PROFESSOR: Well, I hope you appreciate that we have inducted you into some real magic, the magic of building languages, really building new languages. What have we looked at? We've looked at an Escher picture language: this language invented by Peter Henderson. We looked at digital logic language. Let's see. We've looked at the query language.

And the thing you should realize is, even though these were toy examples, they really are the kernels of really useful things. So, for instance, the Escher picture language was taken by Henry Wu, who's a student at MIT, and developed into a real language for laying out PC boards based just on extending those structures. And the digital logic language, Jerry mentioned when he showed it to you, was really extended to be used as the basis for a simulator that was used to design a real computer. And the query language, of course, is kind of the germ of prologue.

So we built all of these languages, they're all based on LISP. A lot of people ask what particular problems is LISP good for solving for? The answer is LISP is not good for solving any particular problems. What LISP is good for is constructing within it the right language to solve the problems you want to solve, and that's how you should think about it.

So all of these languages were based on LISP. Now, what's LISP based on? Where's that come from? Well, we looked at that too. We looked at the meta-circular evaluator and said well, LISP is based on LISP. And when we start looking at that, we've got to do some real magic, right? So what does that mean, right? Why operators, and fixed points, and the idea that what this means is that LISP is somehow the fixed-point equation for this funny set of things which are defined in terms of themselves. Now, it's real magic.

Well, today, for a final piece of magic, we're going to make all the magic go away. We already know how to do that. The idea is, we're going to take the register machine architecture and show how to implement LISP on terms of that. And, remember, the idea of the register machine is that there's a fixed and finite part of the machine. There's a finite-state controller, which does some particular thing with a particular amount of hardware. There are particular data paths: the operation the machine does. And then, in order to implement recursion and sustain the illusion of infinity, there's some large amount of memory, which is the stack.

So, if we implement LISP in terms of a register machine, then everything ought to become, at

this point, completely concrete. All the magic should go away. And, by the end of this talk, I want you get the feeling that, as opposed to this very mysterious meta-circular evaluator, that a LISP evaluator really is something that's concrete enough that you can hold in the palm of your hand. You should be able to imagine holding a LISP interpreter there.

All right, how are we going to do this? We already have all the ingredients. See, what you learned last time from Jerry is how to take any particular couple of LISP procedures and hand-translate them into something that runs on a register machine. So, to implement all of LISP on a register machine, all we have to do is take the particular procedures that are the meta-circular evaluator and hand-translate them for a register machine. And that does all of LISP, right? So, in principle, we already know how to do this. And, indeed, it's going to be no different, in kind, from translating, say, recursive factorial or recursive Fibonacci. It's just bigger and there's more of it. So it'd just be more details, but nothing really conceptually new.

All right, also, when we've done that, and the thing is completely explicit, and we see how to implement LISP in terms of the actual sequential register operations, that's going to be our final most explicit model of LISP in this course. And, remember, that's a progression through this course. We started out with substitution, which is sort of like algebra. And then we went to the environment model, which talked about the actual frames and how they got linked together. And then we made that more concrete in the meta-circular evaluator.

There are things the meta-circular evaluator doesn't tell us. You should realize that. For instance, it left unanswered the question of how a procedure, like recursive factorial here , somehow takes space that grows. On the other hand, a procedure which also looks syntactically recursive, called fact-iter, somehow doesn't take space. We justify that it doesn't need to take space by showing the substitution model. But we didn't really say how it happens that the machine manages to do that, that that has to do with the details of how arguments are passed to procedures. And that's the thing we didn't see in the meta-circular evaluator precisely because the way arguments got passed to procedures in this LISP depended on the way arguments got passed to procedures in this LISP. But, now, that's going to become extremely explicit.

OK. Well, before going on to the evaluator, let me just give you a sense of what a whole LISP system looks like so you can see the parts we're going to talk about and the parts we're not going to talk about. Let's see, over here is a happy LISP user, and the LISP user is talking to something called the reader. The reader's job in life is to take characters from the user and

turn them into data structures in something called a list structure memory.

All right, so the reader is going to take symbols, parentheses, and A's and B's, and ones and threes that you type in, and turn these into actual list structure: pairs, and pointers, and things. And so, by the time evaluator is going, there are no characters in the world. And, of course, in more modern list systems, there's sort of a big morass here that might sit between the user and the reader: Windows systems, and top levels, and mice, and all kinds of things. But conceptually, characters are coming in.

All right, the reader transforms these into pointers to stuff in this memory, and that's what the evaluator sees, OK? The evaluator has a bunch of helpers. It has all possible primitive operators you might want. So there's a completely separate box, a floating point unit, or all sorts of things, which do the primitive operators. And, if you want more special primitives, you build more primitive operators, but they're separate from the evaluator. The evaluator finally gets an answer and communicates that to the printer. And now, the printer's job in life is to take this list structure coming from the evaluator, and turn it back into characters, and communicate them to the user through whatever interface there is.

OK. Well, today, what we're going to talk about is this evaluator. The primitive operators have nothing particular to do with LISP, they're however you like to implement primitive operations. The reader and printer are actually complicated, but we're not going to talk about them. They sort of have to do with details of how you might build up list structure from characters. So that is a long story, but we're not going to talk about it. The list structure memory, we'll talk about next time. So, pretty much, except for the details of reading and printing, the only mystery that's going to be left after you see the evaluator is how you build list structure on conventional memories. But we'll worry about that next time too.

OK. Well, let's start talking about the evaluator. The one that we're going to show you, of course, is not, I think, nothing special about it. It's just a particular register machine that runs LISP. And it has seven registers, and here are the seven registers. There's a register, called EXP, and its job is to hold the expression to be evaluated. And by that, I mean it's going to hold a pointer to someplace in list structure memory that holds the expression to be evaluated.

There's a register, called ENV, which holds the environment in which this expression is to be evaluated. And, again, I made a pointer. The environment is some data structure. There's a register, called FUN, which will hold the procedure to be applied when you go to apply a

procedure. A register, called ARGL, which wants the list of evaluated arguments.

What you can start seeing here is the basic structure of the evaluator. Remember how evaluators work. There's a piece that takes expressions and environments, and there's a piece that takes functions, or procedures and arguments. And going back and forth around here is the eval/apply loop. So those are the basic pieces of the eval and apply.

Then there's some other things, there's continue. You just saw before how the continue register is used to implement recursion and stack discipline. There's a register that's going to hold the result of some evaluation. And then, besides that, there's one temporary register, called UNEV, which typically, in the evaluator, is going to be used to hold temporary pieces of the expression you're working on, which you haven't gotten around to evaluate yet, right? So there's my machine: a seven-register machine. And, of course, you might want to make a machine with a lot more registers to get better performance, but this is just a tiny, minimal one.

Well, how about the data paths? This machine has a lot of special operations for LISP. So, here are some typical data paths. A typical one might be, oh, assign to the VAL register the contents of the EXP register. In terms of those diagrams you saw, that's a little button on some arrow.

Here's a more complicated one. It says branch, if the thing in the expression register is a conditional to some label here, called the ev-conditional. And you can imagine this implemented in a lot of different ways. You might imagine this conditional test as a special purpose sub-routine, and conditional might be represented as some data abstraction that you don't care about at this level of detail. So that might be done as a sub-routine. This might be a machine with hardware-types, and conditional might be testing some bits for a particular code. There are all sorts of ways that's beneath the level of abstraction we're looking at.

Another kind of operation, and there are a lot of different operations assigned to EXP, the first clause of what's in EXP. This might be part of processing a conditional. And, again, first clause is some selector whose details we don't care about. And you can, again, imagine that as a sub-routine which'll do some list operations, or you can imagine that as something that's built directly into hardware. The reason I keep saying you can imagine it built directly into hardware is even though there are a lot of operations, there are still a fixed number of them. I forget how many, maybe 150. So, it's plausible to think of building these directly into hardware.

Here's a more complicated one. You can see this has to do with looking up the values of

variables. It says assign to the VAL register the result of looking up the variable value of some particular expression, which, in this case, is supposed to be a variable in some environment. And this'll be some operation that searches through the environment structure, however it is represented, and goes and looks up that variable. And, again, that's below the level of detail that we're thinking about. This has to do with the details of the data structures for representing environments.

But, anyway, there is this fixed and finite number of operations in the register machine. Well, what's its overall structure? Those are some typical operations. Remember what we have to do, we have to take the meta-circular evaluator-- and here's a piece of the meta-circular evaluator. This is the one using abstract syntax that's in the book. It's a little bit different from the one that Jerry shows you. And the main thing to remember about the evaluator is that it's doing some sort of case analysis on the kinds of expressions: so if it's either self-evaluated, or quoted, or whatever else. And then, in the general case where the expression it's looking at is an application, there's some tricky recursions going on.

First of all, eval has to call itself both to evaluate the operator and to evaluate all the operands. So there's this sort of red recursion of values walking down the tree that's really the easy recursion. That's just a val walking down this tree of expressions. Then, in the evaluator, there's a hard recursion. There's the red to green. Eval calls apply. That's the case where evaluating a procedure or argument reduces to applying the procedure to the list of arguments.

And then, apply comes over here. Apply takes a procedure and arguments and, in the general case where there's a compound procedure, apply goes around and green calls red. Apply comes around and calls eval again. Eval's the body of the procedure in the result of extending the environment with the parameters of the procedure by binding the arguments. Except in the primitive case, where it just calls something else primitive-apply, which is not really the business of the evaluator. So this sort of red to green, to red to green, that's the eval/apply loop, and that's the thing that we're going to want to see in the evaluator.

All right. Well, it won't surprise you at all that the two big pieces of this evaluator correspond to eval and apply. There's a piece called eval-dispatch, and a piece called apply-dispatch. And, before we get into the details of the code, the way to understand this is to think, again, in terms of these pieces of the evaluator having contracts with the rest of the world. What do they do from the outside before getting into the grungy details?

Well, the contract for eval-dispatch-- remember, it corresponds to eval. It's got to evaluate an expression in an environment. So, in particular, what this one is going to do, eval-dispatch will assume that, when you call it, that the expression you want to evaluate is in the EXP register. The environment in which you want the evaluation to take place is in the ENV register. And continue tells you the place where the machine should go next when the evaluation is done. Eval-dispatch's contract is that it'll actually perform that evaluation, and, at the end of which, it'll end up at the place specified by continue. The result of the evaluation will be in the VAL register. And it just warns you, it makes no promises about what happens to the registers. All other registers might be destroyed. So, there's one piece, OK?

Together, the pieces, apply-dispatch that corresponds to apply, it's got to apply a procedure to some arguments, so it assumes that this register, ARGL, contains a list of the evaluated arguments. FUN contains the procedure. Those correspond to the arguments to the apply procedure in the meta-circular evaluator.

And apply, in this particular evaluator, we're going to use a discipline which says the place the machine should go to next when apply is done is, at the moment apply-dispatch is called at the top of the stack, that's just discipline for the way this particular machine's organized. And now apply's contract is given all that. It'll perform the application. The result of that application will end up in VAL. The stack will be popped. And, again, the contents of all the other registers may be destroyed, all right? So that's the basic organization of this machine. Let's break for a little bit and see if there are any questions, and then we'll do a real example.

Well, let's take the register machine now, and actually step through, and really, in real detail, so you see completely concrete how some expressions are evaluated, all right? So, let's start with a very simple expression. Let's evaluate the expression 1. And we need an environment, so let's imagine that somewhere there's an environment, we'll call it E,0. And just, since we'll use these later, we obviously don't really need anything to evaluate 1. But, just for reference later, let's assume that E,0 has in it an X that's bound to 3 and a Y that's bound to 4, OK?

And now what we're going to do is we're going to evaluate 1 in this environment, and so the ENV register has a pointer to this environment, E,0, all right? So let's watch that thing go. What I'm going to do is step through the code. And, let's see, I'll be the controller. And now what I need, since this gets rather complicated, is a very little execution unit. So here's the execution unit, OK? OK.

OK. All right, now we're going to start. We're going to start the machine at eval-dispatch, right? That's the beginning of this. Eval-dispatch is going to look at the expression in dispatch, just like eval where we look at the very first thing. We branch on whether or not this expression is self-evaluating. Self-evaluating is some abstraction we put into the machine-- it's going to be true for numbers-- to a place called ev-self-eval, right?

So me, being the controller, looks at ev-self-eval, so we'll go over to there. Ev-self-eval says fine, assign to val whatever is in the expression unit, OK? And I have a bug because what I didn't do when I initialized this machine is also say what's supposed to happen when it's done, so I should have started out the machine with done being in the continue register, OK? So we assign to VAL. And now go to fetch of continue, and [? the value changed. ?] OK.

OK, let's try something harder. Let's reset the machine here, and we'll put in the expression register, X, OK? Start again at eval-dispatch. Check, is it self-evaluating? No. Is it a variable? Yes. We go off to ev-variable. It says assign to VAL, look up the variable value in the expression register, OK? Go to fetch of continue.

PROFESSOR: Done.

PROFESSOR: OK. All right. Well, that's the basic idea. That's a simple operation of the machine. Now, let's actually do something a little bit more interesting. Let's look at the expression the sum of x and y. OK. And now we'll see how you start unrolling these expression trees, OK?

Well, start again at eval-dispatch, all right? Self-evaluating? No. Variable? No. All the other special forms which I didn't write down, like quote, and lambda, and set, and whatever, it's none of those. It turns out to be an application, so we go off to ev-application, OK? Ev-application, remember what it's going to do overall. It is going to evaluate the operator. It's going to evaluate the arguments, and then it's going to go apply them. So, before we start, since we're being very literal, we'd better remember that, somewhere in this environment, it's linked to another environment in which plus is bound to the primitive procedure plus before we get an unknown variable in our machine.

OK, so we're at ev-application. OK, assign to UNEV the operands of what's in the expression register, OK? Those are the operands. UNEV's a temporary register where we're going to save them.

PROFESSOR: I'm assigning.

PROFESSOR: Assign to x the operator. Now, notice we've destroyed that expression in x, but the piece that we need is now in UNEV. OK. Now, we're going to get set up to recursively evaluate the operator. Save the continue register on the stack. Save the environment. Save UNEV. OK, assign to continue a label called eval-args.

Now, what have we done? We've set up for a recursive call. We're about to go to eval-dispatch. We've set up for a recursive call to eval-dispatch. What did we do? We took the things we're going to need later, those operands that were in UNEV; the environment in which we're going to eventually have to, maybe, evaluate those operands; the place we eventually want to go to, which, in this case, was done; we've saved them on the stack. The reason we saved them on the stack is because eval-dispatch makes no promises about what registers it may destroy. So all that stuff is saved on the stack.

Now, we've set up eval-dispatch's contract. There's a new expression, which is the operator plus; a new environment, although, in this case, it's the same one; and a new place to go to when you're done, which is eval-args. So that's set up. Now, we're going to go off to eval-dispatch. Here we are back at eval-dispatch. It's not self-evaluating. Oh, it's a variable, so we'd better go off to ev-variable, right? Ev-variable is assigned to VAL. Look up the variable value of the expression, OK? So VAL is the primitive procedure plus, OK? And go to fetch of continue.

PROFESSOR: Eval-args.

PROFESSOR: Right, which is now eval-args not done. So we come back here at eval-args, and what do we do? We're going to restore the stuff that we saved, so we restore UNEV. And notice, there, it wasn't necessary, although, in general, it would be. It might be some arbitrary evaluation that happened. We restore ENV. OK, we assign to FUN fetch of VAL.

OK, now, we're going to go off and start evaluating some arguments. Well, first thing we'd better do is save FUN because some arbitrary stuff might happen in that evaluation. We initialize the argument list. Assign to argl an empty argument list, and go to eval-arg-loop, OK? At eval-arg-loop, the idea of this is we're going to evaluate the pieces of the expressions that are in UNEV, one by one, and move them from unevaluated in UNEV to evaluated in the arg list, OK? So we save argl. We assign to x the first operand of the stuff in UNEV.

Now, we check and see if that was the last operand. In this case, it is not, all right? So we save the environment. We save UNEV because those are all things we might need later. We're

going to need the environment to do some more evaluations. We're going to need UNEV to look at what the rest of those arguments were. We're going to assign continue a place called accumulate-args, or accumulate-arg.

OK, now, we've set up for another call to eval-dispatch, OK? All right, now, let me short-circuit this so we don't go through the details of eval-dispatch. Eval-dispatch's contract says I'm going to end up, the world will end up, with the value of evaluating this expression in this environment in the VAL register, and I'll end up there. So we short-circuit all of this, and a 3 ends up in VAL. And, when we return from eval-dispatch, we're going to return to accumulate-arg.

PROFESSOR: Accumulate-arg.

PROFESSOR: With 3 in the VAL register, OK? So that short-circuited that evaluation. Now, what do we do? We're going to go back and look at the rest of the arguments, so we restore UNEV. We restore ENV. We restore argl. One thing.

PROFESSOR: Oops! Parity error.

[LAUGHTER]

PROFESSOR: Restore argl.

PROFESSOR: OK. OK, we assign to argl consing on fetch of the value register to what's in argl. OK, we assign to UNEV the rest of the operands in fetch of UNEV, and we go back to eval-arg-loop.

PROFESSOR: Eval-arg-loop.

PROFESSOR: OK. Now, we're about to do the next argument, so the first thing we do is save argl. OK, we assign to x the first operand of fetch of UNEV. OK, we test and see if that's the last operand. In this case, it is, so we're going to go to a special place that says evaluate the last argument because, notice, after evaluating the argument, we don't need the environment any more. That's going to be the difference.

So here, at eval-last-arg, which is assigned to accumulate-last-arg, now, we're set up again for eval-dispatch. We've got a place to go to when we're done. We've got an expression. We've got an environment. OK, so we'll short-circuit the call to eval-dispatch. And what'll happen is there's a y there, it's 4 in that environment, so VAL will end up with 4 in it. And, then, we're

going to end up at accumulate-last-arg, OK? So, at accumulate-last-arg, we restore argl. We assign to argl cons of fetch of the new value onto it, so we cons a 4 onto that. We restore what was saved in the function register. And notice, in this case, it had not been destroyed, but, in general, it will be.

And now, we're ready to go off to apply-dispatch, all right? So we've just gone through the eval. We evaluated the argument, the operator, and the arguments, and now, we're about to apply them. So we come off to apply-dispatch here, OK? We come off to apply-dispatch, and we're going to check whether it's a primitive or a compound procedure.

PROFESSOR: Yes.

PROFESSOR: All right. So, in this case, it's a primitive procedure, and we go off to primitive-apply. So we go off to primitive-apply, and it says assign to VAL the result of applying primitive procedure of the function to the argument list.

PROFESSOR: I don't know how to add. I'm just an execution unit.

PROFESSOR: Well, I don't know how to add either. I'm just the evaluator, so we need a primitive operator. Let's see, so the primitive operator, what's the sum of 3 and 4?

AUDIENCE: 7.

PROFESSOR: OK, 7.

PROFESSOR: Thank you.

PROFESSOR: Now, we restore continue, and we go to fetch of continue.

PROFESSOR: Done.

PROFESSOR: OK. Well, that was in as much detail as you will ever see. We'll never do it in as much detail again. One very important thing to notice is that we just executed a recursive procedure, right? This whole thing, we used a stack and the evaluator was recursive. A lot of people think the reason that you need a stack and recursion in an evaluator is because you might be evaluating recursive procedures like factorial or Fibonacci. It's not true. So you notice we did recursion here, and all we evaluated was plus X, Y, all right? The reason that you need recursion in the evaluator is because the evaluation process, itself, is recursive, all right? It's not because the procedure that you might be evaluating in LISP is a recursive procedure. So

that's an important thing that people get confused about a lot.

The other thing to notice is that, when we're done here, we're really done. Not only are we at done, but there's no accumulated stuff on the stack, right? The machine is back to its initial state, all right? So that's part of what it means to be done. Another way to say that is the evaluation process has reduced the expression, plus X, Y, to the value here, 7. And by reduced, I mean a very particular thing. It means that there's nothing left on the stack. The machine is now in the same state, except there's something in the value register. It's not part of a sub-problem of anything. There's nothing to go back to. OK. Let's break. Question?

AUDIENCE: The question here, in the stack, is because the data may be recursive. You may have embedded expressions, for instance.

PROFESSOR: Yes, because you might have embedded expressions. But, again, don't confuse that with what people sometimes mean by the data may be recursive, which is to say you have these list-structured, recursive data list operations. That has nothing to do with it. It's simply that the expressions contain sub-expressions. Yeah?

AUDIENCE: Why is it that the order of the arguments in the arg list got reversed?

PROFESSOR: Ah! Yes, I should've mentioned that. Here, the reason the order is reversed-- it's a question of what you mean by reversed. I believe it was Newton. In the very early part of optics, people realized that, when you look through the lens of your eye, the image was up-side down. And there was a lot of argument about why that didn't mean you saw things up-side down. So it's sort of the same issue. Reversed from what? So we just need some convention. The reason that they're coming at 4, 3 is because we're taking UNEV and consing the result onto argl. So you have to realize you've made that convention.

The place that you have to realize that-- well, there's actually two places. One is in apply-primitive-operator, which has to realize that the arguments to primitives go in, in the opposite order from the way you're writing them down. And the other one is, we'll see later when you actually go to bind a function's parameters, you should realize the arguments are going to come in from the opposite order of the variables to which you're binding them. So, if you just keep track of that, there's no problem. Also, this is completely arbitrary because, if we'd done, say, an iteration through a vector assigning them, they might come out in the other order, OK? So it's just a convention of the way this particular evaluator works. All right, let's take a break.

We just saw evaluating an expression and, of course, that was very simple one. But, in essence, it would be no different if it was some big nested expression, so there would just be deeper recursion on the stack. But what I want to do now is show you the last piece. I want to walk you around this eval and apply loop, right? That's the thing we haven't seen, really. We haven't seen any compound procedures where applying a procedure reduces to evaluating the body of the procedure, so let's just suppose we had this. Suppose we were looking at the procedure define F of A and B to be the sum of A and B. So, as we typed in that procedure previously, and now we're going to evaluate F of X and Y, again, in this environment, E,0, where X is bound to 3 and Y is bound to 4.

When the defined is executed, remember, there's a lambda here, and lambdas create procedures. And, basically, what will happen is, in E,0, we'll end up with a binding for F, which will say F is a procedure, and its args are A and B, and its body is plus a,b. So that's what the environment would have looked like had we made that definition.

Then, when we go to evaluate F of X and Y, we'll go through exactly the same process that we did before. It's even the same expression. The only difference is that F, instead of having primitive plus in it, will have this thing. And so we'll go through exactly the same process, except this time, when we end up at apply-dispatch, the function register, instead of having primitive plus, will have a thing that will represent it saying procedure, where the args are A and B, and the body is plus A, B. And, again, what I mean, by its ENV, I mean there's a pointer to it, so don't worry that I'm writing a lot of stuff there. There's a pointer to this procedure data structure.

OK, so, we're in exactly the same situation. We get to apply-dispatch, so, here, we come to apply-dispatch. Last time, we branched off to a primitive procedure. Here, it says oh, we now have a compound procedure, so we're going to go off to compound-apply. Now, what's compound-apply? Well, remember what the meta-circular evaluator did? Compound-apply said we're going to evaluate the body of the procedure in some new environment.

Where does that new environment come from? We take the environment that was packaged with the procedure, we bind the parameters of the procedure to the arguments that we're passing in, and use that as a new frame to extend the procedure environment. And that's the environment in which we evaluate the procedure body, right? That's going around the apply/eval loop. That's apply coming back to call eval, all right?

OK. So, now, that's all we have to do in compound-apply. What are we going to do? We're going to manufacture a new environment. And we're going to manufacture a new environment, let's see, that we'll call E,1. E,1 is going to be some environment where the parameters of the procedure, where A is bound to 3 and B is bound to 4, and it's linked to E,0 because that's where f is defined. And, in this environment, we're going to evaluate the body of the procedure.

So let's look at that, all right? All right, here we are at compound-apply, which says assign to the expression register the body of the procedure that's in the function register. So I assign to the expression register the procedure body, OK? That's going to be evaluated in an environment which is formed by making some bindings using information determined by the procedure-- that's what's in FUN-- and the argument list.

And let's not worry about exactly what that does, but you can see the information's there. So make bindings will say oh, the procedure, itself, had an environment attached to it. I didn't write that quite here. I should've said in environment because every procedure gets built with an environment. So, from that environment, it knows what the procedure's definition environment is. It knows what the arguments are. It looks at argl, and then you see a reversal convention here. It just has to know that argl is reversed, and it builds this frame, E,1. All right, so, let's assume that that's what make bindings returns, so it assigns to ENV this thing, E,1.

All right, the next thing it says is restore continue. Remember what continue was here? It got put up in the last segment. Continue got stored. That was the original done, which said what are you going to do after you're done with this particular application? It was one of the very first things that happened when we evaluated the application.

And now, finally, we're going to restore continue. Remember apply-dispatch's contract. It assumes that where it should go to next was on the stack, and there it was on the stack. Continue has done, and now we're going to go back to eval-dispatch. We're set up again. We have an expression, an environment, and a place to go to. We're not going to go through that because it's sort of the same expression.

OK, but the thing, again, to notice is, at this point, we have reduced the original expression, F,X,Y, right? We've reduced evaluating F,X,Y in environment E,0 to evaluate plus A, B in E,1. And notice, nothing's on the stack, right? It's a reduction. At this point, the machine does not contain, as part of its state, the fact that it's in the middle of evaluating some procedure called

f, that's gone, right? There's no accumulated state, OK?

Again, that's a very important idea. That's the meaning of, when we used to write in the substitution model, this expression reduces to that expression. And you don't have to remember anything. And here, you see the meaning of reduction. At this point, there is nothing on the stack. See, that has very important consequences. Let's go back and look at iterative factorial, all right?

Remember, this was some sort of loop and doing iter. And we kept saying that's an iterative procedure, right? And what we wrote, remember, are things like, we said, fact-iter of 5. We wrote things like reduces to iter of 1, and 1, and 5, which reduces to iter of 1, and 2, and 5, and so on, and so on, and so on. And we kept saying well, look, you don't have to build up any storage to do that. And we waved our hands, and said in principle, there's no storage needed. Now, you see no storage needed. Each of these is a real reduction, right?

As you walk through these expressions, what you'll see are these expressions on the stack in some particular environment, and then these expressions in the EXP register in some particular environment. And, at each point, there'll be no accumulated stuff on the stack because each one's a real reduction, OK?

All right, so, for example, just to go through it in a little bit more care, if I start out with an expression that says something like, oh, say, fact-iter of 5 in some environment that will, at some point, create an environment in which n is down to 5. Let's call that-- And, at some point, the machine will reduce this whole thing to a thing that says that's really iter of 1, and 1, and n, evaluated in this environment, E,1 with nothing on the stack. See, at this moment, the machine is not remembering that evaluating this expression, iter-- which is the loop-- is part of this thing called iterative factorial. It's not remembering that. It's just reducing the expression to that, right? If we look again at the body of iterative factorial, this expression has reduced to that expression. Oh, I shouldn't have the n there. It's a slightly different convention from the slide to the program, OK?

And, then, what's the body of iter? Well, iter's going to be an it, and I won't go through the details of if. It'll evaluate the predicate. In this case, it'll be false. And this iter will now reduce to the expression iter of whatever it says, star, counter product, and-- what does it say-- plus counter 1 in some other environment, by this time, E,2, where E,2 will be set up having bindings for product and counter, right? And it'll reduce to that, right? It won't be remembering

that it's part of something that it has to return to. And when iter calls iter again, it'll reduce to another thing that looks like this in some environment, E,3, which has new bindings for product and counter. So, if you're wondering, see, if you've always been queasy about how it is we've been saying those procedures, that look syntactically recursive, are, in fact, iterative, run in constant space, well, I don't know if this makes you less queasy, but at least it shows you what's happening. There really isn't any buildup there.

Now, you might ask well, is there buildup in principle in these environment frames? And the answer is yeah, you have to make these new environment frames, but you don't have to hang onto them when you're done. They can be garbage collected, or the space can be reused automatically. But you see the control structure of the evaluator is really using this idea that you actually have a reduction, so these procedures really are iterative procedures. All right, let's stop for questions. All right, let's break.

Let me contrast the iterative procedure just so you'll see where space does build up with a recursive procedure, so you can see the difference. Let's look at the evaluation of recursive factorial, all right? So, here's fact-recursive, or standard factorial definition. We said this one is still a recursive procedure, but this is actually a recursive process.

And then, just to link it back to the way we started, we said oh, you can see that it's going to be recursive process by the substitution model because, if I say recursive factorial of 5, that turns into 5 times-- what is it, fact-rec, or record fact-- 5 times recursive factorial of 4, which turns into 5 times 4 times fact-rec of 3, which returns into 5 times 4 times 3 times, and so on, right? The idea is there was this chain of stuff building up, which justified, in the substitution model, the fact that it's recursive. And now, let's actually see that chain of stuff build up and where it is in the machine, OK?

All right, well, let's imagine we're going to start out again. We'll tell it to evaluate recursive factorial of 5 in some environment, again, E,0 where recursive factorial is defined, OK? Well, now we know what's eventually going to happen. This is going to come along, it'll evaluate those things, figure out it's a procedure, build somewhere over here an environment, E,1, which has n bound to 5, which hangs off of E,0, which would be, presumably, the definition environment of recursive factorial, OK?

And, in this environment, it's going to go off and evaluate the body. So, again, the evaluation here will reduce to evaluating the body in E,1. That's going to look at an if, and I won't go

through the details of if. It'll look at the predicate. It'll decide it eventually has to evaluate the alternative. So this whole thing, again, will reduce to the alternative of recursive factorial, the alternative clause, which says that this whole thing reduces to times n of recursive factorial of n minus 1 in the environment E,1, OK? So the original expression, now, is going to reduce to evaluating that expression, all right?

Now we have an application. We did an application before. Remember what happens in an application? The first thing you do is you go off and you save the value of the continue register on the stack. So the stack here is going to have done in it. And then you're going to set up to evaluate the sub-parts, OK? So here we go off to evaluate the sub-parts.

First thing we're going to do is evaluate the operator. What happens when we evaluate an operator? Well, we arrange things so that the operator ends up in the expression register. The environments in the ENV register continue someplace where we're going to go evaluate the arguments. And, on the stack, we've saved the original continue, which is where we wanted to be when we're all done. And then the things we needed when we're going to get done evaluating the operator, the things we'll need to evaluate the arguments, namely, the environment and those arguments, those unevaluated arguments, so there they are sitting on the stack. And we're about to go off to evaluate the operator.

Well, when we return from this particular call-- so we're about to call eval-dispatch here-- when we return from this call, the value of that operator, which, in this case, is going to be the primitive multiplier procedure, will end up in the FUN register, all right? We're going to evaluate some arguments. They will evaluate in here. That'll give us 5, in this case. We're going to put that in the argl register, and then we'll go off to evaluate the second operand.

So, at the point where we go off to evaluate the second operand-- and I'll skip details like computing, and minus 1, and all of that-- but, when we go off to evaluate the second operand, that will eventually reduce to another call to fact-recursive. And, what we've got on the stack here is the operator from that combination that we're going to use it in and the other argument, OK? So, now, we're set up for another call to recursive factorial. And, when we're done with this one, we're going to go to accumulate the last arg. And remember what that'll do? That'll say oh, whatever the result of this has to get combined with that, and we're going to multiply them.

But, notice now, we're at another recursive factorial. We're about to call eval-dispatch again,

except we haven't really reduced it because there's stuff on the stack now. The stuff on the stack says oh, when you get back, you'd better multiply it by the 5 you had hanging there. So, when we go off to make another call, we evaluate the n minus 1. That gives us another environment in which the new n's going to be down to 4. And we're about to call eval-dispatch again, right?

We get another call. That 4 is going to end up in the same situation. We'll end up with another call to fact-recursive n. And sitting on the stack will be the stuff from the original one and, now, the subsidiary one we're doing. And both of them are waiting for the same thing. They're going to go to accumulate a last argument. And then, of course, when we go to the fourth call, the same thing happens, right? And this goes on, and on, and on.

And what you see here on the stack, exactly what's sitting here on the stack, the thing that says times and 5. And what you're going to do with that is accumulate that into a last argument. That's exactly this, right? This is exactly where that stuff is hanging. Effectively, the operator you're going to apply, the other argument that it's got to be multiplied by when you get back and the parentheses, which says yeah, what you wanted to do was accumulate them. So, you see, the substitution model is not such a lie. That really is, in some sense, what's sitting right on the stack. OK.

All right, so that, in some sense, should explain for you, or at least convince you, that, somehow, this evaluator is managing to take these procedures and execute some of them iteratively and some of them recursively, even though, as syntactically, they look like recursive procedures. How's it managing to do that? Well, the basic reason it's managing to do that is the evaluator is set up to save only what it needs later.

So, for example, at the point where you've reduced evaluating an expression and an environment to applying a procedure to some arguments, it doesn't need that original environment anymore because any environment stuff will be packaged inside the procedures where the application's going to happen. All right, similarly, when you're going along evaluating an argument list, when you've finished evaluating the list, when you're finished evaluating the last argument, you don't need that argument list any more, right? And you don't need the environment where those arguments would be evaluated, OK? So the basic reason that this interpreter is being so smart is that it's not being smart at all, it's being stupid. It's just saying I'm only going to save what I really need.

Well, let me show you here. Here's the actual thing that's making a tail recursive. Remember, it's the restore of continue. It's saying when I go off to evaluate the procedure body, I should tell eval to come back to the place where that original evaluation was supposed to come back to. So, in some sense, you want to say what's the actual line that makes a tail recursive? It's that one.

If I wanted to build a non-tail recursive evaluator, for some strange reason, all I would need to do is, instead of restoring continue at this point, I'd set up a label down here called, "Where to come back after you've finished applying the procedure." Instead, I'd set continue to that. I'd go to eval-dispatch, and then eval-dispatch would come back here. At that point, I would restore continue and go to the original one. So here, the only consequence of that would be to make it non-tail recursive. It would give you exactly the same answers, except, if you did that iterative factorial and all those iterative procedures, it would execute recursively.

Well, I lied to you a little bit, but just a little bit, because I showed you a slightly over-simplified evaluator where it assumes that each procedure body has only one expression. Remember, in general, a procedure has a sequence of expressions in it. So there's nothing really conceptually new. Let me just show you the actual evaluator that handles sequences of expressions.

This is compound-apply now, and the only difference from the old one is that, instead of going off to eval directly, it takes the whole body of the procedure, which, in this case, is a sequence of expressions, and goes off to eval-sequence. And eval-sequence is a little loop that, basically, does these evaluations one at a time. So it does an evaluation. Says oh, when I come back, I'd better come back here to do the next one. And, when I'm all done, when I want to get the last expression, I just restore my continue and go off to eval-dispatch.

And, again, if you wanted for some reason to break tail recursion in this evaluator, all you need to do is not handle the last expression, especially. Just say, after you've done the last expression, come back to some other place after which you restore continue. And, for some reason, a lot of LISP evaluators tended to work that way. And the only consequence of that is that iterative procedures built up stack. And it's not clear why that happened.

All right. Well, let me just sort of summarize, since this is a lot of details in a big program. But the main point is that it's no different, conceptually, from translating any other program. And the main idea is that we have this universal evaluator program, the meta-circular evaluator. If

we translate that into LISP, then we have all of LISP. And that's all we did, OK?

The second point is that the magic's gone away. There should be no more magic in this whole system, right? In principle, it should all be very clear except, maybe, for how list structured memory works, and we'll see that later. But that's not very hard.

The third point is that all this tail recursion came from the discipline of eval being very careful to save only what it needs next time. It's not some arbitrary thing where we're saying well, whenever we call a sub-routine, we'll save all the registers in the world and come back, right? See, sometimes it pays to really worry about efficiency. And, when you're down in the guts of your evaluator machine, it really pays to think about things like that because it makes big consequences.

Well, I hope what this has done is really made the evaluator seem concrete, right? I hope you really believe that somebody could hold a LISP evaluator in the palm of their hand. Maybe to help you believe that, here's a LISP evaluator that I'm holding the palm of my hand, right? And this is a chip which is actually quite a bit more complicated than the evaluator I showed you. Maybe, here's a better picture of it. What there is, is you can see the same overall structure. This is a register array. These are the data paths. Here's a finite state controller. And again, finite state, that's all there is. And somewhere there's external memory that'll worry about things.

And this particular one is very complicated because it's trying to run LISP fast. And it has some very, very fast parallel operations in there like, if you want to index into an array, simultaneously check that the index is an integer, check that it doesn't exceed the array bands, and go off and do the memory access, and do all those things simultaneously. And then, later, if they're all OK, actually get the value there. So there are a lot of complicated operations in these data paths for making LISP run in parallel. It's a completely non-risk philosophy of evaluating LISP.

And then, this microcode is pretty complicated. Let's see, there's what? There's about 389 instructions of 220-bit microcode sitting here because these are very complicated data paths. And the whole thing has about 89,000 transistors, OK? OK. Well, I hope that that takes away a lot of the mystery. Maybe somebody wants to look at this. Yeah. OK. Let's stop. Questions?

AUDIENCE: OK, now, it sounds like what you're saying is that, with the restore continue put in the proper place, that procedures that would invoke a recursive process now invoke an integer process

just by the way that the eval signature is?

PROFESSOR: I think the way I'd prefer to put it is that, with restore continue put in the wrong place, you can cause any syntactically-looking recursive procedure, in fact, to build up stack as it runs. But there's no reason for that, so you might want to play around with it. You can just switch around two or three instructions in the way compound-apply comes back, and you'll get something which isn't tail recursive.

But the thing I wanted to emphasize is there's no magic. It's not as if there's some very clever pre-processing program that's looking at this procedure, factorial iter, and say oh, gee, I really notice that I don't have to push stack in order to do this. Some people think that that's what's going on. It's something much, much more dumb than that, it's this one place you're putting the restore instruction. It's just automatic.

AUDIENCE: OK.

AUDIENCE: But that's not affecting the time complexity is it?

PROFESSOR: No.

AUDIENCE: It's just that it's handling it recursively instead of iteratively. But, in terms of the order of time it takes to finish the operation, it's the same one way or the other, right?

PROFESSOR: Yes. Tail recursion is not going to change the time complexity of anything because, in some sense, it's the same algorithm that's going on. What it's doing is really making this thing run as an iteration, right? Not going to run out of memory counting up to a giant number simply because the stack would get pushed.

See, the thing you really have to believe is that, when we write-- see, we've been writing all these things called iterations, infinite loops, define loop to be called loop. That's is as much an iteration as if we wrote do forever loop, right? It's just syntactic sugar as the difference. These things are real, honest to god, iterations, right? They don't change the time complexity, but they turn them into real iterations. All right, thank you.