PROFESSOR: Well, now that we've given you some power to make independent local state and to model objects, I thought we'd do a bit of programming of a very complicated kind, just to illustrate what you can do with this sort of thing. I suppose, as I said, we were motivated by physical systems and the ways we like to think about physical systems, which is that there are these things that the world is made out of. And each of these things has particular independent local state, and therefore it is a thing. That's what makes it a thing.

And then we're going to say that in the model in the world--we have a world and a model in our minds and in the computer of that world. And what I want to make is a correspondence between the objects in the world and the objects in the computer, the relationships between the objects in the world and the relationships between those same obj...--the model objects in the computer, and the functions that relate things in the world to the functions that relate things in the computer.

This buys us modularity. If we really believe the world is like that, that it's made out of these little pieces, and of course we could arrange our world to be like that, we could only model those things that are like that, then we can inherit the modularity in the world into our programming. That's why we would invent some of this object-oriented programming.

Well, let's take the best kind of objects I know. They're completely--they're completely wonderful: electrical systems. Electrical systems really are the physicist's best, best objects. You see over here I have some piece of machinery. Right here's a piece of machinery. And it's got an electrical wire connecting one part of the machinery with another part of the machinery. And one of the wonderful properties of the electrical world is that I can say this is an object, and this is an object, and they're-- the connection between them is clear. In principle, there is no connection that I didn't describe with these wires.

Let's say if I have light bulbs, a light bulb and a power supply that's plugged into the outlet. Then the connection is perfectly clear. There's no other connections that we know of. If I were to tie a knot in the wire that connects the light bulb to the power supply, the light remains lit up. It doesn't care. That the way the physics is arranged is such that the connection can be made abstract, at least for low frequencies and things like that. So in fact, we have captured all of the connections there really are.

Well, as you can go one step further and talk about the most abstract types of electrical systems we have, digital to dual circuits. And here there are certain kinds of objects. For example, in digital circuits we have things like inverters. We have things like and-gates. We have things like or-gates. We connect them together by sort-of wires which represent abstract signals. We don't really care as physical variables whether these are voltages or currents

or some combination or anything like that, or water, water pressure. These abstract variables represent certain signals. And we build systems by wiring these things together with wires.

So today what I'm going to show you, right now, we're going to build up an invented language in Lisp, embedded in the same sense that Henderson's picture language was embedded, which is not the same sense as the language of pattern match and substitution was done yesterday. The pattern match/substitution language was interpreted by a Lisp program. But the embedding of Henderson's program is that we just build up more and more procedures that encapsulate the structure we want.

So for example here, I'm going to have some various primitive kinds of objects, as you see, that one and that one. I'm going to use wires to combine them. The way I represent attaching-- I can make wires. So let's say A is a wire. And B is a wire. And C is a wire. And D is a wire. And E is wire. And S is a wire.

Well, an or-gate that has both inputs, the inputs being A and B, and the output being Y or D, you notate like this. An and-gate, which has inputs A and B and output C, we notate like that. By making such a sequence of declarations, like this, I can wire together an arbitrary circuit. So I've just told you a set of primitives and means of combination for building digital circuits, when I need more in a real language than abstraction.

And so for example, here I have--here I have a half adder. It's something you all know if you've done any digital design. It's used for adding numbers together on A and B and putting out a sum and a carry. And in fact, the wiring diagram is exactly what I told you. A half adder with things that come out of the box-- you see the box, the boundary, the abstraction is always a box. And there are things that come out of it, A, B, S, and C. Those are the declared variables--declared variables of a lambda expression, which is the one that defines half adder.

And internal to that, I make up some more wires, D and E, which I'm going to use for the interconnect-- here E is this one and D is this wire, the interconnect that doesn't come through the walls of the box-- and wire things together as you just saw. And the nice thing about this that I've just shown you is this language is hierarchical in the right way. If a language isn't hierarchical in the right way, if it turns out that a compound object doesn't look like a primitive, there's something wrong with the language-- at least the way I feel about that.

So here we have--here, instead of starting with mathematical functions, or things that compute mathematical functions, which is what we've been doing up until now, instead of starting with things that look like mathematical functions, or compute such things, we are starting with things that are electrical objects and we build up more electrical objects. And the glue we're using is basically the Lisp structure: lambdas. Lambda is the ultimate glue, if you will.

And of course, half adder itself can be used in a more complicated abstraction called a full adder, which in fact

involves two half adders, as you see here, hooked together with some extra wires, that you see here, S, C1, and C2, and an or-gate, to manufacture a full adder, which takes a input number, another input number, a carry in, and produces output, a sum and a carry out. And out of full adders, you can make real adder chains and big adders.

So we have here a language so far that has primitives, means of combination, and means of abstraction to real language. Now, how are we going to implement this? Well, let's do it easily. Let's look at the primitives. The only problem is we have to implement the primitives. Nothing else has to be implemented, because we're picking up the means of combination and abstraction from Lisp, inheriting them in the embedding.

OK, so let's look at a particular primitive. An inverter is a nice one. Now, inverter has two wires coming in, an in and an out. And somehow, it's going to have to know what to do when a signal comes in. So somehow it's going to have to tell its input wire-- and now we're going to talk about objects and we're going to see this in a little more detail soon-- but it's going to have to tell its input wire that when you change, tell me. So this object, the object which is the inverter has to tell the object which is the input wire, hi, my name is George. And my, my job is to do something with results when you change. So when you change, you get a change, tell me about it. Because I've got to do something with that.

Well, that's done down here by adding an action on the input wire called invert-in, where invert-in is defined over here to be a procedure of no arguments, which gets the logical not of the signal on the input wire. And after some delay, which is the inverter delay, all these electrical objects have delays, we'll do the following thing-- set the signal on the output wire to the new value. A very simple program.

Now, you have to imagine that the output wire has to be sensitive and know that when its signal changes, it may have to tell other guys, hey, wake up. My value has changed. So when you hook together inverter with an and-gate or something like that, there has to be a lot of communication going on in order to make sure that the signal propagates right. And down here is nothing very exciting. This is just the definition of logical not for some particular representations of the logical values-- 1, 0 in this case.

And we can look at things more complicated like and-gates. And-gates take two inputs, A1 and A2, we'll call them, and produce an output. But the structure of the and-gate is identical to the one we just saw. There's one called an and-action procedure that's defined, which is the thing that gets called when an input is changed. And what it does, of course, is nothing more than compute the logical and of the signals on the inputs. And after some delay, called the and-gate delay, calls this procedure, which sets a signal on the output to a new value.

Now, how I implement these things is all wishful thinking. As you see here, I have an assignment operation. It's not set. It's a derived assignment operation in the same way we had functions that were derived from CAR and CDR.

So I, by convention, label that with an exclamation point. And over here, you see there's an action, which is to inform the wire, called A1 locally in this and-gate, to call the and-action procedure when it gets changed, and the wire A2 to call the and-action procedure when it gets changed. All very simple.

Well, let's talk a little bit about this communication that must occur between these various parts. Suppose, for example, I have a very simple circuit which contains an and with wires A and B. And that connects through a wire called C to an inverter which has a wire output called D. What are the comput...--here's the physical world. It's an abstraction of the physical world. Now I can buy these out of little pieces that you get at Radio Shack for a few cents. And there are boxes that act like this, which have little numbers on them like LS04 or something.

Now supposing I were to try to say what's the computational model. What is the thing that corresponds to that, that part of reality in the mind of us and in the computer? Well, I have to assign for every object in the world an object in the computer, and for every relationship in the world between them a relationship in the computer. That's my goal.

So let's do that. Well, I have some sort of thing called the signal, A. This is A. It's a signal. It's a cloudy thing like that. And I have another one down here which I'm going to call B. It's another signal. Now this signal--these two signals are somehow going to have to hook together into a box, let's call it this, which is the and-gate, action procedure. That's the and-gate's action procedure.

And it's going to produce--well, it's going to interact with a signal object, which we call C--a wire object, excuse me, we call C. And then the-- this is going to put out again, or connect to, another action procedure which is one associated with the inverter in the world, not. And I'm going to have another--another wire, which we'll call D.

So here's my layout of stuff. Now we have to say what's inside them and what they have to know to compute. Well, every--every one of these wires has to know what the value of the signal that's on that wire is. So there's going to be some variable inside here, we'll call it signal. And he owns a value. So there must be some environment associated with this. And for each one of these, there must be an environment that binds signal. And there must be a signal here, therefore. And presumably, signal's a value that's either 1 or 0, and signal.

Now, we also have to have some list of people to inform if the signal here changes. We're going to have to inform this. So I've got that list. We'll call it the Action Procedures, AP. And it's presumably a list. But the first thing on the list, in this case, is this guy. And the action procedures of this one happens to have some list of stuff. There might be other people who are sharing A, who are looking at it. So there might be other guys on this list, like somebody over there that we don't know about. It's the other guy attached to A.

And the action procedure here also has to point to that, the list of action procedures. And of course, that means

this one, its action procedures has to point up to here. This is the things-- the people it has to inform. And this guy has some too. But I don't know what they are because I didn't draw it in my diagram. It's the things connected to D.

Now, it's also the case that when the and-action procedure is awakened, saying one of the people who know that you've told--one of the people you've told to wake you up if their signal changes, you have to go look and ask them what's their signal so you can do the and, and produce a signal for this one. So there has to be, for example, information here saying A1, my A1 is this guy, and my A2 is this guy. And not only that, when I do my and, I'm going to have to tell this guy something. So I need an output-- being this guy.

And similarly, this guy's going to have a thing called the input that he interrogates to find out what the value of the signal on the input is, when the signal wakes up and says, I've changed, and sends a message this way saying, I've changed. This guy says, OK, what's your value now? When he gets that value, then he's going to have to say, OK, output changes this guy, changes this guy. And so on. And so I have to have at least that much connected-ness.

Now, let's go back and look, for example, at the and-gate. Here we are back on this slide. And we can see some of these parts. For any particular and-gate, there is an A1, there is an A2, and the output. And those are, those are an environment that was created at the--those produce a frame at the time and-gate was called, a frame where A1, A2, and output are--have as their values, they're bound to the wires which, they are--which were passed in. In that environment, I constructed a procedure-- this one right there. And-action procedure was constructed in that environment. That was the result of evaluating a lambda expression.

So it hangs onto the frame where these were defined. Local--part of its local state is that. The and-action procedure, therefore, has access to A1, A2, and output as we see here. A1, A2, and output.

Now, we haven't looked inside of a wire yet. That's all that remains. Let's look at a wire. Like the overhead, very good. Well, the wire, again, is a, is a somewhat complicated mess. Ooh, wrong one. It's a big complicated mess, like that. But let's look at it in detail and see what's going on.

Well, the wire is one of these. And it has to have two things that are part of it, that it's state. One of them is the signal we see here. In other words, when we call make-wire to make a wire, then the first thing we do is we create some variables which are the signal and the action procedures for this wire. And in that context, we define various functions--or procedures, excuse me, procedures.

One of them is called set-my-signal to a new value. And what that does is takes a new value in. If that's equal to my current value of my signal, I'm done. Otherwise, I set the signal to the new value and call each of the action

procedures that I've been, that I've been--what's the right word?-- introduced to. I get introduced when the and-gate was applied to me. I add action procedure at the bottom.

Also, I have to define a way of accepting an action procedure-- which is what you see here--- which increments my action procedures using set to the result of CONSing up a new process--a procedure, which is passed to me, on to my actions procedures list. And for technical reasons, I have to call that procedure one. So I'm not going to tell you anything about that, that has to do with event-driven simulations and getting them started, which takes a little bit of thinking.

And finally, I'm going to define a thing called the dispatcher, which is a way of passing a message to a wire, which is going to be used to extract from it various information, like what is the current signal value? What is the method of setting your signal? I want to get that out of it. How do I--how do I add another action procedure? And I'm going to return that dispatch, that procedure as a value.

So the wire that I've constructed is a message accepting object which accepts a message like, like what's your method of adding action procedures? In fact, it'll give me a procedure, which is the add action procedure, which I can then apply to an action procedure to create another action procedure in the wire. So that's a permission. So it's given me permission to change your action procedures.

And in fact, you can see that over here. Next slide. Ah. This is nothing very interesting. The call each of the action procedures is just a CDRing down a list. And I'm not going to even talk about that anymore. We're too advanced for that. However, if I want to get a signal from a wire, I ask the wire-- which is, what is the wire? The wire is the dispatch returned by creating the wire. It's a procedure. I call that dispatch on the message get-signal. And what I should expect to get is a method of getting a signal. Or actually, I get the signal.

If I want to set a signal, I want to change a signal, then what I'm going to do is take a wire as an argument and a new value for the signal, I'm going to ask the wire for permission to set its signal and use that permission, which is a procedure, on the new value. And if we go back to the overhead here, thank you, if we go back to the overhead here, we see that the method-- if I ask for the method of setting the signal, that's over here, it's set-my-signal, a procedure that's defined inside the wire, which if we look over here is the thing that says set my internal value called the signal, my internal variable, which is the signal, to the new value, which is passed to me as an argument, and then call each of the action procedures waking them up. Very simple.

Going back to that slide, we also have the one last thing-- which I suppose now you can easily work out for yourself-- is the way you add an action. You take a wire--a wire and an action procedure. And I ask the wire for permission to add an action. Getting that permission, I use that permission to give it an action procedure. So that's a real object.

There's a few more details about this. For example, how am I going to control this thing? How do I do these delays? Let's look at that for a second. The next one here. Let's see. We know when we looked at the and-gate or the not-gate that when a signal changed on the input, there was a delay. And then it was going to call the procedure, which was going to change the output.

Well, how are we going to do this? We're going to make up some mechanism, a fairly complicated mechanism at that, which we're going to have to be very careful about. But after a delay, we're going to do an action. A delay is a number, and an action is a procedure. What that's going to be is they're going to have a special structure called an agenda, which is a thing that organizes time and actions. And we're going to see that in a while. I don't want to get into that right now.

But the agenda has a moment at which--at which something happens. We're setting up for later at some moment, which is the sum of the time, which is the delay time plus the current time, which the agenda thinks is now. We're going to set up to do this action, and add that to the agenda.

And the way this machine will now run is very simple. We have a thing called propagate, which is the way things run. If the agenda is empty, we're done--if there's nothing more to be done. Otherwise, we're going to take the first item off the agenda, and that's a procedure of no arguments. So that we're going to see extra parentheses here. We call that on no arguments. That takes the action. Then we remove that first item from the agenda, and we go around the propagation loop. So that's the overall structure of this thing.

Now, there's a, a few other things we can look at. And then we're going to look into the agenda a little while from now. Now the overhead again. Well, in order to set this thing going, I just want to show you some behavior out of this simulator. By the way, you may think this simulator is very simple, and probably too simple to be useful. The fact of the matter is that this simulator has been used to manufacture a fairly large computer. So this is a real live example.

Actually, not exactly this simulator, because I'll tell you the difference. The difference is that there were many more different kinds of primitives. There's not just the word inverter or and-gate. There were things like edge-triggered, flip-flops, and latches, transparent latches, and adders, and things like that. And the difficulty with that is that there's pages and pages of the definitions of all these primitives with numbers like LS04. And then there's many more parameters for them. It's not just one delay. There's things like set up times and hold times and all that. But with the exception of that part of the complexity, the structure of the simulator that we use for building a real computer, that works is exactly what you're seeing here.

Well in any case, what we have here is a few simple things. Like, there's inverter delays being set up and making

a new agenda. And then we can make some inputs. There's input-1, input-2, a sum and a carry, which are wires. I'm going to put a special kind of object called a probe onto, onto some of the wires, onto sum and onto carry. A probe is a, can object that has the property that when you change a wire it's attached to, it types out a message. It's an easy thing to do.

And then once we have that, of course, the way you put the probe on, the first thing it does, it says, the current value of the sum at time 0 is 0 because I just noticed it. And the value of the carry at time 0, this is the time, is 0. And then we go off and we build some structure. Like, we can build a structure here that says you have a half-adder on input-1, input-2, sum, and carry. And we're going to set the signal on input-1 to 1. We do some propagation. At time 8, which you could see going through this thing if you wanted to, the new value of sum became 1. And the thing says I'm done. That wasn't very interesting.

But we can send it some more signals. Like, we set-signal on input-2 to be one. And at that time if we propagate, then it carried at 11, the carry becomes 1, and at 16, the sum's new value becomes 0. And you might want to work out that, if you like, about the digital circuitry. It's true, and it works. And it's not very interesting. But that's the kind of behavior we get out of this thing.

So what I've shown you right now is a large-scale picture, how you, at a bigger, big scale, you implement an event-driven simulation of some sort. And how you might organize it to have nice hierarchical structure allowing you to build abstract boxes that you can instantiate. But I haven't told you any of the details about how this agenda and things like that work. That we'll do next. And that's going to involve change and mutation of data and things like that. Are there any questions now, before I go on? Thank you. Let's take a break.

Well, we've been making a simulation. And the simulation is an event-driven simulation where the objects in the world are the objects in the computer. And the changes of state that are happening in the world in time are organized to be time in the computer, so that if something happens after something else in the world, then we have it happen after, after the corresponding events happen in the same order in the computer. That's where we have assignments, when we make that alignment.

Right now I want to show you a way of organizing time, which is an agenda or priority queue, it's sometimes called. We'll do some--we'll do a little bit of just understanding what are the things we need to be able to do to make agendas. And so we're going to have--and so right now over here, I'm going to write down a bunch of primitive operations for manipulating agendas. I'm not going to show you the code for them because they're all very simple, and you've got listings of all that anyway.

So what do we have? We have things like make-agenda which produces a new agenda. We can ask--we get the current-time of an agenda, which gives me a number, a time. We can get--we can ask whether an agenda is

empty, empty-agenda. And that produces either a true or a false.

We can add an object to an agenda. Actually, what we add to an agenda is an operation--an action to be done. And that takes a time, the action itself, and the agenda I want to add it to. That inserts it in the appropriate place in the agenda. I can get the first item off an agenda, the first thing I have to do, which is going to give me an action. And I can remove the first item from an agenda. That's what I have to be able to do with agendas. That is a big complicated mess. From an agenda.

Well, let's see how we can organize this thing as a data structure a bit. Well, an agenda is going to be some kind of list. And it's going to be a list that I'm going to have to be able to modify. So we have to talk about modifying of lists, because I'm going to add things to it, and delete things from it, and things like that. It's organized by time. It's probably good to keep it in sorted order.

But sometimes there are lots of things that happen at the same time--approximate same time. What I have to do is say, group things by the time at which they're supposed to happen. So I'm going to make an agenda as a list of segments. And so I'm going to draw you a data structure for an agenda, a perfectly reasonable one.

Here's an agenda. It's a thing that begins with a name. I'm going to do it right now out of list structure. It's got a header. There's a reason for the header. We're going to see the reason soon.

And it will have a segment. It will have--it will be a list of segments. Supposing this agenda has two segments, they're the car's-- successive car's of this list. Each segment is going to have a time-- say for example, 10-- that says that the things that happen in this segment are at time 10.

And what I'm going to have in here is another data structure which I'm not going to describe, which is a queue of things to do at time 10. It's a queue. And we'll talk about that in a second. But abstractly, the queue is just a list of things to do at a particular time. And I can add things to a queue. This is a queue. There's a time, there's a segment.

Now, I may have another segment in this agenda. Supposing this is stuff that happens at time 30. It has, of course, another queue of things that are queued up to be done at time 30. Well, there are various things I have to be able to do to an agenda.

Supposing I want to add to an agenda another thing to be done at time 10. Well, that's not very hard. I'm going to walk down here, looking for the segment of time 10. It is possible that there is no segment of time 10. We'll cover that case in a second. But if I find a segment of time 10, then if I want to add another thing to be done at time 10, I just increase that queue-- "just increase" isn't such an obvious idea. But I increase the things to be done at that

time.

Now, supposing I want to add something to be done at time 20. There is no segment for time 20. I'm going to have to create a new segment. I want my time 20 segment to exist between time 10 and time 30. Well, that takes a little work. I'm going to have to do a CONS. I'm going to have to make a new element of the agenda list--list of segments. I'm going to have to change. Here's change. I'm going to have to change the CDR of the CDR of the agenda to point that a new CONS of the new segment and the CDR of the CDR of the CDR of the agenda, the CD-D-D-DR.

And this is going to have a new segment now of time 20 with its own queue, which now has one element in it. If I wanted to add something at the end, I'm going to have to replace the CDR of this, of this list with something. We're going to have to change that piece of data structure. So I'm going to need new primitives for doing this. But I'm just showing you why I need them.

And finally, if I wanted to add a thing to be done at time 5, I'm going to have to change this one, because I'm going to have to add it in over here, which is why I planned ahead and had a header cell, which has a place. If I'm going to change things, I have to have places for the change. I have to have a place to make the change.

If I remove things from the agenda, that's not so hard. Removing them from the beginning is pretty easy, which is the only case I have. I can go looking for the first, the first segment. I see if it has a non-empty queue. If it has a non-empty queue, well, I'm going to delete one element from the queue, like that. If the queue ever becomes empty, then I have to delete the whole segment. And then this, this changes to point to here. So it's quite a complicated data structure manipulation going on, the details of which are not really very exciting.

Now, let's talk about queues. They're similar. Because each of these agendas has a queue in it. What's a queue? A queue is going to have the following primitive operations. To make a queue, this gives me a new queue. I'm going to have to be able to insert into a queue a new item. I'm going to have to be able to delete from a queue the first item in the queue. And I want to be able to get the first thing in the queue from some queue. I also have to be able to test whether a queue is empty.

And when you invent things like this, I want you to be very careful to use the kinds of conventions I use for naming things. Notice that I'm careful to say these change something and that tests it. And presumably, I did the same thing over here. OK, and there should be an empty test over here.

OK, well, how would I make a queue? A queue wants to be something I can add to at the end of, and pick up the thing at the beginning of. I should be able to delete from the beginning and add to the end. Well, I'm going to show you a very simple structure for that. We can make this out of CONSes as well.

Here's a queue. It has--it has a queue header, which contains two parts-- a front pointer and a rear pointer. And here I have a queue with two items in it. The first item, I don't know, it's perhaps a 1. And the second item, I don't know, let's give it a 2. The reason why I want two pointers in here, a front pointer and a rear pointer, is so I can add to the end without having to chase down from the beginning.

So for example, if I wanted to add one more item to this queue, if I want to add on another item to be worried about later, all I have to do is make a CONS, which contains that item, say a 3. That's for inserting 3 into the queue. Then I have to change this pointer here to here. And I have to change this one to point to the new rear.

If I wish to take the first element of the queue, the first item, I just go chasing down the front pointer until I find the first one and pick it up. If I wish to delete the first item from the queue, delete-queue, all I do is move the front pointer along this way. The new front of the queue is now this. So queues are very simple too.

So what you see now is that I need a certain number of new primitive operations. And I'm going to give them some names. And then we're going to look into how they work, and how they're used. We have set the CAR of some pair, or a thing produced by CONSing, to a new value. And set the CDR of a pair to a new value. And then we're going to look into how they work.

I needed setting CAR over here to delete the first element of the queue. This is the CAR, and I had to set it. I had to be able to set the CDR to be able to move the rear pointer, or to be able to increment the queue here. All of the operations I did were made out of those that I just showed you on the, on the last blackboard. Good. Let's pause the time, and take a little break then.

When we originally introduced pairs made out of CONS, made by CONS, we only said a few axioms about them, which were of the form-- what were they-- for all X and Y, the CAR of the CONS of X and Y is X and the CDR of the CONS of X and Y is Y. Now, these say nothing about whether a CONS has an identity like a person. In fact, all they say is something sort of abstract, that a CONS is the parts it's made out of. And of course, two things are made out of the same parts, they're the same, at least from the point of view of these axioms.

But by introducing assignment-- in fact, mutable data is a kind of assignment, we have a set CAR and a set CDR-- by introducing those, these axioms no longer tell the whole story. And they're still true if written exactly like this. But they don't tell the whole story. Because if I'm going to set a particular CAR in a particular CONS, the questions are, well, is that setting all CARs and all CONSes of the same two things or not? If I--if we use CONSes to make up things like rational numbers, or things like 3 over 4, supposing I had two three-fourths. Are they the same one-- or are they different?

Well, in the case of numbers, it doesn't matter. Because there's no meaning to changing the denominator of a

number. What you could do is make a number which has a different denominator. But the concept of changing a number which has to have a different denominator is sort of a very weird, and sort of not supported by what you think of as mathematics. However, when these CONSes represent things in the physical world, then changing something like the CAR is like removing a piece of the fingernail. And so CONSes have an identity.

Let me show you what I mean about identity, first of all. Let's do some little example here. Supposing I define A to the CONS of 1 and 2. Well, what that means, first of all, is that somewhere in some environment I've made a symbol A to have a value which is a pair consisting of pointers to a 1 and a pointer to a 2, just like that. Now, supposing I also say define B to be the CONS-- it doesn't matter, but I like it better, it's prettier-- of A and A.

Well, first of all, I'm using the name A twice. At this moment, I'm going to think of CONSes as having identity. This is the same one. And so what that means is I make another pair, which I'm going to call B. And it contains two pointers to A. At this point, I have three names for this object. A is its name. The CAR of B is its name. And the CDR of B is its name. It has several aliases, they're called.

Now, supposing I do something like set-the-CAR, the CAR of the CAR of B to 3. What that means is I find the CAR of B, that's this. I set the CAR of that to be 3, changing this. I've changed A. If I were to ask what's the CAR of A-- of A now? I would get out 3, even though here we see that A was the CONS of 1 and 2.

I caused A to change by changing B. There is sharing here. That's sometimes what we want. Surely in the queues and things like that, that's exactly what we defined our--organized our data structures to facilitate-- sharing. But inadvertent sharing, unanticipated interactions between objects, is the source of most of the bugs that occur in complicated programs. So by introducing this possibility of things having identity and sharing and having multiple names for the same thing, we get a lot of power. But we're going to pay for it with lots of complexity and bugs.

So also, for example, if I just looked at this just to drive that home, the CADR of B, which has nothing to do with even the CAR of B, apparently. The CADR of B, what's that? Take that CDR of B and now take the CAR of that. Oh, that's 3 also. So I can have non-local interactions by sharing. And I have to be very careful of that.

Well, so far, of course, it seems I've introduced several different assignment operators-- set, set CAR, set CDR. Well, maybe I should just get rid of set CAR and set CDR. Maybe they're not worthwhile. Well, the answer is that once you let the camel's nose into the tent, the rest of him follows. All I have to have is set, and I can make all of the--all of the bad things that can happen.

Let's play with that a little bit. A couple of days ago, when we introduced compound data, you saw Hal show you a definition of CONS in terms of a message acceptor. I'm going to show you even a more horrible thing, a definition of CONS in terms of nothing but air, hot air. What is the definition of CONS, of the old functional kind, in terms of

purely lambdic expressions, procedures? Because I'm going to then modify this definition to get assignment to be only one kind of assignment, to get rid of the set CAR and set CDR in terms of set.

So what if I define CONS of X and Y to be a procedure of one argument called a message M, which calls that message on X and Y? This [? idea ?] was invented by Alonzo Church, who was the greatest programmer of the 20th century, although he never saw a computer. It was done in the 1930s. He was a logician, I suppose at Princeton at the time. Define CAR of X to be the result of applying X to that procedure of two arguments, A and D, which selects A. I will define CDR of X to be that procedure, to be the result of applying X to that procedure of A and D, which selects D.

Now, you may not recognize this as CAR, CDR, and CONS. But I'm going to demonstrate to you that it satisfies the original axioms, just once. And then we're going to do some playing of games. Consider the problem CAR of CONS of, say, 35 and 47. Well, what is that? It is the result of taking car of the result of substituting 35 and 47 for X and Y in the body of this. Well, that's easy enough. That's CAR of the result of substituting into lambda of M, M of 35 and 47.

Well, what this is, is the result of substituting this object for X in the body of that. So that's just lambda of M-- that's substituted, because this object is being substituted for X, which is the beginning of a list, lambda of M-- M of 35 and 47, applied to that procedure of A and D, which gives me A. Well, that's the result of substituting this for M here. So that's the same thing as lambda of A, D, A, applied to 35 and 47. Oh, well that's 35. That's substituting 35 for A and for 47 for D in A. So I don't need any data at all, not even numbers. This is Alonso Church's hack.

Well, now we're going to do something nasty to him. Being a logician, he wouldn't like this. But as programmers, let's look at the overhead. And here we go. I'm going to change the definition of CONS. It's almost the same as Alonzo Church's, but not quite. What do we have here? The CONS of two arguments, X and Y, is going to be that procedure of one argument M, which supplies M to X and Y as before, but also to two permissions, the permission to set X to N and the permission to set Y to N, given that I have an N.

So besides the things that I had here in Church's definition, what I have is that the thing that CONS returns will apply its argument to not just the values of the X and Y that the CONS is made of, but also permissions to set X and Y to new values. Now, of course, just as before, CAR is exactly the same. The CAR of X is nothing more than applying X, as in Church's definition, to a procedure, in this case, of four arguments, which selects out the first one. And just as we did before, that will be the value of X that was contained in the procedure which is the result of evaluating this lambda expression in the environment where X and Y are defined over here. That's the value of CONS.

Now, however, the exciting part. CDR, of course, is the same. The exciting part, set CAR and set CDR. Well,

they're nothing very complicated anymore. Set CAR of a CONS X to a new value Y is nothing more than applying that CONS, which is the procedure of four--the procedure of one argument which applies its argument to four things, to a procedure which is of four arguments-- the value of X, the value of Y, permission to set X, the permission to set Y-- and using it--using that permission to set X to the new value. And similarly, set-cdr is the same thing.

So what you've just seen is that I didn't introduce any new primitives at all. Whether or not I want to implement it this way is a matter of engineering. And the answer is of course I don't implement it this way for reasons that have to do with engineering. However in principle, logically, once I introduced one assignment operator, I've assigned-- I've introduced them all. Are there any questions? Yes, David.

AUDIENCE: I can follow you up until you get--I can follow all of that. But when we bring in the permissions, defining CONS in terms of the lambda N, I don't follow where N gets passed.

PROFESSOR: Oh, I'm sorry. I'll show you. Let's follow it. Of course, we could do it on the blackboard. It's not so hard. But it's also easy here.

Supposing I wish to set-cdr of X to Y. See that right there. set-cdr of X to Y. X is presumably a CONS, a thing resulting from evaluating CONS. Therefore X comes from a place over here, that that X is of the result of evaluating this lambda expression. Right? That when I evaluated that lambda expression, I evaluated it in an environment where the arguments to CONS were defined.

That means that as free variables in this lambda expression, there is the--there are in the frame, which is the parent frame of this lambda expression, the procedure resulting from this lambda expression, X and Y have places. And it's possible to set them. I set them to an N, which is the argument of the permission. The permission is a procedure which is passed to M, which is the argument that the CONS object gets passed.

Now, let's go back here in the set-cdr The CONS object, which is the first argument of set-cdr gets passed an argument. That--there's a procedure of four things, indeed, because that's the same thing as this M over here, which is applied to four objects. The object over here, SD, is, in fact, this permission. When I use SD, I apply it to Y, right there. So that comes from this.

AUDIENCE: So what do you--

PROFESSOR: So to finish that, the N that was here is the Y which is here. How's that?

AUDIENCE: Right, OK. Now, when you do a set-cdr, X is the value the CDR is going to become.

PROFESSOR: The X over here. I'm sorry, that's not true. The X is--set-cdr has two arguments-- The CONS I'm changing and the value I'm changing it to. So you have them backwards, that's all. Are there any other questions? Well, thank you. It's time for lunch.