[MUSIC PLAYING]

PROFESSOR: Well, last time we talked about compound data, and there were two main points to that business. First of all, there was a methodology of data abstraction, and the point of that was that you could isolate the way that data objects are used from the way that they're represented: this idea that there's this guy, George, and you go out make a contract with him; and it's his business to represent the data objects; and at the moment you are using them, you don't think about George's problem. And then secondly, there was this particular way that Lisp has of gluing together things to form objects called pairs, and that's done with cons, car and cdr. And the way that cons, car and cdr are implemented is basically irrelevant.

That's sort of George's problem of how to build those things. It could be done as primitives. It could be done using procedures in some weird way, but we're not going to worry about that. And as an example, we looked at rational number arithmetic. We looked at vectors, and here's just a review of vectors. Here's an operation that takes the sum of of two vectors, so we want to add this vector, v1, and this vector, v2, and we get the sum. And the sum is the vector whose coordinates are the sum of the coordinates of the pieces you're adding. So I can say, to define make-vect, right, to add two vectors I make a vector, whose x coordinate is the sum of the two x coordinates, and whose y coordinate is the sum of the two y coordinates.

And then similarly, we could have an operation that scales vectors, so here's a procedure scale that multiplies a vector, v, by some number, s. So here's v, v goes from there to there and I scale v, and I get a vector in the same direction that's longer. And again, to scale a vector, I multiply the successive coordinates. So I make a vector, whose x coordinate is the scale factor times the x coordinate and whose y coordinate is the scale factor times the y coordinate. So those are two operations that are implemented using the representation of vectors.

And the representation of vectors, for instance, is something that we can build in terms of pairs. So George has gone out and implemented for us make-vector and x coordinate and y coordinate, and this could be done, for instance, using cons, car and cdr; and notice here, I wrote this in a slightly different way. The procedures we've seen before, I've said something like say, make-vector of x and y: cons of x and y. And here I just wrote make-vector cons.

And that means something slightly different. Previously we'd say, define make-vector to be a procedure that takes two arguments, x and y, and does cons of x and y. And here I am saying define make-vector to be the thing that cons is, and that's almost the same as the other way we've been writing things. And I just want you to get used to the idea that procedures can be objects, and that you can name them. OK, well there's vector representation, and again, if that was all there was to it, this would all be pretty boring.

And the point is, remember, that you can use cons to glue together not just numbers to form pairs, but to glue together arbitrary things. So for instance, if we'd like to represent a line segment, say the line segment that goes from a certain vector: say, the segment from the vector 2,3 to the point represented by the vector 5,1. If we want to represent that line segment, then we can build that as a pair of pairs. So again, we can represent line segments.

We can make a constructor that makes a segment using cons, selects out the start of a segment, selects out the end point of the segment; and then if we actually look at that, if we peel away the abstraction layers, and say what's that really is a pair of pairs, we'd say well that's a pair. Here's the segment. It's car, right, it's car pointer is a pair, and it's cdr is also a pair, and then what the car is-- here's the car, that itself is a pair of 2 and 3. And similarly the cdr is a pair of 2 and 3. And let me remind you again, that a lot of people have some idea that if I'd taken this arrow and somehow written it to point down, that would mean something else. That's irrelevant. It's only how these are connected and not whether this arrow happens to go vertically or horizontally.

And again just to remind you, there was this notion of closure. See, closure was the thing that allowed us to start building up complexity, that didn't trap us in pairs. Particularly what I mean is the things that we make, having combined things using cons to get a pair, those things themselves can be combined using cons to make more complicated things. Or as a mathematician might say, the set of data objects in List is closed under the operation of forming pairs. That's the thing that allows us to build complexity. And that seems obvious, but remember, a lot of the things in the computer languages that people use are not closed.

So for example, forming arrays in basic and Fortran is not a closed operation, because you can make an array of numbers or character strings or something, but you can't make an array of arrays. And when you look at means of combination, you should be should be asking yourself whether things are closed under that means of combination. Well in any case, because we can form pairs of pairs, we can start using pairs to glue things together in all sorts of different ways. So for instance if I'd like to glue together the four things, 1, 2, 3 and 4, there are a lot of ways I can do it. I could, for example, like we did with that line segment, I could make a pair that had a 1 and a 2 and a 3 and a 4, right?

Or if I liked, I could do something like this. I could make a pair, whose first thing is a pair, whose car is 1, and his cdr is itself a pair that has the 2 and the 3, and then I could put the 4 up here. So you see, there are a lot of different ways that I can start using pairs to glue things together, and so it'll be a good idea to establish some kind of conventions, right, that allow us to deal with this thing in some conventional way, so we're not constantly making an ad hoc choice. And List has a particular convention for representing a sequence of things as, essentially, a chain of pairs, and that's called a List.

And what a List is is essentially just a convention for representing a sequence. I would represent the sequence 1, 2, 3 and 4 by a sequence of pairs. I'd put 1 here and then the cdr of this would point to another pair whose car was the next thing in the sequence, and the cdr would point to another pair whose car was the next thing in the sequence-- so there's 3-- and then another one. So for each item in the sequence, I'll get a pair. And now there are no more, so I put a special marker that means there's nothing more in the List. OK, so that's a conventional way to glue things together if you want to represent a sequence, right. And what it is is a bunch of pairs, the successive cars of each pair are the items that you want to glue together, and the cdr pointer points to the next pair.

Now if I actually wanted to construct that, what I would type into List is this: I'd actually construct that as saying, well this thing is the cons of 1 onto the cons of 2 onto the cons of 3 onto the cons of 4 onto, well, this thing nil. And what nil is is a name for the end of List marker. It's a special name, which means this is the end of the List. OK, so that's how I would actually construct that. Of course, it's a terrible drag to constantly have to write something like the cons of 1 onto the cons of 2 onto the cons of 3, whenever you want to make this thing.

So List has an operation that's called List, and List is just an abbreviation for this nest of conses. So I could say, I could construct that by saying that is the List of 1, 2, 3 and 4. And all this is is another way, a piece of syntactic sugar, a more convenient way for writing that chain of conses-- cons of cons of cons of cons of cons of cons onto nil. So for example, I could build this thing and say, I'll define 1-TO-4 to be the List of 1, 2, 3 and 4. OK, well notice some of the consequences of using this convention. First of all if I have this List, this 1, 2, 3 and 4, the car of the whole thing is the first element in the List, right. How do I get 2? Well, 2 would be the car of the cdr of this thing 1-TO-4, it would be 2, right. I take this thing, I take the cdr of it, which is this much, and the car of that is 2, and then similarly, the car of the cdr of the cdr of 1-TO-4, cdr, cdr, car-- would give me 3, and so on.

Let's take a look at that on the computer screen for a second. I could come up to List, and I could type define 1-TO-4 to be the List of 1, 2, 3 and 4, right. And I'll tell that to List, and it says, fine, that's the definition of 1-TO-4. And I could say, for instance, what's the car of the cdr of the cdr of 1-TO-4, close paren, close paren. Right, so the car of the cdr of the cdr would be 3. Right, or I could say, what's 1-TO-4 itself. And you see what List typed out is 1, 2, 3, 4, enclosed in parentheses, and this notation, typing the elements of the List enclosed in parentheses is List's conventional way for printing back this chain of pairs that represents a sequence.

So for example, if I said, what's the cdr of 1-TO-4, that's going to be the rest of the List. That's the thing pointed to by the first pair, which is, again, a sequence that starts off with 2. Or for example, I go off and say, what's the cdr of the cdr of 1-TO-4; then that's 3,4. Or if I say, what's the cdr of the cdr of the cdr of the cdr of 1-TO-4, and I'm down there looking at the end of List pointer itself, and List prints that as just open paren, close paren. You can think of that as a List with nothing in there. All right, see at the end what I did there was I looked at the cdr of the cdr of the cdr of 1-TO-4, and I'm just left with the end of List pointer itself. And that gets printed as open close.

All right, well that's a conventional way you can see for working down a List by taking successive cdrs of things. It's called cdring down a List. And of course it's pretty much of a drag to type all those cdrs by hand. You don't do that. You write procedures that do that. And in fact one very, very common thing to do in List is to write procedures that, sort of, take a List of things and do something to every element in List, and return you a List of the results.

So what I mean for example, is I might write a procedure called Scale-List, and Scale-List I might say I want to scale by 10 the entire List 1-TO-4, and that would return for me the List 10, 20, 30, 40. [UNINTELLIGIBLE PHRASE] Right, it returns List, and well you can see that there's going to be some kind of recursive strategy for doing it. How would I actually write that procedure? The idea would be, well if you'd like to build up a List where you've multiplied every element by 10, what you'd say is well you imagine that you'd taken the rest of the List-- right, the thing represented by the cdr of the List, and suppose I'd already built a List where each of these was multiplied by 10-- that would be Scale-List of the cdr of the List. And then all I have to do is multiply the car of the List by 10, and then cons that onto the rest, and I'll get a List.

Right and then similarly, to have scaled the cdr of the List, I'll scale the cdr of that and cons onto that 2 multiplied by 10. And finally when I get all the way down to the end, and I only have this end of List pointer. All right, this thing whose name is nil-- well I just returned an end of List pointer. So there's a recursive strategy for doing that. Here's the actual procedure that does that. Right, this is an example of the general strategy of cdr-ing down a List and so called cons-ing up the result, right.

So to Scale a List l by some scale factor s, what do I do? Well there's a test, and List has the predicate called null. Null means is this thing the end of List pointer, or another way to think of that is are there any elements in this List, right. But in any case if I'm looking at the end of List pointer, then I just return the end of List pointer. I just return nil, otherwise I cons together the result of doing what I'm going to do to the first element in the List, namely taking the car of l and multiplying it by s, and I cons that onto recursively scaling the rest of the List.

OK, so again, the general idea is that you recursively do something to the rest of the List, to the cdr of the List, and then you cons that onto actually doing something to the first element of the List. When you get down to the end here, you return the end of List pointer, and that's a general pattern for doing something to a List. Well of course you should know by now that the very fact that there's a general pattern there means I shouldn't be writing this procedure at all. What I should do is write a procedure that's the general pattern itself that says, do something to everything in the List and define this thing in terms of that.

Right, make some higher order procedure, and here's the higher order procedure that does that. It's called MAP, and what MAP does is it takes a List, takes a List l, and it takes a procedure p, and it returns the List of the elements gotten by applying p to each successive element in the List. All right, so p to v1, p to v2, p of en. Right, so I think of taking this List and transforming it by applying p to each element. And you see all this procedure is is exactly the general strategy I said. Instead of multiply by 10, it's do the procedure. If the List is empty, return nil. Otherwise, apply p to the first element of the List. Right, apply p to car of l, and cons that onto the result of applying p to everything in the cdr of the List, so that's a general procedure called MAP. And I could define Scale-List in terms of MAP. Let me show you that first.

But I could say Scale-List is another way to define it is just MAP along the List by the procedure, which takes an item and multiplies it by s. Right, so this is really the way I should think about scaling the List, build that actual recursion into the general strategy, not to every particular procedure I write. And of course, one of the values of doing this is that you start to see commonality. Right, again you're capturing general patterns of usage. For instance, if I said MAP, the square procedure, down this List 1-TO-4, then I'd end up with 1, 4, 9 and 16. Right, or if I said MAP down this List, lambda of x plus x10, if I MAP that down 1-TO-4, then I'd get the List where everything had 10 added to it: right, so I'd get 11, 12, 13, 14. And you can see that's going to be a very, very common idea: doing something to every element in the List.

One thing you might think about is writing MAP in an iterative style. The one I wrote happens to evolve a recursive process, but we could just as easily have made one that evolves an iterative process. But see the interesting thing about it is that once you start thinking in terms of MAP-- see, once you say scale is just MAP, you stop thinking about whether it's iterative or recursive, and you just say, well there's this aggregate, there's this List, and what I do is transform every item in the List, and I stop thinking about the particular control structure in order.

That's a very, very important idea, and it, I guess it really comes out of APL. It's, sort of, the really important idea in APL that you stop thinking about control structures, and you start thinking about operations on aggregates, and then about halfway through this course, we'll see when we talk about something called stream processing, how that view of the world really comes into its glory. This is just us a, sort of, cute idea. But we'll see much more applications of that later on. Well let me mention that there's something that's very similar to MAP that's also a useful idea, and that's-- see, MAP says I take a List, I apply something to each item, and I return a List of the successive values.

There's another thing I might do, which is very, very similar, which is take a List and some action you want to do and then do it to each item in the List in sequence. Don't make a List of the values, just do this particular action, and that's something that's very much like MAP. It's called for-each, and for-each takes a procedure and a List, and what it's going to do is do something to every item in the List. So basically what it does: it says if the List is not empty, right, if the List is not null, then what I do is, I apply my procedure to the first item in the List, and then I do this thing to the rest of the List. I apply for-each to the cdr of the List.

All right, so I do it to the first of the List, do it to the rest of the List, and of course, when I call it recursively, that's going to do it to the rest of the rest of the List and so on. And finally, when I get done, I have to just do something to say I'm done, so we'll return the message "done." So that's very, very similar to MAP. It's mostly different in what it returns. And so for example, if I had some procedure that printed things on the screen, if I wanted to print everything in the List, I could say for-each, print this List. Or if I had a List of figures, and I wanted to draw them on the display, I could say for-each, display on the screen this figure. Let's take questions.

AUDIENCE: Does it create a new copy with something done to it, unless you explicitly tell it to do that? Is that correct?

PROFESSOR: Right. Yeah, that's right. For-each does not create a List. It just sort of does something. So if you have a bunch of things you want to do and you're not worried about values like printing something, or drawing something on the screen, or ringing the bell on the terminal, or for something, you can say for-each, you know, do this for-each of those things in the List, whereas MAP actually builds you this new collection of values that you might want to use. It's just a subtle difference between them.

AUDIENCE: Could you write MAP using for-each, so that you did some sort of cons or something to build the List back up?

PROFESSOR: Well, sort of. I mean, I probably could. I can't think of how to do it right offhand, but yeah, I could arrange something.

AUDIENCE: The vital difference between MAP and for-each is one is recursive and the other is not in the sense you defined early yesterday, I believe.

PROFESSOR: Yeah, about MAP and for-each and recursion. Yeah, that's a good point. For the MAP procedure I wrote, that happens to be a recursive process. And the reason for that is that when you've done this thing to the rest of the List, you're waiting for that value so that you can stick it on to the beginning of the List, whereas for-each doesn't really have any values to wait for. So that turns out to be an iterative process. That's not fundamental. I could have defined MAP so that it's evolved by an iterative process. I just didn't happen to.

AUDIENCE: If you were to cons for each with a List that had embedded Lists, I imagine it would work, right? It would give you the internal elements of each of those internal Lists?

PROFESSOR: OK, the question is if I [UNINTELLIGIBLE] for each or MAP, for that matter, with a List that had Lists in it-- although we haven't really looked at that yet-- would that work. The answer is yes in the sense I mean work and no in the sense that you mean work, because all that-- see if I give you a List, where hanging off here is, you know, is something that's not a number, maybe another List or you know, another cons or something, for-each just says do something to each item in this List. It goes down successively looking at the cdrs.

AUDIENCE: OK.

PROFESSOR: And as far as it's concerned, the first item in this List is whatever is hanging off here.

AUDIENCE: Mhm.

PROFESSOR: That might or might not be the right thing.

AUDIENCE: So it wouldn't go down into the--

PROFESSOR: Absolutely not. I could certainly write something else. There's another, what you're looking for is a common pattern of usage called tree recursion, where you take a List, and you actually go all the way down to the what's called the leaves of the tree. And you could write such a thing, but that's not for-each and it's not MAP. Remember, these things are really being very simple minded. OK, no more questions? All right, let's break.

[MUSIC PLAYING]

PROFESSOR: What I'd like to do now is spend the rest of this time talking about one example, and this example, I think, pretty much summarizes everything that we've done up until now: all right, and that's List structure and issues of abstraction, and representation and capturing commonality with higher order procedures, and also is going to introduce something we haven't really talked about a lot yet-- what I said is the major third theme in this course: meta-linguistic abstraction, which is the idea that one of the ways of tackling complexity in engineering design is to build a suitable powerful language.

You might recall what I said was pretty much the very most important thing that we're going to tell you in this course is that when you think about a language, you think about it in terms of what are the primitives; what are the means of combination-- right, what are the things that allow you to build bigger things; and then what are the means of abstraction. How do you take those bigger things that you've built and put black boxes around them and use them as elements in making something even more complicated?

Now the particular language I'm going to talk about is an example that was made up by a friend of ours called Peter Henderson. Peter Henderson is at the University of Stirling in Scotland. And what this language is about is making figures that sort of look like this. This is this is a woodcut by Escher called "Square Limit." You, sort of, see it has this complicated, kind of, recursive, sort of, recursive kind of figure, where there's this fish pattern in the middle and things sort of bleed out smaller and smaller in self similar ways.

Anyway, Peter Henderson's language was for describing figures that look like that and designing new ones that look like that and drawing them on a display screen. There's another theme that we'll see illustrated by this example, and that's the issue of what Gerry and I have already mentioned a lot: that there's no real difference, in some sense, between procedures and data. And anyway I hope by the end of this morning, if you're not already, you will be completely confused about what the difference between procedures and data are, if you're not confused about that already.

Well in any case, let's start describing Peter's language. I should start by telling you what the primitives are. This language is very simple because there's only one primitive. A primitive is not quite what you think it is. There's only one primitive called a picture, and a picture is not quite what you think it is. Here's an example. This is a picture of George. The idea is that a picture in this language is going to be something that draws a figure scaled to fit a rectangle that you specify.

So here you see in [? Saint ?] [? Lawrence's ?] outline of a rectangle, that's not really part of the picture, but the picture-- you'll give it a rectangle, and it will draw this figure scaled to fit the rectangle. So for example, there's George, and here, this is also George. It's the same picture, right, just scaled to fit a different rectangle. Here's George as a fat kid. That's the same George. It's all the same figure. All of these three things are the same picture in this language. I'm just giving it different rectangles to scale itself in. OK, those are the primitives. That is the primitive.

Now let's start talking about the means of combination and the operations. There is, for example, an operation called Rotate. And what Rotate does is, if I have a picture, say a picture that draws an "A" in some rectangle that I give it, the Rotate of that-- say the Rotate by 90 degrees would, if I give it a rectangle, draw the same image, but again, scaled to fit that rectangle. So that's Rotate by 90 degrees. There's another operation called Flip that can flip something, either horizontally or vertically. All right, so those are, sort of, operations, or you can think of those as means of combination of one element.

I can put things together. There's a means of combination called Beside, and what Beside does: it'll take two pictures, let's say A and B-- and by picture I mean something that's going to draw an image in a specified rectangle-- and what Beside will do-- I have to say, Beside of A and B, the side of two pictures and some number, s. And s will be a number between zero and one. And Beside will draw a picture that looks like this. It will take the rectangle you give it and scale its base by s. Say s is 0.5.

And then over here it will draw-- it'll put the first picture, and over here it'll put the second picture. Or for instance if I gave it a different value of s, if I said Beside with a 0.25, it would do the same thing, except the A would be much skinnier. So it would draw something like that. So there's a means of combination Beside, and similarly there's an Above, which does the same thing except it puts them vertically instead of horizontally.

Well let's look at that. All right, there's George and his kid brother, which is, right, constructed by taking George and putting him Beside the Above-- taking the empty picture, and there's a thing called the empty picture, which does the obvious thing-- putting the empty picture above a copy of George, and then putting that whole thing Beside George. Here's something called P which is, again, George Beside Flipping George, I think, horizontally in this case, and then Rotating the whole result 180 degrees and putting them Beside one another with the basic rectangle divided at 0.5, right, and I can call that P. And then I can take P, and put it above the Flipped copy of itself, and I can call that Q.

Notice how rapidly that we've built up complexity, just in, you know, 15 seconds, you've gotten from George to that thing Q. Why is that? How are how we able to do that so fast? The answer is the closure property. See, it's the fact that when I take a picture and put it Beside another picture, that's then, again, a picture that I can go and Rotate and Flip or put Above something else. Right, and when I take that element P, which is the Beside or the Flip or the Rotate of something, that's, again, a picture. Right, the world of pictures is closed under those means of combination. So whenever I have something, I can turn right around and use that as an element in something else. So maybe better than List and segments, that just gives you an image for how fast you can build up complexity, because operations are closed.

OK, well before we go on with building more things, let's talk about how this language is actually implemented. The basic element that sits under the table here is a thing called a rectangle, and what a rectangle is going to be, it's a thing that specified by an origin that's going to be some vector that says where the rectangle starts. And then there's going to be some other vector that I'm going to call the horizontal part of the rectangle, and another picture called the vertical part of the rectangle. And those three pieces are the elements: where the lower vertex is, how you get to the next vertex over here, and how you get to the vertex over there. The three vectors specify a rectangle.

Now to actually build rectangles, what I'll assume is that we have a constructor called "make rectangle," or "make-rect," and selectors for horiz and vert and origin that get out the pieces of that rectangle. And well, you know a lot of ways you can do this now. You can do it by using pairs in some way or other standard List or not. But in any case, the implementation of these things, that's George's problem. It's just a data representation problem. So let's assume we have these rectangles to work with. OK.

Now the idea of this, remember what's got to happen. Somehow we have to worry about taking the figure and scaling it to fit some rectangle that you give it, that's the basic thing you have to arrange, that these pictures can do. How do we think about that? Well, one way to think about that is that any time I give you a rectangle, that defines, in some sense, a transformation from the standard square into that rectangle. Let me say what I mean. By the standard square, I'll mean something, which is a square whose coordinates are 0,0, and 1,0, and 0,1 and 1,1. And there's some sort of the obvious scaling transformation, which maps this to that and this to that, and sort of, stretches everything uniformly.

So we take a line segment like this and end up mapping it to a line segment like that, so some point xy goes to some other point up there. And although it's not important, with a little vector algebra, you could write that formula. The thing that xy goes to, the point that xy goes to is gotten by taking the origin of the rectangle and then adding that as a vector to-- well, take x, the x coordinate, which is something between zero and one, multiply that by the horizontal vector of the rectangle; and take the y coordinate, which is also something between zero and one and multiply that by the vertical vector of the rectangle. That's just a little linear algebra. Anyway, that's the formula, which is the right obvious transformation that takes things into the unit square, into the interior of that rectangle.

OK well, let's actually look at that as a procedure. So what we want is the thing which tells us that particular transformation that a rectangle defines. So here's the procedure. I'll call it coordinate-map. Coordinate-map is the thing that takes as its argument a rectangle and returns for you a procedure on points. Right, so for each rectangle you get a way of transforming a point xy into that rectangle. And how do you get it? Well I just-- writing in List what I wrote there on the blackboard-- I add to the origin of the rectangle the result of adding-- I take the horizontal part of the rectangle; I scale that by the x coordinate of the point. I take the vertical vector of the rectangle. I scale that by the y coordinate of the point, and then add all those three things up.

That's the procedure. That is the procedure that I'm going to apply to a point. And this whole thing is generated for each rectangle. So any rectangle defines a coordinate MAP, which is a procedure on points. OK. All right, so for example, George here, my original George, might have been something that I specified by segments in the unit square, and then for each rectangle I give this thing, I'm going to draw those segments inside that rectangle. How actually do I do that? Well I take each segment in my original reference George that was specified, and to each of the end points of those segments, I applied the coordinate MAP of the particular rectangle I want to draw it in. So for example, this lower rectangle, this George as a fat kid rectangle, has its coordinate MAP.

And if I want to draw this image, what I do is for each segment here, say for this segment, I transformed that point by the coordinate MAP, transform that point by the coordinate MAP. That will give me this point and that point and draw the segment between them. Right, that's the idea. Right, and if I give it a different rectangle like this one, that's a different coordinate MAP, so I get a different image of those line segments. Well how do we actually get a picture to start with?

I can build a picture to start with out of a List of line segments initially. Here's a procedure that builds what I'll call a primitive picture, meaning one I, sort of, got that didn't come out of Beside or Rotate or something. It starts with a List of line segments, and now it does what I said. What's a picture have to be? First of all it's a procedure that's defined on rectangles. What does it do? It says for each-- this is going to be a List of line segments-- for each segment, for each s, which is a segment in this List of segments, well it draws a line. What line does it draw? It gets the start point of that segment, transforms that by the coordinate MAP of the rectangle.

That's the first new point it wants to do. Then it takes the endpoint of the segment, transforms that by the coordinate MAP of the rectangle, and then draws a line between. Let's assume drawline is some primitive that's built into the system that actually draws a line on the display. All right, so it transforms the endpoints by the coordinate MAP of the rectangle, draws a line between them, does that for each s in this List of segments. And now remember again, a picture is a procedure that takes a rectangle as argument. So when you hand it a rectangle, this is what it does: draws those lines. All right, so there's-- how would I actually use this thing?

Let's make it a little bit more concrete. Right, I would say for instance, define R to be make-rectangle of some stuff, and I'd have to specify some vectors here using make-vector. And then I could say, define say, G to be make-picture, and then some stuff. And what I'd have to specify here is a List of line segments, right, using make segment. Make-segment might be made out of vectors, and vectors might be made out of points. And then if I actually wanted to see the image of G inside a rectangle, well a picture is a procedure that takes a rectangle as argument. So if I then called G with an input of R, that would cause whatever image G is worrying about to be drawn inside the rectangle R. Right, so that's how you'd use that.

[MUSIC PLAYING]

PROFESSOR: Well why is it that I say this example is nice? You probably don't think it's nice. You probably think it's more weird than nice. Right, representing these pictures as procedures, which do complicated things with rectangles. So why is it nice? The reason it's nice is that once you've implemented the primitives in this way, the means of combination just fall out by implementing procedures. Let me show you what I mean. Suppose we want to implement Beside. So I'd like to-- suppose I've got a picture. Let's call it P1. P1 is going to be-- and now remember what a picture really is. It's a thing that if you can hand it some rectangle, it will cause an image to be drawn in whatever rectangle you hand it.

And suppose P2 two is some other picture, and you hand that a rectangle. And whatever rectangle you hand it, it draws some picture. And now if I'd like to implement Beside of P1 and P2 with a scale factor A, well what does that have to be? That's got to be picture. It's got to be a thing that you hand it a rectangle, and it draws something in that rectangle. So if hand Beside this rectangle-- let's hand it a rectangle. Well what's it going to do? it's going to take this rectangle and split it into two at a ratio of A and one minus A. And it will say, oh sure, now I've got two rectangles. And now it goes off to P1 and says P1, well draw yourself in this rectangle, and goes off to P2, and says, P2, fine, draw yourself in this rectangle.

The only computation it has to do is figure out what these rectangles are. Remember a rectangle is specified by an origin and a horizontal vector and a vertical vector, so it's got to figure out what these things are. So for this first rectangle, the origin turns out to be the origin of the original rectangle, and the vertical vector is the same as the vertical vector of the original rectangle. The horizontal vector is the horizontal vector of the original rectangle scaled by A. And that's the first rectangle.

The second rectangle, the origin is the original origin plus that horizontal vector scaled by A. The horizontal vector of the second rectangle is the rest of the horizontal vector of the first one, which is 1 minus A times the original H, and the vertical vector is still v. But basically it goes and constructs these two rectangles, and the important point is having constructed the rectangles, it says OK, p1, you draw yourself in there, and p2, you draw yourself in there, and that's all Beside has to do. All right, let's look at that piece of code. Beside of a picture and another picture with some scaling ratio is first of all, since it's a picture, a procedure that's going to take a rectangle as argument.

What's it going to do? It says, p1 draw yourself in some rectangle and p2 draw yourself in some other rectangle. And now what are those rectangles? Well here's the computation. It makes a rectangle, and this is the algebra I just did on the board: the origin, something; the horizontal vector, something; and the vertical vector, something. And for p2, the rectangle it wants has some other origin and horizontal vector and vertical vector. But the important point is that all it's saying is, p1, go do your thing in one rectangle, and p2, go do your thing in another rectangle. That's all the Beside has to do.

OK, similarly Rotate-- see if I have this picture A, and I want to look at say rotating A by 90 degrees, what that should mean is, well take this rectangle, which is origin and horizontal vector and vertical vector, and now pretend that it's really the rectangle that looks like this, which has an origin and a horizontal vector up here, and a vertical vector there, and now draw yourself with respect to that rectangle. Let me show you that as a procedure.

All right, so we'll Rotate 90 of the picture, because again, a procedure for rectangle, which says, OK picture, draw yourself in some rectangle; and then this algebra is the transformation on the rectangle. It's the one which makes it look like the rectangle is sideways, the origin is someplace else and the vertical vector is someplace else, and the horizontal vector is someplace else, and vertical vector is someplace else. OK? OK. OK, again notice, the crucial thing that's going on here is you're using the representation of pictures as procedures to automatically get the closure property, because what happens is, Beside just has this thing p1. Beside doesn't care if that's a primitive picture or it's line segments or if p1 is, itself, the result of doing Aboves or Besides or Rotates.

All Beside has to know about, say, p1 is that if you hand p1 a rectangle, it will cause something to be drawn. And above that level, Beside just doesn't-- it's none of its business how p1 accomplishes that drawing. All right, so you're using the procedural representation to ensure this closure. OK. So implementing pictures as procedures makes these means of combination, you know, both pretty simple and also, I think, elegant. But that's not the real punchline. The real punchline comes when you look at the means of abstraction in this language. Because what have we done? We've implemented the means of combination themselves as procedures.

And what that means is that when we go to abstract in this language, everything that List supplies us for manipulating procedures is automatically available to do things in this picture language. The technical term I want to say is not only is this language implemented in List, obviously it is, but the language is nicely embedded in List. What I mean is by embedding the language in this way, all the power of List is automatically available as an extension to whatever you want to do.

And what do I mean by that? Example: say, suppose I want to make a thing that takes four pictures A, B, C and D, and makes a configuration that looks like this. Well you might call that, you know, four pictures or something, four-pict configuration. How do I do that? Well I can obviously do that. I just write a procedure that takes B above D and A above C and puts those things beside each other. So I automatically have List's ability to do procedure composition. And I didn't have to make that specifically in the picture language. It's automatic from the fact that the means of combination are themselves procedures.

Or suppose I wanted to do something a little bit more complicated. I wanted to put in a parameter so that for each of these, I could independently specify a rotation by 90 degrees. That's just putting a parameter in the procedure. It's automatically there. Right, it automatically comes from the embedding. Or even more, suppose I wanted to, you know, use recursion.

Let's look at a recursive means of combination on pictures. I could say define-- let's see if you can figure out what this one is-- suppose I say define what it means to right-push a picture, right-push a picture and some integer N and some scale factor A. I'll define this to say if N equals 0, then the answer is the picture. Otherwise I'm going to put-- oops, name change: P. Otherwise, I'm going to take P and put it beside the results of recursively right-pushing P with N minus 1 and A and use a scale factor of A. OK, so if N0 , it's P. Otherwise I put P with a scale factor of A-- I'm sorry I didn't align this right-- recursively beside the result of right-pushing P, N minus 1 times with a scale factor of A.

There's a recursive means of combination. What's that look like? Well, here's what it looks like. There's George right-pushed against himself twice with a scale factor of 0.75. OK. Where'd that come from? How did I get all this fancy recursion? And the answer is just automatic, absolutely automatic. Since these are procedures, the embedding says, well sure, I can define recursive procedures. I didn't have to arrange that. And of course, we can do more complicated things of the same sort. I could make something that does an up-push. Right, that sort of goes like this, by recursively putting something above.

Or I could make something that, sort of, was this scheme. I might start out with a picture and then, sort of, recursively both push it aside and above, and that might put something there. And then up here I put the same recursive thing, and I might end up with something like this. Right, so there's a procedure that's a little bit more complicated than right-push but not much. I just do an Above and a Beside, rather than just a Beside.

Now if I take that and apply that with the idea of putting four pictures together, which I can surely do; and I go and I apply that to Q, which we defined before, right, what I end up with this is this thing, which is, sort of, the square limit of Q, done twice. Right, and then we can compare that with Escher's "Square Limit." And you see, it's sort of the same idea. Escher's is, of course, much, much prettier. If we go back and look at George, right, if we go look at George here-- see, I started with a fairly arbitrary design, this picture of George and did things with it.

Right, whereas if we go look at the Escher picture, right, the Escher picture is not an arbitrary design. It's this very, very clever thing, so that when you take this fish body and Rotate it and shrink it down, it bleeds into the next one really nicely. And of course with George, I didn't really do anything like that. So if we look at George, right, there's a little bit of match up, but not very nice, and it's pretty arbitrary. One very nice project, by the way, would be to write a procedure that could take some basic figure like this George thing and start moving the ends of the lines around, so you got a really nice one when you went and did that "Square Limit" process. That'd be a really nice thing to think about.

Well so, we can combine things. We can recursive procedures. We can do all kinds of things, and that's all automatic. Right, the important point, the difference between merely implementing something in a language and embedding something in the language, so that you don't lose the original power of the language, and what List is great at, see List is a lousy language for doing any particular problem. What it's good for is figuring out the right language that you want and embedding that in List. That's the real power of this approach to design.

Of course, we can go further. See, you saw the other thing that we can do in List is capture general methods of doing things as higher order procedures. And you probably just from me drawing it got the idea that right-push and the analogous thing where you push something up and up and up and up and this corner push thing are all generalizations of a common kind of idea. So just to illustrate and give you practice in looking at a fairly convoluted use of higher order procedures, let me show you the general idea of pushing some means of combination to recursively repeat it.

So here's a good one to puzzle out. We'll define it what it means to push using a means of combination. Comb is going to be something like the Beside or Above. Well what's that going to be. That's going to be a procedure, remember what Beside actually was, right. It took a picture, took two pictures and a scale factor. Using that I produced something that took a level number and a picture and a scale factor, that I called right-push. So this is going to be something that takes a picture, a level number and a scale factor, and it's going to say-- I'm going to do some repeated operation. I'm going to repeatedly apply the procedure which takes a picture and applies the means of combination to the picture and the original picture and the one I took in here and the scale factor, and I do the thing which repeats this procedure N times, and I apply that whole thing to my original picture.

Repeated here, in case you haven't seen it, is another higher order procedure that takes a procedure and a number and returns for you another procedure that applies this procedure N times. And I think some of you have already written repeated as an exercise, but if you haven't, it's a very good exercise in thinking about higher order procedures. But in any case, the result of this repeated is what I apply to picture. And having done that, that's going to capture-- that is the thing, the way I got from the idea of Beside to the idea of right-push So having done that, I could say define right-push to be push of Beside. Or if I say, define up-push to be push of Beside, I'd get the analogous thing or define corner-push to be push of some appropriate thing that did both the Beside and Above, or I could push anything. Anyway this is, if you're having trouble with lambdas, this is an excellent exercise in figuring out what this means.

OK, well there's a lot to learn from this example. The main point I've been dwelling on is the notion of nicely embedding a language inside another language. Right, so that all the power of this language like List of the surrounding language is still accessible to you and appears as a natural extension of the language that you built. That's one thing that this example shows very well. OK. Another thing is, if you go back and think about that, what's procedures and what's data. You know, by the time we get up to here, my God, what's going on. I mean, this is some procedure, and it takes a picture and an argument, and what's a picture. Well, a picture itself, as you remember, was a procedure, and that took a rectangle. And a rectangle is some abstraction.

And I hope now that by now you're completely lost as to the question of what in the system is procedure and what's data. You see, there isn't any difference. There really isn't. And you might think of a picture sometimes as a procedure and sometimes as data, but that's just, sort of, you know, making you feel comfortable. It's really both in some sense or neither in some sense. OK, there's a more general point about the structure of the system as creating a language, viewing the engineering design process as one of creating language or rather one of creating a sort of sequence of layers of language. You see, there's this methodology, or maybe I should say mythology, that's, sort of, charitably called software, quote, engineering.

All right, and what does it say, it's says well, you go and you figure out your task, and you figure out exactly what you want to do. And once you figure out exactly what you want to do, you find out that it breaks out into three sub-tasks, and you go and you start working on-- and you work on this sub-task, and you figure out exactly what that is. And you find out that that breaks down into three sub-tasks, and you specify them completely, and you go and you work on those two, and you work on this sub-one, and you specify that exactly. And then finally when you're done, you come back way up here, and you work on your second sub-task, and specify that out and work it out. And then you end up with-- you end up at the end with this beautiful edifice.

Right, you end up with a marvelous tree, where you've broken your task into sub-tasks and broken each of these into sub-tasks and broken those into sub-tasks, right. And each of these nodes is exactly and precisely defined to do the wonderful, beautiful task to make it fit into the whole edifice, right. That's this mythology. See only a computer scientist could possibly believe that you build a complex system like that, right. Contrast that with this Henderson example. It didn't work like that.

What happened was that there was a sequence of layers of language. What happened? There was a layer of a thing that allowed us to build primitive pictures. There's primitive pictures and that was a language. I didn't say much about it. We talked about how to construct George, but that was a language where you talked about vectors and line segments and points and where they sat in the unit square. And then on top of that, right, on top of that-- so this is the language of primitive pictures. Right, talking about line segments in particular pictures in the unit square. On top of that was a whole language. There was a language of geometric combinators, a language of geometric positions, which talks about things like Above and Beside and right-push and Rotate. And those things, sort of, happened with reference to the things that are talked about in this language.

And then if we like, we saw that above that there was sort of a language of schemes of combination. For example, push, which talked about repeatedly doing something over with a scale factor. And the things that were being discussed in that language were, sort of, the things that happened down here. So what you have is, at each level, the objects that are being talked about are the things that were erected at the previous level. What's the difference between this thing and this thing? The answer is that over here in the tree, each node, and in fact, each decomposition down here, is being designed to do a specific task, whereas in the other scheme, what you have is a full range of linguistic power at each level.

See what's happening there, at any level, it's not being set up to do a particular task. It's being set up to talk about a whole range of things. The consequence of that for design is that something that's designed in that method is likely to be more robust, where by robust, I mean that if you go and make some change in your description, it's more likely to be captured by a corresponding change, in the way that the language is implemented at the next level up, right, because you've made these levels full. So you're not talking about a particular thing like Beside. You've given yourself a whole vocabulary to express things of that sort, so if you go and change your specifications a little bit, it's more likely that your methodology will able to adapt to capture that change, whereas a design like this is not going to be robust, because if I go and change something that's in here, that might affect the entire way that I decomposed everything down, further down the tree.

Right, so very big difference in outlook in decomposition, levels of language rather than, sort of, a strict hierarchy. Not only that, but when you have levels of language you've given yourself a different vocabularies for talking about the design at different levels. So if we go back and look at George one last time, if I wanted to change this picture George, see suddenly I have a whole different ways of describing the change. Like for example, I may want to go to the basic primitive design and move the endpoint of some vector. That's a change that I would discuss at the lowest level. I would say the endpoint is somewhere else.

Or I might come up and say, well the next thing I wanted to do, this little replicated element, I might want to do by something else. I might want to put a scale factor in that Beside. That's a change that I would discuss at the next level of design, the level of combinators. Or I might want to say, I might want to change the basic way that I took this pattern and made some recursive decomposition, maybe not bleeding out toward the corners or something else. That would be a change that I would discuss at the highest level. And because I've structured the system to be this way, I have all these vocabularies for talking about change in different ways and a lot of flexibility to decide which one's appropriate.

OK, well that's sort of a big point about the difference in software methodology that comes out from List, and it all comes, again, out of the notion that really, the design process is not so much implementing programs as implementing languages. And that's really the powerful of List. OK, thank you. Let's take a break.