ProgrammingParadigms-Lecture21

Instructor (Jerry Cain):Hey, everyone. We're online. I have a good number of handouts for you today; they're all going out. They've been posted since yesterday. Two Scheme handouts for material that I, just briefly, touched on in the last five minutes of last Friday, and today, and Wednesday, and Friday. Still, probably, we'll be covering Scheme. Your next assignment goes out today; it's not due for a while. I'm not making it due until the Wednesday evening after the weekend. History has shown that the hardest part about this entire assignment is getting the first of the eight or so functions that you have to write, written. Because that means you have to get used to CALA, which is a development environment, and getting used to all of the parentheses, and understanding how to test your code, and things like that. After you get that working, then the assignment is, actually I think, on the easier side, as far as 107 assignments go. But recognize that the first function, for a lot of people, is actually a lot of work. It's not unheard of for some people to spend, like, an hour or to 90 minutes on just the first function, which is like seven lines of code, and then, after that it's just smooth sailing. But they have to iron out the rough patches with the first function, then it gets easier. Okay? We will have a discussion section, tomorrow. The examples in this section handout for tomorrow are being passed out, right now. It's actually a pretty difficult set of problems. They're old final exam questions that I've given in past quarters. But nonetheless, I mean, it's good material for understanding all the little quirks and neat things about Scheme. So I'll let you work with those tomorrow, with Ben Newman, who will be teaching it. What I thought I'd do is, I thought I'd introduce you, a little bit more formally, to the development environment and show you how that's gonna contribute to your Assignment 7 experience. Okay? So let me bring this down, create some mood lighting, and let's see where we are with regard to getting everything to fit on the screen. Please, please, please, please, please let it fit on the screen. I don't know what that's about, right there, a little mushroom. It's a microphone, I think. Oh, there we go. Okay, so let's look at – make sure that this fits on the screen. I'm not sure what people in TV-land can see. It looks like they can see my entire computer screen, but it's projecting larger up there, for some reason. But as far as I can tell – we're working on it. This is you driving, right? It's actually not a disaster, if you just wanna leave it. Yeah, 138's good. That's me up top. Let me just – we'll figure it out. Okay. Everybody see that, in the room? See, all they see is Jerry at the bottom. So on the lanes, and on the pods, where I know I've tested it, it also should be working on this as well. If you want, you can deal with the Scheme environment like I introduced, last week, or you can type in 4, and have it confirm that it's 4. Or you can type in hello and have it confirm that it's really hello. But if you want to do things that are meaningful, from a mathematical standpoint, you can do that and come back with 21.

So this is shell-like environment, where you happen to be dealing with the Scheme language. Now, I don't want to imply that, somehow, programming and Scheme is all command line driven; it's not. Normally, what you will do – let's spell quick correctly – let's – sorry, exit. Let me do this – let me, actually – well, this is huge. Oops. There we go. Okay. That's a little too small for this screen, though. Hold on a second. It's gonna have to be this. Okay, good enough. What I want to do is, I want to split this; I want to bring this, and open up a shell, down below and I'll play with CALA right here, in the

lower half. But what I'm gonna do up here is, I'm gonna prepare a little coat-snip at the one I finished with on Friday, where I would define, in place, this thing. If you recall, I called it sorted. And I said it was capable of sorting the sequence, but in the end, I said we're gonna deal with Scheme's equivalent of a function pointer, which we call a function object in Scheme. And the intent here, is that I'm defining this two-argument Scheme function that takes the list and then lock them into a data type. I want it to be heterogeneous by specification, but I don't care whether it's a string list, or a list list, or an int list, or what have you. And then, I pass in this thing that knows how to compare pairs of elements that reside in that list. And this is what I wrote last time. I wrote, "Either it's the case that this list is so small that it can't help but be sorted, or it's the case that this CNP, when levied against the char of the sequence and the cuda" – I'm sorry, "the char of – of the cuda of the sequence" – is that fitting on the screen? Yes, good. There's that, "and it's the case that, at the same time, sorted cuda of sequence, with the same comparison function, kinda just works out with this recursively." Okay?

So I can bring this down so you can see it. And that's what I left you with on Friday. Does that make sense? How's that turning out on the screen? I think that actually looks okay on the screen, so TV people, people in TV-land shouldn't be yelling at me. Okay? I'm gonna save this file, and there's a command in CALA, this actually common in most Scheme interpreters where you can load a code file. So in the past, we've prepared dot C and dot CC files ahead of time, and then gone into a shell and actually typed make to build the executable. What we do now is, we actually prepare our Scheme functions in a file that's suffixed by dot Scm for Scheme and then, we actually load them into the interpreter, using this load command that I'm doing down here. Does that make sense to people? Yes, no? Okay. So if this all works out, do that, and then I can type in, "Is it the case that this list is sorted according to that predicate right there?" And hopefully, it comes back with the truth. And it does, the green – the green T. Okay? If I do this, and I jump all over the place, and I ask whether or not it respects the greater than and equal to predicate, it should come back with a false, and it does. Okay? Make sense to people? Okay. So that's the over-arching idea as to how you interact with a Scheme developmental environment. I give you a dot SCM file for the "Where am I?" assignment, that's going out today. It has tons of code that's already defined in there, but you're gonna go through this – this update the Scheme file, save, and then type load in a parallel Kawa shell, okay, just to kinda bring in whatever code you've most recently written. Okay? Does that make sense to everybody? Okay, very good. So there's that. I will put this to bed; I may come back to it. That's fine. But I want to write code on the board because I think it's nicer to create the code, than to have it prepared ahead of time.

There is this one idiom from C + + with iterators, and from Assignment 3 with vector math, that I want to see the equivalent of in Scheme. Do you understand that, if I write this as a function, sum all – actually, I don't want to do that, I'm sorry. Let's say, not sum all. Let's say double all, and I give it this list. And without writing code, I think it's pretty clear that the output of this should just be 2 4 6 8. If I have another function, called increment all, and I type in this right here, I expect it to spit out 2 3 4 5. Okay? Do you understand how those are algorithmically similar? They both visit every single element in the list, using char cuda recursion. Okay? They both apply some functionality to each

char, okay, but the output is a list of exactly the same length as the incoming list where each thing that ever served as a char of a cuda is transformed by either the double operation, or the increment operation. Does that make sense? Okay? Except for the fact that this is Scheme, it kind of screams vector Map. Does that make sense to everybody? Okay. Well, it's not like Map, as a verb, was coined for C and C + + purposes. It actually exists everywhere, and rather than doing this, what I could do is, I could define a double function, which is not a built-in, but I want to just do this and I can equate the double operation of an x with that as an expression. I could just do it on one line; I don't have to do it over multiple lines if I don't want to. Okay? I can also define the increment function to be associated with this successor thing. Okay? Does that make sense? There is an operation in pure Scheme called Map, and it takes 2 or more arguments. We're gonna deal with it as if it's a pure 2 argument function. The second argument – I'm sorry – the first argument to Map can just be the name of some previously defined function, whether it's a built-in or something you just defined. And then, you can specify what list you should be Mapping over. Okay? And the result of that call, right there, if I typed it in at the prompt, would be 2 4 6 8. If I do the same thing with the Map operation, but Map incr, this function over the list instead, then I should get out 2 3 4 5, and be done with it. Okay? Does that make sense? Now, Map is a little bit more sophisticated than I'm making it out to be here. I'm making it look like it Maps unary functions over single lists. Okay? That will change as we get a little bit more experience with it, but I want you to understand that this is kind of the more functional approach to this up there. If you actually implement that and that, then you're implementing it in terms of exposed char cuda recursion, with both implementations.

Does that make sense? Yes, no? Think about what the implementation of that would look like. What I could do, instead, is I could either use the built-in Map, or I'll just pretend for a minute that the Map function isn't a built-in and we're gonna define it in a second. We can extract the notion of a Map to use char cuda recursion regardless of what this function turns out to be. Okay? And recognize that this can be passed in and caught in a variable like CNP was in the sorted question mark predicate function. Okay? So I'm actually going to – I'll give you the full story on Map. If you want to, you can – you can Map a lot of interesting functions over lists, if you want to. If I want to Map the char function over a list, I could certainly do it. It better be a list of lists, and that would spit out the list 1 4 11. Okay? It's still the case that it transforms the list, right there, of length 3 into another list of length 3. And it uses this operation right there to figure out how to transform each thing that comes up as a char in the recursion to some element in the final list. Yes? Okay. If I want to do this: Map, cuda 1 2 4 8 2 11, and I can do that. This would give me the list 2, the list 8 2, and the empty list. And I, once again, get a list with 3 top-level elements inside. Okay? The Map function – we're not gonna implement it this way, but I just want to advertise what it is. It's kind of neat that it can do this. If I want to Map the cons function – now, cons is different than all of the other examples, or all of the other Mapping functions I've chosen in the past because it's not unary. Okay?

Well, cuda and char and ink – inker and double are all unary functions, which makes them compatible with a single list to follow it. But if I want to, with a real Map, I want to Map cons, what I can do is, I can provide two lists, where one list provides all the first

arguments to the cons calls, and the second provides all the second arguments. So I could do this, and what happens is that the Mapping function takes cons and applies it to the 1, and the 4 list to synthesize the first element. Okay? The next element, the second element of the product, has a 2 cons onto the front of the empty list. So the product of this thing, right here, would be the list 1 4, the list of 2 just by itself, and the list 8 2 5. Okay? Does that make sense? If I wanted to, I could Map the plus function over as many lists as I want to and now, I have to add these together, but it would give me 1 10 and 16. The output is a list of length 2 because all of the input lists are of length 2 and this assimilates all of the chars of the 3 arguments into one sum and then it assimilates all the chars in the cudas into the second argument and then it realizes that all the lists are empty. Okay. Map is robust enough that, if you give it lists of different lengths, if I were to sneak in a third element in the middle list there, it wouldn't freak out. It actually just terminates when the smallest of all the lists reaches its end. Okay? So that particular one would be ignored because the other lists or at least one of the lists is of length 2. Okay? That make sense to people? Okay.

This is the purely functional way of actually visiting, or transforming an incoming list, normally, or a pair, or a triple of incoming lists into another list, okay, that's completely unrelated from a memory standpoint, but it kind of emulates what foreach does in – the foreach algorithm does in C + +, and what vector Map does from our Assignment 3. Okay? We can implement Map. I'm not gonna implement the binary version yet. I just want to use what we've learned, based on the last ten minutes of last Friday, and in the first few minutes of today's lecture, to implement our generic Map function. I can define Map, but I'm not gonna call it Map because that's the name of a built-in. I'm gonna call it my – and I'm gonna say unary Map to emphasize the fact that I'm implementing a subset of the real Map function. Okay? And I'm gonna pass in an actual function and I'm gonna pass in – I'll call it a sequence. Somebody asked about dot dot dot, last time and the equivalent of dot dot dot. Do you remember that question? Okay? I will use this to motivate the equivalent of dot dot dot, when I implement the generic version of my Map, later on. Okay? But right now, I'm just dealing with one unary function, and I'm dealing with one list, which I'm calling seq. Okay? If it's the case, regardless of what f n turns out to be, if seq is null, then I return the empty list, and f n has nothing to do with it. Otherwise, what I want to do is, I want to cons onto the front whatever I get by Applying f n to the char of the sequence, okay, to the result of calling my unary Map, mum, of f n against the cuda of the sequence. That ends the my unary map; that ends the cons; that ends the f; that ends the entire definition. Okay? Except for the word f n which, in this case, is the name of a local variable that's ostensibly bound to a function, this would, more or less, be the implementation of double all and increment all that I crossed out, up there. Okay? I would've hard-coded double and incr into two separate implementations. I'm trying to abstract a way and say I'm gonna make this general enough that I can pass in the function that I'd like to be applied to all the elements inside. Okay? Does that make sense to people? Yes, no? Okay. So there's that.

So this Map function – you may say, "Is it all that useful?" And the answer is I think it's kind of useful. I want to show you a different way of actually flattening all of the elements in the list. This is, I think, fascinating how it does this. Okay? If I put this list up

here – we're revisiting the flatten problem that we dealt with last time. Do 1 2 – actually, you know what? Let me just – let me introduce a couple of other functions, before I do this. I want to get to this, but I'll get there in five minutes. There are a couple other functions that are built-in to Scheme, that I want you to understand. There are – one's called Apply; one's called Eval. Eval is easier to introduce, and it's a little bit quirkier. Do you understand that, when I type in open parend plus, space one, space two, space three, space four, space five, close parend, that the interpreter actually reads it, tokenizes it, recognizes that most of them are integers, and it somehow, figures out how to invoke the plus function, okay, based on what I typed in. Eval recognizes the fact that everything, at least from – whether it's sorting a file, or it's typed in at the shell, comes in as a stream of text that needs to be tokenized and evaluated. Okay? If you type in – this looks weird, but if you type this in, with that quote there, that suppresses evaluation. Right? This would actually come back, and it would list this as the output of that operation. Eval, even though this isn't – you wouldn't type this in this way because you would just go with the more direct way of adding the first three integers – but if you type this in like that, whatever its 1 argument is, it evaluates it. In this case, it evaluates just to the list constant plus 1 2 3 because it's quoted, and then it evaluates it as if it were typed in at the command prompt in the interpreter. Does that sit well with everybody? So this would, slightly less direct means, be a way of actually printing out the sum of the first 3 integers. Okay? Now, this is certainly the less common of the two operations. Apply is cool; Eval is neat because it advertises the fact that functions and data are really exactly the same thing in Scheme; that everything is expressed as a list. Okay? The function – the code that implements Eval in the interpreter for this Eval, is the very code that gets invoked every time you hit enter, okay, at the interpreter. It reads in the text, up to that – to the last matching parentheses, and then it evokes some Eval function, behind the scenes, to get it to produce a 6, or whatever it wants to produce. Okay? Apply is a little bit different. Apply actually allows you to specify which function should be, basically, pressed against all of the arguments that follow. That's another way of computing 6 in a less direct way. Okay? But it won't always be the case that we know that the last argument is the list constant 1 2 3. It could be something that's procedurally or functionally generated. Okay?

What Apply does is it always takes exactly 2 arguments. The first one is always supposed to identify some block of Scheme code. What follows it is always supposed to be a list of the type of data that can actually be levied against by this type of operation. So what Apply does is, it effectively takes the plus; it cons it onto the front of this thing and then, it evaluates it in this Eval sense. Okay? So this would come back with a 6. Okay? More meaningful version – more meaningful example of where you would use Apply? If I just assume that I don't ever have an empty list, if I want to define the – let's say the average function. Literally, I want to compute the average of all the numbers in a list, and I'll just say num list. And I'll assume that they're all doubles, so that I don't have any fractions and truncation and things like that to deal with. So I have all of these scores, like 39.5 and 40, and 29.5, and all these types of scores we saw on the mid-term. If I wanted to, I could write my sum all function, using char cuda recursion, okay, to synthesize the sum of all the scores, and then divide by the length of the list, or I could do this. That right there. Okay? That's a more meaningful, less contrived example because I don't know what num

list is. And if I really do pass in num list as a list of scores, I have to, somehow, get the plus to be effectively onto the front of that anyway. I could use char recursive cursion. I can't use Map because Map transforms a list of length n to a list of length n; I don't want that. I want just to take a list of length n and produce a single number out of it. Okay? So the Apply function is this quick way of actually saying, "You know what, I really wanted a plus sign onto the front." or "I wanted the function to be the very first element of this list that's right here. So can you just, please, tuck that onto the front and then evaluate it like it was there all along?"

Okay? Does that make sense? That is a much cleaner way to do it. The drawback of exposed char cuda recursion is that it's asymmetric, and it advertises the asymmetry that's involved in visiting the char of the entire list first, and then the char of the cuda second, etc. Whereas, both Map and Apply make it look like all of the elements in a list are peers, and the simultaneously contribute to the overall effort – or the overall computation. When you Map the double function over the list 1 2 3 4 5, all the way through 10, it's like it just dumps out the list 2 4 6 8, etc., okay, in one fell swoop, without actually exposing the fact that there's char cuda recursion involved. The same thing with Apply. Rather than actually doing this sum all function, where you recursively, you know, compute the sum all of the cuda and then add, with a plus sign to the char of the entire list to that, you just say, "Okay, num list, all of the elements, you're participating in a plus." Okay? "I'm applying the plus to all of you. Cooperate and come up with your sum." Okay? And that's the way you actually feel about it, and that's the much more functional way of actually dealing with everything. Okay? The fact that recursion's involved, that has nothing to do with the functional paradigm, really. That has more to do with the fact that the central data structure in the language is the list, which is a recursive data structure. Okay? That make sense to people? Okay. As far as Eval is concerned, when would you use this? It requires a lot more – the examples are much more complicated. Where I've seen Eval used is, I've seen Eval used where Apply couldn't be used. You could imagine applying a plus, or a cons, or a char, or something like that, to arguments. Right?

You can't Apply define, or add, or or because they're special functions that don't necessarily evaluate all of their arguments. You understand what I mean when I say that? Like and, and or short-circuit evaluate, right, so they actually cannot be thought of as normal functions. If you wanted to "Apply," and I have to use it in quotes, "Apply" and to a list of predicates, to figure out whether or not they're all true or not, you couldn't use Apply because it really requires this to be a bona fide function, and and is not one of those. But you could cons onto the front of the list and then evaluate it. Okay? Does that make sense? Okay. I've also seen—and this is really sophisticated stuff – I've actually never coded this myself, but I have seen, programmatically, the definition of new functions have been synthesized textually. Okay? Do you understand what I mean when I say that? Like, you actually can build up, as a string, something that looks like the definition of a function. Okay? And I say – I meant a definition of a function, something like this. So imagine an algorithm, even though we don't an example of this here because I think it requires a really sophisticated setting, in order for you to need this. But you could imagine this being built up as a string in a language. Okay? Or built up as a list,

rather. So if, somehow, you can textually synthesize that right there, and use cons, and char, and cuda to construct all of these symbols in one big, complicated list and then, pass all of that to an Eval statement, you can programmatically introduce the definition of new functions while the program is running. Okay? That's actually a pretty neat idea; some people would say it's dangerous that a code would be this self-aware and evolving while it's running. Some people were like, "I don't – that's fine as long as I'm a good enough programmer and I can handle it. Then I'll just lever it like – be able to use that feature of the programming language."

I've never seen it, but I have heard enough, and read enough about the contribution of Eval and the introduction of evolving functions, and functions on the fly, in things that involve randomization and things like genetic algorithms, where new sentient beings are modeled, you know, while the program is running. And they have different little definitions of how they should execute and respond to the world, in the simulation. Okay? Does that make sense? Okay. Now, I'm gonna focus on this one because I think it has a broader impact on the type of code we're gonna write. What I want to do now is, I want to revisit the flatten problem, now that I know about Map and I also know about this thing called Apply. Okay? Now, I'm gonna revisit the flatten thing. That's the char of sum list I want to flatten. Here is the cuda – I'm sorry, here is another list, another element that I want to be involved in the flattening process. And then, I'll have just 10, like that. And you know what the product of – you know what the product of the flattening of that should turn out to be. It should be 1 2 3 4 5 10, all as peer top-level elements and no intervening parentheses. Parentheses as bookends, okay, but nothing in between, other than the numbers. Does that make sense? When we implemented this, first we recursively generated the flattening of this, and then we either consd or appended, or prepended, this element right here onto the front of it. Okay? In this case, we would flatten this. It wouldn't be very hard, but we would just flatten it. It would be very quick about it. And then we appended this to that right there. Does that make sense? Okay? If this had been an atom, then our [inaudible] list that we just would've cons this onto the front of the recursive flattening. Well, what I'd like to do is say, I could actually – while I'm defining this, this is kind of kooky but, while I'm defining this flatten function, I could implement it recursively. I could recursively flatten all of the elements, where this gets transformed into a 1 2; this gets transformed into a 3 4 5, and this gets transformed into a 10, just temporarily. Okay? Do you understand how the definition of flatten could involve a recursive call to flatten, but not directly, but via Map?

Oh, I want to flatten the entire thing. Oh, I should Map flatten over the list, okay, and then append all of the things that I get back. Does that make sense to people? Okay? If this is the product of flattening the first element; this is the product of flattening the second element, and I just ensure that this is the product of flattening the last element. Okay? Then, I could effectively – I could effectively get the final product by just appending all of these lists. I could Apply – this is gonna come back like that, if I really use a Map call. Okay? I could Apply the append function to the product of the recursive Map call, so that it goes through with a thread needle and just, basically, creates one big sequence out of all of these elements right there. Okay? It's really kind of hot. So let's implement this, okay, and we'll use some leap of faith arguments to defend why we know

it's working. But what I want to do is, I want to define the flatten function and I'm just gonna give it a sequence. Okay? Let's just – let's forget about the base case. Okay? We're not even sure what the base case is. Okay? Here, actually, is a base case. But let me just think about the recursive case, where if I know I'm dealing with a top-level list of many items, then what I want to do is, I want to Map this flatten routine, that's being defined right here, over sequence. Now, think about what that does from a leap of faith standpoint. This transforms a list – this list of length 3, okay, into this list of length 3. Okay? If flatten works, in this, like basically, in this involuted way where it actually – the definition of flatten is compatible with itself. Okay? It's supposed to transform this into this, right here, via this one Map call. Okay?

And after that happens, I can Apply, not plus, but I can Apply the append function to the result of that Mapping. Okay? It's like taking the list of length 3, pulling the left parenthesis in a little bit, and sticking the word append at the front, and say, "Okay, evaluate yourself as if append were there all along." So if I stick in, right there, the word append and evaluate it – and that's what the Apply statement there is doing – it will give me one list with 1 2 3 4 5 10 in it. Okay? So that's fun, I think, but we, actually we only want to do that if this thing really is a list. Okay? If it is the case that the sequence itself is not a list, okay – in other words, if I actually, recursively, hand a 10 to this thing – does that make sense? Okay. Then I want to just return the listification of this sequence. Okay. Otherwise, I want this to be the else flaws; that balances that – I'm sorry, that balances that, balances that. This ends the if, and this ends the define. Okay? So the base case deals with the scenario, when you dive so deeply into the recursion that you've actually arrived at a 1, or a 2, or a 3, or a 5, or a 10. Okay? And you say, "Okay, now I have to start backing up." But because cons append is involved, I have to wrap them in parentheses, so that they behave nicely in the context of the Apply append call. Does that make sense? Okay? Think about what happens – this is recursively flattened, according to that formula. Okay? It just works out. Same thing with this right here. When it recursively calls flatten against the third element in the sequence, it gets this atom, which is not a list, so it has to wrap these things in parentheses so that, when it participates in the Apply append call with all of the other peers that are progenerated recursively, it actually plays nice. Okay? You guys getting this? Okay, very good. Now, some people do not like the fact that I gratuitously put these parentheses around the elements, all of them, just for them to disappear. But if I really just want to illustrate how Apply and Map are working together, then this is still a good vehicle for this function, I think. There's no very easy way for you to look at a list, and know whether all the things inside of them are atoms. So the list 1 2 3 is different than the list 1 2 list 3. Right? Okay? If I could, somehow, detect that 1 2 3, everything inside is a top level atom, then I could come up with a more sophisticated implementation here, that's a little bit faster

But all I'm trying to illustrate is how Apply and Map will end in this one example here. Okay? That make sense? Okay. This is probably the tightest recursion – example of recursion you may have seen, ever. Okay? In C + +, it's buffered with all of this memory allocation. Okay? And it's not so clever in its use of base cases. Scheme is this very expressive – that's the positive PR spin on it – it's this very dense way of expressing algorithms. It's usually as terse as the most articulate person is in describing what the

algorithm's supposed to do. Okay? And that is very difficult to look at and understand how it works. Okay? You guys are good? Okay. I have one other topic I want to introduce, and I'll spend a lot of time, on Wednesday, going over very advanced examples of all of this stuff. But Scheme has been different than everything we've seen before, in the sense that it's almost entirely runtime; there's no compile time element to it, whatsoever. It is weakly typed, which doesn't mean that there aren't types involved; it means that all kinds of type checking is deferred until runtime. Okay? And if there are problems, then it just presents the problems, if it ever comes up, while the code is running. There's one next thing, actually exists in extensions to C and C + +, but it's not part of the core language. I want to think about how we would implement this function. I want to define a function called translate. And I don't mean translate in a linguistic sense; I mean translate in a distance sense. I want to take a list of points in one-dimensional space – so like points on the number line – and I want to shift all of them by a certain delta. Okay? So I'll just say something called points, and I'll say delta, right here. Now, before I go in and fill in the body here, this is how I want it to work. I want to be able to call translate against 2 5 8 11 25, and I want to be able to pass in 100. Okay? And I want it to spit out 102, 105, 108, 111, 125, just like that. Okay? I want it to take the lists – the first argument that's a list of length n – n points, and I want it to generate another list of n points, where everything's been shifted by some delta amount. Okay? Does that make sense? So you look at this and, given that I just taught you Map, 35 minutes ago, you may say, "Okay, well, maybe he wants me to use Map." And the answer is, I do want you to use Map. Okay? But, the problem is that, there's no clear existing function that knows how to translate a number by another number that's not specified, until the function call actually happens. You understand what I mean when that feels a little bit like client data to this Map call right here? It's external to the actual thing being Mapped over, but it, somehow, is involved in the product? Does that make sense? So I kind of want to Map something, right there. I'll just put this big placeholder, and that's supposed to be the function that somehow figures out how to add this number to every single element inside points.

I have a very difficult time naming a function right there because I can't call a function called increment by delta because that type of function is gonna have to take two arguments, not just one. It will have to take one element that gets bound to the char of the list that it's Mapping over, and you'd have to also pass in the 100 to it. Does that make sense? Okay? You guys understand the problem here? Okay? This has to, basically, either be global data, okay, or it has to behave as global data in the execution of whatever function gets placed right here. Now, I'm moving a lot of space for this thing right here. What you can do in Scheme, and a lot of other languages, but not C or C + + is, you can actually embed the definition, and scope the definition of a function, inside another function. Okay? This is the way you do that: You erase the placeholder boundary. There's that. You can actually define a function in Scheme, on the fly, using a keyword called Lambda. And Lambda is just a gesture to the calculus that backs most programming languages, and the way it deals with function call and return. But if I write this right here—just think of it as boilerplate – that means that I am defining a function without giving it a name. Lambda is not the name. Lambda is just a placeholder, meaning it's an anonymous function I'm defining on the fly. Okay? I am saying, quite clearly, that

this anonymous function takes one argument. Why? Because the way we're using Map, right here, we're Mapping over a single list right here, so I need to actually let every single thing that is ever a char of a cuda, okay, actually be passed in and bound to x. The body of this function has to add x to something. It has to take the 2 to a 102, or the 5 to a 105, or the 11 to a 111. Okay? The only way it's gonna do that is if its definition can involve the actual value that delta has adopted on this particular call. Okay? Does that sit well with everybody?

So I can do this, and that ends the Lambda definition. So this, right here, is this anonymous function, whose implementation is framed in terms of the one parameter called x, and something that is effectively global to its scope, called delta. It happens to only exist as a local variable to the outer function, okay, but for the lifetime of this function definition, it exists only long enough for it to be Mapped – for it to be Mapped over this thing called points. This is going to adopt whatever value was served the actual translate call. Okay? So that, if I call this with 100, it constructs an anonymous function, on the fly, where this is the number 100. If I pass in and call this thing a second time, but I put 1,000 there, it constructs a second function, okay, that is, from a memory standpoint, is completely independent of the first indication of this. And it actually just puts this – places a 1,000 there as opposed to a 100. Does that make sense? Okay? So that is how that feature works. It is not anything you've seen in standard C or C + +. Now, it turns out that G + +, which is all about extending C + + because it just doesn't like the original language, I guess. It allows you to define inner functions. I haven't advertised it to you. In fact, I didn't know it until a year ago, when some student showed it to me. I'm like, "Uh, let's pretend that that doesn't exist because I don't want people using it." But this, right here, is actually quite common, in the Scheme world, anyway. Okay? There are two things I want to announce in the last four minutes before I leave you go. You actually can explicitly define inner functions, if you want to. I actually like both ways. I think this one, it's kind of jockier. It's like, "Yeah, I'm not intimidated by – I don't need function names." is kind of what it's saying. "I don't need to wear my seatbelt when I'm coding."

So. But there is a more expressive way of doing it, that I think is fine. If I wanted to define this translate thing, a little bit more – I don't want to say eloquently – it's just – I guess so, a little bit more clearly and be more obtuse about the fact that some inner function is involved, I could define translate of – I'll just call it seq and delta, just like that. I could actually provide an inner definition – oops, keep making this mistake – I can define this inner function, internally, called shift by. And I can actually give it a name where I actually add delta, okay, and then, right afterwards, I can Map this shift by function over the sequence, and that ends the entire define. Now, the reason I don't like this is because it kind of breaks the functional paradigm. I've always equated the definition of some function with one expression. Right? I've – increment is equated with the plus function; translate over here is equated with the Map function. There's always been one expression that was provided as the body of the function, whereas I'm actually providing a sequence here. This is like, Roman numeral No. 1; this is Roman numeral No. 2. Pure Scheme allows you to actually define a sequence of items, okay, and then the expression of whatever the last one evaluates to is what the overall function evaluates to. I don't like this because it isn't purely functional. All of a sudden, it takes on this C or C

+ + like idiom, where it constructs a lot of things piece-meal in order to build up the overall answer. Okay? But you can define an inner function, if you want to; materialize a meaningful name for the short term; define it in terms of local variables in – global symbols and local variables plus n delta, and then use the last line to actually define what the Map should do, or what function should be Mapped over the sequence. Okay? Does that make sense? Okay.

So there's one other thing I should tell you about the define thing. When you write something like this: Sum x and y, and you just – a simple placeholder definition, just to illustrate my point, right here – that right there, is just syntactic sugar for this. Define the sum to be equated with – now, you may be a little weirded out by the syntax, but the second one is actually much more clear about its association of some function with an actual name. Okay? Does that make sense to people? So that, every time you use sum in a call, it evaluates to this Lambda function that's compatible with two additional arguments. And then, this executes where x and y have been replaced by whatever those two additional arguments evaluate to. Okay? Define actually works – if I wanted to do this – actually, I shouldn't use x. Let's say I want to do, like, PI 3.14; you haven't seen this before and I don't want you to use it, but you understand what's probably happening there. It's forever associating capital P, capital I with the number 3.14. This is just, basically, doing the same thing. It just happens to be associating the word sum or the symbol sum, not with a constant – I'm sorry – not with a numeric constant or a string constant, but with a Lambda constant. Okay?

And so every time you use sum, or char, or cuda, or append, or Map, or my unary Map, or whatever, the actual symbol that's at the front, sum plus even, technically – plus and minus and times and all of those – they're all bound to actual Lambda functions. We prefer this because it's probably just a little bit more readable, and it's more consistent with the way we've learned how to program in other languages. But this is much more clear; I want this, but this is functionally equivalent. This advertises quite clearly that sum is being equated with this thing, right here. Okay? Does that make sense? Okay. So Scheme really is all about symbol and symbol evaluation, and functional evaluation, and right down to the actual definition of the functions, and the way they're stored in memory. Okay. So I will cover some more sophisticated examples on Wednesday. You'll also see some sophisticated examples in tomorrow's section, as well. So again, I –

[End of Audio]

Duration: 50 minutes