

Instructor (Jerry Cain): Oh, we're on. Oh, hey. Hey, everyone. I don't have any handouts for today. I still have enough material from Wednesday's handouts to keep you busy for the next year if you want. But certainly, the other lecture works. When I left you last time, I had just introduced primarily – in that, I'd talked about char and cuda and cons and all the little different – different atomic operations I want to deal with. Well, I'll give you an example of our first char cuda recursion problem. I had implemented this: sum, I'm gonna call it sum list, this time. And I will do – just call it num list. Okay, and I want to – even though there is actually exposed iteration in Scheme, I want to just pretend that it's not available to us. I want to take this purely functional and recursive approach to everything we implement, for the most part anyway. And so, if I want to be able to type in, at the prompt, sum of – or sum list, rather, something like this right here, I would like it to come back with a 15. Okay, if I want to go ahead and do something like this, I want it to come back with a zero. And so, what I'm going to exploit here is – I think it's something that's pretty conceptually obvious to everyone – is that the sum of this entire list is gonna be 1 plus whatever the sum of this list is. I'm gonna exploit the fact that cuda is available to me, and it's very, very fast to just kind of recurse on the tail of the list. For the implementation, is this right here. If it is the case that null of num list passes, or comes back with a true, then it evaluates this expression. Then this if evaluates to zero and the overall sum list expression evaluates to zero. Otherwise, I'd like to return the sum of the char of the num list, whatever that turns out to be, okay, and add it to the result of sum listing, the cuda of the num list. Okay, there's that; that ends the plus; that ends the if and that ends the define. Okay? Does that make sense to people? Okay, there are a couple of points to make about this that are not obvious. These two things, I think, if you trust the implementation and you just get really, really good at leap of faith and recursive programming, when you're programming in Scheme and Lisp, these are probably gonna work – I mean I'm sorry, you can look at these and just expect that this and that are the intended answers. And if you trace through the recursion, and use leap of faith, you will be able to confirm that. What I want to point out, here, is that what's interesting about Scheme is that, if you were to do this, at the command prompt you were to type in a request to sum list and you did this – let's say that I did hello 1 2 3 4 5, just like that, right there. It is going to fail; it should fail pretty much in any language, unless plus is overloaded to mean something when it's sitting between a string and an integer. But it is interesting to know how much progress it's going to make before it actually errors out and says, "No, I can't do this for you."

It is very literal in its following of the recursive recipe. Because it fails this node test right here, it doesn't return to zero. It calls char of num list and it prepares this hello to eventually be levied against this with a plus sign, but before it can actually evaluate the plus expression right here, it has to fully commit to the recursive evaluation of this thing right here. Okay? So what's going to do all of this work? And I'm kind of spinning – I'm putting a bad PR spin on it, but I don't want you to think about it that way. It's actually gonna go and assemble this 15, recursively. It's gonna bring the lift up the 15 as the evaluation of the recursive call, and it's gonna, at that moment, do a plus of hello against the 15 and only then, after it's assembled the 15, will it say, "Oh well, I can't do that."

and it'll issue an error. Okay? Does that make sense to people? The reason I'm highlighting that is because it underscores the fact that there is absolutely zero compiled time elements with regard to type checking. The type of compiling that goes on here – when you talk about compiling, it's almost always a fancier word for translation, and that makes sense in the context of C and C++. Anything that's considered to be compilation in Scheme or Lisp is nothing beyond just parsing and making sure that the parentheses match, and the double quotes are matched, and things like that. Does that make sense? After it's been digested, and a link list has been built in memory to store this thing, it evaluates it and you're basically executing the program. Okay? So it's at runtime that tightness matches or, actually, detected and while the code is running, does it say, "Well, that won't work out." You may think that that's awful, but think about the equivalent in C of a heterogeneous list. It would have to involve void stars, okay, if you're gonna really support that kind of heterogeneity. And at runtime, it would also error out, but it would error out in a much more mysterious way, with those segfaults or those bus errors. Okay? At least this, right here, would tell you that there's a tightness match between the string hello and the 15. It might not actually have its real values, but it might say plus doesn't work when sitting in between a string and an integer. Okay? Does that make sense to you people?

Because Scheme is a runtime language, it actually preserves type information about all of the data. For the lifetime of this hello, the hello sits in memory and it's part of a small data structure that is tagged with something of an enumerated type saying, "This thing is a string." You have a little bit of a hint of that from a section problem that went out in week two. The first real discussion session, where I had people CACat note all the strings that were in a Scheme list, okay, the nodes in those lists are the things that really kind of back these types of data structures. And that they really are tagged with list, or integer, or double, or float, or Boolean, or a string and things like that. Okay? Does that make sense to people? Okay. So there's that. Let me just get a little bit more brute force in my recursion here. It doesn't – you don't always have to deal with lists. I will deal with lists, in a bit, in a couple of minutes again, but I could just define Fibonacci on n, okay, to be this, just to introduce a couple of other predicates. I could ask whether or not n comes in as zero. There's a built-in called zero question mark. The question mark is legal part of the syntax, just like it is part of null. It's supposed to stare at you and say, "There's a question being asked about the data that's in argument." And if that's the case, you just go ahead and return zero. Okay? Otherwise, you can do this. In which case, you can return 1. Okay? And otherwise, you can commit to the sum of whatever you get by calling fib against n minus 1 and fib of n minus 2. That ends the plus; that ends the f; that ends the first a f; that ends the define. Okay? It's kind of comical with all of the parentheses but it's much easier to do it in an editor, where you actually have parentheses balancing. Okay? I'm gonna rewrite this because I actually started writing this thinking factorial and forgot that I had two base cases. So I just kind of gracefully went through it like this is what I intended. But I actually would not – I probably wouldn't cascade my base cases like that. Okay? You can actually assemble base cases using or. I showed you how to do it with and before, so let me rewrite this: Define fib of n to be this. I would probably do something like this. If it's the case that – let's say or – you could do zero question mark n, or you could do something like that. I just want some symmetry in the

base cases, like that. Then, I would just return n . And actually, I'm – yeah, that's right, n zero and 1. This, right there, is what gets evaluated as an expression if this, right here, passes.

Otherwise, I'd want a return fib of n minus 1 – I'm sorry, this is not minus. These are – no that's right. N minus 1 and fib of n 1 is 2. That ends the plus; that ends the if; and that ends the define. Okay, I'm assuming you're well behaved and you're not entering in a negative – not a negative number. There are ways to actually flag those, as well. There are trees and error function that you can invoke instead of plus or if or null question mark, you can call the error function and it prints out whatever string happens to be the argument, and it terminates the function call. Okay? Does this make sense to people? Okay. It's just weird getting used to this whole hyper parenthetical environment. And it's very difficult to remember that the parenthesis comes before the function call name, as opposed to afterwards. Okay? Make sense? Question right there?

Student:[Inaudible] after the n , do you need a parenthesis or, no?

Instructor (Jerry Cain):Which n is this?

Student:It's the [inaudible].

Instructor (Jerry Cain):Oh, this right here? No, this is actually occupying – let's say that the if, if not really a function, it's what's called a special form because it doesn't evaluate all of its arguments. But it takes either two or three expressions, afterwards. The first one is always supposed to be something that can be interpreted as a Boolean. So either it has to evaluate as a false or non-false. This, right here, has to be some expression that just evaluates if this test passes. It's actually completely ignored if this test fails. Okay? Does that make sense? And then, this is evaluated.

Now, you're thinking you need, maybe, something like that. Yeah, well that, actually, would try to invoke the n function with zero arguments. Okay? So n is supposed to be evaluated as a stand-alone variable. So now that I've talked about this if thing, I will underscore another feature. Just to show you how true this whole runtime idea really is in the Scheme language, if I do this – let's say that I just type this in at the prompt. I type in if it's the case that zero of zero – it looks pretty stupid, but I specifically want this test to pass. Then, I print out 4. Otherwise, I print out hello and 4.5 and the list 8 2, all added together.

Now, from a type standpoint, that else expression, that's gonna be evaluated if the test fails, it's completely nonsense. And if you know that if you type that in stand-alone, it would completely break down. Everyone believes me there? Okay? Do you understand that it's easy to parse it. It doesn't actually – when it parses this thing and builds a data structure behind the scenes, it does very little other than to say, "That's a token; that's a token; that's a token; and that's made up of tokens." and builds a list behind the scenes.

But only if it really has to commit to evaluating it, does it go ahead and see whether or not the code that's attached to the plus symbol actually can accommodate strings and integers and integer lists all as arguments, combined. But because the evaluation of this thing right here sees a test that passes, it goes and it evaluates 4. This is gonna print out 4, and it's not like it's gonna even bother analyzing whether or not the else expression would've worked out or not. Does that make sense? Okay.

That is emblematic of the type of thing. Some people say it's a feature. Some people say it's actually that's it's the type of thing that they don't like about runtime languages because a compiler would actually say, "You know what, please look at that because I don't think it could ever work out." Even if it's only called once in a million scenarios, I want to know upfront that it's gonna work out. A scripting language or a runtime language, like Scheme, or Python, which we'll talk about later, or Java Script, which we might talk about a little bit, about the last day of class, wouldn't analyze that at all.

So it's technically possible that code that you write doesn't get exercise for weeks, okay, because it just doesn't happen to come up in the workflow of a typical runtime scenario. Okay? Does that make sense to people? But it is emblematic and representative of the type of things that have very little or no compile time element to them. We're so used to very strongly compile time – I'm sorry, we're very used to strongly compile time oriented languages, like C and C++ and even Java, to a large extent, that we look at something like that and say, "Well that's just a blocker." And it really, technically, isn't. You can write whatever you want to here, if you know that it's not gonna be executed. And you wouldn't do that; I'm just illustrating a feature of the language. Okay? Does that make sense to people? Okay, very good. So let me do a couple more things. I want to get a little bit more intense with the type of recursion example I do because I want to illustrate how very conceptually sophisticated algorithms can be captured in a very small snippet of Scheme code. Again, it's like feature and a curse, at the same time. It means that the language itself is very articulate and terse in its ability to state and express algorithms cleanly. You don't need pages and pages to make a point; you need, like, four lines. Okay? Normally, you'd say that that person's a very good speaker but if actually a language, you say that it's clean or terse or expressive. What I want to do – and I love this function, so I'm happy rewriting it. I want to write this function called flatten. Okay? That's actually not an "h," flahen; it is flatten. Okay. And I want it to – I'll just illustrate how it works. I type this in at the prompt. If I give it this, it has the easiest job in the world. All it has to do is, it has to return the same list because the list is already flat. Okay? There are no nested lists, at all. But if I give it something like this, I want it to come back with this right here. Okay?

So I want it to conceptually, even – it will synthesize a new list. I want it to, more or less, look like it's taking the original list and taking all the intervening parentheses, and just removing them. And I'm implying that it has to either be integers or lists of integers; doesn't have to be. This could be the string 3; this could be the string 4. And I would just require that these things be strings in the output. Okay? It's almost as if you're doing this: something of a traversal through the tree, that's more or less implied by that nest of list structure. And you're just doing this inward traversal and you're threading through all of

the integers like it's popcorn on a Christmas tree or something. Okay? And then doing this and saying, "This is where all the atoms live." Okay? And preserve the order in which you actually read them from left to right. Okay? Does that sit well with everybody? Okay. So you look at this and it turns out this would be very difficult, I think, to do in C or C++, not because of the algorithm. The algorithm is actually kind of tricky, but in C and C++, you would have to deal with the manual link list mechanics. You'd have to actually take lists and take the atoms that are in the nodes and, somehow, build new nodes around them and thread everything together. And it would be 50 percent memory management, 50 percent algorithm. Okay? All of the link list mechanics are, more or less, managed for you in Scheme, so there's 50 percent less to think about. Okay? Let me just implement this. I get to illustrate a new data structure. I also will say that I will not – I'm gonna avoid this. There's this one little nuance of the algorithm that kind of makes it more complicated. I just want to pretend that it's not a problem. If I give you something like this, in terms of the empty list, can either be considered a list or it can be considered an atom. In some dialects of Scheme, the empty list actually stands for null and false.

So I just want to pretend that this – that empty lists never actually appear in any of our examples. Okay? I don't want to worry about the edge case of having to deal with those. I just want to have this nice simple thing, where I'm always dealing with real atoms or lists that have some material in them. Okay? So I want to define this flatten thing, and I'm just gonna call it – I'll call it a sequence. Now I have three different things I want to think about: I am always – let me put a couple of examples up on the board. These are the three cases I want to deal with. One's a base case and two have recursive insights that we need. I'm either gonna hand you the overall empty list. That's different. Okay? Eventually, we're gonna account for everything and the recursion's gonna lead to an empty list. I'm just not allowing empty list to reside as elements inside the top-level list, okay, or anywhere in some nested list. Then I have this scenario; and I'm just gonna do this. I normally would commit to any extra values. Okay? I'm also gonna have this scenario. There's nothing in theory that prevents us from handling the scenario where the char of the list is, itself, a list. Okay? How do I flatten that? I do absolutely nothing; I just return the empty list because it's already flattened. It's basically like the vacuous list that, not only is it flattened, it's totally deflated. Okay? This right here, what I want to do is, I want to just a prepend, or cons the char because it's atomic and not a list, I want to cons the front – I'm sorry, I want to cons this one onto whatever I get by recursively flattening the cuda. Make sense to people? Okay. This is different. What has to happen is that I don't want to cons this element on to the front of whatever I get by flattening this recursively because I don't have a nested list right at the front. What I really want to do is I want to append the flattening of this to this right here. Make sense? Now there was – that didn't technically it all because it might be this. You know, it could be this ridiculously hyperlinked structure that happens to be sitting at the char. So what I really want to is, I either want to cons this really simple atomic element to the front of the flattening, or I want to append the flattening of this to the flattening of the char. Does that make sense?

Okay. So you have to be clear on the difference between cons and append. I'm not gonna do cascaded ifs, like I did right here. I'm gonna use another data structure. There is

something called cond. I like this. There's no real analog – there's a little bit of an analog to this in C and C + +. It's kind of like a switch statement, but it's a switch statement on Boolean tests, as opposed to scalar values. What's presented to cond is a series of pairs. The first item in the pair is a Boolean test; the second item in the pair is the express that gets evaluated if the test actually passes. So what I would do up front is, I would ask whether or not the sequence is null. And if that is true, then the entire cond should evaluate to this thing right here. So that's that; that ends that; that's the pair. I have that right there. So this opens up the list of pairs; this opens up the first pair; this happens to be associated with the test that's inside the pair. Okay? Does that make sense to people? Okay. It's just a lot of parentheses; it's more – it's easy to get, just once you type it in. But the first thing I want to check for is whether or not the sequence is null. Okay? Make sense? I'm actually gonna handle this scenario as a second clause right here because it happens to be a built-in predicate that tests whether or not something is a list. If I fall past the null sequence because it's not null, then I know that I have at least one element in there.

So what I want to do is, if it's the case that the list predicate – that's another built-in; it was in the first of the two handouts I gave in on Friday. It's just this predicate that says, "Is the thing that's serving as my only argument here, is it a list? Because if so, I want to react differently. Is the sequence actually," – I'm sorry, "Is char of sequence – char of sequence, is that a list?" If so, what I want to do is, I want to flatten the char; I want to flatten the cuda and I want to append them. Does that make sense to people? So what I would do is, I would call the append function on the flattening of the char of the sequence, and the flattening of the cuda of the sequence. That ends the flatten; that ends the append; that ends the entire order list. Okay, the entire [inaudible]. There's enough parentheses, had to be smushed in here between u and the end of the all editors, to balance that parend, right there. Okay? The third scenario is supposed to be the else scenario. Okay? I've seen a lot of things happen here. Normally, you want to make sure that the last in the sequence of tests that gets evaluated is guaranteed to evaluate to true. Okay? I've seen some people do this because that certainly evaluates to true. That's the Boolean constant true in Scheme. There actually is a key word that only makes sense in the context of the con statement. You can't use it outside of a con statement, which is actually interesting from a syntactic standpoint. You can't always – you don't see that in a lot of other languages.

You can use the keyword else right there, which basically, is synonymous with true in this context. What you would do is, you would cons the char of the sequence, which requires no flattening because it's not a list, to whatever you get by flattening the cuda of the sequence. [Inaudible] Okay. You guys get what's going on here? Yes, no? Okay. This right here, cond, it actually does expand; it is this macro that syntactically expands to cascaded ifs. Okay? It evaluates this test; if it passes, then the overall cond evaluates to that. Otherwise, it falls through, and evaluates this test, this test right there. If it passes, then it evaluates this thing; otherwise, it ignores it, and falls down here. Else always evaluates to true, so it would certainly evaluate this if it got this far. Okay? If, for whatever reason, you put a test there that also failed and it's potentially the case that all

cond tests fail, then the – not the return value, but the cond – what the cond expression evaluates to is not actually defined. You can't rely on it..

Most implementations just return the empty list, okay, but I don't want you to code that way. I want you to make sure that any con statement that you ever evince to implement an algorithm, has this default case. This is the equivalent of default in a C or C++ switch statement. Okay? I just want you to know that one of your expressions is gonna work out. Okay? Make sense? Now, you notice that there's really no functionality – I mean, think about all the functions that are used: There's cond, of course; there's null; there's list; there's append; there's char; there's cuda; there's flatten itself. At no point, do I invoke any functionality that's specific to ints vs. Booleans vs. strings, which is why it's perfectly happy to flatten lists that have a variety of atomic types inside. Okay? Does that make sense to everybody? Okay. Question over here? No. Okay. What else is going on? Okay. You guys are good here? Okay. Questions on the left. What's up? Yup.

Student: Wasn't that thing you write an [inaudible].

Instructor (Jerry Cain): After else? This right here is cons. Is it just the word you're curious about, or what does this do? I can write it out more cleanly if it's not clear.

Student: [Inaudible]

Instructor (Jerry Cain): This is cons. Remember what cons – I talked about – what's that?

Student: What does it do?

Instructor (Jerry Cain): Cons, basically – cons is short for construct and it takes an atom; and it takes a list; and it evaluates to a list where this is the first element and everything in the list comes afterwards. Okay? So it's, basically, this built-in prepend function. And if you think about it from an implementation standpoint – we're gonna talk about the memory model, next week, as to how this thing works. But if you have a handle on the front of the list, it's very easy to change what the front element is. Which is why char and cuda and cons, which manipulate, either extract or update the front element. This is me, holding a link list here. Why those things are supported operations is because they can run in constant time. Okay? Yup?

Student: So at least in our implementation of Scheme [inaudible], arguments can be evaluated before they're passed into a function?

Instructor (Jerry Cain): They actually do, yes. That is common with, I think, most languages. Not – actually, I don't know of any specific examples, specific languages, that defer the evaluation of arguments until after the char. Like, there's some versions of, well some older dialects of Java script. And I think certain features of Pascal, where they had some kind of copy – like copy – passing like copy semantics. They had some, like, kind of parameter passing Scheme that I'm not familiar with because it was really before my

time. I just know it exists; I just don't know it very well. But in our world, in our Scheme implementation, it is the case that the arguments are evaluated before the function is invoked. Okay?

Everyone doing good? Okay. I have – I want to motivate another example. You have tons of examples in the two handouts I gave you out on Wednesday. I'm not gonna go over all of them because I say that, for every concept, there's probably three examples, and I just don't want to cover all three, when I think one suffices for lecture material. I do want to implement one other function that's clever in its use of or and and, and recursion to determine whether or not an integer list is actually sorted in a way that it respects the less than sign. Okay?

So let me write this function. I want to write this function called sorted, with a question mark, just because I can do that. And I'll just be consistent with the way some of the built-ins use it. And I will allow it to take this thing that I'll call a num list. I probably should call it an integer list because I'm really thinking about integers, specifically. But what I want this to do, is I want it to return true if the list that's fed as an argument is already sorted in non-decreasing order. Okay?

So something like this: Sorted – sorted of something like 1 2 2 4 7, should return true or something equivalent to true. Okay? And if I try to walk the system and do something like this, certainly it's close to sorted in some informal – by some informal definition of close. But because it dips once, it's technically – excuse me, not sorted the way I want it to be. Okay? So there's this; this will come back with a false. I want to implement this with the understanding that the num list really is numbers. So that I can compare things with the less than sign.

Turn it and you can compare strings for equality and inequality. You have to use different flavors of less than and greater than, and equals to. You have to basically use the functions that are designed, better designed to work on strings. So there's that. I've erased this because I want to make room for it. The base case is actually – I don't want to say it's too – it's very clever. It's actually kind of, it's kind of obvious.

But all lists of length zero, and of length 1, are already sorted. Okay? So what I want to do is: I want a return. It doesn't – it almost looks like there's no – I love this implementation because it looks like there's almost no base case. But I exploit or and it's short-circuit evaluation. Okay? And so I really do have a base case here, even though it's not expressed in this pure, if base case, do this scenario. I want it to return the truth or falsity of the following: That either the length of num list, this is another built-in – and I know I'm spraying built-ins at you – but they're all kind of obvious and you should know that they just exist in any language that has lists as a built-in.

If the length of the num list is less than 2, then you have your answer, and you could care less what the second of the disjunct is gonna evaluate to. Okay? If this passes, you're done; the thing is sorted. Otherwise, you actually know that, not only is there a char, but there is a char of a cuda as well. Does that make sense? Okay? So if I get this far, and I'm

evaluating the expression that happens to have an open parend that I just drew right there, then I know that I have at least two elements.

So what I want to do is I want to confirm two things. If I have these two elements, I have x and y. They could actually really be x and y, I mean they just have to valuate to something. And then, I have all of these other things. I need to confirm two things to decide if this thing is actually sorted. I have to see that this thing right there – that x really is less than or equal to y. And that the cuda passes the sorter predicate. Make sense? Okay.

So there's that. So I have this and right here and I want to do this. Less than or equal to of char of num list and – I want to do this; I want to – I'll this – I'll invent – I'm not inventing; I'm actually using another function that's built-in. It's kind of funny that it exists. You have no idea what c a d r is, but you kind of have an idea as to what it's probably getting at. This right here is obviously supposed to be the first element. When you see something like c a d r, you're supposed to actually pretend that this is really two words nested, like cascaded; that it's supposed to be the char of the cuda. Does that make sense?

If you wanted to put char of the cuda with extra nesting calls, you could do it. But if, for whatever reason, you want to get the cuda, of the cuda, of the cuda, of the cuda, of the cuda, that happens to be a built-in. Okay? If you know, structurally, that you want the char of the cuda, of the char of the cuda, and because of the way things are nested together, you want to do that, then you can use that instead. Or you can go forth with the actual exposed char in cuda calls; that's all this evaluates to anyway. Now, you may wonder whether or not, you know, something like this just parses and it figures it out. It does not. It actually stops at four, at least as far as I know. That's the built-in; that's where the built-in stops.

Okay? If you really do programming in Scheme for a living, like you're figuring out when c d d d a r is relevant, it's almost as easy as figuring out when char is relevant. So it's not like you have to go through and, like, understand the full 2 to the 4th different permutations of this right here. Okay? You just want whatever; you just want to know that they exist so that, when I write something like this, you know that I'm really trying to get to the second element. Okay?

That balances that; that balances that call. I also want sorted to just work out on the plain old cuda, just 1 d, of the num list. Ends the and; ends the or; ends the define. Okay? There are some implementations of Scheme, not ours, that understand that matching all the parentheses at the end is a little bit difficult. Some of them have overloaded. Ours does not, but this is funny. Some of them have overloaded to say, "Oh, whatever, just use the square bracket." and it'll just round them out to figure out what it would have been had you actually had the patience to count your parentheses. But unfortunately, ours does not do that. So you really do have to commit to this thing called counting, okay, to actually match everything. Okay? Does that make sense? This is nice because if this thing fails, it doesn't bother with the recursive call. So there are two short-circuit evaluations

here that really do save us time. It is prepared to march through enough recursive calls to come back with a true and it has to do that. It has to be exhaustive in its search of the entire array to come back with a true because it doesn't want to make any mistakes. But it only has to find one flaw in the array, for it to return early with a false. Okay? Does that make sense to people?

Now, it turns out that – I didn't know this until I put it in the handout – but this particular flavor of Scheme allows less than and less than or equal to, to actually take multiple arguments. We're used to seeing something like this, and when we look at that, as soon as you get used to the prefix notation for function evaluation, you look at that and say, "Okay, that's really asking a question as to whether 1 is less than 2," and it comes back with a true. It turns out you can do this as well. I'm not saying you should exploit this, but the is sorted for less than or equal to, for instance this right here, where I have two sixes at the end. This also would evaluate to true. I don't want you to have to remember this; this isn't really intellectually engaging. But it's kind of neat that it was smart enough, or at least robust enough, to recognize that less than or equal to doesn't have to be this implicit binary function. It could really just be a request to see whether or not all neighboring pairs in a list actually respect the common notion of what less than or equal to means, rather. Okay? It doesn't – not surprisingly, it doesn't work for equals or not-equals. Okay? I mean, I guess it could've worked but it's, like, why would you expect them to put down a list of length 20 and ask whether or not they're all equal to one another. Okay? But it does work for all of the inequality operations. Okay? Make sense? Okay, that's fun. So what I want to do here is, I want to speak a little bit about – I'll be more sophisticated in my explanation of this next week, when – but I'll probably draw the same pictures next week. I just want to get to them now so that I can really talk about how function pointers work, okay, the equivalent of function pointers in Scheme. Do you understand that, algorithmically, this is pretty much spot-on as to how it would confirm whether or not any sequence is sorted, except that it's constraining it to be number specific because of its use of that right there? Even this is fine. That's there to compare lengths of a list of two and it's gonna be there regardless of whether we're dealing with string lists, or complex number lists, or whatever. Okay? But this is the thing that's actually requiring that all of the elements in the list actually be numbers. Okay? Let me just give you a little bit of an idea as to what happens when you invoke a function. I've already spoken informally about it, but let me actually draw some pictures.

When you invoke `char of 1 2 3 4`, a couple of things happen. It actually digests this token, okay, and it happens to occupy something of the leading slot in the series of slots that make up a link list. Does that make sense? And you know, by looking at it, that that is a symbol `.` It's supposed to be a test of functionality that tells you how to digest and manipulate the remaining arguments. Okay? It's very close to this; this thing itself points to a list. And it has some tight tagging so that it knows whether something's a list or an integer, or whatnot. But this points to a 1, points to a 2, points to a 3, points to a 4, etc. Okay? Technically, that thing's quoted; so there's really a quote function there, as well. But that's kind of the over-arching idea of what the memory model would look like, for this particular call. Does that make sense? Now, without being too sophisticated about it, I think it's reasonable for you to understand. I just drew a symbol table of functions there.

And among the functions that are defined here, there actually is an entry where the key is the string `char` and it's associated – and I'll just write it this way – with code. Okay? That code that should be invoked whenever `char` is consulted as the function that should be guiding evaluation. Okay? So what happens, and it really does do this behind the scenes, it digests this; it builds this as a data structure. It does any recursive evaluation of arguments that it needs to do, but it will be suppressed here because of the quote. It then goes and finds the `funct` – the code that's associated with this and then, there's some general meta-code thing, behind the scenes, that figures out how to manipulate the arguments, and produce a result based on the contents of this thing, right here. Now, you've seen how we define, not `char` or `cuda` but how we define `flatten` or `sorted` question mark or `sum list` and things like that. We exactly associate the definition of something like `flatten` or `sorted` with code that's expressed in link list form. Does that make sense to people? Think about it; you use lots of parentheses after the define of `num list`. So this, right here, actually is stored in memory much like these things are. Okay? It's just understood to be an in-memory representation of the recipe that needs to be followed, and a series of instructions as to how to manipulate the remaining arguments of the function invocation. It really is this in-memory representation of the functionality. Okay?

So it's almost like – I'm trying to think of the best analogy of this. It's almost like you're reading, from a file, the series of instructions that are allowed. Okay? And you store those instructions in memory and associate them with this keyword, right here. And that there's some way, behind the scenes, that it actually uses this to guide execution of this type of execution statement, right there. Okay? Does that make sense? Okay. Now, there's much more detail to this; I make it sound like you could just write it in an hour. It's not the case at all; it's actually pretty, pretty challenging. But nonetheless, that's the over-arching idea. The reason I'm saying that is because this thing right there, it self-evaluates to this. It's because it occupies the 0th slot, as opposed to the 1th, or the 2nd or the 3rd slot, okay, that this right here is assumed to be a variable that's associated with code as opposed to just raw data. Okay? So in the same way that this evaluates to itself because of the quote, this evaluates to the code. Okay? There is actually a way to type in an anonymous function, right there, and have it guide execution. I'll show you that on Monday. Okay? But `char` really, here, it does evaluate to a block of code; it's like this. In C or C++, this is, at compile time, taken to be an instruction as to where to jump to in memory. In Scheme, it's taken as an instruction as to where – what symbol to look up in the global symbol table of definitions, and assume that there's code associated with it. Okay? The reason I'm saying that, is because at the moment, we've implemented this thing called `sorted` so that it hard-codes this in right there. Okay? If I want to generalize this, okay, and go all, like, `vector sort` on you or, like some kind of – be as a polymorphic in my support of a generic sort is possible, you do it in Scheme. You do it with something that's equivalent to a function pointer. Okay? But rather than passing in the address of the function, you pass in the function itself. Okay? When I say that, you actually pass in the code as a parameter. Okay?

Let me show you what that would look like. At the moment, `sorted` question mark of the list 1 2 3 4, just works according to that recipe, and comes back with a true. What I want to do is this: I want to be able to invoke `sorted`, so that it can take 1 and 3 and 5 and 7.

But recognizing that this could've been an array of – I'm sorry, a list of strings or a list of lists, I actually am gonna pass in that. It's a stand-alone token; there's no spaces in between, so it knows that less than or equal to is the thing that's being passed in is the 2nd argument now. You haven't seen the new prototype for sorted yet, but you can imagine that there's gonna be some variable that catches whatever this evaluates to. And it's gonna evaluate to the code that it's associated with that knows how to compare two or more elements, actually technically one or more elements, okay, to figure out whether or not all of the elements respect, in the way they're listed, respect the less than or equal to predicate. Okay? If I want to do this – this would, just presumably, come back with true because we're assuming that less than or equal to does the right thing – sorted a b d c, and I pass in this thing called string less than, that just happens to be function that's the equivalent of less than on strings. So it operates like normal [inaudible] less than in C++ strings. I'm sorry, less than – yeah, that's right. Okay? I would expect this to return false for the right reasons because this, right here, is failing it. Does that make sense? Okay. So I would expect this to come back with a false. So it turns out that the implementation of this sorted thing doesn't have to change much at all. I am gonna rewrite it fresh, even though it's in the handouts because I don't like changing code unless it's absolutely ridiculous not to, to just change it. But I'm smushed on room here, so let me just write it fresh. I want to define sorted and I'm gonna take – I'll just gonna call it seq, so I'll have a shorter word for it, and I'll just call it comp. Now, the way I'm invoking those Schemish expressions over there, I'm expecting that something has been evaluated, and it's evaluated to code and it's being pushed on to the 2nd of these two variables right here. Okay?

So locally, this comp, it's almost like it's the name of a function that exists elsewhere. Okay? So I'm gonna implement it to be core – I don't know where that is – or less than length of sequence of 2 and – what's with the c? Sorry, char and cuda. And I want to put comp right here. Char of sequence, char of the cuda of sequence, and then, at the same time, I need sorted on the cuda, and finally use a c of sequence with the same comparator, an and, and the or, and the define. So if, at a first pass, the kindergarten explanation here is that comp is function pointer, okay, that can assume the place of a function name in the implementation, that would probably be enough to just make you understand what's going on. Okay? Technically, it is a variable on – that actually has attached to it, whatever this, or this evaluates to. And it doesn't have to be a built-in; it can be anything that we define that knows how to compare 2 or more elements or just 2 elements, in this case, exactly 2 elements in this case, and come back with a true or false. Okay? Does that make sense? So this, right here, it's attached to a list, like 1 3 5 7, or a b d c. This, right here, is also attached to a list. That because of the way it's invoked and it appears as the 0th element in a function call, or in a list, that it's supposed to be associated with code, or a link list that knows how to guide execution and manipulation of everything that follows it. Make sense? Okay, that's great.

So there you have that. There are a couple of other examples in the handout that deal with this. I'm not done with the function pointer idea. I have so many analogs to things you're just accustomed to, in C and C++, the notion of mapping, the notion of anonymous functions, and client data, and all that kind of stuff. There are all these cool things in

Scheme you just have not seen in other languages. Okay? And they exist in extensions to the languages, like extensions to C and C ++, but they're not part of the core language. There's something that I'm gonna talk about, on Monday, is core to Scheme and it doesn't really exist in the core of any of the languages that you've seen before. Okay? So I'll have that; I'll have more examples with function objects, which is what those things are really called. And then, I'll talk about mapping, filtering, things like that. Okay? Have a great weekend.

[End of Audio]

Duration: 52 minutes