[MUSIC PLAYING]

PROFESSOR: Last time, we took a look at an explicit control evaluator for Lisp, and that bridged the gap between all these high-level languages like Lisp and the query language and all of that stuff, bridged the gap between that and a conventional register machine. And in fact, you can think of the explicit control evaluator either as, say, the code for a Lisp interpreter if you wanted to implement it in the assembly language of some conventional register transfer machine, or, if you like, you can think of it as the microcode of some machine that's going to be specially designed to run Lisp.

In either case, what we're doing is we're taking a machine that speaks some low-level language, and we're raising the machine to a high-level language like Lisp by writing an interpreter. So for instance, here, conceptually, is a special purpose machine for computing factorials. It takes in five and puts out 120. And what this special purpose machine is is actually a Lisp interpreter that's configured itself to run factorials, because you fit into it a description of the factorial machine.

So that's what an interpreter is. It configures itself to emulate a machine whose description you read in. Now, inside the Lisp interpreter, what's that? Well, that might be your general register language interpreter that configures itself to behave like a Lisp interpreter, because you put in a whole bunch of instructions in register language. This is the explicit control evaluator. And then it also has some sort of library, a library of primitive operators and Lisp operations and all sorts of things like that. That's the general strategy of interpretation.

And the point is, what we're doing is we're writing an interpreter to raise the machine to the level of the programs that we want to write. Well, there's another strategy, a different one, which is compilation. Compilation's a little bit different. Here--here we might have produced a special purpose machine for, for computing factorials, starting with some sort of machine that speaks register language, except we're going to do a different strategy.

We take our factorial program. We use that as the source code into a compiler. What the compiler will do is translate that factorial program into some register machine language. And this will now be not the explicit control evaluator for Lisp, this will be some register language for computing factorials. So this is the translation of that.

That will go into some sort of loader which will combine this code with code selected from the library to do things like primitive multiplication. And then we'll produce a load module which configures the register language machine to be a special purpose factorial machine. So that's a, that's a different strategy. In interpretation, we're raising the machine to the level of our language, like Lisp. In compilation, we're taking our program and lowering it to the language that's spoken by the machine.

Well, how do these two strategies compare? The compiler can produce code that will execute more efficiently. The essential reason for that is that if you think about the register operations that are running, the interpreter has to produce register operations which, in principle, are going to be general enough to execute any Lisp procedure. Whereas the compiler only has to worry about producing a special bunch of register operations for, for doing the particular Lisp procedure that you've compiled.

Or another way to say that is that the interpreter is a general purpose simulator, that when you read in a Lisp procedure, then those can simulate the program described by that, by that procedure. So the interpreter is worrying about making a general purpose simulator, whereas the compiler, in effect, is configuring the thing to be the machine that the interpreter would have been simulating. So the compiler can be faster.

On the other hand, the interpreter is a nicer environment for debugging. And the reason for that is that we've got the source code actually there. We're interpreting it. That's what we're working with. And we also have the library around. See, the interpreter--the library sitting there is part of the interpreter. The compiler only pulls out from the library what it needs to run the program.

So if you're in the middle of debugging, and you might like to write a little extra program to examine some run time data structure or to produce some computation that you didn't think of when you wrote the program, the interpreter can do that perfectly well, whereas the compiler can't. So there are sort of dual, dual advantages. The compiler will produce code that executes faster. The interpreter is a better environment for debugging.

And most Lisp systems end up having both, end up being configured so you have an interpreter that you use when you're developing your code. Then you can speed it up by compiling. And very often, you can arrange that compiled code and interpreted code can call each other. We'll see how to do that. That's not hard.

In fact, the way we'll-- in the compiler we're going to make, the way we'll arrange for compiled coding and interpreted code to call, to call each other, is that we'll have the compiler use exactly the same register conventions as the interpreter. Well, the idea of a compiler is very much like the idea of an interpreter or evaluator. It's the same thing. See, the evaluator walks over the code and performs some register operations. That's what we did yesterday.

Well, the compiler essentially would like to walk over the code and produce the register operations that the evaluator would have done were it evaluating the thing. And that gives us a model for how to implement a zeroth-order compiler, a very bad compiler but essentially a compiler. A model for doing that is you just take the evaluator, you run it over the code, but instead of executing the actual operations, you just save them away. And that's your compiled code.

So let me give you an example of that. Suppose we're going to compile--suppose we want to compile the expression f of x. So let's assume that we've got f of x in the x register and something in the environment register. And now imagine starting up the evaluator. Well, it looks at the expression and it sees that it's an application. And it branches to a place in the evaluator code we saw called ev-application. And then it begins. It stores away the operands and unev, and then it's going to put the operator in exp, and it's going to go recursively evaluate it.

That's the process that we walk through. And if you start looking at the code, you start seeing some register operations. You see assign to unev the operands, assign to exp the operator, save the environment, generate that, and so on. Well, if we look on the overhead here, we can see, we can see those operations starting to be produced. Here's sort of the first real operation that the evaluator would have done. It pulls the operands out of the exp register and assigns it to unev. And then it assigns something to the expression register, and it saves continue, and it saves env.

And all I'm doing here is writing down the register assignments that the evaluator would have done in executing that code. And can zoom out a little bit. Altogether, there are about 19 operations there. And this is the--this will be the piece of code up until the point where the evaluator branches off to apply-dispatch. And in fact, in this compiler, we're not going to worry about apply-dispatch at all. We're going to have everything--we're going to have both interpreted code and compiled code. Always evaluate procedures, always apply procedures by going to apply-dispatch. That will easily allow interpreted code and compiled code to call each other.

Well, in principle, that's all we need to do. You just run the evaluator. So the compiler's a lot like the evaluator. You run it, except it stashes away these operations instead of actually executing them. Well, that's not, that's not quite true. There's only one little lie in that.

What you have to worry about is if you have a, a predicate. If you have some kind of test you want to do, obviously, at the point when you're compiling it, you don't know which branch of these--of a conditional like this you're going to do. So you can't say which one the evaluator would have done. So all you do there is very simple. You compile both branches.

So you compile a structure that looks like this. That'll compile into something that says, the code, the code for P. And it puts its results in, say, the val register. So you walk the interpreter over the predicate and make sure that the result would go into the val register. And then you compile an instruction that says, branch if, if val is true, to a place we'll call label one.

Then we, we will put the code for B to walk the interpreter--walk the interpreter over B. And then go to put in an instruction that says, go to the next thing, whatever, whatever was supposed to happen after this thing was done.

You put in that instruction. And here you put label one. And here you put the code for A. And you put go to next thing.

So that's how you treat a conditional. You generate a little block like that. And other than that, this zeroth-order compiler is the same as the evaluator. It's just stashing away the instructions instead of executing them. That seems pretty simple, but we've gained something by that. See, already that's going to be more efficient than the evaluator. Because, if you watch the evaluator run, it's not only generating the register operations we wrote down, it's also doing things to decide which ones to generate.

So the very first thing it does, say, here for instance, is go do some tests and decide that this is an application, and then branch off to the place that, that handles applications. In other words, what the evaluator's doing is simultaneously analyzing the code to see what to do, and running these operations. And when you-- if you run the evaluator a million times, that analysis phase happens a million times, whereas in the compiler, it's happened once, and then you just have the register operations themselves.

Ok, that's a, a zeroth-order compiler, but it is a wretched, wretched compiler. It's really dumb. Let's--let's go back and, and look at this overhead. So look at look at some of the operations this thing is doing. We're supposedly looking at the operations and interpreting f of x. Now, look here what it's doing. For example, here it assigns to exp the operator in fetch of exp. But see, there's no reason to do that, because this is-- the compiler knows that the operator, fetch of exp, is f right here.

So there's no reason why this instruction should say that. It should say, we'll assign to exp, f. Or in fact, you don't need exp at all. There's no reason it should have exp at all. What, what did exp get used for? Well, if we come down here, we're going to assign to val, look up the stuff in exp in the environment. So what we really should do is get rid of the exp register altogether, and just change this instruction to say, assign to val, look up the variable value of the symbol f in the environment.

Similarly, back up here, we don't need unev at all, because we know what the operands of fetch of exp are for this piece of code. It's the, it's the list x. So in some sense, you don't want unev and exp at all. See, what they really are in some sense, those aren't registers of the actual machine that's supposed to run. Those are registers that have to do with arranging the thing that can simulate that machine.

So they're always going to hold expressions which, from the compiler's point of view, are just constants, so can be put right into the code. So you can forget about all the operations worrying about exp and unev and just use those constants. Similarly, again, if we go, go back and look here, there are things like assign to continue eval-args. Now, that has nothing to do with anything. That was just the evaluator keeping track of where it should go next, to evaluate the arguments in some, in some application.

But of course, that's irrelevant to the compiler, because you-- the analysis phase will have already done that. So this is completely irrelevant. So a lot of these, these assignments to continue have not to do where the running machine is supposed to continue in keeping track of its state. It has to, to do with where the evaluator analysis should continue, and those are completely irrelevant. So we can get rid of them.

Ok, well, if we, if we simply do that, make those kinds of optimizations, get rid, get rid of worrying about exp and unev, and get rid of these irrelevant register assignments to continue, then we can take this literal code, these sort of 19 instructions that the, that the evaluator would have done, and then replace them. Let's look at the, at the slide. Replace them by--we get rid of about half of them. And again, this is just sort of filtering what the evaluator would have done by getting rid of the irrelevant stuff.

And you see, for instance, here the--where the evaluator said, assign val, look up variable value, fetch of exp, here we have put in the constant f. Here we've put in the constant x. So there's a, there's a little better compiler. It's still pretty dumb. It's still doing a lot of dumb things. Again, if we go look at the slide again, look at the very beginning here, we see a save the environment, assign something to the val register, and restore the environment.

Where'd that come from? That came from the evaluator back here saying, oh, I'm in the middle of evaluating an application. So I'm going to recursively call eval dispatch. So I'd better save the thing I'm going to need later, which is the environment. This was the result of recursively calling eval dispatch. It was evaluating the symbol f in that case.

Then it came back from eval dispatch, restored the environment. But in fact, the actual thing it ended up doing in the evaluation is not going to hurt the environment at all. So there's no reason to be saving the environment and restoring the environment here. Similarly, here I'm saving the argument list. That's a piece of the argument evaluation loop, saving the argument list, and here you restore it.

But the actual thing that you ended up doing didn't trash the argument list. So there was no reason to save it. So another way to say, another way to say that is that the, the evaluator has to be maximally pessimistic, because as far from its point of view it's just going off to evaluate something. So it better save what it's going to need later.

But once you've done the analysis, the compiler is in a position to say, well, what actually did I need to save? And doesn't need to do any-- it doesn't need to be as careful as the evaluator, because it knows what it actually needs. Well, in any case, if we do that and eliminate all those redundant saves and restores, then we can get it down to this. And you see there are actually only three instructions that we actually need, down from the initial 11 or so, or the initial 20 or so in the original one.

And that's just saying, of those register operations, which ones did we actually need? Let me just sort of summarize that in another way, just to show you in a little better picture. Here's a picture of starting-- This is looking at all the saves and restores. So here's the expression, f of x, and then this traces through, on the bottom here, the various places in the evaluator that were passed when the evaluation happened.

And then here, here you see arrows. Arrow down means register saved. So the first thing that happened is the environment got saved. And over here, the environment got restored. And these-- so there are all the pairs of stack operations. Now, if you go ahead and say, well, let's remember that we don't--that unev, for instance, is a completely useless register. And if we use the constant structure of the code, well, we don't need, we don't need to save unev. We don't need unev at all.

And then, depending on how we set up the discipline of the--of calling other things that apply, we may or may not need to save continue. That's the first step I did. And then we can look and see what's actually, what's actually needed. See, we don't-- didn't really need to save env or cross-evaluating f, because it wouldn't, it wouldn't trash it. So if we take advantage of that, and see the evaluation of f here, doesn't really need to worry about, about hurting env. And similarly, the evaluation of x here, when the evaluator did that it said, oh, I'd better preserve the function register around that, because I might need it later. And I better preserve the argument list.

Whereas the compiler is now in a position to know, well, we didn't really need to save-- to do those saves and restores. So in fact, all of the stack operations done by the evaluator turned out to be unnecessary or overly pessimistic. And the compiler is in a position to know that. Well that's the basic idea. We take the evaluator, we eliminate the things that you don't need, that in some sense have nothing to do with the compiler at all, just the evaluator, and then you see which stack operations are unnecessary.

That's the basic structure of the compiler that's described in the book. Let me just show you how that examples a little bit too simple. To see how you, how you actually save a lot, let's look at a little bit more complicated expression. F of G of X and 1. And I'm not going to go through all the code. There's a, there's a fair pile of it. I think there are, there are something like 16 pairs of register saves and restores as the evaluator walks through that. Here's a diagram of them.

Let's see. You see what's going on. You start out by--the evaluator says, oh, I'm about to do an application. I'll preserve the environment. I'll restore it here. Then I'm about to do the first operand. Here it recursively goes to the evaluator. The evaluator says, oh, this is an application, I'll save the environment, do the operator of that combination, restore it here.

This save--this restore matches that save. And so on. There's unev here, which turns out to be completely unnecessary, continues getting bumped around here. The function register is getting, getting saved across the first operands, across the operands. All sorts of things are going on. But if you say, well, what of those really were the business of the compiler as opposed to the evaluator, you get rid of a whole bunch.

And then on top of that, if you say things like, the evaluation of F doesn't hurt the environment register, or simply looking up the symbol X, you don't have to protect the function register against that. So you come down to just a couple of, a couple of pairs here. And still, you can do a little better.

Look what's going on here with the environment register. The environment register comes along and says, oh, here's a combination. This evaluator, by the way, doesn't know anything about G. So here it says, so it says, I'd better save the environment register, because evaluating G might be some arbitrary piece of code that would trash it, and I'm going to need it later, after this argument, for doing the second argument. So that's why this one didn't go away, because the compiler made no assumptions about what G would do.

On the other hand, if you look at what the second argument is, that's just looking up one. That doesn't need this environment register. So there's no reason to save it. So in fact, you can get rid of that one, too. And from this whole pile of, of register operations, if you simply do a little bit of reasoning like that, you get down to, I think, just two pairs of saves and restores. And those, in fact, could go away further if you, if you knew something about G.

So again, the general idea is that the reason the compiler can be better is that the interpreter doesn't know what it's about to encounter. It has to be maximally pessimistic in saving things to protect itself. The compiler only has to deal with what actually had to be saved. And there are two reasons that something might not have to be saved. One is that what you're protecting it against, in fact, didn't trash the register, like it was just a variable look-up. And the other one is, that the thing that you were saving it for might turn out not to actually need it.

So those are the two basic pieces of knowledge that the compiler can take advantage of in making the code more efficient. Let's break for questions.

AUDIENCE: You kept saying that the uneval register, unev register didn't need to be used at all. Does that mean that you could just map a six-register machine? Or is that, in this particular example, it didn't need to be used?

PROFESSOR: For the compiler, you could generate code for the six-register, five, right? Because that exp goes away also. Assuming--yeah, you can get rid of both exp and unev, because, see, those are data structures of the evaluator. Those are all things that would be constants from the point of view of the compiler. The only thing is this particular compiler is set up so that interpreted code and compiled code can coexist.

So the way to think about it is, is maybe you build a chip which is the evaluator, and what the compiler might do is generate code for that chip. It just wouldn't use two of the registers. All right, let's take a break.

[MUSIC PLAYING]

We just looked at what the compiler is supposed to do. Now let's very briefly look at how, how this gets accomplished. And I'm going to give no details. There's, there's a giant pile of code in the book that gives all the details. But what I want to do is just show you the, the essential idea here. Worry about the details some other time.

Let's imagine that we're compiling an expression that looks like there's some operator, and there are two arguments. Now, the-- what's the code that the compiler should generate? Well, first of all, it should recursively go off and compile the operator. So it says, I'll compile the operator. And where I'm going to need that is to be in the function register, eventually. So I'll compile some instructions that will compile the operator and end up with the result in the function register.

The next thing it's going to do, another piece is to say, well, I have to compile the first argument. So it calls itself recursively. And let's say the result will go into val. And then what it's going to need to do is start setting up the argument list. So it'll say, assign to argl cons of fetch-- so it generates this literal instruction-- fetch of val onto empty list.

However, it might have to work-- when it gets here, it's going to need the environment. It's going to need whatever environment was here in order to do this evaluation of the first argument. So it has to ensure that the compilation of this operand, or it has to protect the function register against whatever might happen in the compilation of this operand.

So it puts a note here and says, oh, this piece should be done preserving the environment register. Similarly, here, after it gets done compiling the first operand, it's going to say, I better compile-- I'm going to need to know the environment for the second operand. So it puts a little note here, saying, yeah, this is also done preserving env. Now it goes on and says, well, the next chunk of code is the one that's going to compile the second argument.

And let's say it'll compile it with a targeted to val, as they say. And then it'll generate the literal instruction, building up the argument list. So it'll say, assign to argl cons of the new value it just got onto the old argument list. However, in order to have the old argument list, it better have arranged that the argument list didn't get trashed by whatever happened in here.

So it puts a little note here and says, oh, this has to be done preserving argl. Now it's got the argument list set up. And it's all ready to go to apply dispatch. It generates this literal instruction. Because now it's got the arguments in argl and the operator in fun, but wait, it's only got the operator in fun if it had ensured that this block of code didn't trash what was in the function register.

So it puts a little note here and says, oh, yes, all this stuff here had better be done preserving the function register. So that's the little--so when it starts ticking--so basically, what the compiler does is append a whole bunch of code sequences. See, what it's got in it is little primitive pieces of things, like how to look up a symbol, how to do a conditional. Those are all little pieces of things.

And then it appends them together in this sort of discipline. So the basic means of combining things is to append two code sequences. That's what's going on here. And it's a little bit tricky. The idea is that it appends two code sequences, taking care to preserve a register. So the actual append operation looks like this. What it wants to do is say, if-- here's what it means to append two code sequences. So if sequence one needs register-- I should change this. Append sequence one to sequence two, preserving some register. Let me say, and. So it's clear that sequence one comes first.

So if sequence two needs the register and sequence one modifies the register, then the instructions that the compiler spits out are, save the register. Here's the code. You generate this code. Save the register, and then you put out the recursively compiled stuff for sequence one. And then you restore the register.

And then you put out the recursively compiled stuff for sequence two. That's in the case where you need to do it. Sequence two actually needs the register, and sequence one actually clobbers it. So that's sort of if. Otherwise, all you spit out is sequence one followed by sequence two. So that's the basic operation for sticking together these bits of code fragments, these bits of instructions into a sequence.

And you see, from this point of view, the difference between the interpreter and the compiler, in some sense, is that where the compiler has these preserving notes, and says, maybe I'll actually generate the saves and restores and maybe I won't, the interpreter being maximally pessimistic always has a save and restore here. That's the essential difference.

Well, in order to do this, of course, the compiler needs some theory of what code sequences need and modifier registers. So the tiny little fragments that you put in, like the basic primitive code fragments, say, what are the operations that you do when you look up a variable? What are the sequence of things that you do when you compile a constant or apply a function? Those have little notations in there about what they need and what they modify.

So the bottom-level data structures-- Well, I'll say this. A code sequence to the compiler looks like this. It has the actual sequence of instructions. And then, along with it, there's the set of registers modified. And then there's the set of registers needed. So that's the information the compiler has that it draws on in order to be able to do this operation.

And where do those come from? Well, those come from, you might expect, for the very primitive ones, we're going to put them in by hand. And then, when we combine two sequences, we'll figure out what these things should be. So for example, a very primitive one, let's see. How about doing a register assignment.

So a primitive sequence might say, oh, it's code fragment. Its code instruction is assigned to R1, fetch of R2. So this is an example. That might be an example of a sequence of instructions. And along with that, it'll say, oh, what I need to remember is that that modifies R1, and then it needs R2. So when you're first building this compiler, you put in little fragments of stuff like that.

And now, when it combines two sequences, if I'm going to combine, let's say, sequence one, that modifies a bunch of registers M1, and needs a bunch of registers N1. And I'm going to combine that with sequence two. That modifies a bunch of registers M2, and needs a bunch of registers N2.

Then, well, we can reason it out. The new code fragment, sequence one, and-- followed by sequence two, well, what's it going to modify? The things that it will modify are the things that are modified either by sequence one or sequence two. So the union of these two sets are what the new thing modifies. And then you say, well, what is this--what registers is it going to need?

It's going to need the things that are, first of all, needed by sequence one. So what it needs is sequence one. And then, well, not quite all of the ones that are needed by sequence one. What it needs are the ones that are needed by sequence two that have not been set up by sequence one. So it's sort of the union of the things that sequence two needs minus the ones that sequence one modifies. Because it worries about setting them up.

So there's the basic structure of the compiler. The way you do register optimizations is you have some strategies for what needs to be preserved. That depends on a data structure. Well, it depends on the operation of what it means to put things together. Preserving something, that depends on knowing what registers are needed and modified by these code fragments.

That depends on having little data structures, which say, a code sequence is the actual instructions, what they modify and what they need. That comes from, at the primitive level, building it in. At the primitive level, it's going to be completely obvious what something needs and modifies. Plus, this particular way that says, when I build up bigger ones, here's how I generate the new set of registers modified and the new set of registers needed.

And that's the whole-- well, I shouldn't say that's the whole thing. That's the whole thing except for about 30 pages of details in the book. But it is a perfectly usable rudimentary compiler. Let me kind of show you what it does.

Suppose we start out with recursive factorial. And these slides are going to be much too small to read. I just want to flash through the code and show you about how much it is. That starts out with--here's a first block of it, where it compiles a procedure entry and does a bunch of assignments. And this thing is basically up through the part where it sets up to do the predicate and test whether the predicate's true.

The second part is what results from-- in the recursive call to fact of n minus one. And this last part is coming back from that and then taking care of the constant case. So that's about how much code it would produce for factorial.

We could make this compiler much, much better, of course. The main way we could make it better is to allow the compiler to make any assumptions at all about what happens when you call a procedure. So this compiler, for instance, doesn't even know, say, that multiplication is something that could be coded in line. Instead, it sets up this whole mechanism. It goes to apply-dispatch.

That's a tremendous waste, because what you do every time you go to apply-dispatch is you have to concept this argument list, because it's a very general thing you're going to. In any real compiler, of course, you're going to have registers for holding arguments. And you're going to start preserving and saving the way you use those registers similar to the same strategy here.

So that's probably the very main way that this particular compiler in the book could be fixed. There are other things like looking up variable values and making more efficient primitive operations and all sorts of things. Essentially, a good Lisp compiler can absorb an arbitrary amount of effort. And probably one of the reasons that Lisp is slow with compared to languages like FORTRAN is that, if you look over history at the amount of effort that's gone into building Lisp compilers, it's nowhere near the amount of effort that's gone into FORTRAN compilers. And maybe that's something that will change over the next couple of years.

OK, let's break. Questions?

AUDIENCE: One of the very first classes-- I don't know if it was during class or after class- you showed me the, say, addition has a primitive that we don't see, and-percent add or something like that. Is that because, if you're doing inline code you'd want to just do it for two operators, operands? But if you had more operands, you'd want to do something special?

PROFESSOR: Yeah, you're looking in the actual scheme implementation. There's a plus, and a plus is some operator. And then if you go look inside the code for plus, you see something called-- I forget-- and-percent plus or something like that. And what's going on there is that particular kind of optimization. Because, see, general plus takes an arbitrary number of arguments.

So the most general plus says, oh, if I have an argument list, I'd better cons it up in some list and then figure out how many there were or something like that. That's terribly inefficient, especially since most of the time you're probably adding two numbers. You don't want to really have to cons this argument list. So what you'd like to do is build the code for plus with a bunch of entries.

So most of what it's doing is the same. However, there might be a special entry that you'd go to if you knew there were only two arguments. And those you'll put in registers. They won't be in an argument list and you won't have to [UNINTELLIGIBLE]. That's how a lot of these things work. OK, let's take a break.

[MUSIC PLAYING]