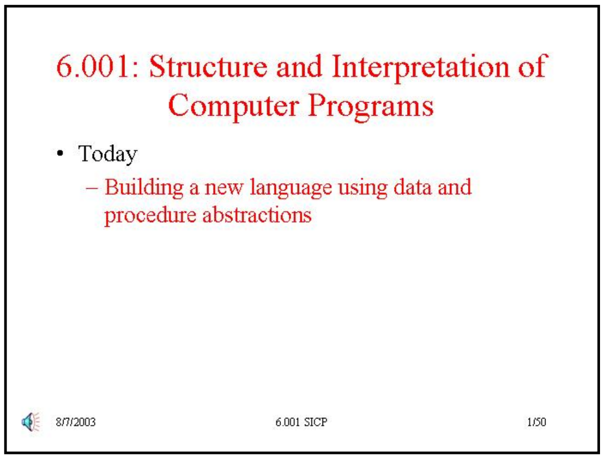
6.001 Notes: Section 30.1

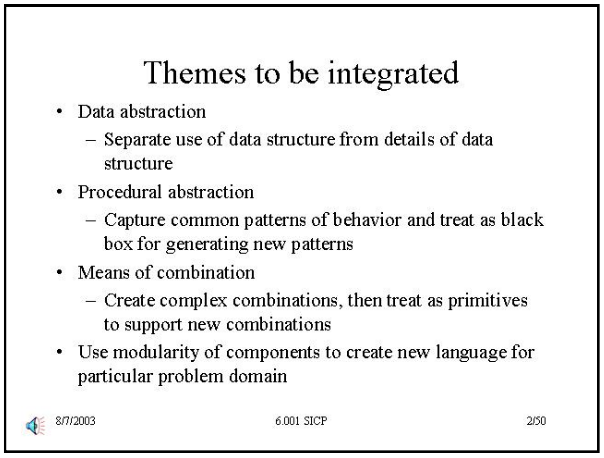
Slide 30.1.1
In this lecture, we are going to go back to several themes that we have been exploring over the past few weeks, and stitch them together into a single demonstration. We are going to see how quickly we can describe and control a complex system, by using the tools we have been building for dealing with abstractions.



6.001: Structure and Interpretation of Computer Programs

- Today
  - Building a new language using data and procedure abstractions

8/7/2003 6.001 SICP 1/50



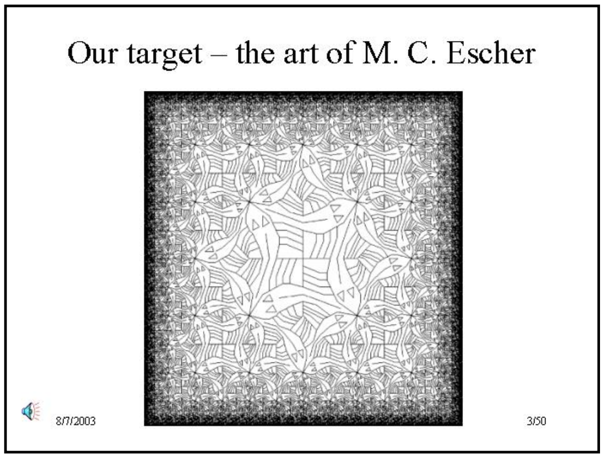
Themes to be integrated

- 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

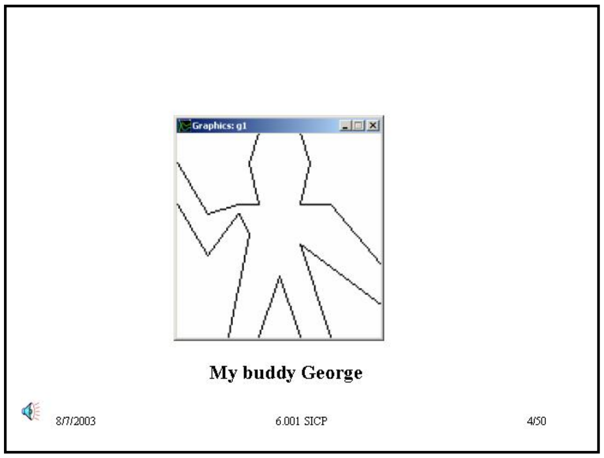
8/7/2003 6.001 SICP 2/50

Slide 30.1.2
The themes we are going to utilize include the following.
We are going to build on the idea of data abstraction, especially the idea of separating the use of a data structure from the details of its implementation. We are going to see how that abstraction barrier enables us to quickly describe complex structures without getting lost in their details, and how to focus on the use of such structures as entire units, while being assured that the interior details will be handled correctly.
We are going to also build on the idea of procedural abstraction, especially the idea of capturing common patterns inside a black box, and using such abstractions to capture more complex patterns.
And we are going to build on the idea of means of combination, that is, the idea that we can create simple methods for combining primitive objects into complex things, then treating the result as a primitive within a still more complex thing. This will allow us to control complexity by utilizing a modular decomposition of the problem domain.
So our goal is to pull these pieces together to build a new language, one that is specifically designed for a particular problem domain.

Slide 30.1.3
So what problem domain should we use? Well, we are going to create a language that describes pictures such as this famous one by M.C. Escher, called "quadratlimit" or "square limit". Not only will our language let us describe the process by which such pictures can be created, it will also let us create our own variations on this theme, leading to pictures that have a resemblance to the kind of elegant structure shown here in this Escher print.


Our target – the art of M. C. Escher

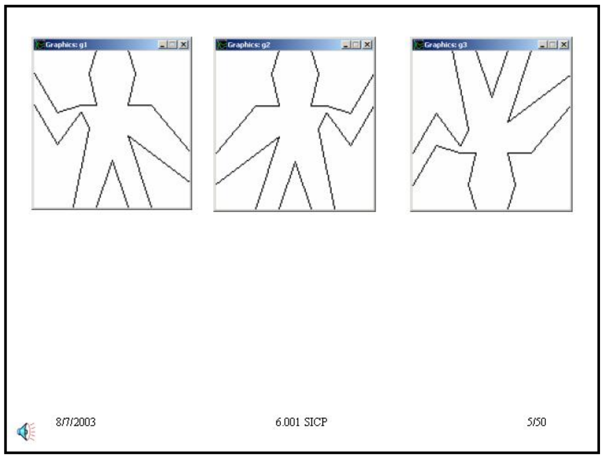
8/7/2003 3/50

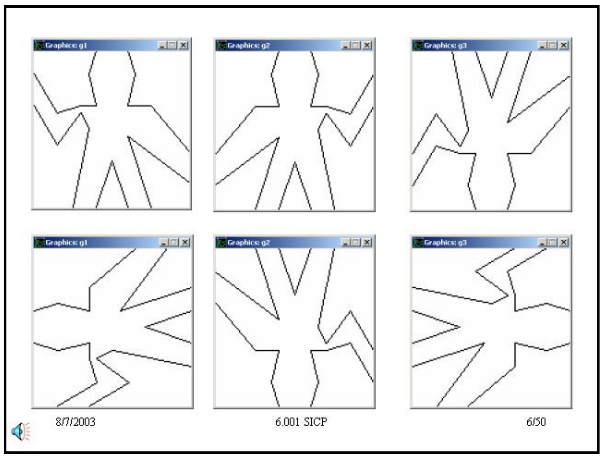

My buddy George

8/7/2003 6.001 SICP 4/50

Slide 30.1.4
So how do we describe such a system? Well, let's start with a simpler example. Here is a picture of my friend George. At an abstract level, what kinds of things would I like to do with George?

Slide 30.1.5
First, I might like to flip him, either about the vertical axis or about the horizontal one. By this I mean literally taking this portrait of George and spinning it 180 degrees out of the plane, then setting it back down.


8/7/2003 6.001 SICP 5/50

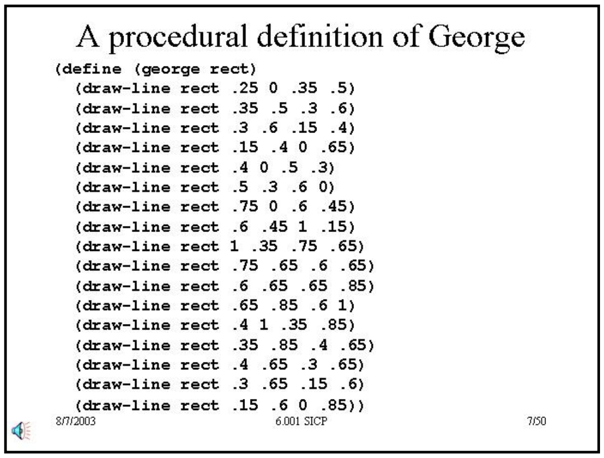

8/7/2003 6.001 SICP 6/50

Slide 30.1.6
Alternatively, I might like to rotate him about an axis coming out the picture, causing him to do a cartwheel as I rotate his picture by increments of 90 degrees. Conceptually this is easy. If I think of George as a picture, I can easily envision grabbing the whole picture and doing something to it. But how do I do this in practice?

Slide 30.1.7
Here is one straightforward way. Assume that we have some primitive drawing method called

`draw-line`,

that takes a rectangle as input, and a set of coordinate values (e.g. x and y start point, and x and y end point), and draws a line from start to end within that rectangle. Note, details of the rectangle are irrelevant, we are just burying them beneath the abstraction. Thus, the first expression would draw a line starting .25 units to the right of the lower left hand corner, and ending at a point .35 units to the right and .5 units up from the lower left hand corner.



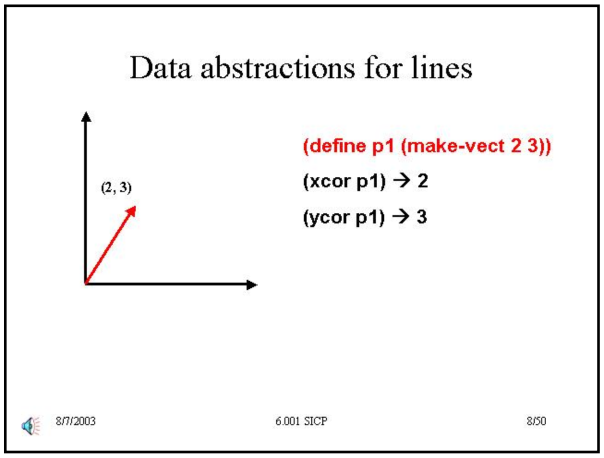
A procedural definition of George

```
(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))
```

8/7/2003 6.001 SICP 7/50

Given that, here is a definition of George. Note what this does. Each expression gives a start and end point, relative to the origin of the rectangle, and `draw-line` then sketches the line between those points.

Blech! This is far too specific, right? Given this definition of George, how do I create a rotated George or a flipped George? This is not obvious, and for an important reason. Here I have intertwined the action of drawing with the data representing George. I have not separated those actions, and moreover, I have chosen a very low-level representation for the elements of George. I really need to isolate these two aspects if I am to have any hope of drawing different pictures of George.



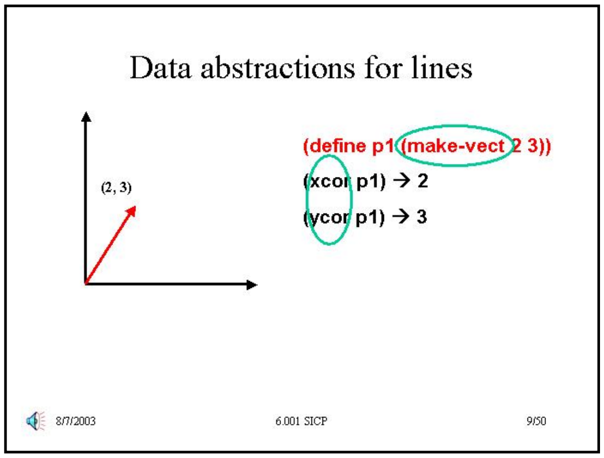
Data abstractions for lines

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

8/7/2003 6.001 SICP 8/50

Slide 30.1.8
So let's fix this. First, we need some data abstractions, to isolate points from the use of points. What is a point or a vector? It is just a way of gluing together an x coordinate and a y coordinate, so we can create an abstraction for this. This has a constructor (called `make-vect`) and two selectors (called `xcor` and `ycor`).

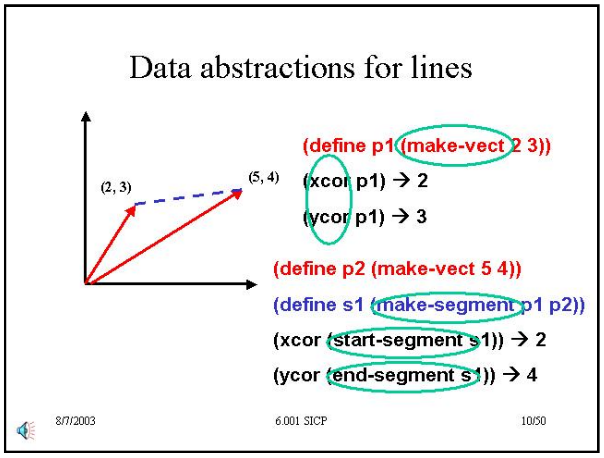
Slide 30.1.9
Note that there is an inherent contract between these two components: whatever method we use to glue things together in the constructor, we can get the parts back out using the selectors, but the details of how we do that don't matter to things that simply want to use these objects.



Data abstractions for lines

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

8/7/2003 6.001 SICP 9/50



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
```

8/7/2003 6.001 SICP 10/50

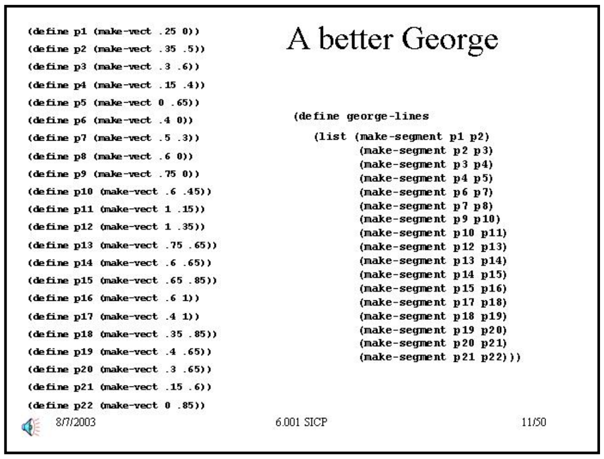
Slide 30.1.10

Similarly, we can glue two endpoints or vectors together to create a line segment. This again is a data abstraction, with a contract between the constructor (`make-segment`) and the selectors (`start-segment` and `end-segment`).

So this gives us a way of abstracting vectors and segments. Note the key point: I don't need to know the details of how vectors and segments are built, I just rely on the contract. This means we could think of George in terms of the appropriate elements, namely lines, rather than details of how those lines are represented.

Slide 30.1.11

So here is George in this format. Now, it looks like we have just put some window dressing around the line segments. But hang on, as we will see how treating George as an abstraction is going to make life much easier for us. In particular, note that here we have created an abstraction for the points, and a separate abstraction for the line segments. Moreover, these are now defined with respect to some coordinate frame; they are not actually being drawn yet. So we have also separated the act of drawing from the representation of the data to be drawn.

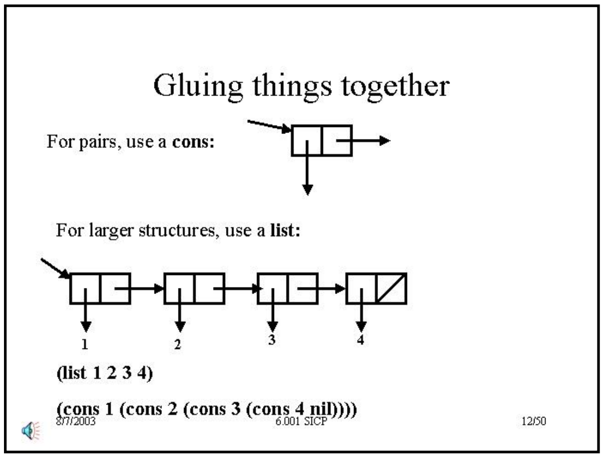


A better George

```
(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)))
```

8/7/2003 6.001 SICP 11/50



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))))

8/7/2003 6.001 SICP 12/50

Slide 30.1.12

First though, how could we actually build these vectors and segments?

Well, you saw this in the last lecture. For pairs of things (i.e. things that come naturally in twos), we could just use a cons cell or a pair. And for larger collections, we could use lists. And of courses lists are just sequences of cons cells glued together into a spine, with the elements hanging off of it.

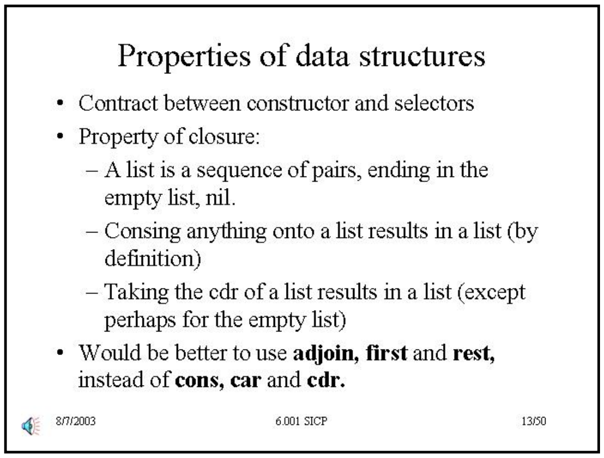
Slide 30.1.13

Remember that there are several important properties to pairs and lists. They have a contract between constructor and selectors. They have the property of **closure**, that is, that the result of creating an instance of an object can itself be used to create a new object.

This is worth exploring a bit more carefully. So what is a list? It is a sequence of pairs, ending in the special symbol nil, or empty list. Thus, consing anything onto a list gives you a new sequence of pairs, ending in the empty list, hence a list. Similarly, taking the cdr of a sequence of pairs ending in the empty list results in a shorter sequence of pairs ending in the empty list, and hence is a list. Thus lists are closed under the operations of cons and cdr.

Note that this is not quite right. What happens if you try to take the cdr of nil? In MIT Scheme you get an error, but by this definition it would be better to just return the empty list.

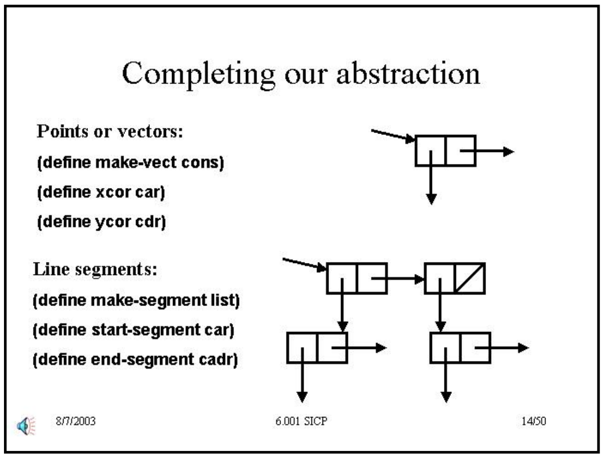
Also notice that it would really be better to have distinctive operations for lists, as compared to pairs, e.g. adjoin, first and rest, instead of cons, car and cdr, to distinguish operations on lists from operations on pairs. For historical reasons, we stick with the latter, even though the better would be conceptually much cleaner.



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.**

8/7/2003 6.001 SICP 13/50



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)

8/7/2003 6.001 SICP 14/50

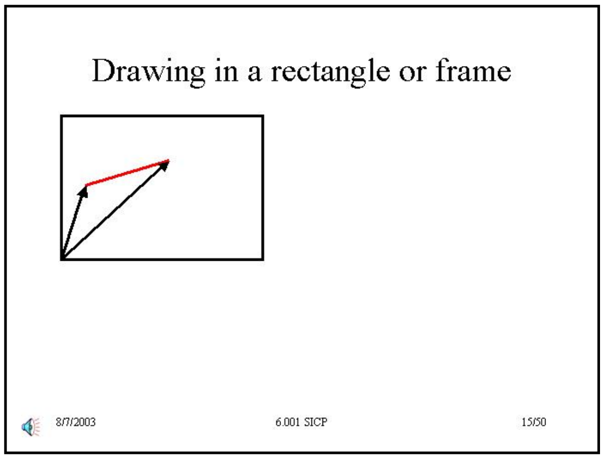
Slide 30.1.14

So let's use this to build a specific version of our abstraction. Note how our abstraction nicely inherits its contract from the underlying contract for pairs and lists. And note how points create a nested structure underneath lines. That is a line segment is a list of two elements, each of which points to another abstraction, namely a pair representing a vector.

Slide 30.1.15

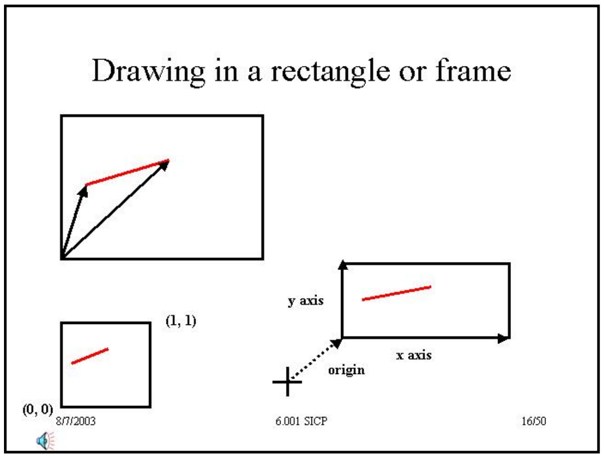
Okay, now let's put the pieces together. George is just defined as a collection (a list in this case) of line segments. All we need to do is take a rectangle or frame (which is just a pair of orthogonal line segments), and draw these segments within that rectangle.

But as we start thinking towards the kinds of operations we did on George earlier, we would like to be able to draw George in different frames. So we would like to be able to define any frame, either of different size, or even non-orthogonal, and draw George inside of it. What does that mean? Ideally, we could take George (defined as a set of segments within a frame) and stretch those segments to fit within a new frame.



Drawing in a rectangle or frame

8/7/2003 6.001 SICP 15/50



Drawing in a rectangle or frame

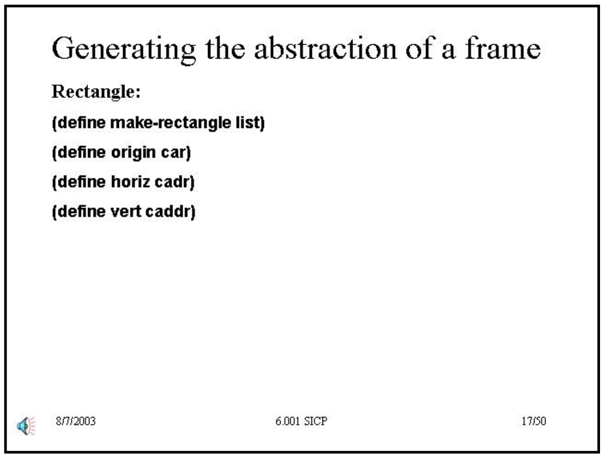
Slide 30.1.16

Here is a nice way to build that abstraction. We'll create our picture within the constraints of a default rectangle, of size 1. That is, our initial set of segments will have the property that their x and y values all lie between 0 and 1. Imagine those segments being attached to a sheet of rubber that fits over a square frame as shown in the lower left.

Then, if we provide some other rectangle, which may be shifted over from the first, and which may have a different aspect ratio, we want our method for drawing to take that sheet of rubber and stretch it to fit over the new rectangle's frame.

Slide 30.1.17

Well that is fine, we just need another data structure. Note that this one is built out of vectors, so we are constructing another abstraction within our system. In particular, a rectangle is now defined to have a vector to its origin or starting point, and two vectors that specify the extent of the horizontal and vertical axes of the frame.

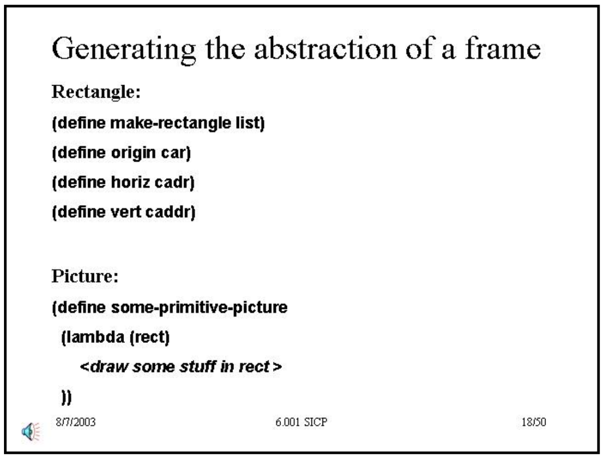


Generating the abstraction of a frame

Rectangle:

```
(define make-rectangle list)
(define origin car)
(define horiz cadr)
(define vert caddr)
```

8/7/2003
6.001 SICP
17/50



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>
  ))
```

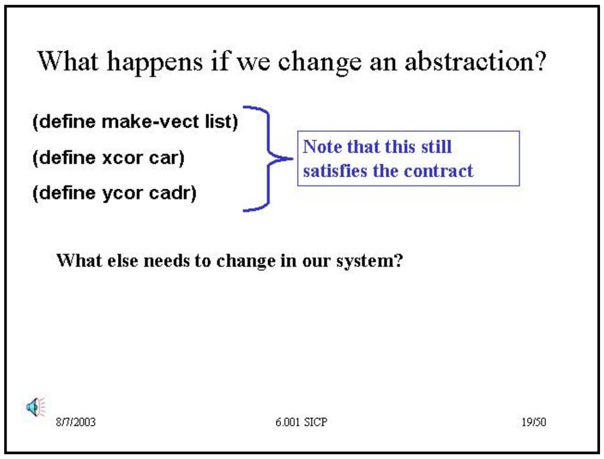
8/7/2003
6.001 SICP
18/50

Slide 30.1.18

And then a picture is just a procedure that takes a rectangle (one of these abstractions) and does some stuff to draw lines within that rectangle. We will return shortly to what this actually does.

Slide 30.1.19

Now a key issue in building data abstractions is that it should insulate the details of the abstraction from the actual use of the abstraction. To stress this, suppose we make the following change to our data structures: we change `make-vect` to `list`, and we change `ycor` to `cadr`, which is short for `(car (cdr .))`. Note that is version still satisfies our contract on the abstraction. What else has to change inside of our system?



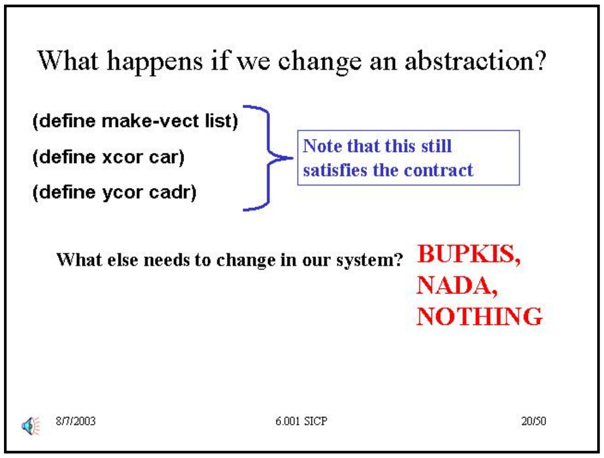
What happens if we change an abstraction?

```
(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?

8/7/2003
6.001 SICP
19/50



What happens if we change an abstraction?

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

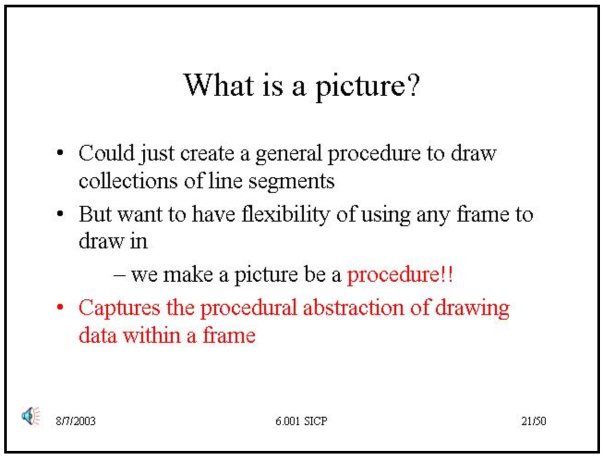
Note that this still satisfies the contract

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

8/7/2003 6.001 SICP 20/50

Slide 30.1.20
Absolutely nothing!
This is a key point about the data abstractions. The modularity of the abstraction isolates changes within the abstraction from the use of the abstraction. Hence any code that uses these abstractions will still run, even though the details within the abstraction have changed.

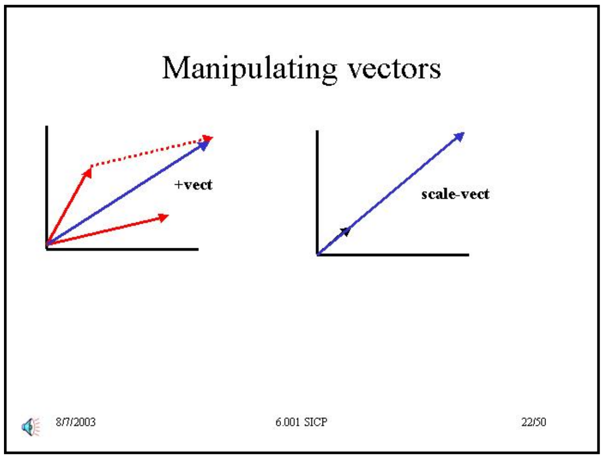
Slide 30.1.21
Okay, so we can create the pieces of George. But how do we actually draw?
This is where the weird part comes in! We could just create a procedure to draw line segments. But we want the flexibility of being able to use any frame to draw the same picture.
So we make a picture be a procedure!! This definitely sounds weird!! A picture sounds like it should be a data structure, a collection of geometric entities, but we are going to make it a procedure. Inside the procedure will reside those geometric entities, but that procedure will take as input a rectangle, then scale the elements to fit within that rectangle, and display the result.
This seems odd, right? In principle, a picture is data, but we are choosing to represent it as a procedural abstraction that captures the process of drawing data in a frame.
Why? Primarily for flexibility. In this way, we have one procedure with inherent data, but it provides an infinite number of versions of the picture. This abstraction allows for very easy manipulation of a picture structure to get new versions. So let's see how that happens!



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

8/7/2003 6.001 SICP 21/50

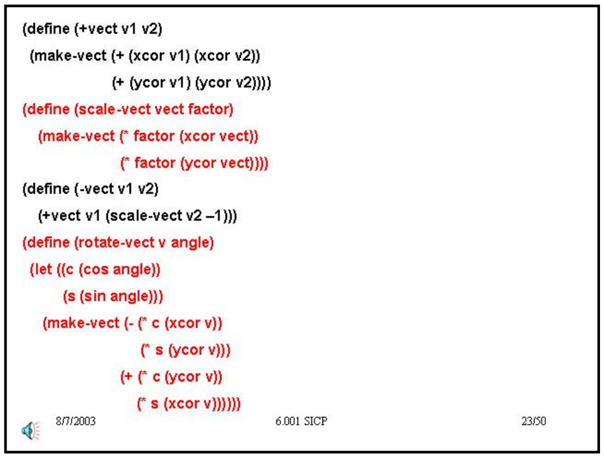


Manipulating vectors

8/7/2003 6.001 SICP 22/50

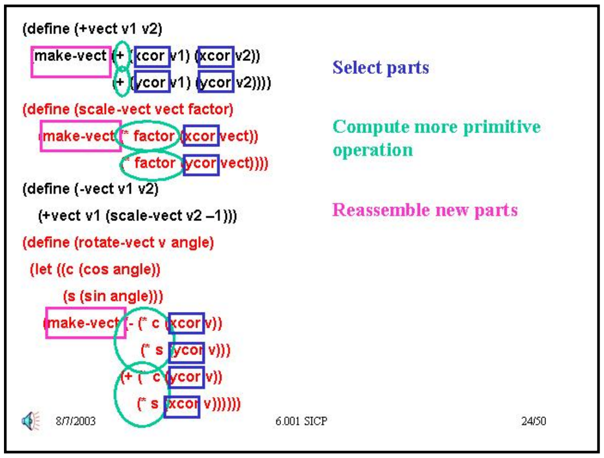
Slide 30.1.22
First, we need to manipulate the pieces of a picture, so that means we need ways to manipulate vectors themselves. Let's look at some standard things we would like to do with vectors. We would like to take two vectors and add them together to get a new vector. And we would like to stretch or scale a vector by scaling its x and y coordinates by the same amount.

Slide 30.1.23
So here is code to do this. For example, the first procedure takes two vectors and adds them together to get a new vector. It does this by extracting the x and y components of the vectors, adding them separately and then creating a new vector with those values for the components. The second procedure takes a vector and a number, and stretches (or shrinks) the vector by that amount.
Rotation is just a matter of applying some trigonometry to the vector to create a new vector.
A key thing to observe is how we inherit closure from the underlying representation. For example, in +vect, v1 could be the result of some other +vect operation.



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

8/7/2003 6.001 SICP 23/50



```
(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

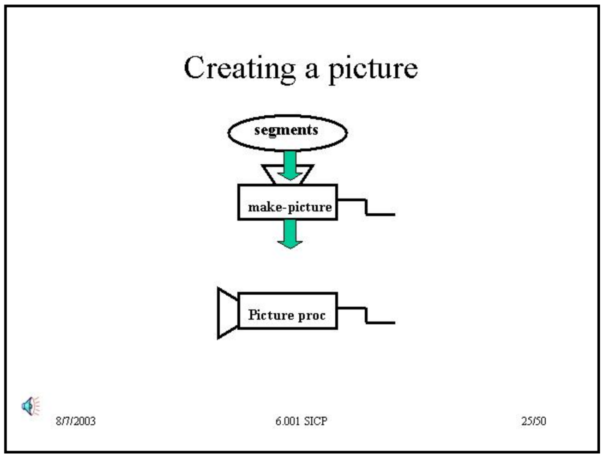
Compute more primitive operation

Reassemble new parts

8/7/2003 6.001 SICP 24/50

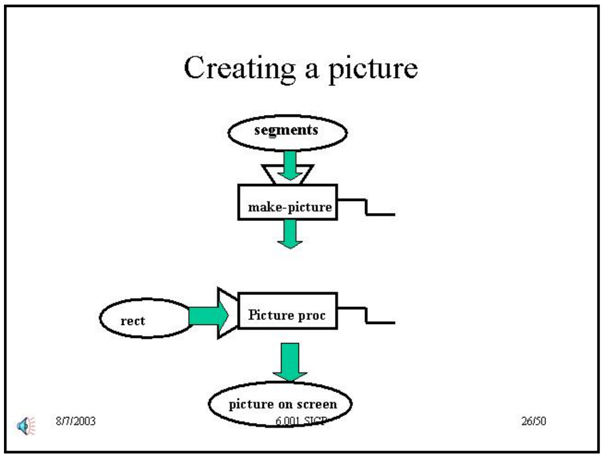
Slide 30.1.24
Also note the form: we extract the pieces, do some simpler operations, and then construct a new version of the same type of object.
And notice how this nicely isolates the changes beneath the abstraction barrier. If we change the abstraction implementation for vectors, none of these procedures need to change!!

Slide 30.1.25
Now we can use all these pieces to assemble a picture. The basic idea is to take a list of segments (defined as pairs of vectors using our nice data abstraction). This then gets passed to a procedure that generates a picture, that is, a new procedure with the data embedded within it.
Note that this `make-picture` procedure is a higher order procedure, it takes as input a list and creates as output a new procedure.



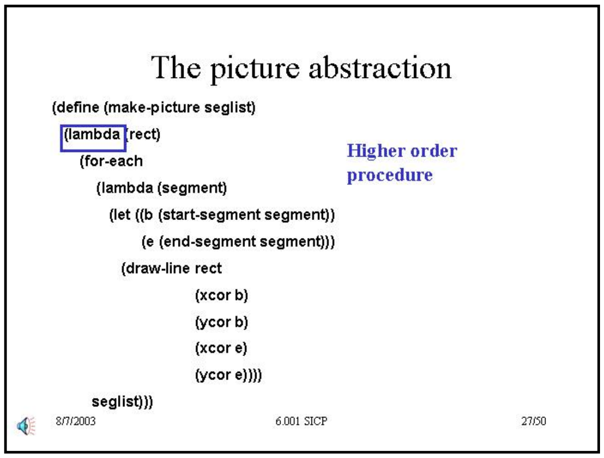
Creating a picture

8/7/2003 6.001 SICP 25/50


Creating a picture

Slide 30.1.26
To use a picture, we simply give it a rectangle (as a data abstraction) and the picture procedure will then display it on the screen inside that rectangle.

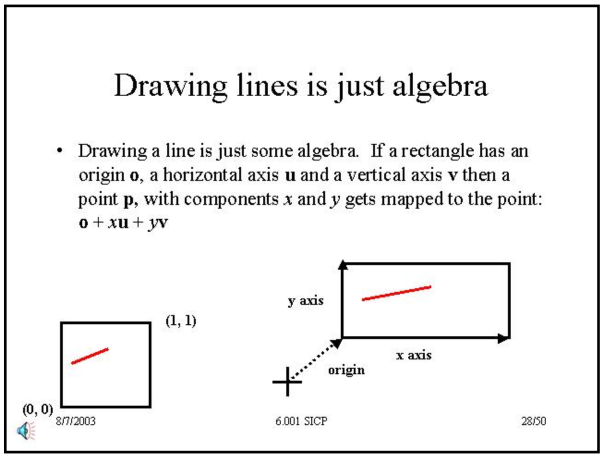
Slide 30.1.27
So here is the code for doing that.
Note that this is a higher order procedure: it takes a list as input, and returns a procedure as output. That procedure when given a rectangle as input, will ask each of the lines segments in the data structure embedded within the procedure to draw itself, appropriately scaled within the rectangle. `For-each` is just like `map`, except that it doesn't accumulate an answer as it walks down the list applying its internal lambda to each element.
The key thing to note is how we are using standard list operations to capture this procedural abstraction of 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

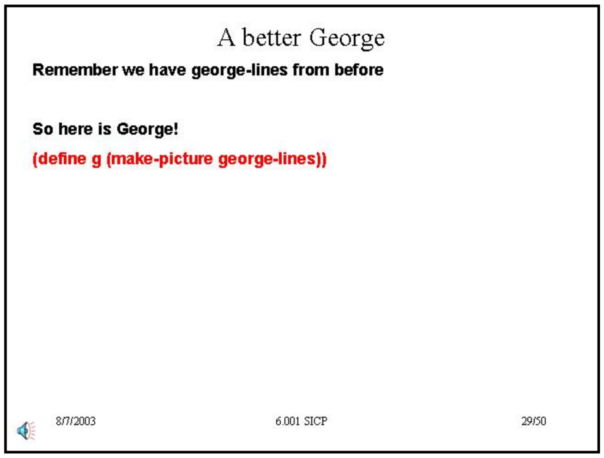
8/7/2003
6.001 SICP
27/50


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

Slide 30.1.28
Just to be careful, what should `draw-line` do? The idea is that this procedure is given a rectangle, which contains within it an origin vector, an x axis and a y axis. Draw-line takes an x and y coordinate value of a point (defined in a canonical rectangle), and scales the new horizontal and vertical axes by those amounts, and then shifts this by the offset to the origin, using the vector algebra shown. Doing this for two points automatically shifts and stretches a line to fit within the new rectangle.

Slide 30.1.29
So, just to complete this idea, here is a better definition of George.



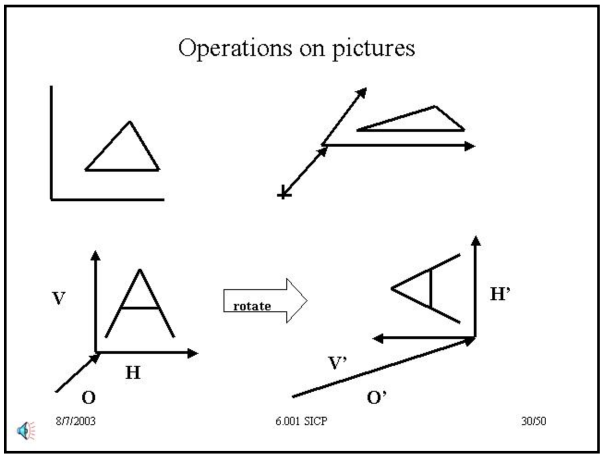
A better George

Remember we have george-lines from before

So here is George!

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

8/7/2003 6.001 SICP 29/50

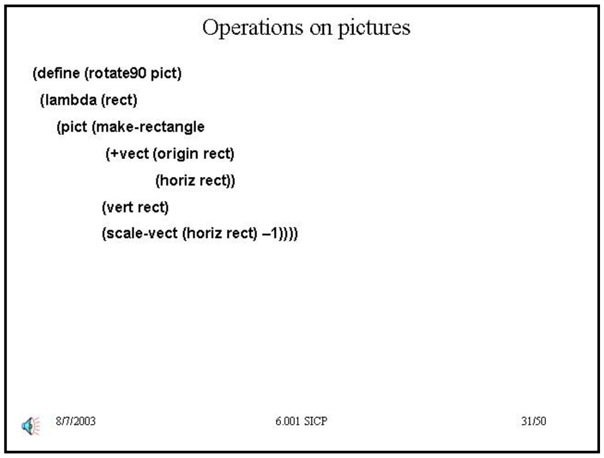


Operations on pictures

8/7/2003 6.001 SICP 30/50

Slide 30.1.30
So what is the big deal? Well, George is now both a data abstraction (`george-lines`: a set of segments) and a procedure (`g`: a process for drawing those lines within a rectangle). Note that `g` contains the information about the segments within it as part of the procedural abstraction
This makes it quite easy to use George as a building block in other pictures, and that is what we want to turn to next.
Again, remember what a picture is: it's a procedure that takes a rectangle as input, and scales its line segments to draw within the rectangle (or frame).
So we can easily generalize this idea to arbitrary frames, not just rectangular ones. Remember that a frame is just a set of three vectors, an origin and two axes, so by picking vectors for axes that are not orthogonal, we get skewing for free.
To rotate a picture, we can just shift the axes, in particular, make the old horizontal axis vertical, and the old vertical axis the negative horizontal axis. If we do this, then we can see that drawing the picture within this new coordinate frame will accomplish the task of rotating the original picture.

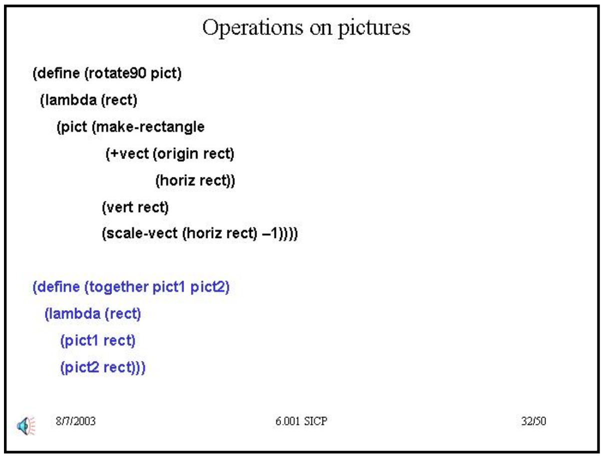
Slide 30.1.31
And we can easily build code to do this. Note what this does, in a very cool way. Rotate90 returns a picture, that is a procedure of one argument, a rectangle. That new picture asks the old picture to draw itself, but in a new frame. And that frame simply comes about by creating a new origin, a new horizontal axis and a new vertical axis, just as we sketched.
Also notice how nicely the data abstractions preserve the cleanliness of this code. It is very easy to see what is being done here.



Operations on pictures

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

8/7/2003 6.001 SICP 31/50


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)))
```

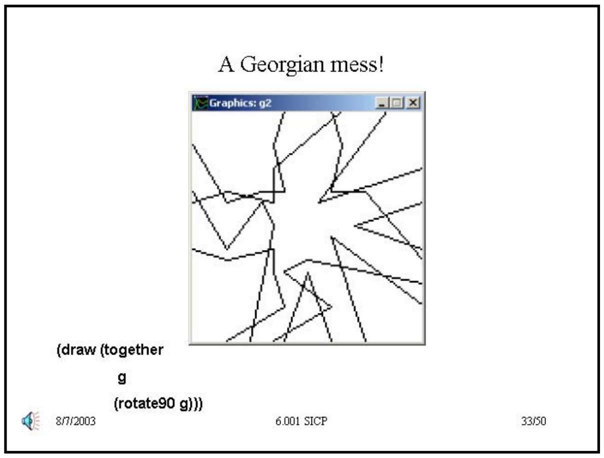
8/7/2003 6.001 SICP 32/50

Slide 30.1.32

Of course there is nothing that says we can only deal with a single picture. Together simply asks two pictures to draw themselves in the same frame. It does so since each picture is a procedure that draws within a rectangle, and by supplying the same rectangle to each picture, we will get a combination of the two.

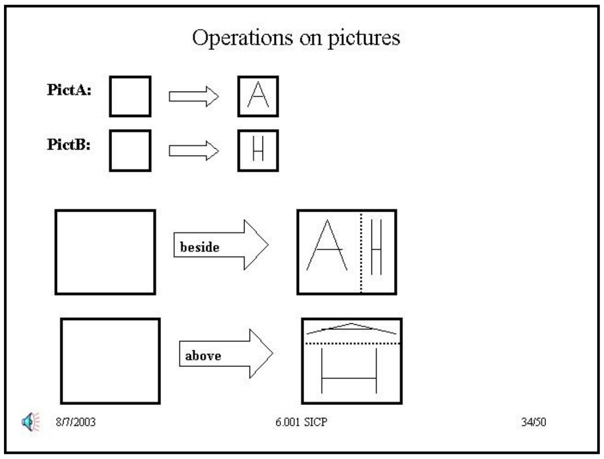
Slide 30.1.33

And here is an example in action!


A Georgian mess!

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

8/7/2003 6.001 SICP 33/50


Operations on pictures

8/7/2003 6.001 SICP 34/50

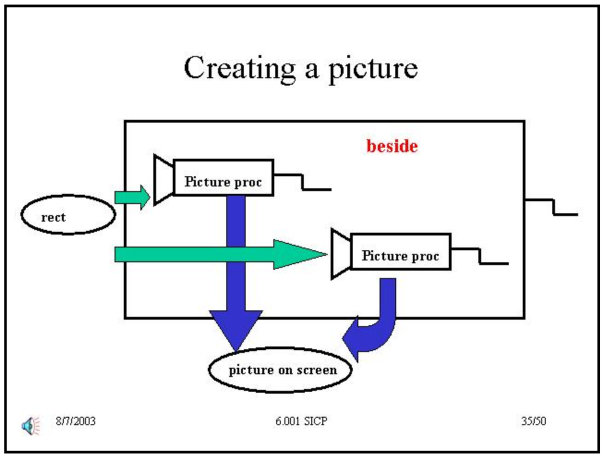
Slide 30.1.34

Now, suppose we have two pictures that draw different things, and we want to combine them. In other words, how do we create a means of combination for pictures?

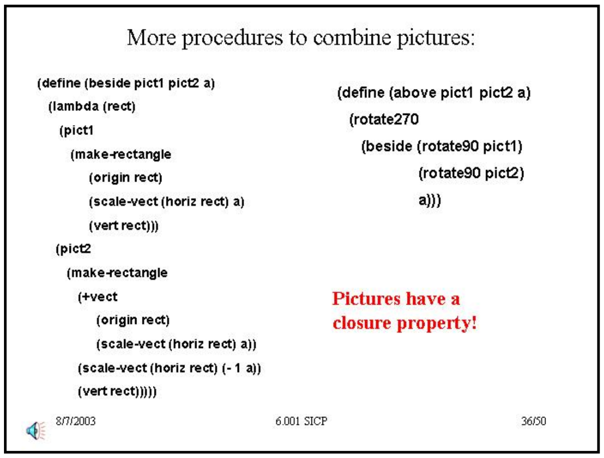
Suppose we start with two different pictures that draw different things. Since a picture is a procedure that takes a frame and draws into it, we could give each of these different pictures a different frame. Specifically, if we pick a division point and split a frame into two parts, each of those parts can act like a frame, and we can draw different pictures into each part. Thus, `beside` should draw two pictures, scaled appropriately, next to one another, and `above` should do the obvious thing in the vertical direction.

Slide 30.1.35

Conceptually, an operation like `beside` has within it two picture procedures. When given a rectangle, each picture draws itself in its share of the rectangle, thus combining two primitive pictures into a more complex one.



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!

8/7/2003
6.001 SICP
36/50

Slide 30.1.36

And here is the code to do it. Let's step through `beside` to see what it does. `Beside` takes two pictures (remember these are procedures that draw in rectangles) and a ratio. It creates a new frame by simply shrinking the horizontal axis by that ratio, and otherwise uses the same origin and vertical axis. It then asks the first picture to draw a version of itself in that frame (which will result in a picture that has been squeezed along the horizontal axis). `Beside` creates a second frame, with the same vertical extent, and a horizontal extent that is set up to fill the remainder of the original frame. Here, however, we need to shift the origin, or starting point, over to the end of the first frame, hence the bit of vector algebra in the second rectangle. And then we ask the second picture to draw itself within this frame.

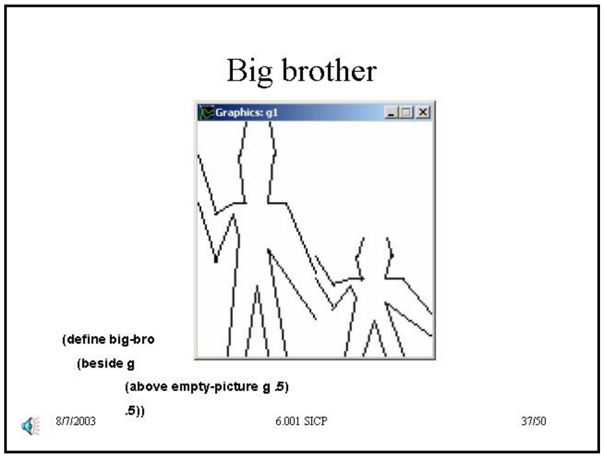
Note the key point here. We can treat pictures as black box abstractions! Thus we can combine pictures with other pictures without worrying about the details of the pictures. Note how the abstraction simply allows us to think about the frames of pictures as vector manipulations, and the pictures themselves come along for free.

Moreover, pictures have the property of closure. Thus we can use our combiners (things like above, beside) to produce new abstractions that can then be used as primitives within some other combination.

Slide 30.1.37

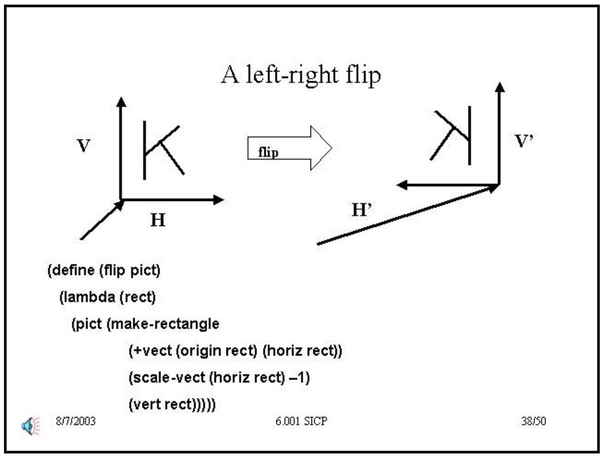
Here is an example of George and his little brother. Notice the elegant combination here. Nowhere is there a specification of the line segments; we are simply taking an abstract notion of a picture and saying: given an empty picture and a picture of George, `above` will create a new picture (notice the higher order procedure abstraction). That procedure can then be combined with George in a `beside` fashion to create another picture.

Notice how the closure property nicely guarantees that combining pictures in this way provides new pictures that can further be combined.



Big brother

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

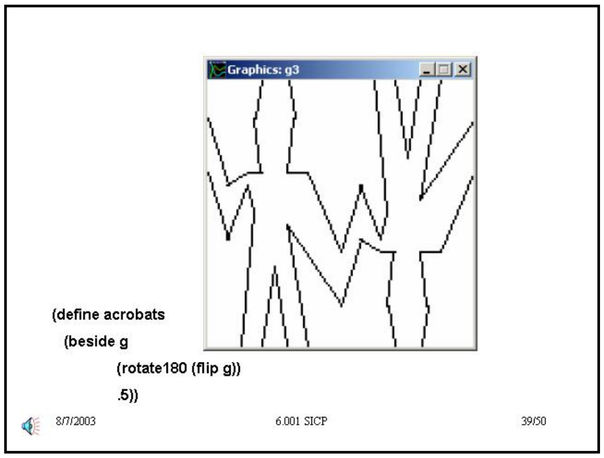

A left-right flip

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

8/7/2003 6.001 SICP 38/50

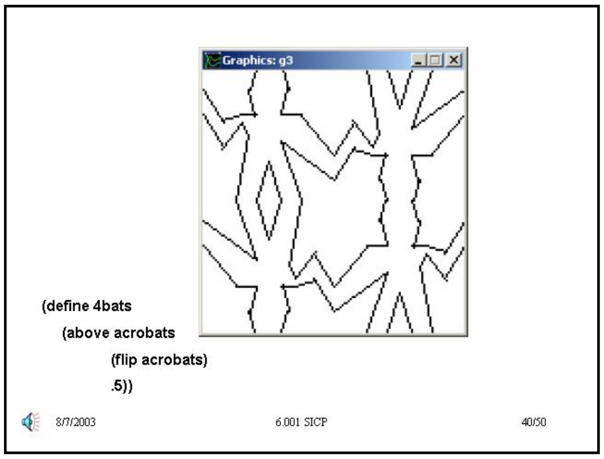
Slide 30.1.38
So let's see how far we can push this idea. Here is another operation on a picture, flipping about a vertical axis. Notice again how flip takes in a picture (a procedure), creates a new picture procedure, and does it by constructing a new rectangle out of the original one by using the appropriate abstractions.

Slide 30.1.39
So we can use this to put things together in interesting ways.



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

8/7/2003 6.001 SICP 39/50

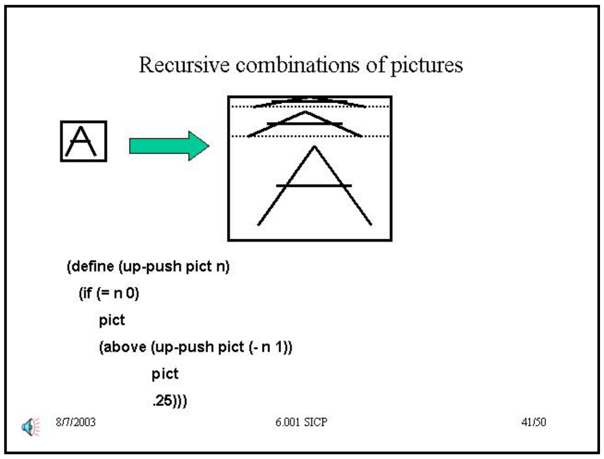


```
(define 4bats
  (above acrobats
         (flip acrobats)
         .5))
```

8/7/2003 6.001 SICP 40/50

Slide 30.1.40
And that includes more complex combinations built on top of combinations.
Trace through the procedures to check that these take the right inputs and give back the right output.
And again notice how closure is nicely allowing us to combine complex things, then treat the result as a primitive and combine again.

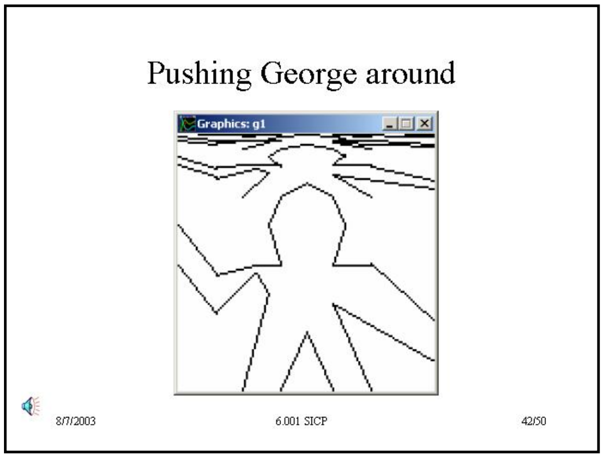
Slide 30.1.41
So how about recursive application of these ideas of combining things? Well, one interesting way would be to draw a picture in some fraction of a frame, then draw it in the same fraction of the remaining part of the frame, and so on, some number of times. This would just be a recursive application of the same idea.
Note how the code accomplishes this. `Up-push` takes a picture as input, and returns a picture. We can see this inductively by noticing that the base case returns a picture. For the recursive case, if we inductively assume that smaller versions of this procedure return pictures, then we see that


Recursive combinations of pictures

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

8/7/2003 6.001 SICP 41/50

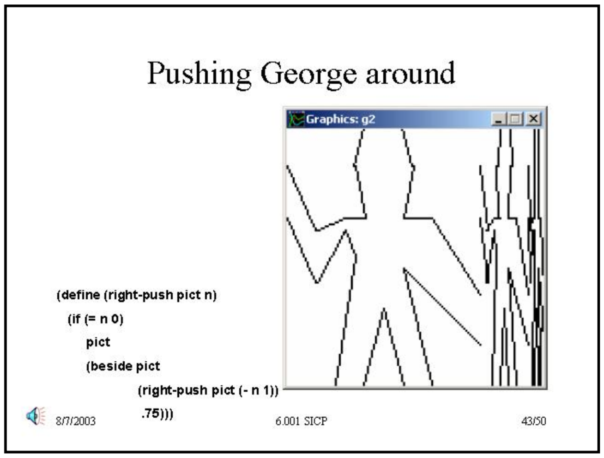
`above` will also return a picture.



Pushing George around

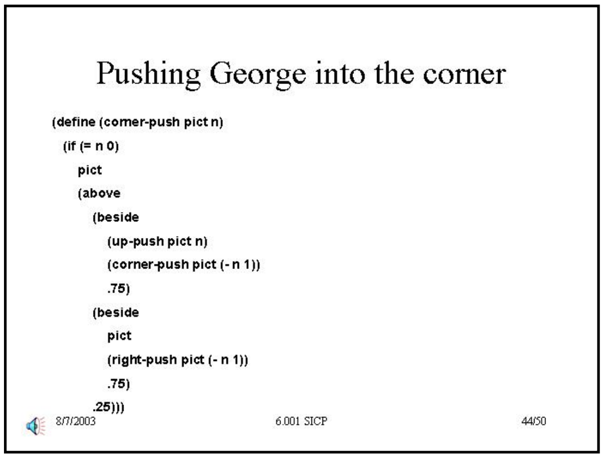
Slide 30.1.42
So we can apply this idea to George.

Slide 30.1.43
And we can do the same thing by pushing things to the side.



Pushing George around

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



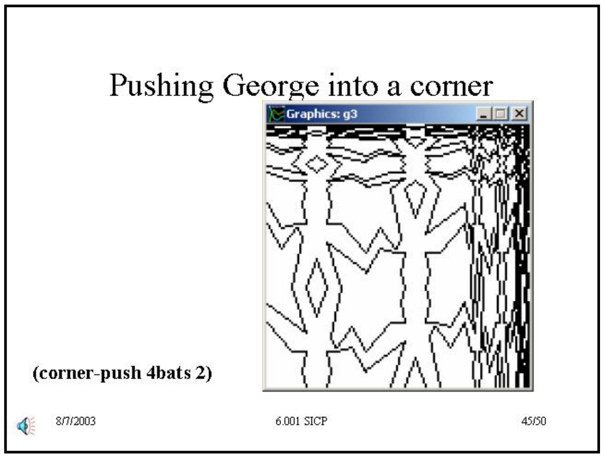
Pushing George into the corner

```
(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)))
```

8/7/2003
6.001 SICP
44/50

Slide 30.1.44
And we can generalize this to push both up and out. Look at the code carefully to note how pictures are combined using higher procedures to return pictures as needed.

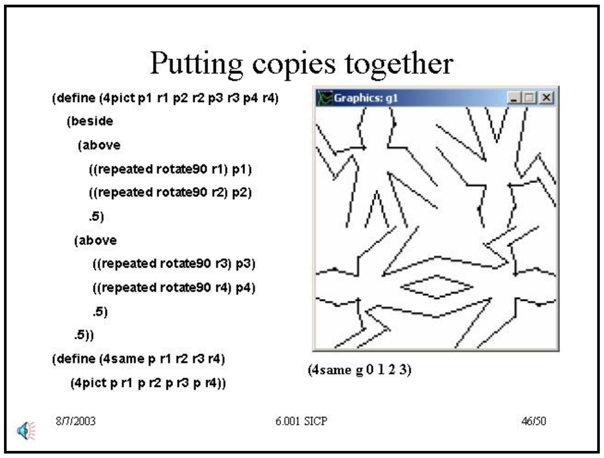
Slide 30.1.45
This lets us push George into a corner.


Pushing George into a corner

(corner-push 4bats 2)

8/7/2003 6.001 SICP 45/50

Slide 30.1.46
Next we can put copies of things together. Here we are using the operation of rotate90 to rotate pictures different amounts. Repeated is just a higher order procedure returns a procedure that applies its first argument (a procedure) a specified number of times in succession (the second argument) to the supplied argument. Again, trace through the code to see how elegantly we have captured the idea of making copies of four different pictures, rotated appropriate amounts.

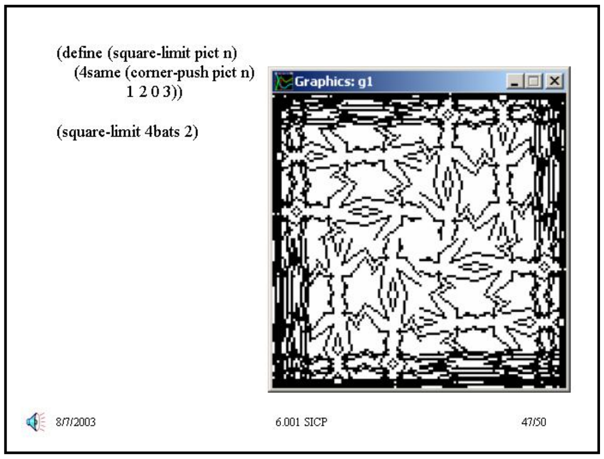

Putting copies together

```
(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)

8/7/2003 6.001 SICP 46/50

Slide 30.1.47
And ultimately, with a simple combination of the things we started with, pushed into a corner, and replicated in four different corners, we get interesting combinations like the one illustrated here.

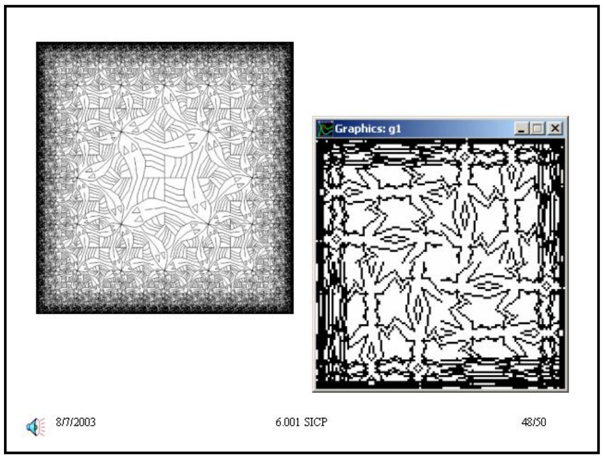


```
(define (square-limit pict n)
   (4same (corner-push pict n)
          1 2 0 3))

(square-limit 4bats 2)
```

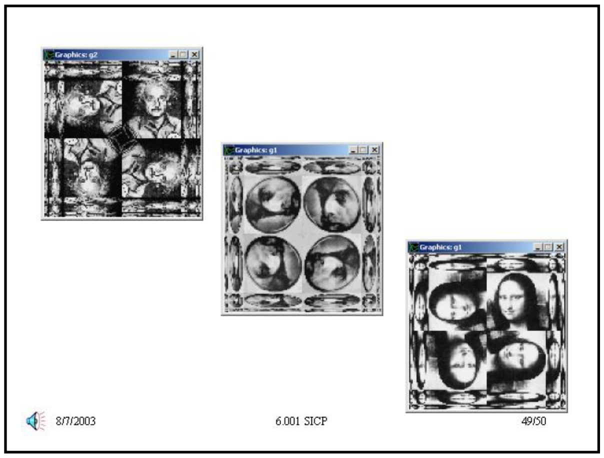
8/7/2003 6.001 SICP 47/50

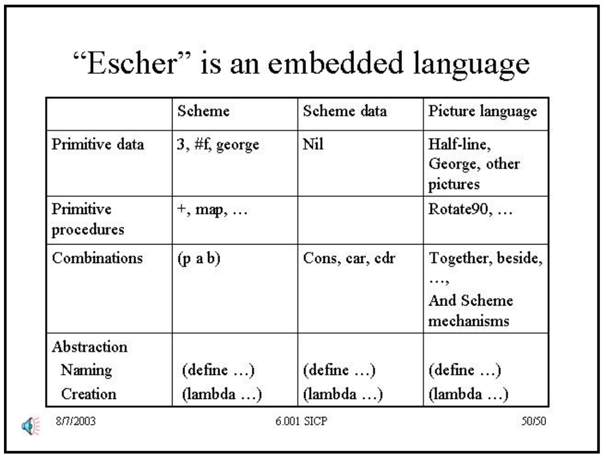
Slide 30.1.48
For comparison, here was our original goal. We aren't quite as good an artist as Escher, but note how we have captured the same recursive behavior of Escher's print in our example.


8/7/2003 6.001 SICP 48/50

Slide 30.1.49

Now, is there anything special about drawing line segments? Of course not. This is just another abstraction issue. So we could instead take a picture, and use exactly the same methods to paint that picture onto a frame. For example, we can get the following, using portraits of Einstein, Escher and Mona Lisa.





"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 …)

8/7/2003 6.001 SICP 50/50

Slide 30.1.50

Now, what is the point of all this?

I claim that what we have done is build a new language: a language for describing and thus creating pictures in the style of Escher. This language is embedded within Scheme, and hence inherits the underlying power of Scheme. But at the same time, there are some nice analogies between the two languages.

First, what are the primitives of Scheme. Here, we have the standard things: primitive elements like numbers, strings, names for things. For Escher, we have a very different notion of primitive data, as here the fundamental unit is a picture! And note that we made a picture as a procedure, so we have really blurred the boundary between data and procedure.

In Scheme, we have primitive procedures for manipulating data objects. In Escher we also have primitive procedures, now oriented towards manipulating pictures.

In Scheme, we have a standard means of combining expressions, namely procedure application. Note that in Escher, we have very elegantly built a similar capability. Our means of combination were ways to gluing picture together. And just like in Scheme, these means of combination had closure, that is, the results of a combination could be treated as a primitive and thus used as input to another combination. Also note that since we built Escher on top of Scheme, for free we got the power of Scheme, for example, the use of recursion.

And finally, we needed a way of naming things so we could treat them as primitives, and Escher inherits exactly that capability from Scheme.

The key issue to see here is how quickly we used abstraction tools to build a new language. We were able to suppress details so that we could focus on the use of the elements, in this case, pictures. We were naturally led to ideas of combining pictures, without ever having to worry about details of the actual pictures. This idea of describing a language for a problem domain in terms of natural primitives, means of combination and means of abstraction is a powerful tool that we will return to many times during the term.