[MUSIC PLAYING]

PROFESSOR: Well, so far in this course we've been talking about procedures, and then just to remind you of this framework that we introduced for talking about languages, we talked about the primitive things that are built into the system. We mentioned some means of combination by which you take the primitive things and you make more complicated things. And then we talked about the means of abstraction, how you can take those complicated things and name them so you can use them as simple building blocks.

And then last time you saw we went even beyond that. We saw that by using higher order procedures, you can actually express general methods for computing things. Like the method of doing something by fixed points, or Newton's method, and so the incredible expressive power you can get just by combining these means of abstraction.

And the crucial idea in all of this is the one that we build a layered system. So for instance, if we're writing the square root procedure, somewhere the square root procedure uses a procedure called good-enough, and between those there is some sort of abstraction boundary. It's almost as if we go out and in writing square root, we go and make a contract with George, and tell George that his job is to write good-enough, and so long as good-enough works, we don't care what it does. We don't care exactly how it's implemented. There are levels of detail here that are George's concern and not ours. So for instance, George might use an absolute value procedure that's written by Harry, and we don't much care about that or even know that, maybe, Harry exists.

So the crucial idea is that when we're building things, we divorce the task of building things from the task of implementing the parts. And in a large system, of course, we have abstraction barriers like this at lots, and lots, and lots of levels. And that's the idea that we've been using so far over and over in implementing procedures.

Well, now what we're going to do is look at the same issues for data. We're going to see that the system has primitive data. In fact, we've already seen that. We've talked about numbers as primitive data. And then we're going to see their means of combination for data. There's glue that allows you to put primitive data together to make more complicated, kind of compound data. And then we're going to see a methodology for abstraction that's a very good thing to

use when you start building up data in terms of simpler data.

And again, the key idea is that you're going to build the system in layers and set up abstraction barriers that isolate the details at the lower layers from the thing that's going on at the upper layers. The details at the lower layers, the ideas, they won't matter. They're going to be George's concern because he signed this contract with us for how the stuff that he implements behaves, and how he implements the thing is his problem.

All right, well let's look at an example. And the example I'm going to talk about is a system that does arithmetic on rational numbers. And what I have in mind is that we should have something in the computer that allows us to ask it, like, what's the sum of 1/2 and 1/4, and somehow the system should say, yeah, that's 3/4. Or we should be able to say what's 3/4 times 2/3, and the system should be able to say, yeah, that's 1/2. Right? And you know what I have in mind. And you also know how to do this from, I don't know, fifth grade or sixth grade.

There are these formulas that say if I have some fraction which is a numerator over a denominator, and I want to add that to some other fraction which is another numerator over another denominator, then the answer is the numerator of the first times the denominator of the second, plus the numerator of the second times the denominator of the first. That's the numerator of the answer, and the denominator is the product of the two denominators.

Right? So there's something from fifth or sixth grade fraction arithmetic. And then similarly, if I want to multiply two things, n1 over d1 multiplied by n2 over d2 is the product of the numerators over the product of the denominators.

So it's no problem at all, but it's absolutely no problem to think about what computation you want to make in adding and multiplying these fractions. But as soon as we go to implement it, we run up across something. We don't have what a rational number is. So we said that the system gives us individual numbers, so we can have 5 and 3, but somehow we don't have a way of saying there's a thing that has both a 3 and a 4 in it, or both a 2 and a 3. It's almost as if we'd like to imagine that somehow there are these clouds, and a cloud somehow has both a numerator and a denominator in it, and that's what we'd like to work in terms of.

Well, how are we going to solve that problem? We're going to solve that problem by using this incredibly powerful design strategy that you've already seen us use over and over. And that's the strategy of wishful thinking. Just like before when we didn't have a procedure, we said, well, let's imagine that that procedure already exists. We'll say, well, let's imagine that we have

these clouds.

Now more precisely what I mean is let's imagine that we have three procedures, one called make-RAT. make-RAT is going to take as arguments two numbers, so I'll call them numerator and denominator, and it'll return for us a cloud-- one of these clouds. I don't really know what a cloud is. It's whatever make-RAT returns, that's its business.

And then we're going to say, suppose we've got one of these clouds, we have a procedure called numer, which takes in a cloud that has an n and a d in it, whatever a cloud is, and I don't know what it is, and returns for us the numerator part. And then we'll assume we have a procedure denom, which again takes in a cloud, whatever a cloud is, and returns for us the denominator [? required. ?]

This is just like before, when if we're building a square root, we assume that we have good enough. Right? And what we'll say is, we'll go find George, and we'll say to George, well, it's your business to make us these procedures. And how you choose to implement these clouds, that's your problem. We don't want to know.

Well, having pushed this task off onto George, then it's pretty easy to do the other part. Once we've got the clouds, it's pretty easy to write the thing that does say addition of rational numbers. You can just say define, well, let's say +RAT. Define +RAT, which will take in two rational numbers, x and y. x and y are each these clouds.

And what does it do? Well, it's going to return for us a rational number. What rational number is it? Well, we've got the formulas there. The numerator of it is the sum of the product of the numerator of x and the denominator of y. It's one thing in the sum. And the other thing in the numerator is the product of the numerator of y and the denominator of x. The star, close the plus.

Right, that's the first argument to make-RAT, which is the numerator of the thing I'm constructing. And then the rest of the thing goes into make-RAT is the denominator of the answer, which is the product of the denominator of x and the denominator of y. Like that. OK? So there is the analog of doing rational number addition. And it's no problem at all, assuming that we have these clouds.

And of course, we can do multiplication in the same way. Define how to get the product of two rational numbers, call it *RAT. Takes in two of these clouds, x and y, it returns a rational

number, make-RAT, whose numerator is the product of the numerators-- numerator of x times the numerator of y. And the denominator of the thing it's going to return is the product of the denominators.

Well, except that I haven't told you what these clouds are, that's all there is to it. See, what did I do? I assumed by wishful thinking that I had a new kind of data object. And in particular, I assumed I had ways of creating these data objects. Make-RAT creates one of these things. This is called a constructor. All right, I have a thing that constructs such data objects. And then I assume I have things that, having made these things, I have ways of getting the parts out. Those are called selectors.

And so formally, what I said is I assumed I had procedures that are constructors and selectors for these data objects, and then I went off and used them. That's no different in kind from saying I assume I have a procedure good-enough, and I go use it to implement square root.

OK, well before we go on, let's ask the question of why do we want to do this in the first place? See, why do we want a procedure like +RAT that takes in two rational numbers and returns a rational number? See, another way to think about this is, well, here's this formula. And I've also got to implement something that adds rational numbers.

One other way to think about is, well, there's this thing, and I type in four numbers, an n1, and a d1, and an n2, and a d2. And it sets some registers in the machine to this numerator and this denominator. So I might say, well, why don't I just add rational numbers by I type in four numbers, numerators and denominators, and get out two numbers, which is a numerator and a denominator.

Why are we worrying about building things like this anyway? Well, the answer is, suppose you want to think about expressing something like this, suppose I'd like to express the idea of taking two rational numbers, x plus y, say, and multiplying that by the sum of two other rational numbers. Well, the way I do it, having things like +RAT and *RAT, is I'd say, oh yeah, what that is is just the product. That's *RAT of the sum of x and y and the sum of s and t.

So except for syntax, I get an expression that looks like the way I want to think about it mathematically. I want to say there are two numbers. There's a thing which is the sum of them, and there's a thing which is the sum of these two. That's this and this. And then I multiply them. So I get an expression that matches this expression.

If I did the other thing, if I said, well, the way I want to think about this is I type into my machine four numbers, which are the numerators and the denominators of x and y, and then four more numbers, which are the numerators and denominators of s and t. And then what I'd be sitting with is, well, what would I do? I'd add these, and somehow I'd have to have two temporary variables, which are the numerators and denominators of this sum, and I'd go off and store them someplace.

And then I'd go over here, I'd type in four more numbers, I'd get two more temporary variables, which are the numerators and denominators of s and t. And then finally, I put those together by multiplying them. You see, what's starting to happen, there are all these temporary variables, which are sort of the guts of the internals of these rational numbers that start hanging out all over the system.

And of course, if I had more and more complicated expressions, there'd be more and more guts hanging out that confuse my programming. And those of you who sort of programmed things like that, where you're just adding numbers in assembly language, you sort of see you have to suddenly be concerned with these temporary variables.

But more importantly than confusing my programming, they're going to confuse my mind. Because the whole name of this game is that we'd like the programming language to express the concepts that we have in our heads, like rational numbers are things that you can add and then take that result and multiply them.

Let's break for questions. Yeah?

AUDIENCE: I don't quite see the need- when we had make-RAT with the numerator and denominator, we had to have the numerator and denominator to pass as parameters to create the cloud, and then we extracted to get back what we had to have originally.

PROFESSOR: That's right. So the question is, I sort of have the numerator and the denominator, why am I worrying about having the cloud given that I have to get the pieces out? That's sort of what I tried to say at the end, but let me try and say it again, because that's really the crucial question.

The point is, I want to carry this numerator and denominator around together all the time. And it's almost as if I want to know, yeah, there's a numerator and denominator in there, but also, I would like to say, fine, but from another point of view, that's x. And I carry x around, and I

name it as x, and I hold it. And I can say things like, the sum of x and y, rather than just have-- see, it's not so bad when I only think about x, but if I have a system with 10 rational numbers, suddenly I have 20 numerators and denominators, which are not necessarily-- if I don't link them, then it's just 20 arbitrary numbers that are not linked in any particular way.

It's a lot like saying, well, I have these instructions that are the body of the procedures, why do I want to package them and say it's the procedure? It's exactly the same idea.

No? OK. Let's break, let's just stretch and get somebody-- [INAUDIBLE]

[MUSIC PLAYING]

OK, well, we've been working on this rational number arithmetic system, and then what we did, the important thing about what we did, is we thought about the problem by breaking it into two pieces. We said, assume there is this contract with George, and George has figured out the way to how to construct these clouds, provided us procedures make-RAT, which was a constructor, and selectors, which are numerator and denominator. And then in terms of that, we went off and implemented addition and multiplication of rational numbers.

Well, now let's go look at George's problem. How can we go and package together a numerator and a denominator and actually make one of these clouds? See, what we need is a kind of glue, a glue for data objects that allows us to put things together. And Lisp provides such a glue, and that glue is called list structure. List structure is a way of gluing things together, and more precisely, Lisp provides a way of constructing things called pairs.

There's a primitive operator in Lisp called cons. We can take a look at it. There's a thing called cons. Cons is an operator which takes in two arguments called x and y, and it returns for us a thing called a pair. All right, so a thing called a pair that has a first part a second part.

So cons takes two objects. There's a thing called a pair. The first part of the cons is x, and the second part of the cons is y. And that's what it builds. And then we also assume we have ways of getting things out. If you're given a pair, there's a thing called car, and car of a pair, p, gives you out the first part of the pair, p. And there's a thing called cdr, and cdr of the pair, p, gives you the second part of the pair, p. OK, so that's how we construct things.

There's also a conventional way of drawing pictures of these things. Just like we write down that as the conventional way of writing Plato's idea of two, the way we could draw a diagram to represent cons of two and three is like this. We draw a little box. And so here's the box we're

talking about, and this box has two arrows coming out of it. And say the first part of this pair is 2, and the second part of this pair is 3. And this notation has a name, it's called box and pointer notation.

By the way, let me say right now that a lot of people get confused that there's some significance to the geometric way I drew these pointers, the directions. Like some people think it'd be different if I took this pointer and turned it up here, and put the 3 out here. That has no significance. All right? It's merely you have a bunch of arrows, these pointers, and the boxes. The only issue is how they're connected, not the geometric arrangement of whether I write the pointer across, or up, or down.

Now it's completely un-obvious, probably, why that's called list structure. We're not actually going to talk about that today. We'll see that next time.

So those are pairs, there's cons that constructs them. And what I'm going to know about cons, and car, and cdr, is precisely that if I have any x and y, all right, if I have any things x and y, and I use cons to construct a pair, then the car of that pair is going to be x, the thing I put in, and the cdr of that pair is going to be y. That's the behavior of these operators, cons, car, and cdr.

Given them, it's pretty clear how George can go off and construct his rational numbers. After all, all he has to do-- remember George's problem was to implement make-RAT, numerator, and denom. So all George has to do is say define make-RAT of some n and a d-- so all I have to do is cons them. That's cons of n and d.

And then if I want to get the numerator out, I would say define the numerator, numer, of some rational number, x. If the rational number's implemented as a pair, then all I have to do is get out the car of x. And then similarly, define the denom is going to be the cdr, the other thing I put into the pair.

Well, now we're in business. That's a complete implementation of rational numbers. Let's use it. Suppose I want to say, so I want to think about how to add 1/2 plus 1/4 and watch the system work. Well, the way I'd use that is I'd say, well, maybe define a. I have to make a 1/2. Well, that's a rational number with numerator 1 and denominator 2, so a will be make-RAT of 1 and 2.

And then I'll construct the 1/4. I'll say define d to be make-RAT of 1 and 4. And if I'd like to look

at the answer-- well, assuming I don't have a special thing that prints rational numbers, or I could make one-- I could say, for instance, define the answer to be +RAT of a and b, and now I can say, what's the answer? What are the numerators and denominators of the answer?

So if I'm adding 1/2 and 1/4, I'll say, what is the numerator of the answer? And the system is going to type out, well, 6. Bad news. And if I say what's the denominator of the answer, the system's going to type out 8. So instead of what I would really like, which is for it to say that 1/2 and 1/4 is 3/4, this foolish machine is going to say, no, it's 6/8.

Well, that's sort of bad news. Where's the bug? Why does it do that, after all? Well, it's the way that we just had +RAT. +RAT just took the-- it said you add the numerator times the denominator, you add that to the numerator times the denominator, and put that over the product of the two denominators, and that's why you get 6/8.

So what was wrong with our implementation of +RAT? What's wrong with that rational number arithmetic stuff that we did before the break? Well, the answer is one way to look at it is absolutely nothing's wrong. That's perfectly good implementation. It follows the sixth grade, fifth grade mathematic for adding fractions.

One thing we can say is, well, that's George's problem. Like, boy, wasn't George dumb to say that he can make a rational number simply by sticking together the numerator and the denominator? Wouldn't it be better for George, when he made a rational number, to reduce the stuff to lowest terms? And what I mean is, wouldn't it be better for George, instead of using this version of make-RAT, to use this one on the slide? Or instead of just saying cons together n and d, what you do is compute the greatest common divisor of n and d, and gcd is the procedure which, well, for all we care is a primitive, which computes the greatest common divisor of two numbers.

So the way I can construct a rational number is get the greatest common divisor of the two numbers, and I'm going to call that g, and then instead of consing together n and d, I'll divide them through. I'll cons together the quotient of n by the the gcd and the quotient of d by the gcd. And that will reduce the rational number to lowest terms. So when I do this addition, when +RAT calls make-RAT-- and for the definition of +RAT it had a make-RAT in there-- just by the fact that it's constructing that, the thing will get reduced to lowest terms automatically.

OK, that is a complete system. For rational number arithmetic, let's look at what we've done. All right, we said we want to build rational number arithmetic, and we had a thing called +RAT.

We implemented that. And I showed you multiplying rational numbers, and although I didn't put them up there, presumably we'd like to have something that subtracts rational numbers, and I don't know, all sorts of things. Things that test equality in division, and maybe things that print rational numbers in some particular way.

And we implemented those in terms of pairs. These pairs, cons, car, and cdr that are built into Lisp. But the important thing is that between these and these, we set up an abstraction barrier. We set up a layer of abstraction.

And what was that layer of abstraction? That layer of abstraction was precisely the constructor and the selectors. This layer was make-RAT, and numer, and denom. This methodology, another way to say what it's doing, is that we are separating the way something is used, separating the use of data objects, from the representation of data objects. So up here, we have the way that rational numbers are used, do arithmetic on them. Down here, we have the way that they're represented, and they're separated by this boundary. The boundary is the constructors and selectors.

And this methodology has a name. This is called data abstraction. Data abstraction is sort of the programming methodology of setting up data objects by postulating constructors and selectors to isolate use from representation. Well, so why? I mean, after all, we didn't have to do it this way. It's perfectly possible to do rational number addition without having any compound data objects, and here on the slide is one example.

We certainly could have defined +RAT, which takes in things x and y, and we'll say, well what are these rational numbers really? So really, they're just pairs, and the numerator's the car and the denominator's the cdr. So what we'll do is we'll take the car of x times the cdr of y, multiply them. Take the car of y times the cdr of x, multiply them. Add them. Take the cdr of x and the cdr of y, multiply them, and then constitute together. Well, that sort of does the same thing.

But this ignores the problem of reducing things to lowest terms, but let's not worry about that for a minute. But so what? Why don't we do it that way? Right? After all, there are sort of fewer procedures to define, and it's a lot more straightforward. It saves all this self-righteous BS about talking about data abstraction. We just sort of do it. I mean, who knows, maybe it's even marginally more efficient depending on whatever compiler were using for this.

What's the point of isolating the use from the representation? Well, it goes back to this notion of naming. Remember, one of the most important principles in programming is the same as one of the most important principles in sorcery, all right? That's if you have the name of the spirit, you get control over it.

And if you go back and look at the slide, you see what's in there is we have this thing +RAT, but nowhere in the system, if I have a +RAT and a -RAT and a *RAT, and things that look like that, nowhere in the system do I have a thing that I can point at which is a rational number. I don't have, in a system like that, the idea of rational number as a conceptual entity.

Well, what's the advantage of that? What's the advantage of isolating the idea of rational numbers as a conceptual entity, and really naming it with make-RAT, numerator, and denominator. Well, one advantage is you might want to have alternative representations. See, before I showed you that one way George can solve this things not reduced to lowest terms problem, is when you build a rational number, you divide up by the greatest common denominator.

Another way to do that is shown over here. I can have an alternative representation for rational numbers where when you make a rational number, you just cons them. However, when you go to select out the numerator, at that point you compute the gcd of the stuff that's sitting in that pair, and divide out by the gcd. And similarly, when I get the denominator, at that point when I go to get the denominator, I'll divide out by the gcd.

So the difference would be in the old representation, when ans was constructed here, say what's 6 and 8, in the first way, the 6 and 8 would have got reduced when they got stuck into that pair, numerator would select out 3. And in the way I just showed you, well, ans would get 6 and 8 put in, and then at the point where I said numerator, some computation would get done to put out 3 instead of 6. So those are two different ways I might do it.

Which one's better? Well, it depends, right? If I'm making a system where I am mostly constructing rational numbers and hardly ever looking at them, then it's probably better not to do that gcd computation when I construct them. If I'm doing a system where I look at things a lot more than I construct them, then it's probably better to do the work when I construct them. So there's a choice there.

But the real issue is that you might not be able to decide at the moment you're worrying about these rational numbers. See, in general, as systems designers, you're forced with the

necessity to make decisions about how you're going to do things, and in general, the way you'd like to retain flexibility is to never make up your mind about anything until you're forced to do it.

The problem is, there's a very, very narrow line between deferring decisions and outright procrastination. So you'd like to make progress, but also at the same time, never be bound by the consequences of your decisions. Data abstraction's one way of doing this. What we did is we used wishful thinking. See, we gave a name to the decision. We said, make-RAT, numerator, and denominator will stand for however it's going to be done, and however it's going to be done is George's problem.

But really, what that was doing is giving a name to the decision of how we're going to do it, and then continuing as if we made the decision. And then eventually, when we really wanted it to work, coming back and facing what we really had to do. And in fact, we'll see a couple times from now that you may never have to choose any particular representation, ever, ever. Anyway, that's a very powerful design technique. It's the key to the reason people use data abstraction. And we're going to see that idea again and again. Let's stop for questions.

AUDIENCE: What does this decision making through abstraction layers do to the axiom of do all your design before any of your code?

PROFESSOR: Well, that's someone's axiom, and I bet that's the axiom of someone who hasn't implemented very large computer systems very much. I said that computer science is a lot like magic, and it's sort of good that it's like magic. There's a bad part of computer science that's a lot like religion. And in general, I think people who really believe that you design everything before you implement it basically are people who haven't designed very many things.

The real power is that you can pretend that you've made the decision and then later on figure out which one is right, which decision you ought to have made. And when you can do that, you have the best of both worlds.

AUDIENCE: Can you explain the difference between let and define?

PROFESSOR: Oh, OK. Let is a way to establish local names. Let me give you sort of the half answer. And I'll say, later on we can talk about the whole very complicated thing. But the big difference for now is that, see, when you're typing at Lisp, you're typing in this environment where you're making definitions. And when you say define a to be 5, if I say define a to be 5, then from then on the

thing will remember that a is 5.

Let is a way to set up a local context where there's a definition. So if I type something like, saying let a-- no, I shouldn't say a-- if I said let z be 10, and within that context, tell me what the sum of z and z is. So if I typed in this expression to Lisp, and then this would put out 20. However, then if I said what's z, the computer would say that's an unbound variable.

So let is a way of setting up a context where you can make definitions. But those definitions are local to this context. And of course, if I'd said a in here, I'd still get 20. But this a would not interfere at all with this one. So if I type this, and then type this, and then say what's a? a will still be 5. So there's some other subtle differences between let and define, but that's the most important one.

All right, well, we've looked at implementing this little system for doing arithmetic on rational numbers as an example of this methodology of data abstraction. And that's a way of controlling complexity in large systems. But, see, like procedure definition, and like all the ways we're going to talk about for controlling complexity, the real power of these things show up not when you sort of do these things in themselves, like it's not such a great thing that we've done rational number arithmetic, it's that you can use these as building blocks for making more complicated things.

So it's no wonderful idea that you can just put two numbers together to form a pair. If that's all you ever wanted to do, there are tons of ways that you can do that. The real issue is can you do that in such a way so that the things that you build become building blocks for doing something even more complex? So whenever someone shows you a method for controlling complexity, you should say, yeah, that's great, but what can I build with it?

So for example, let me just run through another thing that's a lot like the rational number one. Suppose we would like to represent points in the plane. You sort of say, well, there's a point, and we're going to call that point p. And that point might have coordinates, like this might be the point 1 comma 2. The x-coordinate might be 1, and it's y-coordinate might be 2. And we'll make a little system for manipulating points in the plane.

And again, we can do that-- here's a little example of that. It can represent vectors, the same as points in the plane, and we'll say, yep, there's a constructor called make-vector, make-vector's going to take two coordinates, and here we can implement them if we like as pairs, but the important thing is that there's a constructor. And then given some vector, p, we can

find its x-coordinate, or we can get its y-coordinate. So there's a constructor and selectors for points in the plane.

Well, given points in the plane, we might want to use them to build something. So for instance, we might want to talk about, we might have a point, p, and a point, q, and p might be the point 1, 2, and q might be the point 2, 3. And we might want to talk about the line segment that starts at p and ends at q. And that might be the segment s. So we might want to build points for vectors in terms of numbers, and segments in terms of vectors. So we can represent line segments in exactly the same way.

All right, so the line segment from p to q, we'll say there's a constructor, make-segment. And make up names for the selectors, the starting point of the segment and the ending point of the segment. And again, we can implement a segment using cons as a pair of points, and car and cdr get out the two points that we put together to get the segment.

Well, now having done that, we can have some operations on them. Like we could say, what's the midpoint of a line segment? So here's the midpoint of a line segment, that's going to be the points whose coordinates are the averages of the coordinates of the endpoints. OK, there's the midpoint.

So to get the midpoint of a line segment, s, we'll just say grab the starting point to the segment, grab the ending point of the segment, and now make a vector-- make a point whose coordinates are the average of the x-coordinate of the first point and the x-coordinate of the second point, and whose y-coordinate is the average of the y-coordinates. So there's an implementation of midpoint.

And then similarly, we can build something like the length of the segment. The length of the segment is a thing whose-- use Pythagoras's rule, the length of the segment is the square root of the d x squared plus d y squared. We'll say to get the length of a line segment, we'll let dx be the difference of the x-coordinate of one endpoint and the x-coordinate of the other endpoint, and we'll let dy be the difference of the y-coordinates. And then we'll take the square root of the sum of the squares of dx and dy, that's what this says. All right, so there's an implementation of length.

And again, what we built is a layered system. We built a system which has, well, say up here there's segments. And then there's an abstraction barrier. The abstraction barrier separates the implementation of segments from the implementation of vectors and points, and what that

abstraction barrier is are the constructors and selectors. It's make-segment, and segment-start, and segment-end.

And then there are vectors. And vectors in turn are built on top of pairs and numbers. So I'll say pairs and numbers. And that has its own abstraction barrier, which is make-vector, and x-coordinate, and y-coordinate. So we have, again, a layered system. You're starting to see that there are layers here.

I ought to mention, there is a very important thing that I kind of took for granted. And it's sort of so natural, but on the other hand it's a very important thing. Notice that in order to represent this segment s, I said this segment is a pair of points.

And a point is a pair of numbers. And if I were going to draw the box and pointers structure for that, I would say, oh, the segment is, given those particular representations that I showed you, I'd say this segment s is a pair, and the first thing in the pair is a vector, and the vector is a pair of numbers. And that's this, that's p. And the other thing in the segment is q, which is itself a pair of numbers.

So I almost took it for granted when I said that cons allows you to put things together. But it's very easy to not appreciate that, because notice, some of the things I can put together can themselves be pairs. And let me introduce a word that I'll talk about more next time, it's one of my favorite words, called closure. And by closure I mean that the means of combination in your system are such that when you put things together using them, like we make a pair, you can then put those together with the same means of combination. So I can have not only a pair of numbers, but I can have a pair of pairs.

So for instance, making arrays in a language like Fortran is not a closed means of combination, because I can make an array of numbers, but I can't make an array of arrays. And one of the things that you should ask, one of your tests of quality for a means of combination that someone shows you, is gee, are the things you make closed under that means of combination? So pairs would not be nearly so interesting if all I could do was make a pair of numbers. I couldn't build very much structure at all.

OK, well, we'll come back to that. I just wanted to mention it now. You'll hear a lot about closure later on.

You can also see the potential for losing control of complexity as you have a layered system if

you don't use data abstraction. Let's go back and look at this slide for length. Length works and is a simple thing because I can say, when I want to get this value, I can say, oh, that is the x-coordinate of the first endpoint of the segment. And each of these things, each of these selectors, x-coordinate and endpoint, stand for a decision choice whose details I don't have to look at.

So I could perfectly well, again, just like rational numbers I did before, I could say, oh well, gee, a segment really is a pair of pairs. And the x-coordinate of the first endpoint or the segment really is the-- well, what is it? It's the car of the car of the segment. So I could perfectly well go and redefine length. I could say, define the length of some segment s.

And I could start off writing something like, well, we'll let dx be-- well, what's it have to be? It's got to be the difference of the two coordinates, so that's the difference of, the first one is the car of the car of s, subtracted from the first one, the car of the other half of it, the cdr of s. All right, and then dy would be-- well, let's see, I'd get the y-coordinate, so it'd be the difference of the cdr of the car of s, and the cdr of the cdr of s, sort of go on.

You can see that's much harder to read than the program I had before. But worse than that, suppose you'd gone and implemented length? And then the next day, George comes to you and says, I'm sorry, I changed my mind. I want to write points with the x-coordinate first. So you come back you stare at this code and say, oh gee, what was that? That was the car, so I have to change this to cdr, and this is cdr, and this now has to be car. And this has to be car.

And you sort of do that, and then the next day George comes back and says, sorry, the guys designing the display would like lines to be painted in the opposite direction, so I have to write the endpoint first in the order. And then you come back and you stare at this code, and say, gee, what was it talking about? Oh yeah, well I've got to change this one to cdr, and this one becomes car, this one comes car, and this becomes cdr.

And you go up and do that, and then the next day, George comes back and says, I'm sorry, what I really meant is that the segments always have to be painted from left to right on the screen. And then you sort of, it's clear, you just go and punch George in the mouth at that point. But you see, as soon as we have a 10 layer system, you see how that complexity immediately builds up to the point where even something like this gets out of control.

So again, the way we've gotten out of that is we've named that spirit. We built a system where there is a thing, which is the representation choice for how you're going to talk about vectors.

And choices about that representation are localized right there. They don't have their guts spilling over into things like how you compute the length and how you compute the midpoint. And that's the real power of this system. OK, we're explicit about them, so that we have control over them. All right, questions?

AUDIENCE: What happens in the case where you don't want to be treating objects in terms of pairs? For instance, in three-dimensional space, you'd have three coordinates. Or even in the case where you have n-dimensional space, what happens?

PROFESSOR: Right, OK. Well, this is a preview of what I'll say tomorrow. But the point is, once you have two things, you have as many things as you want. All right? Because if I want to make three things, I could start making things like a pair whose first thing is 1, and whose second thing is another pair that, say, has 2 and 3 in it. And so on, a hundred things. I can nest them out of pairs.

I made a pretty arbitrary decision about how to do it, and you can immediately see there are lots of ways to do that. What we'll start talking about next time are conventions for how to do things like that. But notice that what this really depends on is I can make pairs of pairs. If all I could do was make pairs of numbers, I'd be stuck.

OK. Let's break.

[MUSIC PLAYING]

All right, well, we've just gone off and done a couple of simple examples of data abstraction. Now I want to do something more complicated. We're going to talk about what it means. And this will be harder, because it's always much harder in computer programming to talk about what something means than to go off and do it.

But let's go back to almost the very beginning. Let's go back to the point where I said, we just assumed that there were procedures, make-RAT, and numer, and denom. Let's go back to where we had this, at the very beginning, constructors and selectors, and went off and defined the rational number arithmetic. And remember, I said at that point we were sort of done, except for George. Well, what is it that we'd actually done at that point? What was it that was done?

Well, what I want to say is, what was done after we'd implemented the operations and terms of these, was that we had defined a rational number representation in terms of abstract data.

What do I mean by abstract data? Well, the idea is that at that point, when we had our +RAT and our *RAT, that any implementation of make-RAT, and numerator, and denominator that George supplied us with, could be the basis for a rational number representation. Like, it wasn't our concern where you divided through to get the greatest common denominator, or any of that.

So the idea is that what we built is a rational arithmetic system that would sit on top of any representation. What do I mean by any representation? I mean, certainly it can't be the case that all I mean is George can reach in a bag and pull out three arbitrary procedures and say, well, fine, now that's the implementation. That can't be what I mean.

What I've got to mean is that there's some way of saying whether three procedures are going to be suitable as a basis for rational number representation. If we think about it, what suitable might mean is if I have to assume something like this, I have to say that if x is the result of say, doing make-RAT of n and d, then the numerator of x divided by the denominator of x is equal to n over d.

See, what that is is that's George's contract. What we mean by writing a contract for rational numbers, if you think about it, this is the right thing. And the two ones we showed do the right thing. See, if I'm taking out greatest common divisors, it doesn't matter whether I take them out or not, or the place where I take them, because the idea is I'm going to divide through.

But see, this is George's contract. So what we really say to George is your business is to go off and find us three procedures, make-RAT, and numerator, and denominator, that fulfill this contract for any choice of n and d. And that's what we mean by we can use that as the basis for a rational number representation. And other than that, it fulfills this contract. We don't care how he does it. It's not our business. It's below the layer of abstraction.

In fact, if we want to say, what is a rational number really? See, what's it really, without having to talk about going below the layer of abstraction, what we're forced into saying is a rational number really is sort of this axiom, is three procedures, make-RAT, numerator, and denominator, that satisfy this axiom. In some sense, abstractly, that's what a rational number is really.

That's sort of easy words to listen to, because what you have in your head, of course, is well, for all this thing about saying that's what a rational number is really, you actually just saw that

we built rational numbers. See, what we really did is we built rational numbers on top of pairs. So for all I'm saying abstractly, we can say a rational number really is just this axiom. You can listen to that comfortably, because you're saying, well, yeah, but really it's actually pairs, and I'm just annoying you by trying to be abstract.

Well, let me, as an antidote for that, let me do something that I think is really going to terrify you. I mean, it's really going to bring you face to face with the sort of existential reality of this abstraction that we're talking about. And what I'm going to talk about is, what are pairs really? See, what did I tell you about pairs? I tricked you, right?

I said that Lisp has this primitive called cons that builds pairs. But what did I really tell you about? If you go back and said, let's look on this slide, all I really told you about pairs is that there happens to be this property, these properties of cons, car, and cdr. And all I really said about pairs is that there's a thing called cons, and a thing called car, and a thing called cdr.

And it is the case that if I build cons of x, y and take car of it, I get x. And if I build cons of x, y and get cdr of it, I get y. And even though I lulled you into thinking that there's something in Lisp that does that, so you pretended you knew what it was, in fact, I didn't tell you any more about pairs than this tells you about rational numbers. It's just some axiom for pairs.

Well, to drive that home, let me really scare you, and show you what we might build pairs in terms of. And what you're going to see is that we can build rational numbers, and line segments, and vectors, and all of this stuff in terms of pairs, and we're going to see below here that pairs can be built out of nothing at all. Pure abstraction.

So let me show you on this slide an implementation of cons, car, and cdr. And we'll look at it again in a second, but notice that their procedure definitions of cons, car, and cdr, you don't see any data in there, what you see is a lambda. So cons here is going to return-- is a procedure that returns a procedure, just like AVERAGE DAMP.

Cons of a and b returns a procedure of an argument called pick, and it says, if pick is equal to 1, I'm going to return a, and if pick is equal to 2, I'm going to return b, and that's what cons is going to be. Car of a thing x, car of a pair x, is going to be x applied to 1. And notice that makes sense. You might not understand why or how I'm doing such a thing, but at least it makes sense, because the thing constructed by cons is a procedure, and car applies that to 1.

And similarly, cdr applies that thing to 2. OK, now I claimed that this is a representation of

cons, car, and cdr, and notice there's no data in it. All right, it's built out of air. It's just procedures. There's no data objects at all in that representation. Well, what could that possibly mean? Well, if you really believe this stuff, then you have to believe that in order to show that that's a representation for cons, car, and cdr, all I have to do is show that it satisfies the axiom.

See, all I should have to convince you of is, for example, that gee, that car of cons of 37 and 49 is 37 for arbitrary values of 37 and 49. And cdr the same way. See, if I really can demonstrate to you that that weird procedure definition, in terms of [? air ?], has the property that it satisfies this, then you just have to grant me that that is a possible implementation of cons, car, and cdr, on which I can build everything else.

Well, let's look at that. And this will be practice in the substitution model. How could we check this? We sort of know how to do that. It's just the same substitution model. Let's look. We start out, and we say, what's car of cons of 37 and 49? What do we do? Cons is some procedure. Its value is cons was a procedure of a and b. The thing returned by cons is its procedure body with 37 and 49 substituted for the parameters. It'll be 37 substituted for a and 49 substituted for b.

So this expression has the same meaning as this expression. Its car of, and the body of cons was this thing that started with lambda. And it says, so if pick is equal to 1, where pick is this other argument, if pick is equal to 1, it's 37, that's where a was, and if pick is equal to 2, it's 49. So that's the first step. I'm just going through mechanical substitution. And remember, at this point in the course, if you're confused about what things mean, go mechanically through the substitution model.

Well, what is this reduced to? Car said, take your argument, which in this case is this, and apply it to 1. That was the definition of car. So if I look at car, if I do that, the answer is, well, it's that argument, this was the argument to car, applied to 1. Well, what does that mean? I take 1, and I substitute it in the body here for this value of pick, which is the name of the argument, what do I get? Well, I get the thing that says if 1 equals 1 it's 37, and if 1 equals 2 it's 49, so the answer's 37. And similarly, if I'd taken cdr, that would apply it to 2, and I'd get 49.

So you see, what I've demonstrated is that that completely weird implementation of cons, car, and cdr, satisfies the axioms. So it's a perfectly valid way of building, in fact, all of the data objects we're going to see in Lisp. So they all, if you like, can be built on sort of existential nothing. And as far as you know, that's how it works. You couldn't tell. If all you're ever going

to do with pairs is construct them with cons and look at them with car and cdr, you couldn't possibly tell how this thing works.

Now, it might give you a sort of warm feeling inside if I say, well, yeah, in fact, for various reasons there happens to be a primitive called cons, car, and cdr, and if it's too scary, if this kind of stuff is too scary, you don't have to look inside of it. So that might make you feel better, but the point is, it really could work this way, and it wouldn't make any difference to the system at all. So in some sense, we don't need data at all to build these data abstractions. We can do everything in terms of procedures.

OK, well, why did I terrify you in this way? First, I really want to reinforce this idea of abstraction, that you really can do these things abstractly. Secondly, I want to introduce an idea we're going to see more and more of in this course, which is we're going to blur the line between what's data and what's a procedure.

See, in this funny implementation it turned out that cons of something happened to be represented in terms of a procedure, even though we think of it as data. While here that's sort of a mathematical trick, but one of the things we'll see is that a lot of the very important programming techniques that we're going to get to sort of depend very crucially on blurring this traditional line between what you consider a procedure and what you consider data. We're going to see more and more of that, especially next time.

OK, questions?

AUDIENCE: If you asked the system to print a, what would happen?

PROFESSOR: The question is, what would happen if I asked the system to print a. Given this representation, you already know the answer. The answer is compound procedure a, just like last time. It'd say compound procedure. It might say a little bit more. It might say compound procedure lambda or something or other, depending on details of how I named it. But it's a procedure.

And the only reason for that is I haven't told the system anything special about how to print such things. Now, it's in fact true that with the actual implementation of cons that to be built in the system, it would print something else. It would print, say, this is a pair.

AUDIENCE: When you define cons, and then you pass it into values, how does it know where to look for the cons, because you can use cons over and over again? How does it know where to look to know which a and b it's supposed to pull back out? I don't know if I'm expressing that quite

right. Where is it stored?

PROFESSOR: OK, the question is, I sort of have a cons with a 37 and a 49, and I might make another cons with a 1 and a 2, and I might have one called a, and I might have one called b. And the question is, how does it know? And why don't they get confused? And that's a very good question. See, you have to really believe that the procedures are objects.

It's sort of like saying-- let's try another simpler example. Suppose I ask for the square root of 3. So I asked for the square root of 5, and then I ask for the square of 20. You're probably not the least bit bothered that I can take square root and apply it to 5, and then I can take square root and apply it to 20. And there's sort of no issue, gee, doesn't it get confused about whether it's working on 5 or 20? There's no issue about that because you're thinking of a procedure which goes off and does something.

Now, in some sense you're asking me the same question. But it's really bothering you, and it's bothering you for a really good reason. Because when I write that, you're saying gee, this is, I know, sort of a procedure. But it's not a procedure that's just running. It's just sort of a procedure sitting there. And how can it be that sometimes this procedure has 37 and 49, and there might be another one which has 5 and 6 in there, and why don't they get confused?

So there's something very, very important that's bothering you. And it's really crucial to what's going on. We're suddenly saying that procedures are not just the act of doing something. Procedures are conceptual entities, objects, and if I built cons of 37 and 49, that's a particular procedure that sits there. And it's different from cons of 3 and 4. That's another procedure that sits there.

AUDIENCE: Both of them exist independently.

PROFESSOR: And exists independently.

AUDIENCE: And they both can be referenced by car and cdr.

PROFESSOR: And they both would be referenced by car and cdr. Just like I could increment this, and I could increment that. They're objects. And that's sort of where we're going. See, the fact that you're asking the question shows that you're really starting to think about the implications of what's going on. It's the difference between saying a procedure is just the act of doing something. And a procedure is a real object that has existence.

AUDIENCE: So when the procedure gets built, the actual values are now substituted for a and b--

PROFESSOR: That's right.

AUDIENCE: And then that procedure exists as lambda, and pick is what's actually passed in.

PROFESSOR: Yes, when cons gets called, and the result of cons is a new procedure that's constructed, that new procedure has an argument that's called pick.

AUDIENCE: But it no longer has an a and b. The a and b are the actual values that are passed through.

PROFESSOR: And it has-- right, according to the substitution model, what it now has is not those arbitrary names a and b, it somehow has that 37 and 49 in there. But you're right, that's a hard thing to think about it, and it's different from the way you've been thinking about procedures.

AUDIENCE: And if I have again cons of 37 and 49, it's a different object?

PROFESSOR: And if you make another cons of 37 and 49, you're into a wonderful philosophical problem, which is going to be what the lecture about halfway through this course is about. Which is, if I cons 37 and 49, and I do it again, is that the same thing, or is it a different thing? And how could you tell? And when could it possibly matter?

And that's sort of like saying, is that the same thing as this? Or is this the same thing as that? It's the same kind of question. And that's a very, very deep question. And I can't answer in less than an hour. But we will.