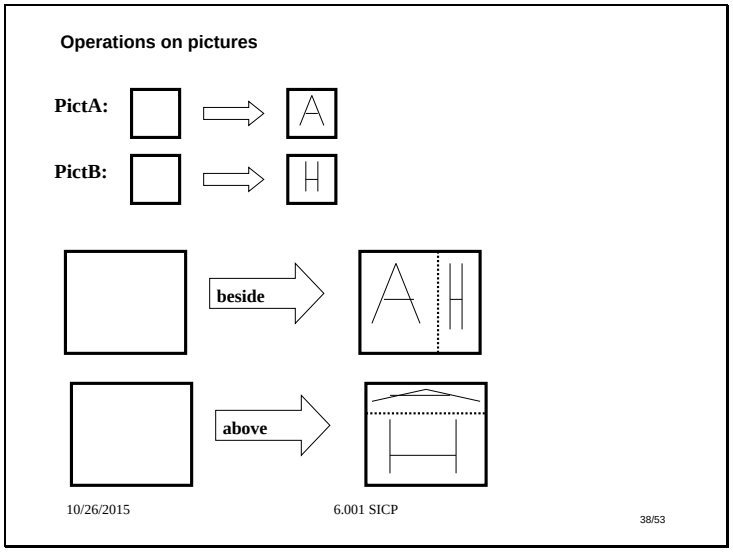
Slide 37

A Georgian mess!

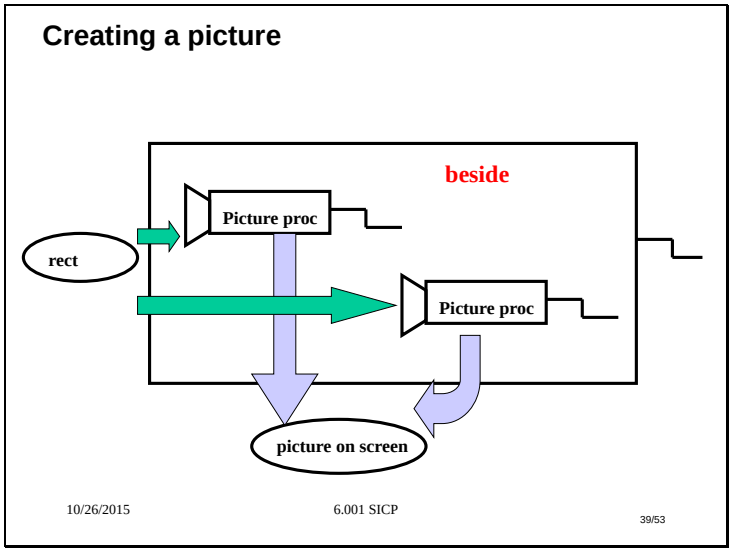



```
(draw (together
        g
        (rotate90 g)))
```

Slide 38


Operations on pictures
PictA:
PictB:
beside
above
10/26/2015
6.001 SICP
38/53

Slide 39


Creating a picture
beside
Picture proc
rect
Picture proc
picture on screen
10/26/2015
6.001 SICP
39/53

Slide 40

More procedures to combine pictures:

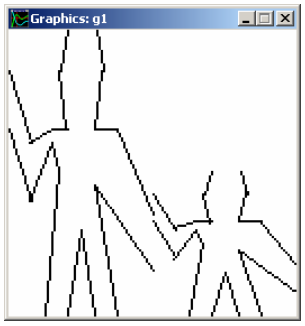
```
(define (beside pict1 pict2 a)
  (lambda (rect)
    (pict1
      (make-rectangle
        (origin rect)
        (scale-vect (horiz rect) a)
        (vert rect)))
    (pict2
      (make-rectangle
        (+vect
          (origin rect)
          (scale-vect (horiz rect) a))
        (scale-vect (horiz rect) (- 1 a))
        (vert rect)))))
```

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

Pictures have a closure property!

Slide 41

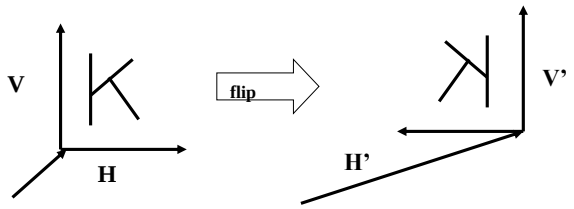
Big brother




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

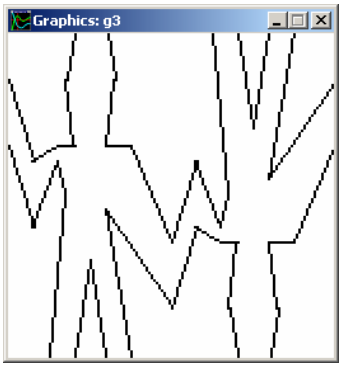
Slide 42

A left-right flip



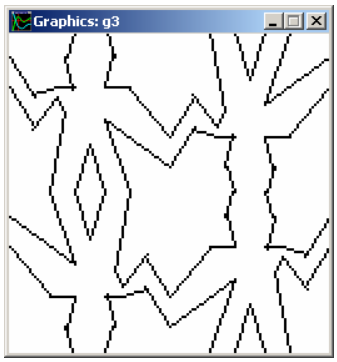

```
(define (flip pict)
  (lambda (rect)
    (pict (make-rectangle
            (+vect (origin rect) (horiz rect))
            (scale-vect (horiz rect) –1)
            (vert rect)))))
```

Slide 43




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

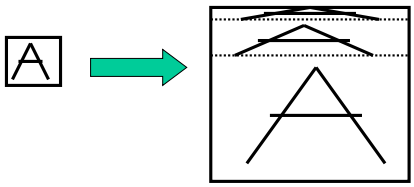
Slide 44




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

Slide 45

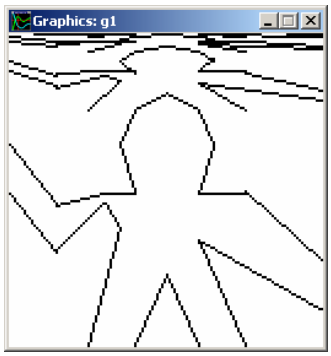
Recursive combinations of pictures



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

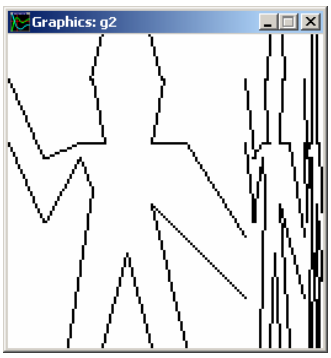
Slide 46

Pushing George around



Slide 47

Pushing George around

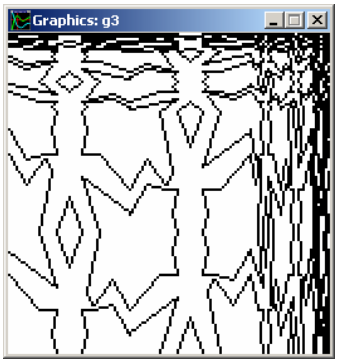



```
(define (right-push pict n)
  (if (= n 0)
      pict
      (beside pict
              (right-push pict (- n 1))
              .75)))
```

Slide 48

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)))
```

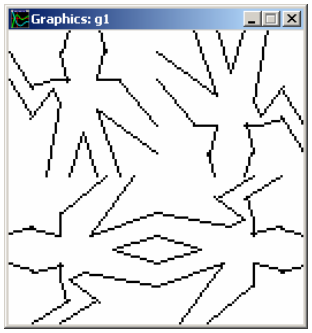


(corner-push 4bats 2)

Slide 49

Putting copies together

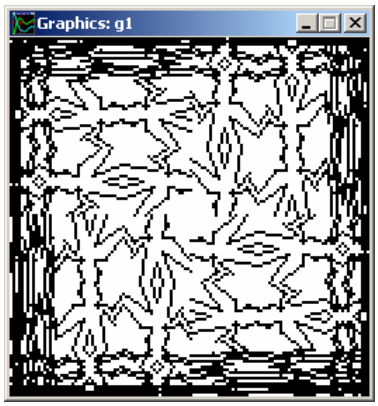
```
(define (4pict p1 r1 p2 r2 p3 r3 p4 r4)
  (beside
    (above
      ((repeated rotate90 r1) p1)
      ((repeated rotate90 r2) p2)
      .5)
    (above
      ((repeated rotate90 r3) p3)
      ((repeated rotate90 r4) p4)
      .5)
    .5))
(define (4same p r1 r2 r3 r4)
  (4pict p r1 p r2 p r3 p r4))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



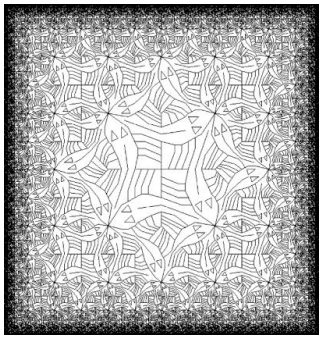
(4same g 0 1 2 3)

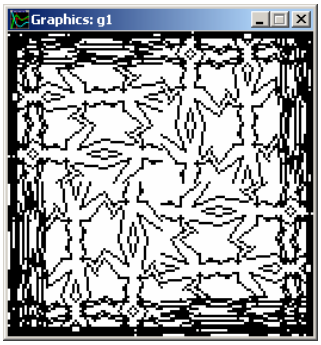
Slide 50

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

(square-limit 4bats 2)
```

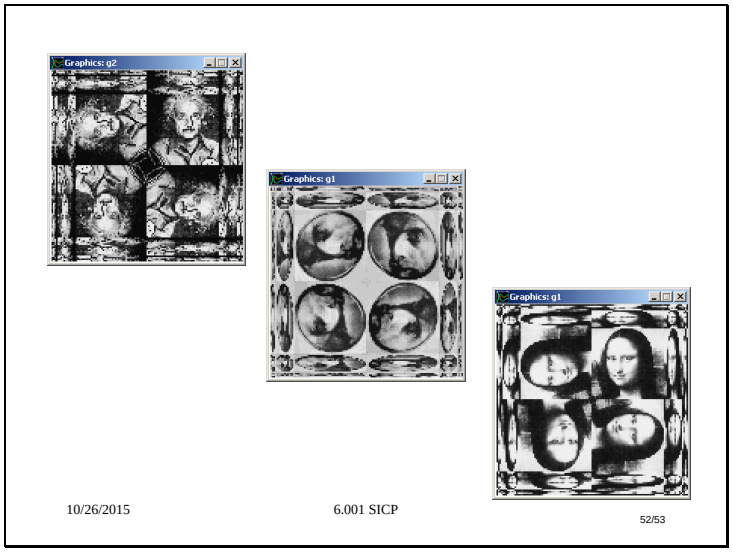


Slide 51





Slide 52


Graphics: g2
Graphics: g1
Graphics: g1
10/26/2015
6.001 SICP
52/53

Slide 53

"Escher" is an embedded language

	Scheme	Scheme data	Picture language
Primitive data	3, #f, george	Nil	Half-line, George, other pictures
Primitive procedures	+, map, …		Rotate90, …
Combinations	(p a b)	Cons, car, cdr	Together, beside, …, And Scheme mechanisms
Abstraction Naming Creation	 (define …) (lambda …)	 (define …) (lambda …)	 (define …) (lambda …)