PROFESSOR: Well, yesterday we learned a bit about symbolic manipulation, and we wrote a rather stylized program to implement a pile of calculus rule from the calculus book. Here on the transparencies, we see a bunch of calculus rules from such a book. And, of course, what we did is sort of translate these rules into the language of the computer. But, of course, that's a sort of funny strategy. Why should we have to translate these rules into the language of the computer? And what do I really mean by that?

These are--the program we wrote yesterday was very stylized. It was a conditional, a dispatch on the type of the expression as observed by the rules. What we see here are rules that say if the object being the derivative is being taken of, if that expression is a constant, then do one thing. If it's a variable, do another thing. If it's a product of a constant times a variable, do something and so on. There's sort of a dispatch there on a type.

Well, since it has such a stylized behavior and structure, is there some other way of writing this program that's more clear? Well, what's a rule, first of all? What are these rules?

Let's think about that. Rules have parts. If you look at these rules in detail, what you see, for example, is the rule has a left-hand side and a right-hand side. Each of these rules has a left-hand side and the right-hand side. The left-hand side is somehow compared with the expression you're trying to take the derivative of. The right-hand side is the replacement for that expression. So all rules on this page are something like this.

I have patterns, and somehow, I have to produce, given a pattern, a skeleton. This is a rule. A pattern is something that matches, and a skeleton is something you substitute into in order to get a new expression. So what that means is that the pattern is matched against the expression, which is the source expression. And the result of the application of the rule is to produce a new expression, which I'll call a target, by instantiation of a skeleton. That's called instantiation. So that is the process by which these rules are described.

What I'd like to do today is build a language and a means of interpreting that language, a means of executing that language, where that language allows us to directly express these rules. And what we're going to do is instead of bringing the rules to the level of the computer by writing a program that is those rules in the computer's language-- at the moment, in a Lisp-- we're going to bring the computer to the level of us by writing a way by which the computer can understand rules of this sort.

This is slightly emphasizing the idea that we had last time that we're trying to make a solution to a class of problems rather than a particular one. The problem is if I want to write rules for a different piece of mathematics, say, to simple algebraic simplification or something like that, or manipulation of trigonometric functions, I would have to write a different program in using yesterday's method. Whereas I would like to encapsulate all of the

things that are common to both of those programs, meaning the idea of matching, instantiation, the control structure, which turns out to be very complicated for such a thing, I'd like to encapsulate that separately from the rules themselves.

So let's look at, first of all, a representation. I'd like to use the overhead here. I'd like-- there it is. I'd like to look at a representation of the rules of calculus for derivatives in a sort of simple language that I'm writing right here. Now, I'm going to avoid--I'm going to avoid worrying about syntax. We can easily pretty this, and I'm not interested in making-- this is indeed ugly. This doesn't look like the beautiful text set dx by dt or something that I'd like to write, but that's not essential. That's sort of an accidental phenomenon.

Here, we're just worrying about the fact that the structure of the rules is that there is a left-hand side here, represents the thing I want to match against the derivative expression. This is the representation I'm going to say for the derivative of a constant, which we will call c with respect to the variable we will call v. And what we will get on the right-hand side is 0. So this represents a rule.

The next rule will be the derivative of a variable, which we will call v with respect to the same variable v, and we get a 1. However, if we have the derivative of a variable called u with respect to a different variables v, we will get 0. I just want you look at these rules a little bit and see how they fit together. For example, over here, we're going to have the derivative of the sum of an expression called x1 and an expression called x2. These things that begin with question marks are called pattern variables in the language that we're inventing, and you see we're just making it up, so pattern variables for matching.

And so in this-- here we have the derivative of the sum of the expression which we will call x1. And the expression we will call x2 with respect to the variable we call v will be-- here is the right-hand side: the sum of the derivative of that expression x1 with respect to v-- the right-hand side is the skeleton-- and the derivative of x2 with respect to v. Colons here will stand for substitution objects. They're--we'll call them skeleton evaluations.

So let me put up here on the blackboard for a second some syntax so we'll know what's going on for this rule language. First of all, we're going to have to worry about the pattern matching. We're going to have things like a symbol like foo matches exactly itself. The expression f of a and b will be used to match any list whose first element is f, whose second element is a, and whose third element is b.

Also, another thing we might have in a pattern is that-- a question mark with some variable like x. And what that means, it says matches anything, which we will call x. Question mark c x will match only constants. So this is something which matches a constant colon x. And question mark v x will match a variable, which we call x.

This is sort of the language we're making up now. If I match two things against each other, then they are

compared element by element. But elements in the pattern may contain these syntactic variables, pattern variables, which will be used to match arbitrary objects. And we'll get that object as the value in the name x here, for example.

Now, when we make skeletons for instantiation. Well, then we have things like this. foo, a symbol, instantiates to itself. Something which is a list like f of a and b, instantiates to-- well, f instantiates to a 3-list, a list of three elements, okay, which are the results of instantiating each of f, a, and b. And x well--we instantiate to the value of x as in the matched pattern.

So going back to the overhead here, we see--we see that all of those kinds of objects, we see here a pattern variable which matches a constant, a pattern variable which matches a variable, a pattern variable which will match anything. And if we have two instances of the same name, like this is the derivative of the expression which is a variable only whose name will be v with respect to some arbitrary expression which we will call v, since this v appears twice, we're going to want that to mean they have to be the same.

The only consistent match is that those are the same. So here, we're making up a language. And in fact, that's a very nice thing to be doing. It's so much fun to make up a language. And you do this all the time. And the really most powerful design things you ever do are sort of making up a language to solve problems like this.

Now, here we go back here and look at some of these rules. Well, there's a whole set of them. I mean, there's one for addition and one for multiplication, just like we had before. The derivative of the product of x1 and x2 with respect to v is the sum of the product of x1 and the derivative x2 with respect to v and the product of the derivative of x1 and x2. And here we have exponentiation. And, of course, we run off the end down here. We get as many as we like. But the whole thing over here, I'm giving this--this list of rules the name "derivative rules."

What would we do with such a thing once we have it? Well, one of the nicest ideas, first of all, is I'm going to write for you, and we're going to play with it all day. What I'm going to write for you is a program called simplifier, the general-purpose simplifier. And we're going to say something like define dsimp to be a simplifier of the derivative rules. And what simplifier is going to do is, given a set of rules, it will produce for me a procedure which will simplify expressions containing the things that are referred to by these rules.

So here will be a procedure constructed for your purposes to simplify things with derivatives in them such that, after that, if we're typing at some list system, and we get a prompt, and we say dsimp, for example, of the derivative of the sum of x and y with respect to x-- note the quote here because I'm talking about the expression which is the derivative-- then I will get back as a result plus 1 0. Because the derivative of x plus y is the derivative of x plus derivative y. The derivative of x with respect to x is 1. The derivative of y with respect to x is 0. It's not what we're going to get. I haven't put any simplification at that level-- algebraic simplification-- yet.

Of course, once we have such a thing, then we can--then we can look at other rules. So, for example, we can, if we go to the slide, OK? Here, for example, are other rules that we might have, algebraic manipulation rules, ones that would be used for simplifying algebraic expressions. For example, just looking at some of these, the left-hand side says any operator applied to a constant e1 and a constant e2 is the result of evaluating that operator on the constants e1 and e2. Or an operator, applied to e1, any expression e1 and a constant e2, is going to move the constant forward. So that'll turn into the operator with e2 followed by e1. Why I did that, I don't know. It wouldn't work if I had division, for example. So there's a bug in the rules, if you like.

So the sum of 0 and e is e. The product of 1 and any expression e is e. The product of 0 and any expression e is 0. Just looking at some more of these rules, we could have arbitrarily complicated ones. We could have things like the product of the constant e1 and any constant e2 with e3 is the result of multiplying the result of--multiplying now the constants e1 and e2 together and putting e3 there. So it says combine the constants that I had, which was if I had a product of e1 and e2 and e3 just multiply--I mean and e1 and e2 are both constants, multiply them.

And you can make up the rules as you like. There are lots of them here. There are things as complicated, for example, as-- oh, I suppose down here some distributive law, you see. The product of any object c and the sum of d and e gives the result as the same as the sum of the product of c and d and the product of c and e.

Now, what exactly these rules are doesn't very much interest me. We're going to be writing the language that will allow us to interpret these rules so that we can, in fact, make up whatever rules we like, another whole language of programming. Well, let's see. I haven't told you how we're going to do this. And, of course, for a while, we're going to work on that. But there's a real question of what is--what am I going to do at all at a large scale? How do these rules work? How is the simplifier program going to manipulate these rules with your expression to produce a reasonable answer?

Well, first, I'd like to think about these rules as being some sort of deck of them. So here I have a whole bunch of rules, right? Each rule-- here's a rule-- has a pattern and a skeleton. I'm trying to make up a control structure for this.

Now, what I have is a matcher, and I have something which is an instantiater. And I'm going to pass from the matcher to the instantiater some set of meaning for the pattern variables, a dictionary, I'll call it. A dictionary, which will say x was matched against the following subexpression and y was matched against another following subexpression. And from the instantiater, I will be making expressions, and they will go into the matcher. They will be expressions. And the patterns of the rules will be fed into the matcher, and the skeletons from the same rule will be fed into the instantiater.

Now, this is a little complicated because when you have something like an algebraic expression, where someth-- the rules are intended to be able to allow you to substitute equal for equal. These are equal transformation rules. So all subexpressions of the expression should be looked at. You give it an expression, this thing, and the rules should be cycled around.

First of all, for every subexpression of the expression you feed in, all of the rules must be tried and looked at. And if any rule matches, then this process occurs. The dictionary--the dictionary is to have some values in it. The instantiater makes a new expression, which is basically replaces that part of the expression that was matched in your original expression. And then, then, of course, we're going to recheck that, going to go around these rules again, seeing if that could be simplified further. And then, then we're going to do that for every subexpression until the thing no longer changes.

You can think of this as sort of an organic process. You've got some sort of stew, right? You've got bacteria or something, or enzymes in some, in some gooey mess. And there's these--and these enzymes change things. They attach to your expression, change it, and then they go away. And they have to match. The key-in-lock phenomenon. They match, they change it, they go away. You can imagine it as a parallel process of some sort. So you stick an expression into this mess, and after a while, you take it out, and it's been simplified. And it just keeps changing until it no longer can be changed. But these enzymes can attach to any part of the, of the expression.

OK, at this point, I'd like to stop and ask for questions. Yes.

AUDIENCE: This implies that the matching program and the instantiation program are separate programs; is that right? Or is that-- they are.

PROFESSOR: They're separate little pieces. They fit together in a larger structure.

AUDIENCE: So I'm going through and matching and passing the information about what I matched to an instantiater, which makes the changes. And then I pass that back to the matcher?

PROFESSOR: It won't make a change. It will make a new expression, which has, which has substituted the values of the pattern variable that were matched on the left-hand side for the variables that are mentioned, the skeleton variables or evaluation variables or whatever I called them, on the right-hand side.

AUDIENCE: And then that's passed back into the matcher?

PROFESSOR: Then this is going to go around again. This is going to go through this mess until it no longer changes.

AUDIENCE: And it seems that there would be a danger of getting into a recursive loop.

PROFESSOR: Yes. Yes, if you do not write your rules nicely, you are-- indeed, in any programming language you invent, if it's sufficiently powerful to do anything, you can write programs that will go into infinite loops. And indeed, writing a program for doing algebraic manipulation for long will produce infinite loops. Go ahead.

AUDIENCE: Some language designers feel that this feature is so important that it should become part of the basic language, for example, scheme in this case. What are your thoughts on--

PROFESSOR: Which language feature?

AUDIENCE: The pairs matching. It's all application of such rules should be--

PROFESSOR: Oh, you mean like Prolog?

AUDIENCE: Like Prolog, but it becomes a more general--

PROFESSOR: It's possible. OK, I think my feeling about that is that I would like to teach you how to do it so you don't depend upon some language designer.

AUDIENCE: OK.

PROFESSOR: You make it yourself. You can roll your own. Thank you.

Well, let's see. Now we have to tell you how it works. It conveniently breaks up into various pieces. I'd like to look now at the matcher. The matcher has the following basic structure. It's a box that takes as its input an expression and a pattern, and it turns out a dictionary.

A dictionary, remember, is a mapping of pattern variables to the values that were found by matching, and it puts out another dictionary, which is the result of augmenting this dictionary by what was found in matching this expression against this pattern. So that's the matcher.

Now, this is a rather complicated program, and we can look at it on the overhead over here and see, ha, ha, it's very complicated. I just want you to look at the shape of it. It's too complicated to look at except in pieces. However, it's a fairly large, complicated program with a lot of sort of indented structure. At the largest scale-- you don't try to read those characters, but at the largest scale, you see that there is a case analysis, which is all these cases lined up. What we're now going to do is look at this in a bit more detail, attempting to understand how it works.

Let's go now to the first slide, showing some of the structure of the matcher at a large scale. And we see that the matcher, the matcher takes as its input a pattern, an expression, and a dictionary. And there is a case analysis here, which is made out of several cases, some of which have been left out over here, and the general case, which I'd like you to see.

Let's consider this general case. It's a very important pattern. The problem is that we have to examine two trees simultaneously. One of the trees is the tree of the expression, and the other is the tree of the pattern. We have to compare them with each other so that the subexpressions of the expression are matched against subexpressions of the pattern.

Looking at that in a bit more detail, suppose I had a pattern, a pattern, which was the sum of the product of a thing which we will call x and a thing which we will call y, and the sum of that, and the same thing we call y. So we're looking for a sum of a product whose second--whose second argument is the same as the second argument of the sum. That's a thing you might be looking for. Well, that, as a pattern, looks like this. There is a tree, which consists of a sum, and a product with a pattern variable question mark x and question mark y, the other pattern variable, and question mark y, just looking at the same, just writing down the list structure in a different way.

Now, suppose we were matching that against an expression which matches it, the sum of, say, the product of 3 and x and, say, x. That's another tree. It's the sum of the product of 3 and x and of x. So what I want to do is traverse these two trees simultaneously. And what I'd like to do is walk them like this. I'm going to say are these the same? This is a complicated object. Let's look at the left branches. Well, that could be the car. How does that look? Oh yes, the plus looks just fine. But the next thing here is a complicated thing. Let's look at that. Oh yes, that's pretty fine, too. They're both asterisks.

Now, whoops! My pattern variable, it matches against the 3. Remember, x equals 3 now. That's in my dictionary, and the dictionary's going to follow along with me: x equals three. Ah yes, x equals 3 and y equals x, different x. The pattern x is the expression x, the pattern y. Oh yes, the pattern variable y, I've already got a value for it. It's x. Is this an x? Oh yeah, sure it is. That's fine. Yep, done. I now have a dictionary, which I've accumulated by making this walk.

Well, now let's look at this general case here and see how that works. Here we have it. I take in a pattern variable--a pattern, an expression, and a dictionary. And now I'm going to do a complicated thing here, which is the general case. The expression is made out of two parts: a left and a right half, in general. Anything that's complicated is made out of two pieces in a Lisp system.

Well, now what do we have here? I'm going to match the car's of the two expressions against each other with respect to the dictionary I already have, producing a dictionary as its value, which I will then use for matching the

cdr's against each other. So that's how the dictionary travels, threads the entire structure. And then the result of that is the dictionary for the match of the car and the cdr, and that's what's going to be returned as a value.

Now, at any point, a match might fail. It may be the case, for example, if we go back and look at an expression that doesn't quite match, like supposing this was a 4. Well, now these two don't match any more, because the x that had to be-- sorry, the y that had to be x here and this y has to be 4. But x and 4 were not the same object syntactically. So this wouldn't match, and that would be rejected sometimes, so matches may fail.

Now, of course, because this matcher takes the dictionary from the previous match as input, it must be able to propagate the failures. And so that's what the first clause of this conditional does.

It's also true that if it turned out that the pattern was not atomic-- see, if the pattern was atomic, I'd go into this stuff, which we haven't looked at yet. But if the pattern is not atomic and the expression is atomic-- it's not made out of pieces-- then that must be a failure, and so we go over here. If the pattern is not atomic and the pattern is not a pattern variable-- I have to remind myself of that-- then we go over here. So that way, failures may occur.

OK, so now let's look at the insides of this thing. Well, the first place to look is what happens if I have an atomic pattern? That's very simple. A pattern that's not made out of any pieces: foo. That's a nice atomic pattern. Well, here's what we see. If the pattern is atomic, then if the expression is atomic, then if they are the same thing, then the dictionary I get is the same one as I had before. Nothing's changed. It's just that I matched plus against plus, asterisk against asterisk, x against x. That's all fine.

However, if the pattern is not the one which is the expression, if I have two separate atomic objects, then it was matching plus against asterisk, which case I fail. Or if it turns out that the pattern is atomic but the expression is complicated, it's not atomic, then I get a failure. That's very simple.

Now, what about the various kinds of pattern variables? We had three kinds. I give them the names. They're arbitrary constants, arbitrary variables, and arbitrary expressions. A question mark x is an arbitrary expression. A question mark cx is an arbitrary constant, and a question mark vx is an arbitrary variable.

Well, what do we do here? Looking at this, we see that if I have an arbitrary constant, if the pattern is an arbitrary constant, then it had better be the case that the expression had better be a constant. If the expression is not a constant, then that match fails. If it is a constant, however, then I wish to extend the dictionary. I wish to extend the dictionary with that pattern being remembered to be that expression using the old dictionary as a starting point.

So really, for arbitrary variables, I have to check first if the expression is a variable by matching against. If so, it's worth extending the dictionary so that the pattern is remembered to be matched against that expression, given the

original dictionary, and this makes a new dictionary.

Now, it has to check. There's a sorts of failure inside extend dictionary, which is that-- if one of these pattern variables already has a value and I'm trying to match the thing against something else which is not equivalent to the one that I've already matched it against once, then a failure will come flying out of here, too. And I will see that some time.

And finally, an arbitrary expression does not have to check anything syntactic about the expression that's being matched, so all it does is it's an extension of the dictionary.

So you've just seen a complete, very simple matcher. Now, one of the things that's rather remarkable about this is people pay an awful lot of money these days for someone to make a, quote, AI expert system that has nothing more in it than a matcher and maybe an instantiater like this. But it's very easy to do, and now, of course, you can start up a little start-up company and make a couple of megabucks in the next week taking some people for a ride. 20 years ago, this was remarkable, this kind of program. But now, this is sort of easy. You can teach it to freshmen.

Well, now there's an instantiater as well. The problem is they're all going off and making more money than I do. But that's always been true of universities. As expression, the purpose of the instantiater is to make expressions given a dictionary and a skeleton. And that's not very hard at all. We'll see that very simply in the next, the next slide here.

To instantiate a skeleton, given a particular dictionary-- oh, this is easy. We're going to do a recursive tree walk over the skeleton. And for everything which is a skeleton variable-- I don't know, call it a skeleton evaluation. That's the name and the abstract syntax that I give it in this program: a skeleton evaluation, a thing beginning with a colon in the rules. For anything of that case, I'm going to look up the answer in the dictionary, and we'll worry about that in a second. Let's look at this as a whole.

Here, I have-- I'm going to instantiate a skeleton, given a dictionary. Well, I'm going to define some internal loop right there, and it's going to do something very simple. Even if a skeleton--even if a skeleton is simple and atomic, in which case it's nothing more than giving the skeleton back as an answer, or in the general case, it's complicated, in which case I'm going to make up the expression which is the result of instantiating-- calling this loop recursively-- instantiating the car of the skeleton and the cdr.

So here is a recursive tree walk. However, if it turns out to be a skeleton evaluation, a colon expression in the skeleton, then what I'm going to do is find the expression that's in the colon-- the CADR in this case. It's a piece of abstract syntax here, so I can change my representation of rules. I'm going to evaluate that relative to this

dictionary, whatever evaluation means. We'll find out a lot about that sometime. And the result of that is my answer. so. I start up this loop-- here's my initialization-- by calling it with the whole skeleton, and this will just do a recursive decomposition into pieces.

Now, one more little bit of detail is what happens inside evaluate? I can't tell you that in great detail. I'll tell you a little bit of it. Later, we're going to see--look into this in much more detail. To evaluate some form, some expression with respect to a dictionary, if the expression is an atomic object, well, I'm going to go look it up. Nothing very exciting there. Otherwise, I'm going to do something complicated here, which is I'm going to apply a procedure which is the result of looking up the operator part in something that we're going to find out about someday.

I want you realize you're seeing magic now. This magic will become clear very soon, but not today. Then I'm looking at--looking up all the pieces, all the arguments to that in the dictionary. So I don't want you to look at this in detail. I want you to say that there's more going on here, and we're going to see more about this. But it's-- the magic is going to stop. This part has to do with Lisp, and it's the end of that.

OK, so now we know about matching and instantiation. Are there any questions for this segment?

AUDIENCE: I have a question.

PROFESSOR: Yes.

AUDIENCE: Is it possible to bring up a previous slide? It's about this define match pattern.

PROFESSOR: Yes. You'd like to see the overall slide define match pattern. Can somebody put up the-- no, the overhead. That's the biggest scale one. What part would you like to see?

AUDIENCE: Well, the top would be fine. Any of the parts where you're passing failed.

PROFESSOR: Yes.

AUDIENCE: The idea is to pass failed back to the dictionary; is that right?

PROFESSOR: The dictionary is the answer to a match, right? And it is either some mapping or there's no match. It doesn't match.

AUDIENCE: Right.

PROFESSOR: So what you're seeing over here is, in fact, because the fact that a match may have another match pass in the dictionary, as you see in the general case down here. Here's the general case where a match passes

another match to the dictionary. When I match the cdr's, I match them in the dictionary that is resulting from matching the car's. OK, that's what I have here. So because of that, if the match of the car's fails, then it may be necessary that the match of the cdr's propagates that failure, and that's what the first line is.

AUDIENCE: OK, well, I'm still unclear what matches-- what comes out of one instance of the match?

PROFESSOR: One of two possibilities. Either the symbol failed, which means there is no match.

AUDIENCE: Right.

PROFESSOR: Or some mapping, which is an abstract thing right now, and you should know about the structure of it, which relates the pattern variables to their values as picked up in the match.

AUDIENCE: OK, so it is--

PROFESSOR: That's constructed by extend dictionary.

AUDIENCE: So the recursive nature brings about the fact that if ever a failed gets passed out of any calling of match, then the first condition will pick it up--

PROFESSOR: And just propagate it along without any further ado, right.

AUDIENCE: Oh, right. OK.

PROFESSOR: That's just the fastest way to get that failure out of there. Yes.

AUDIENCE: If I don't fail, that means that I've matched a pattern, and I run the procedure extend dict and then pass in the pattern in the expression. But the substitution will not be made at that point; is that right? I'm just--

PROFESSOR: No, no. There's no substitution being there because there's no skeleton to be substituted in.

AUDIENCE: Right. So what--

PROFESSOR: All you've got there is we're making up the dictionary for later substitution.

AUDIENCE: And what would the dictionary look like? Is it ordered pairs?

PROFESSOR: That's--that's not told to you. We're being abstract.

AUDIENCE: OK.

PROFESSOR: Why do you want to know? What it is, it's a function. It's a function.

AUDIENCE: Well, the reason I want to know is--

PROFESSOR: A function abstractly is a set of ordered pairs. It could be implemented as a set of list pairs. It could be implemented as some fancy table mechanism. It could be implemented as a function. And somehow, I'm building up a function. But I'm not telling you. That's up to George, who's going to build that later.

I know you really badly want to write concrete things. I'm not going to let you do that.

AUDIENCE: Well, let me at least ask, what is the important information there that's being passed to extend dict? I want to pass the pattern I found--

PROFESSOR: Yes. The pattern that's matched against the expression. You want to have the pattern, which happens to be in those cases pattern variables, right? All of those three cases for extend dict are pattern variables.

AUDIENCE: Right.

PROFESSOR: So you have a pattern variable that is to be given a value in a dictionary.

AUDIENCE: Mm-hmm.

PROFESSOR: The value is the expression that it matched against. The dictionary is the set of things I've already figured out that I have memorized or learned. And I am going to make a new dictionary, which is extended from the original one by having that pattern variable have a value with the new dictionary.

AUDIENCE: I guess what I don't understand is why can't the substitution be made right as soon as you find--

PROFESSOR: How do I know what I'm going to substitute? I don't know anything about this skeleton. This pattern, this matcher is an independent unit.

AUDIENCE: Oh, I see. OK.

PROFESSOR: Right?

AUDIENCE: Yeah.

PROFESSOR: I take the matcher. I apply the matcher. If it matches, then it was worth doing instantiation.

AUDIENCE: OK, good. Yeah.

PROFESSOR: OK?

AUDIENCE: Can you just do that answer again using that example on the board? You know, what you just passed back to the matcher.

PROFESSOR: Oh yes. OK, yes. You're looking at this example. At this point when I'm traversing this structure, I get to here: x. I have some dictionary, presumably an empty dictionary at this point if this is the whole expression. So I have an empty dictionary, and I've matched x against 3. So now, after this point, the dictionary contains x is 3, OK?

Now, I continue walking along here. I see y. Now, this is a particular x, a pattern x. I see y, a pattern y. The dictionary says, oh yes, the pattern y is the symbol x because I've got a match there. So the dictionary now contains at this point two entries. The pattern x is 3, and the pattern y is the expression x. Now, I get that, I can walk along further. I say, oh, pattern y also wants to be 4. But that isn't possible, producing a failure. Thank you. Let's take a break.

OK, you're seeing your first very big and hairy program. Now, of course, one of the goals of this subsegment is to get you to be able to read something like this and not be afraid of it. This one's only about four pages of code. By the end of the subject, I hope a 50-page program will not look particularly frightening. But I don't expect-- and I don't want you to think that I expect you to be getting it as it's coming out. You're supposed to feel the flavor of this, OK? And then you're supposed to think about it because it is a big program. There's a lot of stuff inside this program.

Now, I've told you about the language we're implementing, the pattern match substitution language. I showed you some rules. And I've told you about matching and instantiation, which are the two halves of how a rule works. Now we have to understand the control structure by which the rules are applied to the expressions so as to do algebraic simplification.

Now, that's also a big complicated mess. The problem is that there is a variety of interlocking, interwoven loops, if you will, involved in this. For one thing, I have to apply-- I have to examine every subexpression of my expression that I'm trying to simplify. That we know how to do. It's a car cdr recursion of some sort, or something like that, and some sort of tree walk. And that's going to be happening.

Now, for every such place, every node that I get to in doing my traversal of the expression I'm trying to simplify, I want to apply all of the rules. Every rule is going to look at every node. I'm going to rotate the rules around.

Now, either a rule will or will not match. If the rule does not match, then it's not very interesting. If the rule does match, then I'm going to replace that node in the expression by an alternate expression. I'm actually going to

make a new expression, which contains-- everything contains that new value, the result of substituting into the skeleton, instantiating the skeleton for that rule at this level. But no one knows whether that thing that I instantiated there is in simplified form. So we're going to have to simplify that, somehow to call the simplifier on the thing that I just constructed. And then when that's done, then I sort of can build that into the expression I want as my answer.

Now, there is a basic idea here, which I will call a garbage- in, garbage-out simplifier. It's a kind of recursive simplifier. And what happens is the way you simplify something is that simple objects like variables are simple. Compound objects, well, I don't know. What I'm going to do is I'm going to build up from simple objects, trying to make simple things by assuming that the pieces they're made out of are simple. That's what's happening here.

Well, now, if we look at the first slide-- no, overhead, overhead. If we look at the overhead, we see a very complicated program like we saw before for the matcher, so complicated that you can't read it like that. I just want you to get the feel of the shape of it, and the shape of it is that this program has various subprograms in it. One of them--this part is the part for traversing the expression, and this part is the part for trying rules.

Now, of course, we can look at that in some more detail. Let's look at--let's look at the first transparency, right? The simplifier is made out of several parts. Now, remember at the very beginning, the simplifier is the thing which takes a rules--a set of rules and produces a program which will simplify it relative to them.

So here we have our simplifier. It takes a rule set. And in the context where that rule set is defined, there are various other definitions that are done here. And then the result of this simplifier procedure is, in fact, one of the procedures that was defined. Simplify x. What I'm returning as the value of calling the simplifier on a set of rules is a procedure, the simplify x procedure, which is defined in that context, which is a simplification procedure appropriate for using those set of rules. That's what I have there.

Now, the first two of these procedures, this one and this one, are together going to be the recursive traversal of an expression. This one is the general simplification for any expression, and this is the thing which simplifies a list of parts of an expression. Nothing more. For each of those, we're going to do something complicated, which involves trying the rules.

Now, we should look at the various parts. Well let's look first at the recursive traversal of an expression. And this is done in a sort of simple way. This is a little nest of recursive procedures. And what we have here are two procedures-- one for simplifying an expression, and one for simplifying parts of an expression. And the way this works is very simple. If the expression I'm trying to simplify is a compound expression, I'm going to simplify all the parts of it. And that's calling--that procedure, simplify parts, is going to make up a new expression with all the parts simplified, which I'm then going to try the rules on over here.

If it turns out that the expression is not compound, if it's simple, like just a symbol or something like pi, then in any case, I'm going to try the rules on it because it might be that I want in my set of rules to expand pi to 3.14159265358979, dot, dot, dot. But I may not. But there is no reason not to do it.

Now, if I want to simplify the parts, well, that's easy too. Either the expression is an empty one, there's no more parts, in which case I have the empty expression. Otherwise, I'm going to make a new expression by cons, which is the result of simplifying the first part of the expression, the car, and simplifying the rest of the expression, which is the cdr.

Now, the reason why I'm showing you this sort of stuff this way is because I want you get the feeling for the various patterns that are very important when writing programs. And this could be written a different way. There's another way to write simplified expressions so there would be only one of them. There would only be one little procedure here. Let me just write that on the blackboard to give you a feeling for that.

This in another idiom, if you will. To simplify an expression called x, what am I going to do? I'm going to try the rules on the following situation. If-- on the following expression-- compound, just like we had before. If the expression is compound, well, what am I going to do? I'm going to simplify all the parts. But I already have a cdr recursion, a common pattern of usage, which has been captured as a high-order procedure. It's called map. So I'll just write that here.

Map simplify the expression, all the parts of the expression. This says apply the simplification operation, which is this one, every part of the expression, and then that cuts those up into a list. It's every element of the list which the expression is assumed to be made out of, and otherwise, I have the expression. So I don't need the helper procedure, simplify parts, because that's really this. So sometimes, you just write it this way. It doesn't matter very much.

Well, now let's take a look at-- let's just look at how you try rules. If you look at this slide, we see this is a complicated mess also. I'm trying rules on an expression. It turns out the expression I'm trying it on is some subexpression now of the expression I started with. Because the thing I just arranged allowed us to try every subexpression.

So now here we're taking in a subexpression of the expression we started with. That's what this is. And what we're going to define here is a procedure called scan, which is going to try every rule. And we're going to start it up on the whole set of rules. This is going to go cdr-ing down the rules, if you will, looking for a rule to apply. And when it finds one, it'll do the job.

Well, let's take a look at how try rules works. It's very simple: the scan rules. Scan rules, the way of scanning. Well,

is it so simple? It's a big program, of course. We take a bunch of rules, which is a sublist of the list of rules. We've tried some of them already, and they've not been appropriate, so we get to some here. We get to move to the next one. If there are no more rules, well then, there's nothing I can do with this expression, and it's simplified.

However, if it turns out that there are still rules to be done, then let's match the pattern of the first rule against the expression using the empty dictionary to start with and use that as the dictionary. If that happens to be a failure, try the rest of the rules. That's all it says here. It says discard that rule. Otherwise, well, I'm going to get the skeleton of the first rule, instantiate that relative to the dictionary, and simplify the result, and that's the expression I want.

So although that was a complicated program, every complicated program is made out of a lot of simple pieces. Now, the pattern of recursions here is very complicated. And one of the most important things is not to think about that. If you try to think about the actual pattern by which this does something, you're going to get very confused. I would. This is not a matter of you can do this with practice. These patterns are hard. But you don't have to think about it. The key to this-- it's very good programming and very good design-- is to know what not to think about.

The fact is, going back to this slide, I don't have to think about it because I have specifications in my mind for what simplify x does. I don't have to know how it does it. And it may, in fact, call scan somehow through try rules, which it does. And somehow, I've got another recursion going on here. But since I know that simplify x is assumed by wishful thinking to produce the simplified result, then I don't have to think about it anymore. I've used it. I've used it in a reasonable way. I will get a reasonable answer. And you have to learn how to program that way-- with abandon.

Well, there's very little left of this thing. All there is left is a few details associated with what a dictionary is. And those of you who've been itching to know what a dictionary is, well, I will flip it up and not tell you anything about it. Dictionaries are easy. It's represented in terms of something else called an A list, which is a particular pattern of usage for making tables in lists. They're easy. They're made out of pairs, as was asked a bit ago. And there are special procedures for dealing with such things called assq, and you can find them in manuals.

I'm not terribly excited about it. The only interesting thing here in extend dictionary is I have to extend the dictionary with a pattern, a datum, and a dictionary. This pattern is, in fact, at this point a pattern variable. And what do I want to do? I want to pull out the name of that pattern variable, the pattern variable name, and I'm going to look up in the dictionary and see if it already has a value. If not, I'm going to add a new one in. If it does have one, if it has a value, then it had better be equal to the one that was already stored away. And if that's the case, the dictionary is what I expected it to be. Otherwise, I fail. So that's easy, too. If you open up any program, you're going to find inside of it lots of little pieces, all of which are easy.

So at this point, I suppose, I've just told you some million-dollar valuable information. And I suppose at this point we're pretty much done with this program. I'd like to ask about questions.

AUDIENCE: Yes, can you give me the words that describe the specification for a simplified expression?

PROFESSOR: Sure. A simplified expression takes an expression and produces a simplified expression. That's it, OK? How it does it is very easy. In compound expressions, all the pieces are simplified, and then the rules are tried on the result. And for simple expressions, you just try all the rules.

AUDIENCE: So an expression is simplified by virtue of the rules?

PROFESSOR: That's, of course, true.

AUDIENCE: Right.

PROFESSOR: And the way this works is that simplifi expression, as you see here, what it does is it breaks the expression down into the smallest pieces, simplifies building up from the bottom using the rules to be the simplifier, to do the manipulations, and constructs a new expression as the result. Eventually, one of things you see is that the rules themselves, the try rules, call a simplified expression on the results when it changes something, the results of a match. I'm sorry, the results of instantiation of a skeleton for a rule that has matched. So the spec of a simplified expression is that any expression you put into it comes out simplified according to those rules. Thank you. Let's take a break.