[MUSIC-- "JESU, JOY OF MAN'S DESIRING" BY JOHANN SEBASTIAN BACH] PROFESSOR: So far in this course we've been talking a lot about data abstraction. And remember the idea is that we build systems that have these horizontal barriers in them, these abstraction barriers that separate use, the way you might use some data object, from the way you might represent it. Or another way to think of that is up here you have the boss who's going to be using some sort of data object. And down here is George who's implemented it. Now this notion of separating use from representation so you can think about these two problems separately is a very, very powerful programming methodology, data abstraction.

On the other hand, it's not really sufficient for really complex systems. And the problem with this is George. Or actually, the problem is that there are a lot of Georges. Let's be concrete. Let's suppose there is George, and there's also Martha. OK, now George and Martha are both working on this system, both designing representations, and absolutely are incompatible. They wouldn't cooperate on a representation under any circumstances.

And the problem is you would like to have some system where both George and Martha are designing representations, and yet, if you're above this abstraction barrier you don't want to have to worry about that, whether something is done by George or by Martha. And you don't want George and Martha to interfere with each other. Somehow in designing a system, you not only want these horizontal barriers, but you also want some kind of vertical barrier to keep George and Martha separate.

Let me be a little bit more concrete. Imagine that you're thinking about personnel records for a large company with a lot of loosely linked divisions that don't cooperate very well either. And imagine even that this company is formed by merging a whole bunch of companies that already have their personnel record system set up. And imagine that once these divisions are all linked in some kind of very sophisticated satellite network, and all these databases are put together. And what you'd like to do is, from any place in the company, to be able to say things like, oh, what's the name in a personnel record? Or, what's the job description in a personnel record? And not have to worry about the fact that each division obviously is going to have completely separate conventions for how you might implement these records. From this point you don't want to know about that.

Well how could you possibly do that? One way, of course, is to send down an edict from somewhere that everybody has to change their format to some fixed compatible thing. That's what people often try, and of course it never works. Another thing that you might want to do is somehow arrange it so you can have these vertical barriers. So that when you ask for the name of a personnel record, somehow, whatever format it happens to be, name will figure out how to do the right thing. We want name to be, so-called, a generic operator. Generic operator means what it sort of precisely does depends on the kind of data that it's looking at.

More than that, you'd like to design the system so that the next time a new division comes into the company they don't have to make any big changes in what they're already doing to link into this system, and the rest of the company doesn't have to make any big changes to admit their stuff to the system. So that's the problem you should be thinking about. Like it's sort of just your work. You want to be able to include new things by making minimal changes.

OK, well that's the problem that we'll be talking about today. And you should have this sort of distributed personnel record system in your mind. But actually the one I'll be talking about is a problem that's a little bit more self-contained than that. that'll bring up the issues, I think, more clearly. That's the problem of doing a system that does arithmetic on complex numbers. So let's take a look here.

Just as a little review, there are things called complex numbers. Complex number you can think of as a point in the plane, or z. And you can represent a point either by its real-part and its imaginary-part. So if this is z and its real-part is this much, and its imaginary-part is that much, and you write z equals x plus iy.

Or another way to represent a complex number is by saying, what's the distance from the origin, and what's the angle? So that represents a complex number as its radius times an angle. This one's called-- the original one's called rectangular form, rectangular representation, real- and imaginary-part, or polar representation. Magnitude and angle-- and if you know the real- and imaginary-part, you can figure out the magnitude and angle. If you know x and y, you can get r by this formula. Square root of sum of the squares, and you can get the angle as an arctangent.

Or conversely, if you knew r and A you could figure out x and y. x is r times the cosine of A, and y is r times the sine of A. All right, so there's these two. They're complex numbers. You can think of them either in polar form or rectangular form. What we would like to do is make a system that does arithmetic on complex numbers. In other words, what we'd like--

just like the rational number example-- is to have some operations plus c, which is going to take two complex numbers and add them, subtract them, and multiply them, and divide them.

OK, well there's little bit of mathematics behind it. What are the actual formulas for manipulating such things? And it's sort of not important where they come from, but just as an implementer let's see-- if you want to add two complex numbers it's pretty easy to get its real-part and its imaginary-part. The real-part of the sum of two complex numbers, the real-part of the z1 plus z2 is the real-part of z1 plus the real-part of z2. And the imaginary-part of z1 plus z2 is the imaginary part of z1 plus the imaginary part of z2. So it's pretty easy to add complex numbers. You just add the corresponding parts and make a new complex number with those parts.

If you want to multiply them, it's kind of nice to do it in polar form. Because if you have two complex numbers, the magnitude of their product is here, the product of the magnitudes. And the angle of the product is the sum of the angles. So that's sort of mathematics that allows you to do arithmetic on complex numbers.

Let's actually think about the implementation. Well we do it just like rational numbers. We come down, we assume we have some constructors and selectors. What would we like? Well let's assume that we make a data object cloud, which is a complex number that has some stuff in it, and that we can get out from a complex number the real-part, or the imaginary-part, or the magnitude, or the angle.

We want some ways of making complex numbers-- not only selectors, but constructors. So we'll assume we have a thing called make-rectangular. What make-rectangular is going to do is take a real-part and an imaginary-part and construct a complex number with those parts. Similarly, we can have make-polar which will take a magnitude and an angle, and construct a complex number which has that magnitude and angle.

So here's a system. We'll have two constructors and four selectors. And now, just like before, in terms of that abstract data we'll go ahead and implement our complex number operations. And here you can see translated into Lisp code just the arithmetic formulas I put down before. If I want to add two complex numbers I will make a complex number out of its real- and imaginary-parts. The real part of the complex number I'm going to make is the sum of the real-parts. The imaginary part of the complex number I'm going to make is the sum of the imaginary-parts. I put those together, make a complex number. That's how I implement complex number addition.

Subtraction is essentially the same. All I do is subtract the parts rather than add them. To multiply two complex numbers, I use the other formula. I'll make a complex number out of a magnitude and angle. The magnitude is going to be the product of the magnitudes of the two complex numbers I'm multiplying. And the angle is going to be the sum of the angles of the two complex numbers I'm multiplying. So there's multiplication.

And then division, division is almost the same. Here I divide the magnitudes and subtract the angles.

Now I've implemented the operations. And what do we do? We call on George. We've done the use, let's worry about the representation. We'll call on George and say to George, go ahead and build us a complex number representation. Well that's fine. George can say, we'll implement a complex number simply as a pair that has the real-part and the imaginary-part. So if I want to make a complex number with a certain real-part and an imaginary-part, I'll just use cons to form a pair, and that will-- that's George's representation of a complex number.

So if I want to get out the real-part of something, I just extract the car, the first part. If I want to get the imaginary-part, I extract the cdr. How do I deal with the magnitude and angle? Well if I want to extract the magnitude of one of these things, I get the square root of the sum of the square of the car plus the square of the cdr. If I want to get the angle, I compute the arctangent of the cdr in the car. This is a list procedure for computing arctangent. And if somebody hands me a magnitude and an angle and says, make me a complex number, well I compute the real-part and the imaginary-part, or our cosine of a and our sine of a, and stick them together into a pair.

OK so we're done. In fact, what I just did, conceptually, is absolutely no different from the rational number representation that we looked at last time. It's the same sort of idea. You implement the operators, you pick a representation. Nothing different.

Now let's worry about Martha. See, Martha has a different idea. She doesn't want to represent a complex number as a pair of a real-part and an imaginary-part. What she would like to do is represent a complex number as a pair of a magnitude and an angle. So if instead of calling up George we ask Martha to design our representation, we get something like this. We get make-polar. Sure, if I give you a magnitude and an angle we're just going to form a pair that has magnitude and angle.

If you want to extract the magnitude, that's easy. You just pull out the car or the pair. If you want to extract the angle, sure, that's easy. You just pull out the cdr. If you want to look for real-parts and imaginary-parts, well then you have to do some work. If you want the real-part, you have to get r cosine a. In other words, r, the car of the pair, times the cosine of the cdr of the pair. So this is r times the cosine of a, and that's the real-part. If you want to get the imaginary-part, it's r times the sine of a.

And if I hand you a real-part and an imaginary-part and say, make me a complex number with that real-part and imaginary-part, well I figure out what the magnitude and angle should be. The magnitude's the square root of the sum of the squares and the angle's the arctangent. I put those together to make a pair. So there's Martha's idea.

Well which is better? Well if you're doing a lot of additions, probably George's is better, because you're doing a lot of real-parts and imaginary-parts. If mostly you're going to be doing multiplications and divisions, then maybe Martha's idea is better. Or maybe, and this is the real point, you can't decide. Or maybe you just have to let them both hang around, for personality reasons. Maybe you just really can't ever decide what you would like.

And again, what we would really like is a system that looks like this. That somehow there's George over here, who has built rectangular complex numbers. And Martha, who has polar complex numbers. And somehow we have operations that can add, and subtract, and multiply, and divide, and it shouldn't matter that there are two incompatible representations of complex numbers floating around this system.

In other words, not only like an abstraction barrier here that has things in it like a real-part, and an imaginary-part, and magnitude, and angle. So not only is there an abstraction barrier that hides the actual representation from us, but also there's some kind of vertical barrier here that allows both of these representations to exist without interfering with each other.

The idea is that the things in here-- real-part, imaginary-part, magnitude, and angle-- will be generic operators. If you ask for the real-part, it will worry about what representation it's looking at.

OK, well how can we do that? There's actually a really obvious idea, if you're used to thinking about complex numbers. If you're used to thinking about compound data. See,

suppose you could just tell by looking at a complex number whether it was constructed by George or Martha. In other words, so it's not that what's floating around here are ordinary, just complex numbers, right? They're fancy, designer complex numbers.

So you look at a complex numbers as it's not just a complex number, it's got a label on it that says, this one is by Martha. Or this is a complex number by George. Right? They're signed. See, and then whenever we looked at a complex number we could just read the label, and then we'd know how you expect to operate on that. In other words, what we want is not just ordinary data objects. We want to introduce the notion of what's called typed data. Typed data means, again, there's some sort of cloud. And what it's got in it is an ordinary data object like we've been thinking about. Pulled out the contents, sort of the actual data. But also a thing called a type, but it's signed by either George or Martha. So we're going to go from regular data to type data.

How do we build that? Well that's easy. We know how to build clouds. We build them out of pairs. So here's a little representation that supports typed data. There's a thing called take a type and attach it to a piece of contents, and we just use cons. And if we have a piece of typed data, we can look at the type, which is the car. We can look at the contents, which is the cdr. Now along with that, the way we use our type data will test, when we're given a piece of data, what type it is. So we have some type predicates with us.

For example, to see whether a complex number is one of George's, whether it's rectangular, we just check to see if the type of that is the symbol rectangular, right? The symbol rectangular. And to check whether a complex number is one of Martha's, we check to see whether the type is the symbol polar. So that's a way to test what kind of number we're looking at.

Now let's think about how we can use that to build the system. So let's suppose that George and Martha were off working separately, and each of them had designed their complex number representation packages. What do they have to do to become part of the system, to exist compatibly? Well it's really pretty easy. Remember, George had this package. Here's George's original package, or half of it. And underlined in red are the changes he has to make. So before, when George made a complex number out of an x and y, he just put them together to make a pair. And the only difference is that now he signs them. He attaches the type, which is the symbol rectangular to that pair.

Everything else George does is the same, except that-- see, George and Martha both have procedures named real-part and imaginary-part. So to allow them both to exist in the same Lisp environment, George had changed the names of his procedures. So we'll say, this is

George's real-part procedure. It's the real-part rectangular procedure, the imaginary-part rectangular procedure. And then here's the rest of George's package. He'd had magnitude and angle, just renames them magnitude rectangular and angle rectangular.

And Martha has to do basically the same thing. Martha previously, when she made a complex number out of a magnitude and angle, she just cons them. Now she attaches the type polar, and she changes the name so her real-part procedure won't conflict in name with George's. It's a real-part-polar, imaginary-part-polar, magnitude polar, and angle polar.

Now we have the system. Right there's George and Martha. And now we've got to get some kind of manager to look at these types. How are these things actually going to work now that George and Martha have supplied us with typed data? Well what we have are a bunch of generic selectors. Generic selectors for complex numbers real-part, imaginary-part, magnitude, and angle. Let's look at them more closely.

What does a real-part do? If I ask for the real part of a complex number, well I look at it. I look at its type. I say, is it rectangular? If so, I apply George's real part procedure to the contents of that complex number. This is a number that has a type on it. I strip off the type using contents and apply George's procedure.

Or is this a polar complex number? If I want the real part, I apply Martha's real part procedure to the contents of that number. So that's how real part works. And then similarly there's imaginary-part, which is almost the same. It looks at the number and if it's rectangular, uses George's imaginary-part procedure. If it's polar, uses Martha's. And then there's a magnitude and an angle.

So there's a system. Has three parts. There's sort of George, and Martha, and the manager. And that's how you get generic operators implemented. Let's look at just a simple example, just to pin it down. But exactly how this is going to work, suppose you're going to be looking at the complex number who's real-part is one, and who's imaginary-part is two. So that would be one plus 2i. What would happen is up here, up here above where the operations have to happen, that number would be represented as a pair of 1 and 2 together with typed data. That would be the contents. And the whole data would be that thing with the symbol rectangular added onto that. And that's the way that complex number would exist in the system.

When you went to take the real-part, the manager would look at this and say, oh it's one of George's. He'll strip off the type and hand down to George the pair 1, 2. And that's the kind of data that George developed his system to use. So it gets stripped down. Later on, if you ask George to construct a complex number, George would construct some complex number as a pair, and before he passes it back up through the manager would attach the type rectangular.

So you see what happens. There's no confusion in this system. It doesn't matter in the least that the pair 1, 2 means something completely different in Martha's world. In Martha's world this pair means the complex number whose magnitude is 1 and whose angle is 2. And there's no confusion, because by the time any pair like this gets handed back through the manager to the main system it's going to have the type polar attached. Whereas this one would have the type rectangular attached.

OK, let's take a break.

[MUSIC-- "JESU, JOY OF MAN'S DESIRING" BY JOHANN SEBASTIAN BACH]

We just looked at a strategy for implementing generic operators. That strategy has a name: it's called dispatch type. And the idea is that you break your system into a bunch of pieces. There's George and Martha, who are making representations, and then there's the manager. Looks at the types on the data and then dispatches them to the right person.

Well what criticisms can we make of that as a system organization? Well first of all there was this little, annoying problem that George and Martha had to change the names of their procedures. George originally had a real-part procedure, and he had to go name it real-part rectangular so it wouldn't interfere with Martha's real-part procedure, which is now named real-part-polar, so it wouldn't interfere with the manager's real-part procedure, who's now named real-part. That's kind of an annoying problem. But I'm not going to talk about that one now. We'll see later on when we think about the structure of Lisp names and environments that there really are ways to package all those so-called name spaces separately so they don't interfere with each other. Not going to think about that problem now.

The problem that I actually want to focus on is what happens when you bring somebody new into the system. What has to happen? Well George and Martha don't care. George is

sitting there in his rectangular world, has his procedures and his types. Martha sits in her polar world. She doesn't care.

But let's look at the manager. What's the manager have to do? The manager comes through and had these operations. There was a test for rectangular and a test for polar. If Harry comes in with some new kind of complex number, and Harry has a new type, Harry type complex number, the manager has to go in and change all those procedures. So the inflexibility in the system, the place where work has to happen to accommodate change, is in the manager. That's pretty annoying.

It's even more annoying when you realize the manager's not doing anything. The manager is just being a paper pusher. Let's look again at these programs. What are they doing? What does real-part do? Real-part says, oh, is it the kind of complex number that George can handle? If so, send it off to George. Is it the kind of complex number that Martha can handle? If so, send it off to Martha. So it's really annoying that the bottleneck in this system, the thing that's preventing flexibility and change, is completely in the bureaucracy. It's not in anybody who's doing any of the work. Not an uncommon situation, unfortunately.

See, what's really going on-- abstractly in the system, there's a table. So what's really happening is somewhere there's a table. There're types. There's polar and rectangular. And Harry's may be over here. And there are operators. There's an operator like real-part. Or imaginary-part. Or a magnitude and angle. And sitting in this table are the right procedures. So sitting here for the type polar and real-part is Martha's procedure real-part-polar. And over here in the table is George's procedure real-part-rectangular. And over here would be, say, Martha's procedure magnitude-polar, and George's procedure magnitude-rectangular, right, and so on. The rest of this table's filled in. And that's really what's going on.

So in some sense, all the manager is doing is acting as this table. Well how do we fix our system? How do you fix bureaucracies a lot of the time? What you do is you get rid of the manager. We just take the manager and replace him by a computer. We're going to automate him out of existence. Namely, instead of having the manager who basically consults this table, we'll have our system use the table directly.

What do I mean by that? Let's assume, again using data abstraction, that we have some kind of data structure that's a table. And we have ways of sticking things in and ways of getting things out. And to be explicit, let me assume that there's an operation called "put." And put is going to take, in this case two things I'll call "keys." Key1 and key2. And a value. And that stores the value in the table under key1 and key2. And then we'll assume there's a thing called "get," such that if later on I say, get me what's in the table stored under key1

and key2, it'll retrieve whatever value was stored there. And let's not worry about how tables are implemented. That's yet another data abstraction, George's problem. And maybe we'll see later-- talk about how you might actually build tables in Lisp.

Well given this organization, what did George and Martha have to do? Well when they build their system, they each have the responsibility to set up their appropriate column in the table. So what George does, for example, when he defines his procedures, all he has to do is go off and put into the table under the type-rectangular. And the name of the operation is real-part, his procedure real-part-rectangular.

So notice what's going into this table. The two keys here are symbols, rectangular and real-part. That's the quote. And what's going into the table is the actual procedure that he wrote, real-part rectangular. And then puts an imaginary part into the table, filed under the keys rectangular- and imaginary-part, and magnitude under the keys rectangular magnitude, angle under rectangular-angle. So that's what George has to do to be part of this system.

Martha similarly sets up the column and the table under polar. Polar and real-part. Is the procedure real-part-polar? And imaginary-part, and magnitude, and angle. So this is what Martha has to do to be part of the system. Everyone who makes a representation has the responsibility for setting up a column in the table.

And what does Harry do when Harry comes in with his brilliant idea for implementing complex numbers? Well he makes whatever procedure he wants and builds a new column in this table.

OK, well what happened to the manager? The manager has been automated out of existence and is replaced by a procedure called operate. And this is the key procedure in the whole system. Let's say define operate. Operate is going to take an operation that you want to do, the name of an operation, and an object that you would like to apply that operation to. So for example, the real-part of some particular complex number, what does it do?

Well the first thing it does, it looks in the table. Goes into the table and tries to find a procedure that's stored in the table. So it gets from the table, using as keys the type of the object and the operator, but looks on the table and sees what's stored under the type of the object and the operator, sees if anything's stored. Let's assume that get is implemented. So if nothing is stored there, it'll return the empty list. So it says, if there's actually something

stored there, if the procedure here is not no, then it'll take the procedure that it found in the table and apply it to the contents of the object. And otherwise if there was nothing stored there, it'll-- well we can decide. In this case let's have it put out an error message saying, undefined operator. No operator for this type. Or some appropriate error message.

OK? And that replaces the manager. How do we really use it? Well what we say is we'll go off and define our generic selectors using operate. We'll say that the real-part of an object is found by operating on the object with the name of the operation being real-part. And then similarly, imaginary-part is operate using the name imaginary-part and magnitude and angle. That's our implementation. That plus the tape plus the operate procedure. And the table effectively replaces what the manager used to do.

Let's just go through that slowly to show you what's going on. Suppose I have one of Martha's complex numbers. It's got magnitude 1 and angle 2. And it's one of Martha's. So it's labeled here, polar. Let's call that z. Suppose that's z. And suppose with this implementation someone comes up and asks for the real-part of z. Well real-part now is defined in terms of operate. So that's equivalent to saying operate with the name of the operator being real-part, the symbol real-part on z. And now operate comes. It's going to look in the table, and it's going to try and find something stored under-- the operation is going to apply by looking in the table under the type of the object. And the type of z is polar. So it's going to look and say, can I get using polar? And the operation name, which was real-part. It's going to look in there and apply that to the contents of z. And that? If everything was set up correctly, this thing is the procedure that Martha put there. This is real-part-polar. And this is z without its type. The thing that Martha originally designed those procedures to work on, which is 1, 2.

And so operate sort of does uniformly what the manager used to do sort of all over the system. It finds the right thing, looks in the table, strips off the type, and passes it down into the person who handles it.

This is another, and, you can see, more flexible for most purposes, way of implementing generic operators. And it's called data-directed programming. And the idea of that is in some sense the data objects themselves, those little complex numbers that are floating around the system, are carrying with them the information about how you should operate on them.

Let's break for questions. Yes.

AUDIENCE: What do you have stored in that data object? You have the data itself, you have its type, and you have the operations for that type? Or where are the operations that you found?

PROFESSOR: OK, let me-- yeah, that's a good question. Because it raises other possibilities of how you might do it. And of course there are a lot of possibilities. In this particular implementation, what's sitting in this data object, for example, is the data itself-- which in this case is a pair of 1 and 2-- and also a symbol. This is the symbol, the word P-O-L-A-R, and that's what's sitting in this data object.

Where are the operations themselves? The operations are sitting in the table. So in this table, the rows and columns of the table are labeled by symbols. So when I store something in this table, the key might be the symbol polar and the symbol magnitude. And I think by writing it this way I've been very confusing. Because what's really sitting here isn't-- when I wrote magnitude polar, what I mean is the procedure magnitude polar. And probably what I really should have written-- except it's too small for me to write in this little space-- is something like lambda of z, the thing that Martha wrote to implement. And then you can see from that, there's another way that I alluded to of solving this name conflict problem, which is that George and Martha never have to name their procedures at all. They can just stick the anonymous things generated by lambda directly into the table.

There's also another thing that your question raises, is the possibility that maybe what I would like somehow is to store in this data object not the symbol P-O-L-A-R but maybe actually all the operations themselves. And that's another way to organize the system, called message passing. So there are a lot of ways you can do it.

AUDIENCE: Therefore if Martha and George had used the same procedure names, it would be OK because it wouldn't look [UNINTELLIGIBLE].

PROFESSOR: That's right. That's right. See, they wouldn't even have to name their procedures at all. What George could have written instead of saying put in the table under rectangular- and real-part, the procedure real-part rectangular, George could have written put under rectangular real-part, lambda of z, such and such, and such and such. And the system would work completely the same.

AUDIENCE: My question is, Martha could have put key1 key2 real-part, and George could have put key1 key2 real-part, and as long as they defined them differently they wouldn't have had any conflicts, right?

PROFESSOR: Yes, that would all be OK except for the fact that if you imagine George and Martha typing at the same console with the same meanings for all their names, and it would get confused by real-part, but there are ways to arrange that, too. And in principle you're absolutely right. If their names didn't conflict-- it's the objects that go in the table, not the names.

OK, let's take a break.

[MUSIC-- "JESU, JOY OF MAN'S DESIRING" BY JOHANN SEBASTIAN BACH]

All right, well we just looked at data-directed programming as a way of implementing a system that does arithmetic on complex numbers. So I had these operations in it called plus C and minus C, and multiply, and divide, and maybe some others. And that sat on top of-- and this is the key point-- sat on top of two different representations. A rectangular package here, and a polar package. And maybe some more. And we saw that the whole idea is that maybe some more are now very easy to add.

But that doesn't really show the power of this methodology. Shows you what's going on. The power of the methodology only becomes apparent when you start embedding this in some more complex system. What I'm going to do now is embed this in some more complex system. Let's assume that what we really have is a general kind of arithmetic system. So called generic arithmetic system. And at the top level here, somebody can say add two things, or subtract two things, or multiply two things, or divide two things. And underneath that there's an abstraction barrier. And underneath this barrier, is, say, a complex arithmetic package. And you can say, add two complex numbers.

Or you might also have-- remember we did a rational number package-- you might have that sitting there. And there might be a rational thing. And the rational number package, well, has the things we implemented. Plus rat, and times rat, and so on. Or you might have ordinary Lisp numbers. You might say add three and four. So we might have ordinary numbers, in which case we have the Lisp supplied plus, and minus, and times, and slash.

OK, so we might imagine this complex number system sitting in a more complicated generic operator structure at the next level up. Well how can we make that? We already have the idea, we're just going to do it again. We've implemented a rational number package. Let's look at how it has to be changed.

In fact, at this level it doesn't have to be changed at all. This is exactly the code that we wrote last time. To add two rational numbers, remember there was this formula. You make a rational number whose numerator-- the numerator of the first times the denominator of the second, plus the denominator of the first times the numerator of the second. And who's denominator is the product of the denominators. And minus rat, and star rat, and slash rat. And this is exactly the rational number package that we made before. We're ignoring the GCD problem, but let's not worry about that.

As implementers of this rational number package, how do we install it in the generic arithmetic system? Well that's easy. There's only one thing we have to do differently. Whereas previously we said that to make a rational number you built a pair of the numerator and denominator, here we'll not only build the pair, but we'll sign it. We'll attach the type rational. That's the only thing we have to do different, make it a typed data object.

And now we'll stick our operations in the table. We'll put under the symbol rational and the operation add our procedure, plus rat. And, again, note this is a symbol. Right? Quote, unquote, but the actual thing we're putting in the table is the procedure. And for how to subtract, well you subtract rationals with minus rat. And multiply, and divide. And that is exactly and precisely what we have to do to fit inside this generic arithmetic system.

Well how does the whole thing work? See, what we want to do is have some generic operators. Have add and sub and [UNINTELLIGIBLE] be generic operators. So we're going to define add and say, to add x and y, that will be operate-- we were going to call it operate-2. This is our operator procedure, but set up for two arguments using add on x and y. And so this is the analog to operate.

Let's look at the code for second. It's almost like operate. To operate with some operator on an argument 1 and an argument 2, well the first thing we're going to do is check and see if the two arguments have the same type. So we'll say, is the type of the first argument the same as the type of the second argument? And if they're not, we'll go off and complain, and say, that's an error. We don't know how to do that.

If they do have the same type, we'll do exactly what we did before. We'll go look and filed under the type of the argument-- arg 1 and arg 2 have the same type, so it doesn't matter. So we'll look in the table, find the procedure. If there is a procedure there, then we'll apply it to the contents of the argument 1 and the contents of arg 2. And otherwise we'll say, error. Undefined operator. And so there's operate-2. And that's all we have to do.

We just built the complex number package before. How do we embed that complex number package in this generic system? Almost the same. We make a procedure called make-complex that takes whatever George and Martha hand to us and add the type-complex. And then we say, to add complex numbers, plus complex, we use our internal procedure, plus c, and attach a type, make that a complex number.

So our original package had names plus c and minus c that we're using to communicate with George and Martha. And then to communicate with the outside world, we have a thing called plus-complex and minus-complex. And so on. And the only difference is that these return values that are tight. So they can be looked at up here. And these are internal operations.

Let's go look at that slide again. There's one more thing we do. After defining plus-complex, we put under the type complex and the symbol add, that procedure plus complex. And then similarly for subtracting complex numbers, and multiplying them, and dividing them.

OK, how do we install ordinary numbers? Exactly the same way. Come off and say, well we'll make a thing called make-number. Make-number takes a number and attaches a type, which is the symbol number. We build a procedure called plus-number, which is simply, add the two things using the ordinary addition, because in this case we're talking about ordinary numbers, and attach a type to it and make that a number. And then we put into the table under the symbol number and the operation add, this procedure plus-number, and then the same thing for subtracting, and multiplying, and dividing.

Let's look at an example, just to make it clear. Suppose, for instance, I'm going to perform the operation. So I sit up here and I'm going to perform the operation, which looks like multiplying two complex numbers. So I would multiply, say, 3 plus 4i and 2 plus 6i. And that's something that I might want to take hand that to mul. I'll write mul as my generic operator here.

How's that going to work? Well 3 plus 4i, say, sits in the system at this level as something that looks like this. Let's say it was one of George's. So it would have a 3 and a 4. And attached to that would be George's type, which would say rectangular, it came from George. And attached to that-- and this itself would be the data view from the next level up, which it is-- so that itself would be a type-data object which would say complex. So that's what this object would look like up here at the very highest level, where the really super-generic operations are looking at it.

Now what happens, mul eventually's going to come along and say, oh, what's it's type? It's type is complex. Go through to operate-2 and say, oh, what I want to do is apply what's in the table, which is going to be the procedure star complex, on this thing with the type stripped off. So it's going to strip off the type, take that much, and send that down into the complex world.

The complex world looks at its operations and says, oh, I have to apply star c. Star c might say, oh, at some point I want to look at the magnitude of this object that it's in, that it's got. And they'll say, oh, it's rectangular, it's one of George's. So it'll then strip off the next version of type, and hand that down to George to take the magnitude of.

So you see what's going on is that there are these chains of types. And the length of the chain is sort of the number of levels that you're going to be going up in this table. And what a type tells you, every time you have a vertical barrier in this table, where there's some ambiguity about where you should go down to the next level, the type is telling you where to go. And then everybody at the bottom, as they construct data and filter it up, they stick their type back on. So that's the general structure of the system.

OK. Now that we've got this, let's go and make this thing even more complex. Let's talk about adding to the system not only these kinds of numbers, but it's also meaningful to start talking about adding polynomials. Might do arithmetic on polynomials. Like we could have x to the fifteenth plus 2x to the seventh plus 5. That might be some polynomial. And if we have two such gadgets we can add them or multiply them. Let's not worry about dividing them. Just add them, multiply them, then we'll subtract them.

What do we have to do? Well let's think about how we might represent a polynomial. It's going to be some typed data object. So let's say a polynomial to this system might look like a thing that starts with the type polynomial. And then maybe it says the next thing is what variable its in. So I might say I'm a polynomial in the variable x. And then it'll have some information about what the terms are.

And there're just tons of ways to do this, but one way is to say we're going to have a thing called a term-list. And a term-list-- well, in our case we'll use something that looks like this. We'll make it a bunch of pairs which have an order in a coefficient. So this polynomial would be represented by this term-list. And what that means is that this polynomial starts off with a term of order 15 and coefficient 1. And the next thing in it is a term of order 7 and coefficient 2, a term of order 0, which is constant in coefficient 5.

And there are lots and lots of ways, and lots and lots of trade-offs when you really think about making algebraic manipulation packages about exactly how you should represent these things. But this is a fairly standard one. It's useful in a lot of contexts.

OK, well how do we implement our polynomial arithmetic? Let's start out. What we'll do to make a polynomial-- we'll first have a way to make polynomials. We're going to make a polynomial out of variable like x and term-list. And all that does is we'll package them together someway. We'll put the variable together with the term list using cons, and then attached to that the type polynomial.

OK, how do we add two polynomials? To add a polynomial, p1 and p2, and then just for simplicity let's say we will only add things in the same variable. So if they have the same variable, and same variable here is going to be some selector we write, whose details we don't care about. If the two polynomials have the same variable, then we'll do something. If they don't have the same variable, we'll give an error, polynomials not in the same variable.

And if they do have the same variable, what we'll do is we'll make a polynomial whose variable is whatever that variable is, and whose term-list is something we'll call sum-terms. Plus terms will add the two term lists. So we'll add the two term lists to the polynomial. That'll give us a term-list. We'll add on, we'll say it's a polynomial in the variable with that term-list. That's plus poly. And then we're going to put in our table under the type polynomial, add them using plus poly. And of course we really haven't done much. What we've really done is pushed all the work onto this thing, plus-terms, which is supposed to add term-lists.

Let's look at that. Here's an overview of how we might add two term-lists. So L1 and L2 were going to be two term-lists. And a term-list is a bunch of pairs, coefficient in order. And it's a big case analysis. And the first thing we'll check for and see if there are any terms. We're going to recursively work down these term-lists, so eventually we'll get to a place

where either L1 or L2 might be empty. And if either one is empty, our answer will be the other one. So if L1 is empty we'll return L2, and if L2 is empty we'll return L1.

Otherwise there are sort of three interesting cases. What we're going to do is grab the first term in each of those lists, called t1 and t2. And we're going to look at three cases, depending on whether the order of t1 is greater than the order of t2, or less than t2, or the same. Those are the three cases we're going to look at.

Let's look at this case. If the order of t1 is greater than the order of t2, then what that means is that our answer is going to start with this term of the order of t1. Because it won't combine with any lower order terms. So what we do is add the lower order terms. We recursively add together all the terms in the rest of the term-list in L1 and L2. That's going to be the lower order terms of the answer. And then we're going to adjoin to that the highest order term.

And I'm using here a whole bunch of procedures I haven't defined, like a adjoin-term, and rest-terms, and selectors that get order. But you can imagine what those are. So if the first term-list has a higher order than the second, we recursively add all the lower terms and then stick on that last term. The other case, the same way. If the first term has a smaller order, well then we add the first term-list and the rest of the terms in the second one, and adjoin on this highest order term.

So so far nothing's much happened, we've just sort of pushed this thing off into adding lower order terms. The last case where you actually get to a coefficients that you have to add, this will be the case where the orders are equal. What we do is, well again recursively add the lower order terms. But now we have to really combine something. What we do is we make a term whose order is the order of the term we're looking at. By now t1 and t2 have the same order. That's its order. And its coefficient is gotten by adding the coefficient of t1 and the coefficient of t2.

This is a big recursive working down of terms, but really there's only one interesting symbol in this procedure, only one interesting idea. The interesting idea is this add. And the reason that's interesting is because something completely wonderful just happened. We reduced adding polynomials, not to sort of plus, but to the generic add. In other words, by implementing it that way, not only do we have our system where we can have rational numbers, or complex numbers, or ordinary numbers, we've just added on polynomials. But the coefficients of the polynomials can be anything that the system can add. So these could be polynomials whose coefficients are rational numbers or complex numbers, which in turn could be either rectangular, or polar, or ordinary numbers.

So what I mean precisely is our system right now automatically can handle things like adding together polynomials that have this one: 2/3 of x squared plus 5/17 x plus 11/4. Or automatically handle polynomials that look like 3 plus 2i times x to the fifth plus 4 plus 7i, or something. You can automatically handle those things. Why is that? That's merely because, or profoundly because we reduced adding polynomials to adding their coefficients. And adding coefficients was done by the generic add operator, which said, I don't care what your types are as long as I know how to add you. So automatically for free we get the ability to handle that.

What's even better than that, because remember one of the things we did is we put into the table that the way you add polynomials is using plus poly. That means that polynomials themselves are things that can be added. So for instance let me write one here. Here's a polynomial. So this gadget here I'm writing up, this is a polynomial in y whose coefficients are polynomials in x. So you see, simply by saying, polynomials are themselves things that can be added, we can go off and say, well not only can we deal with rationals, or complex, or ordinary numbers, but we can deal with polynomials whose coefficients are rationals, or complex, or ordinary numbers, or polynomials whose coefficients are rationals, or complex, rectangular, polar, or ordinary numbers, or polynomials whose coefficients are rationals, complex, or ordinary numbers. And so on, and so on, and so on.

So this is sort of an infinite or maybe a recursive tower of types that we've built up. And it's all exactly from that one little symbol. A-D-D. Writing "add" instead of "plus" in the polynomial thing.

Slightly different way to think about it is that polynomials are a constructor for types. Namely you give it a type, like integer, and it returns for you polynomials in x whose coefficients are integers. And the important thing about that is that the operations on polynomials reduce to the operations on the coefficients. And there are a lot of things like that.

So for example, let's go back and rational numbers. We thought about rational numbers as an integer over an integer, but there's the general notion of a rational object. Like we might think about 3x plus 7 over x squared plus 1. That's general rational object whose numerator and denominator are polynomials. And to add two of them we use the same formula, numerator times denominator plus denominator times numerator over product of denominators.

How could we install that in our system? Well here's our original rational number arithmetic package. And all we have to do in order to make the entire system continue working with general rational objects, is replace these particular pluses and stars by the generic operator. So if we simply change that procedure to this one, here we've changed plus and star to add a mul, those are absolutely the only change, then suddenly our entire system can start talking about objects that look like this.

So for example, here is a rational object whose numerator is a polynomial in x whose coefficients are rational numbers. Or here is a rational object whose numerator is polynomials in x whose coefficients are rational objects constructed out of complex numbers. And then there are a lot of other things like that.

See, whenever you have a thing where the operations reduce to operations on the pieces, another example would be two by two matrices. I have the idea, there might be a matrix here of general things that I don't care about. But if I add two of them, the answer over here is gotten by adding this one and that one, however they like to add. So I can implement that the same way. And if I do that, then again suddenly my system can start handling things like this. So here's a matrix whose elements happen to be-- we'll say this element here is a rational object whose numerator and denominators are polynomials. And all that comes for free.

What's really going on here? What's really going on is getting rid of this manager who's sitting there poking his nose into who everybody's business is. We built a system that has decentralized control. So when you come into and no one's poking around saying, gee, are you in the official list of people who can be added? Rather you say, well go off and add yourself how your parts like to be added. And the result of that is you can get this very, very, very complex hierarchy where a lot of things just get done and rooted to the right place automatically.

Let's stop for questions.

AUDIENCE: You say you get this for free. One thing that strikes me is that now you've lost kind of the cleanness of the break between what's on top and what's underneath. In other words, now you're defining some of the lower-level procedures in terms of things above their own line. Isn't that dangerous? Or, if nothing more, a little less structured?

PROFESSOR: No, I-- the question is whether that's less structured. Depends on what you mean by structure. All this is doing is recursion. See, it's saying that the way you add these guys is to use that. And that's not less structured, it's just a recursive structure. So I don't think it's particularly any less clean.

AUDIENCE: Now when you want to change the multiplier or the add operator, suddenly you've got tremendous consequences underneath that you're not even sure the extent of.

PROFESSOR: That's right, but it depends what you mean. See, this goes both ways. What would be a good example? I ignored greatest common divisor, for instance. I ignored that problem just to keep the example simple. But if I suddenly decided that plus rat here should do a GCD computation and install that, then that immediately becomes available to all of these, to that guy, and that guy, and that guy, and all the way down.

So it depends what you mean by the coherence of your system. It's certainly true that you might want to have a special different one that didn't filter down through the coefficients, but the nice thing about this particular example is that mostly you do.

AUDIENCE: Isn't that the problem, I think, that you're getting to tied in with the fact that the structuring, the recursiveness of that structuring there is actually in execution as opposed to just definition of the actual types themselves?

PROFESSOR: I think I understand the question. The point is that these types evolve and get more and more complex as the thing's actually running. Is that what--

AUDIENCE: Yes. As it's running.

PROFESSOR: --what you're saying? Yes, the point is--

AUDIENCE: As opposed to the basic definitions.

PROFESSOR: Right. The type structure is sort of recursive. It's not that you can make this finite list of the actual things they might look like before the system runs. It's something that evolves. So if you want to specify that system, you have to do in some other way than by this finite list. You have to do it by a recursive structure.

AUDIENCE: Because the basic structure of the types is pretty clean and simple.

PROFESSOR: Right. Yes?

AUDIENCE: I have a question. I understand once you have your data structure set up, how it pulls off complex and passes that down, and then pulls off rect, passes that down. But if you're just a user and you don't know anything about rect or polar or whatever, how do you initially set up that data structure so that everything goes to the right spot? If I just have the equation over there on the left and I just want to add, multiply complex numbers--

PROFESSOR: Well that's the wonderful thing. If you're just a user you say "mul."

AUDIENCE: And it figures out that I mean complex numbers? Or how do I tell it that I want--

PROFESSOR: Well you're going to have in your hands complex numbers. See what you would have at some level, as a real user, is a constructor for complex numbers.

AUDIENCE: So then I have to make complex numbers?

PROFESSOR: So you have to make them. What you would probably have as a user is some little thing in the reader loop, which would give you some plausible way to type in a complex number, in whatever format you like. Or it might be that you're never typing them in. Someone's just handing you a complex number.

AUDIENCE: OK, so if I had a complex number that had a polynomial in it, I'd have to make my polynomial and then make my complex number.

PROFESSOR: Right if you wanted it constructed from scratch. At some point you construct them from scratch. But what you don't have to know of that is when you have the object you can just say "mul." And it'll multiply. Yeah?

AUDIENCE: I think the question that was being posed here is, say if I want to change my presentation of complexes, or some operation of complex, how much real code I will have to gets around with, or change to change it in one specific operation?

PROFESSOR: [UNINTELLIGIBLE] what you have to change. And the point is that you only have to change what you're changing. See if Martha decides that she would rather-- let's see something silly-- like change the order in the pair. Like angle and magnitude in the other order, she just makes that change locally. And the whole thing will propagate through the system in the right way. Or if suddenly you said, gee, I have another representation for rationals. And I'm going to stick it here, by filing those operations in the table. Then suddenly all of these polynomials whose coefficients are coefficients of coefficients, or whatever, also can automatically have available that representation. That's the power of this particular one.

AUDIENCE: I'm not sure if I can even pose an intelligent sounding question. But somehow this whole thing went really nicely to this beautiful finish where all the things seemed to fall into place. Sort of seemed a little contrived. That's all for the sake, I'm sure, of teaching. I doubt that the guys who first did this-- and I could be wrong-- figured it all out so that when they just all put it all together, you could all of the sudden, blam, do any kind of arithmetic on any kind of object. It seems like maybe they had to play with it for a while and had to bash it and rework it.

And it seems like that's the kind of problem we're really faced with we start trying to design a really complex system, is having lots of different kinds of parts and not even knowing what kinds of operations we're going to want to do on those parts. How to organize the operations in this nice way so that no matter what you do, when you start putting them together everything starts falling out for free.

PROFESSOR: OK, well that's certainly a very intelligent question. One part is this is a very good methodology that people have discovered a lot coming from symbolic algebra. Because there are a lot of complications. To allow you to implement these things before you decide what you want all the operations to be, and all of that. So in some sense it's an

answer that people have discovered by wading through this stuff. In another sense, it is a very contrived example.

AUDIENCE: It seems like to be able to do this you do have to wade through it for a certain amount of time before you can become good at it.

PROFESSOR: Let me show you how terribly contrived this is. So you can write all these wonderful things. But the system that I wrote here, and if we had another half an hour to give this lecture I would have given this part of it, which says, notice that it breaks down if I tell it to do something as foolish as add 3 plus 7/2. Because what will happen is you'll get to operate-2, and operate-2 will say, oh this is type number, and that's type rational. I don't know how to add them.

So you'd like the system at least to be able to say something like, gee, before you do that change that to 3/1. Turn it into a rational number, hand that to the rational package. That's the thing I didn't talk about in this lecture. It's a little bit in the book, which talks about the problem of what's called coercion. Where you wanted-- see, having so carefully set up all of these types as distinct objects, a lot of times you want to also put in knowledge about how to view an ordinary number as a kind of rational. Or view an ordinary number as a kind of complex. That's where the complexity in the system really starts happening, where you talk about, see where do I put that knowledge? Is it rational to know that ordinary numbers might be pieces of [UNINTELLIGIBLE] of them?

Or they're terrible, terrible examples, like if I might want to add a complex number to a rational number. Bad example. 5/7. Then somebody's got to know that I have to convert these to another type, which is complex numbers whose parts might be rationals. And who worries about that? Does complex worry about that? Does rational worry about that? Does plus worry about that?

That's where the real complexity comes in. And that's where it's pretty well sorted out. And a lot of, in fact, all of this message passing stuff was motivated by problems like this. And when you really push it, people are-- somehow the algebraic manipulation problem seems to be so complex that the people who are always at the edge of it are exactly in the state you said. They're wading through this thing, mucking around, seeing what they use, trying to distill stuff.

AUDIENCE: I just want to come back to this issue of complexity once more. It certainly seems to be true that you have a great deal of flexibility in altering the lower level kinds of things. But it is true that you are, in a sense, freezing higher level operations. Or at least if you change them you don't know where all of the changes are going to show up, or how they are.

PROFESSOR: OK, that's an extremely good question. What I have to do is, if I decide there's a new general operation called equality test, then all of these people have to decide whether or not they would like to have an equality test by looking in the table. There're ways to decentralize it even more. That's what I sort of hinted at last time, where I said you could not only have this type as a symbol, but you actually might store in each object the operations that it knows of that.

So you might have things like greatest common divisor, which is a thing here which is defined only for integers, and not in general for rational numbers. So it might be a very, very fragmented system. And then depending on where you want your flexibility, there's a whole spectrum of places that you can build that in. But you're pointing at the place where this starts being weak, that there has to be some agreement on top here about these general operations. Or at least people have to think about them. Or you might decide, you might have a table that's very sparse, that only has a few things in it. But there are lot of ways to play that game. OK, thank you.

[MUSIC: "JESU, JOY OF MAN'S DESIRING" BY JOHANN SEBASTIAN BACH]

MIT OpenCourseWare
http://ocw.mit.edu

6.001 Structure and Interpretation of Computer Programs, Spring 2005

Please use the following citation format:

> Eric Grimson, Peter Szolovits, and Trevor Darrell, *6.001 Structure and Interpretation of Computer Programs, Spring 2005*. (Massachusetts Institute of Technology: MIT OpenCourseWare). http://ocw.mit.edu (accessed MM DD, YYYY). License: Creative Commons Attribution-Noncommercial-Share Alike.

Note: Please use the actual date you accessed this material in your citation.

For more information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms