ProgrammingParadigms-Lecture22

Instructor (Jerry Cain):Everyone, welcome. I have no handouts for you today. I have several handouts from the past days that I have not finished yet and, in fact, if there's one handout that I consider to be the advanced scheme handout it's probably handout 32. I'm gonna be going through two of those three problems that are in there right now. You saw some sophisticated scheme examples in yesterday's section handout. I want to do a few more today. In particular, I want to get this lambda function idea down and out and some significant examples. I only went through a very trivial example of how lambda gets used last time, but I really want to do some complex, dense, recursive scheme functional programming using mapped lambdas today.

The first one I want to do is actually more of a discreet math problem than it is a scheme problem, but we're gonna drive the problem using scheme. It deals with the notion of what is called a power set and those in 103a and 103b over the past year are very familiar with the power sets. The loose definition of a power set is that it is a set that contains all of a sets subsets. So if we talk about the original set just {1,2,3}, but you know what, I'm not gonna draw it that way because we're not gonna draw it that way in scheme. The set {1,2,3}. Okay.

If I want to list every single subset of {1,2,3} I have to basically enumerate all the subsets such that either one can be excluded or included, two can be excluded or included. There's two options for the presence or absence of every single element. So for this particular set right there there are going to be a total of eight subsets. So the actual grouping, they won't necessarily be in this order, but I somehow want my power set function to be able to eat and digest this as a list and synthesize something like this. The order will be a little bit different, but I will do it this way. Two, three, I'm not sure this is the best order, but there's a method to my madness here.

Do you understand how those four – they're all technically subsets of this thing right here. There's no element in any one of those four sets that I've enumerated that can take the element that isn't there. So technically they're all subsets, including the empty set. Okay. Here are the other ones. And that's the full power set expressed using just scheme lists. Okay? Now, the reason I drew it this way is I'm trying to make it clear what recursive structure I'm gonna exploit by my implementation. We all agree that those are all the subsets that exclude the one. Okay? Does that make sense?

These are all the subsets that include the one and I've ordered them in such a way that there's a clear 1:1 relationship between one subset and the one below it. Each one of these four things right there is identical to those except that a one has been prepended as the English word cons as a scheme word to the front of the list. Does that make sense to people? Okay. So it's almost as if I want to recursively generate this right here. I want to recursively generate it again, but the second one I want to cons a one onto the front of it. Basically, I want to map some function over this list right here that knows how to cons a one onto the front of it. Is that so odd everybody?

And then I want to append this list that has no one's whatsoever to the list that has all ones in it. Okay? As far as the – I'll draw it this way. The power set of the empty list is concerned you may think that this generates the empty list. That's technically not true. That would speak if there are absolutely no subsets of the empty set and that's not true. Technically the empty set is its own subset. Okay? It's called an improper subset, but it's technically a subset. So I expect my power set function when given this right here to return that. Okay? So it's a singleton that has its own element, the empty set. Okay?

I always expect the lengths of these lists to be a perfect power of two. So I have two to the zero for this one right here. That's the length. Two to the eighth as this right here. Okay? It's very clever. It's very dense. I'm gonna write this twice. The first time I'm gonna be careless in making the same recursive call twice. Okay? But I want to go ahead and define – I'm just gonna call it PS. Not postscript, it's short for power set. I'm just gonna give – name the variable set. That's not a key word we'll pretend it's not a key word in any scheme dialect. I don't think it's one in ours, but just pretend that I can use that as a variable name without interfering with the name of some built in function.

I just want to do a simple base case check right up front. If it's the case that I have the null set it's like the language is working in our favor. It's awesome. Then according to this specification right here I want that to prompt the entire thing to evaluate to that as a constant. Okay? That technically is incorrect. Okay? That would go in and pull out the empty list and assume it's bound to functionality because of the place it's occupying in the list, but by putting the quote in front that suppresses evaluation. Okay? Makes sense?

Otherwise what needs to happen is that I have to somehow encode the appending of this right here to the appending of that right there. This right here is just the power set of the cdr of a set. Forget that the one is in there. Take the leap of faith. What is power set of {2,3} supposed to generate? It's supposed to generate that. Okay? It's just best to get this on the board and to just show you – explain why the code is correct. I do want to append two things. Okay? I certainly want to generate those two lists and then concatenate them. The first one I get by pure leap of faith of calling power set of cdr of the set. You're just assuming while you're writing this, of course, that the PS function works while you're writing it. That's just the leap of faith principle.

The second thing is actually a little bit more involved. The second half of the entire power set is certainly related to the recursive call, but we somehow have to visit every single subset in the recursively generated power set and cons the missing element onto the front. We were correct to omit the car from anything that had to do with this power set right here. This is the first 50 percent. That's all about excluding the one or the car of the original set. I want to append this to more or less the same thing. I'll call it subset. Where I cons the car of the set onto the front of a subset. Now, that is not over with. Let me just do PS of set. That ends the map. That ends the append. That ends the if. That ends the define.

Let me do you a favor and make it clear what function is being mapped over this power set. I'm sorry. That's not power set of set. It is – that was not quite right. Cdr of set ends the map append if defined. This is the on fly function that I told you about on Monday. Okay? It is interesting because I really am defining it as part of the execution of power set and its implementation is framed. This is the first time we've formally seen this as part of a language. Its implementation is framed in terms of not only this intervariable, but the value of a local variable in the outer scope. Okay? Does that make sense? Okay? Question in the back? Just pencil? Okay.

So I'm mapping. This is a function over whatever the recursively generated power set is. Okay? This accommodates the issue, or deals with the issue, that the one or the car is excluded from the recursively generated power set that says you know what? I'm gonna visit every single subset. If I actually apply some functionality to every single thing in the recursively generated power set every single one of those things is one of a legitimate subsets of the original. The map function visits each subset in the recursively generated power set and it cons's an element onto the front of it.

So it transforms each subset into another subset that has, as opposed to excludes, the car. Okay? Does that make sense to people? Okay. This is all of it. Okay? You can look at that and it's very easy for you to convince yourself that it works because it has all the right pieces and because the lecturer's writing it and you just usually trust him to write the correct code on the board, but it is very difficult to write the very first time you do it from scratch. Okay? It's just very difficult to basically figure out how to invent the lambda and how to get the append call as opposed to cons call to do the right thing for you.

So even though that looks compact and easy and articulates the algorithm pretty nicely it's very difficult to come up with from scratch. So make sure you read through the first page and a half of Handout 32. I went to great pains to try and make sure that you understood in this literary way how we were constructing the power set, so that you can revisit the descriptions in case you ever have to rewrite something like that from scratch. Okay? There is an issue with this. It's not a huge issue from an algorithmic standpoint. It is functionally accurate and functionally correct. It unnecessarily takes an exponential amount of time. Okay?

I'm sorry. I shouldn't say it that way. It takes an exponential amount of time anyway. It actually makes the same recursive call twice. Okay? And so this is the best example I have to introduce this construct that was introduced in an earlier handout, but I just haven't talked about yet. But that is a call right there and that is a call right there. They're exactly the same things. Set is not being updated or being altered by reference in a way you could with ampersands and asterisks from C and C++. All the lists in scheme are these immutable things and given the constructs that we've learned you don't actually modify existing lists. You synthesize new lists out of old ones. Okay?

That's what cons is doing and that's what car and cdr are doing. Okay? What I want to do is I want to figure out whether or not it's possible to just call this thing once and use whatever it evaluates to into two different settings. And the answer is yes or else I wouldn't be talking about it. Okay? There is this construct in scheme, right? It's called a let binding. It is actually very similar to a lambda and, in fact, I'll explain what a let really is. It's really just intactive sugar for calling an inner function. Okay? This is functionally correct. It runs very, very slowly for any set with more than, like, say six or seven elements. This is more intelligent.

Define power set of set. The base case is precisely the same. Null set. Go ahead and return that right there. Otherwise, what I want to do is I want to execute some block of code. Before I execute the block of code I want to evaluate an argument. Okay? You use what's called a let binding. It's basically saying please let this variable equal to whatever this evident expression evaluates to. I only have one let binding here. I'm going to bind PS rest to whatever I get by calling PS against the cdr of a set. That is the cdr. That ends the P set. That ends the pairing. That ends the full let statement. So this right here is balanced by that. This right here is balanced by that right there.

Formally, this is everything between this parentheses and that parentheses is supposed to be a list of pairs. I only happen to have one pair that's necessary for this, but if I had several variables I wanted to initialize and use in what I'm about to write then I could just provide a list of them. There's actually a clear boilerplate in one of your earlier handouts as to how to script that out. Okay? But what I'm basically asking is I'm asking scheme to let me associate this as a variable name. Much like that and that are. Okay? I'm sorry. Much like set is. And just associating it as a single symbol with whatever this evaluates to. Make sense?

It's more than just being clever and only calling this thing once. It actually saves a huge amount of time because now what can happen is I can append PS rest. Functionally exactly the same thing as that right there to whatever I get by mapping the lambda over a subset. The lambda doesn't change. Cons, car, set, onto the front of the subset. That ends the cons. That ends the lambda. And I can map that over PS rest. That ends the map. That ends the append. The let is the define. Okay? Does that make sense to people?

This right here is the let finding. This parentheses doesn't close the let. It closes that one right there. Everything right here up through the paren that, this one right here, is under the jurisdiction of the let statement. So not only does it have set as a variable, but it has this thing called PS arrow rest as well. Okay? That's been associated with the recursive call. I only have to make the recursive call once. I can remember the answer. Okay? I can remember the answer in a code block right here and it saves on running time considerably. Okay? If you have basically the structure of a let statement, basically I'll just make some things up. Like X expression one, Y expression two, Z expression three. This is just me inventing a syntax for making it clear what the let's supposed to look like.

This right there closes that right there. This is either one, usually one, but technically could be a series of expressions that are evaluated under the jurisdiction of the let, which means that you can refer to X, Y, and Z and whatever they've been associated with. So this is just some functionality of X, Y, and Z and whatever other variables were available to you. The point I want to make is that you may look at this and say, okay, the first thing it does is it evaluates the expression one and it associates it with X and then next in this very sequential way it evaluates this and buys it to Y and evaluates this and buys it to Z and then carries on to this.

Scheme doesn’t actually pledge to do it that way. For reasons that will become clear in a second, you cannot assume anything about the order in which those expressions are evaluated and the order in which the variables are bound. You may say, well, does it really matter? And the answer is yes because you may think because of the way you write this that you can refer to X in this expression right here and you can refer to X and Y in that expression right there and you cannot. You have to think about these three things or those N things as being evaluated either in parallel or in whatever order you want. Okay? Does that make sense to people?

The scheme interpreter is free to do whatever it wants to. If you really do want to stipulate that they be executed and the variables be bound in the order that you prescribe them, you have to use a variation on let called let*. The asterisk is this very abusive symbol, which like stares at you and says you’re using a piece of a language I don’t want you to use. Okay? But let* does impose a sequential ordering in the way that things are evaluated. Okay? To the extent that you use let* you’re shifting paradigms. If you go from this purely functional thing where everything is the composition of some simpler function. Okay? Then you’re being purely functional about it.

When you do this, all you’re really doing is C programming where you set X equal to an expression and Y equal to some expression that involves X. Okay? Does that make sense? So this with the let* right there, which I don’t think you have any reason to use in assignment six or seven for that matter, but I’m just talking about it for completeness. There is no compelling reason to use that unless it just doesn’t make sense for you to make repeated function calls and you really do depend on the order in which things are evaluated. I will erase that right there. As far as let is concerned I actually don’t know how it works behind the scenes, but I can give you an idea as to how it more or less works. What theory guides the let construct.

You’ve probably all read a little bit about let in the handout. Maybe you haven’t because assignment six was due two nights ago and scheme isn’t due for another week, so you may just be taking a vacation. But let me just explain what let is more or less equivalent to. When you do this let, and I’ll just write it this way, X something, Y something, and then you write some block of code. A of X and Y. Do you understand that in some ways it’s almost like a function call? It’s like you’re evaluating this argument right here and associate it with a variable called X. You’re evaluating this and associating it with a variable called Y. You’re executing this as if it’s the body of a function. Okay?

This is a more sophisticated clever use of lambda, but I can totally explain what this is more or less compiled to or translated to after it’s been read by the scheme interpreter. I’ve just opened a paren. The thing that normally comes after an open paren is the name of some function. Okay? What I’m gonna do is I’m just gonna put lambda – you’re used to putting symbols there that evaluate the lambda’s. Okay? They’re bound to cope, but you can put the explicit code at the front if you want to. I’m gonna put lambda of X and

Y that calls A of X and Y as part of its implementation. This is the entire car of that list. Okay?

And its body of that lambda function is more or less synonymous with the body of the let statement. Does that make sense? Right here is the first argument to this thing. This thing expects two arguments. I can just put whatever expression one is and expression two is and then call it a day. So what the let thing really ends up being is just a rearrangement or a different way of expressing the application of some anonymous function to the expressions that you're evaluating at the top. Okay? Do you understand what I mean when I say that?

If you look at the let it makes it look like the evaluation of those arguments is happening first and then some code block is executed under the context of those variable assignments. That's exactly what happens with function evaluation, too. You evaluate the parameters. Okay? In whatever order that they want to and scheme doesn't dictate what order the parameters will be evaluated in. So this gets evaluated, this gets evaluated. You certainly do not want to have the result of this thing right here influence the evaluation of that or vice versa, right?

That's a product of the rule that you can't rely on the order in which these are evaluated. Okay? It's unusual for you to see this at the front. You can do it. It's syntactically correct and valid to do it that way. Okay? But this is functionally identical to this and any limitations that I'm imposing just by rule of this right here about what order things are evaluated in can also be said about this. Does that make sense? Now, except for I have a couple other scheme functions I want to write, but except for the fact that there are a library of scheme functions. Like conceptually you know, like, 60 percent of the language all ready.

Scheme is famous for a variety of reasons, but the one I'm thinking about right now is that syntactically the language is very, very easy to learn very, very quickly. Now, it helps that you all know Java and C and C++ very well now. So you can always equate something that looks confusing with some equivalent or near equivalent in C or C++ or Java and so it's easier to learn your fourth language than it is to learn your first one. But there are many curriculum's that until recently, and even today, have decided that they'd rather teach scheme in their introductory classes. For 20, 22 years MIT taught scheme in its introductory computer science class. Cornell still does as far as I know. So does Berkley. They all use the same model. MIT has just recently migrated away from it.

They're in transition right now to teach Python as the first language. Their argument is that there's no exposed dynamic memory allocation. There's no freeing the leading asterisks, ampersands, all of that arrow nonsense. There are classes in some extensions of scheme. There are structs as well. We're not using structs; we're using lists for everything. It's like, oh, you destruct while you use the list and make sure you know that the zero element is always the name and the first element is always the GPA or whatever. Okay? And their argument is that the language is very economical. It's terse, it's expressive. Expressive is good PR term for dense. Okay?

It forces people to think about their abstractions and about their algorithms a lot more quickly than they do in C or C++ where they have to be worried about variable declarations and memory allocation and things like that. Okay? Does that make sense to people? That's the first time I've explained the let thing because I actually never knew what let was equivalent to until last quarter when somebody told me about it. I'm like, ah, I can probably talk about that in 107. Okay? So this explains the theory behind let. That's a legitimate example of where you'd want to use let. Okay? Basically you evaluate that thing right there and that big append is like a lambda that takes one argument. Okay?

Its argument is whatever power set of cdr of set evaluates to. Okay? Just use this as the analog for that. Okay? Now I have a more difficult mapping of a lambda problem. This is a single mapping. Okay? We map lambda over this and it's confusing in its own right, but as far as mappings go it's a standard mapping where the function you're mapping over happens to be a little more complicated than it would normally be. I have this great example. It's another common [inaudible] number theory thing like you would see in 106, 103b or 103x. I want to write a function called permute. I have to assume that 90 percent of you have written this in C or C++ because you all went through 106b or 106x here, but I want this as a function to output all, in this case, six permutations of those numbers.

I'm gonna assume that all of the lists are simple lists. There's no nested list inside the list and that there's no duplicates and I'm just gonna assume its order from one through N. Okay? And not worry about duplicates or anything like that. It'll still work even if I don't do that, but I just want to deal with 1, 2, and 3 or 1 through N. This is supposed to output this. 3, 1, 2. That's 3, 2, 1. Okay? I'm gonna have to do a larger example of this. Just a sample function call in a second. The way I wrote this it's technically illustrating the structure that I want to exploit in my algorithm.

These right here are all the permutations that begin with the number one. These right here are all the permutations that begin with the number two. These are all the ones that begin with the number three. Okay? Without going into the scheme code, if I'm interested in the permutations of 1, 2, 3, 4 I'll just write the ones that have one at the front. 1, 2, 3, 4. 1, 2, 4, 3. 1, 3, 2, 4. 1, 3, 4, 2. 1, 4, 2, 3. 1, 4, 3, 2. Then they're all those. Draw a little box around this. Then there are all those that begin with the two. Then there are all those that begin with the three. Then there are all those that begin with the four. Okay?

There are 24 permutations here. It's not 24 so much as it is four factorial. These right here, like little Russian dolls of permutations. That happens to be, what I've just boxed with the inner rectangle are, all the permutations of the cdr. Okay? However, I'm not gonna draw them in here, but what's in there isn't the permutations of the cdr. It's the permutation of whatever you get by removing this element from the list. These are the permutations of 1, 3, 4. These are all the permutations of 1, 2, 4 and 1, 2, 3. So when I say that this is a permutation of the cdr it's not the best way to say it. It's the permutations of the original list with the one removed. Does that make sense? Okay.

Now, I'm just gonna assume that the remove function is a built in. It actually is, but it's not called remove. I actually wrote it in the handout as if it really is just a built in. I'm sorry. Not a built in. That it is in a built in and I had to write it myself. You may think that mapping has nothing to do with this and it would be a completely reasonable thing to say because we're taking a list of length three and transforming it into a list of length three factorial. We're taking a list of length four and transforming it into a list of four factorial. That make sense? But what I'm gonna try and do and I think I can do it because I've written the code now 17 times.

But you can actually do this because you can use mapping in a way that you're not used to, but I can actually make the one responsible for somehow transforming itself into all of the permutations that begin with the one. Okay? With two transforming itself via mapping, clever mapping, but mapping, into the list of all those things that begin with a two. Does that make sense? So the way I'm gonna draw this is I'm going to rely on mapping to take something like 1, 2, 3 and transform it into the list. I'll call it one perms. I'll call this two perms. And I'll call this three perms.

When I talk about three perms I'm talking about all the permutations of this list right here that begin with a three. Okay? And I want this to expand to that. Whatever mapping function is applied to this list right here has to transform this and this and three into those things right there. Now, I don't want the permutations to be subgrouped with extra parentheses based on what element they begin with. The ultimate answer is I don't want these parentheses. I want all of the things, whether they're one permutation or two permutations or three permutations, to be here. Okay? Make sense?

What I want to do is I want to recognize based on what I've said right here that the overall algorithm for permutations without worrying about base case yet. Might as well just erase with the chalk. I want to take define and I'll call it permute and when I write this thing right here I'll just call it sequence. I really am thinking and framing it in terms of sequence for sequences like 1, 2, 3 or 1, 2, 3, 4. Okay? I basically want to map some function to be determined and by drawing it the way I do you should just know that it's going to be a lambda. Okay? And I want to map that over sequence. In fact, I'm gonna change the word because now I remember what word I used in the handout. Items.

We have no idea what this is, but if we know that something can be put there and we can make it work the thing that makes it work is gonna take this and transform it into that. We can't have those intervening parentheses, but what we could do is we could – I love this, you can't do this in many languages very easily. Okay? I'm given M lists. I want to take those M lists and I want to take an append and I want to put it onto the front and then evaluate it as if the append were at the beginning of them all along. That flattens it to one degree. Make sense? Okay.

Now, unfortunately, this is not easy right here. Okay? We have a mapping function. You know what's going on here, but we have to somehow take a single element out and I'll admit right now that it's gonna be some lambda. I'll say that it's either gonna be a one or a two or a three. I have to somehow take a lam, which is a one and transform it into a list of all the permutations that have one at the front. Does that make sense? Now, this is a run on sentence I'm about to get to, but let me try it. When this thing is one, I have to have the function that's right here. Somehow transform the one into all the permutations with one at the front. I get that by mapping another lambda over all the permutations of the set where one has been removed.

What is the mapping function? It's just like it is for the power set where I cons whatever this element is onto the front of all of the permutations that happen to exclude this element. Okay? That make sense what's going on? Okay. Now, I'm concerned that I don't have enough space. I'm just gonna remember that item has to be drawn at some point. This is the lambda and I'll get to the items at the end. I have items outstanding and I have a base case to worry about right here. What do I do on behalf of each element?

I actually want to map another name. I'm not gonna commit to a name yet. Whatever I get by calling permute of remove of items LM. That ends the remove. That ends the permute. That ends the map. That ends the entire lambda. Okay? There's so many pieces here and I know it's confusing, but I think you're gonna be able to get it. Okay? This right here is just some function that figures out how to get a one into all of the permutations where one is at the front. Okay? In order to do that I have to recursively generate all of the permutations that you get by excluding the one from item. There's a leap of faith argument right here. Okay? Does that make sense?

What function gets applied over these things? Well, these things right here are actually permutations, so I'll call that argument permutation because I'm applying some function to each permutation in the permutation set of the set that excludes one. Okay? What do I put here? I cons a lam onto the front of the permutation. That ends that. That ends the lambda. This was the argument of the map call. That ends the map. That ends the map. I don't want this one right here. We'll just be clear about which ones really need to be written in at the moment. This ends the remove call. This ends the recursive permute call. This ends the inner map. This ends this lambda right here. Okay?

This map has to map over something. It has to map over the thing that actually gives me isolated elements. This is where items go. And that ends the apply call. Okay? Make sense? Now, the one thing I'm not doing here is that this thing is infinitely recursive at the moment. It's gonna eventually try to take the clear of the empty set unless I block it. Up front all I'm gonna do is if I have an end to items list with the same reason as before I'm gonna return that. I'm gonna assume that the empty list has the empty permutation as its only permutation. Zero factorial is one. Okay? So that's why I have a list of one as opposed to a list of zero there. Okay? Does that make sense to people? Okay.

To look at that it could be just, like, morally offended by how dense that recursion is. If you have a double mapping with a double lambda it's really gonna force you to stretch to figure out how this is accomplishing the task and where the actual permutations are being constructed. This cons, this is really the only dynamic memory allocation function that we really talk about. This, an append, are the things that build larger lists out of smaller ones. Okay? Behind the scenes any time cos is invoked that's really a request to build some linked list cell. Okay? That didn't otherwise exist. And it populates the two fields of the linked list cell with that and that. We'll see next time that's exactly how it works from a memory standpoint.

But it's the accumulation of all of these cos calls and this apply append call where it actually extends permutations and merges lists permutations as the recursion on lines. Okay? And it bottoms out when it does this right here. Okay? Make sense? Okay. Very good. If you're getting this then you are – this is the hardest part of scheme. It has very little to do with the language. You understand lambdas. You understand mapping. It's actually combining them with their expressiveness/density to actually get algorithms like this to run really cleanly. You can do this in C. You can do this in C++. You manually manage the swaps. You've written permute, I'm sure, on strings in C++ in 106b or 106x. Okay?

The string class shields you from some of the memory allocation, but schemes win even over C++ and C is that it obscures all of the memory details from you entirely because the list is this central data structure in a way that it's not in C or C++. Okay? So you have built in support for list dissection and extension. Okay? You guys doing okay? Questions? Okay. Also explained in the handout. Okay? This was an old exam question. So were the things that were on the section handout yesterday. This was intended to be the hard scheme question when I gave it. Okay? This was like nine years ago. I'm running out of scheme questions, so you may see it again.

This is really the densest thing to do and it's hard to look at, but scheme does it much more expressively and better, I think, than C and C++ would. What I want to spend today on and at least half of Friday, if not all of Friday, is talking about the memory model that's involved in implementing scheme. I've made a couple of points, but these are takeaway points that are good to remember even if you don't continue programming. Scheme functional paradigm. You don't think in terms of variable assignments or outline form. You don't think about objects. You think about the data, but not really. You just think about functional languages like scheme and ML's another example of one.

There's actually a language at the moment that's all the rage called a Haskell. I might have a coworker of mine come in and talk about Haskell during deadline week. It's about data transformation in this very functional algebraic way, but it's algebra that's extended to include lists and strings and Booleans. Okay? When you program in scheme you program without side effects, or at least you intend to. Certainly you do in the subset of scheme that I've taught you. That means that you rarely rely on your ability to update a local variable and somehow expect that to be reflected in the argument that was passed from the color. Okay?

You can do it. There's an example in the handout. The partition function that I wrote to help implement all of quick sort uses some sort of clever way of actually programming with side effect, but not really. I mean, it really does synthesize the partitioning of an array around a particular number the way you have to in quick sort. All of the lists in scheme, except for a very small subset of operations that I do not teach you, so you pretend that they're not in the language. All of the lists are immutable. That kind of coincides and makes [inaudible] without side effect very easy because if you can't change the list and you can't change the atoms it's very difficult to program by side effect because you can never change a list in place. You always have to synthesize a new one out of old ones.

Those of you who have programmed in Java before, their strings are considered to be immutable. Like you can't actually go into an existing Java string and change an E to an A. You have to generate a new string out of it. Does that make sense to people? C++ and C you know very well how you can change memory behind the back of another variable. Java makes that very difficult, at least with strings, not with objects in general, but with strings. Scheme makes it difficult in general based on the subset that I've taught you. Okay? Make sense? Yeah?

Student: Did you saw it was or wasn't a built in?

Instructor (Jerry Cain):When I wrote it in the handout I thought it was not a built in, it is a built in. I think it's called remove. I forget what it's called because I wrote it from scratch so it's remove in my head because I just use that code.

Student: How does it work if there's [inaudible]?

Instructor (Jerry Cain):Yeah. I actually was specifically saying I'm only dealing with – the remove that's built in does remove all the duplicates. The one I wrote. And I think this is why I ended up – actually, now I'm remembering I think this is why I wrote it. It is my version just removes the first element. The first version of it, but I'm actually prescribing as part of the problem that I don't have duplicates in the incoming list. Okay? Okay.

So let me give you a little bit. We all kind of gravitate. I know you were completely humiliated by assignments one and two in terms of their difficulty and how much they were exposed raw memory, but now I think we'll all kind of gravitate toward how things work behind the scenes and under the hood. I could give you a very simple set of drawings to give you an idea as to what these lists look like in memory. I did a little bit last Friday, I think, but I'll give you some more. When you type in – I'm being very informal here. I'm not actually drawing out the pictures that are really relevant to coa. I'm just drawing out pictures that are accurate enough so that they're a complete enough design, so that you can know that these things can work.

When you type in a four the scheme interpreter is prepared to build a variable in memory that stores that four. What basically happens is it recognizes as it parses it that it's a pure integer. It has a four up top. It actually returns this right here. So it returns a pointer to the data structure that is self-typed and self-identified as an integer and it levies some type of print operation against this thing as part of the read, evaluate, print result loop. Does that make sense to people? Okay.

So the reason this prints out a four is because the thing that is returned by evaluation is expressed as this pointer and so it goes to this thing and says, oh, it's a four, so that's what it's gonna print out. I'm gonna print it out according to what this data type is. When you type in hello it returns a pointer to something that's tagged as text or a string or whatever and how it does this – hello, rather. Rather the actual details are up to the interpreter, but as long as they return this to the read, develop, print loop and print knows how to deal with pointers to these types of cells that are self-type identifying it knows how to interpret the rest of this entire thing. Does that make sense? Okay. Do you have a question right there?

Student: Up there is some kind of encoded the type of data so first strings or schemes we have some kind of –

Instructor (Jerry Cain):All of the data types in scheme are dynamically typed. Java does the same thing actually. The data actually carries piggyback and not information about its own data type. I don't know. You haven't dealt with this part of Java so much. I don't know whether you're into Java or not specifically. I'm assuming that you know a little bit. There's actually a method that you can invoke against all Java objects called get Class. And it returns a class object. So all of a sudden it's getting very meadow on you because you're like, okay, class is sort of things we code up as templates and we actually construct objects in the image of these class definitions.

But programmatically you can actually have a class that models the notion of a general class. Okay? That's what Java does. Java actually has this data structure and every single object has a hook back to the class class that describes the class that the algorithm is a type of. Okay? Scheme isn't quite that sophisticated as far as I know. It may be now, but what I'm talking about right now, and I don't think coa is any more sophisticated than this, is that it actually tags every single data element with information about what the thing really should be interpreted as. Okay? So this obviously prints out hello.

When you type in – and this is the part I'll leave you with right now. I'll write some more code examples on Friday, but I only have like five minutes to talk about this, so I'm just gonna give you some pictures to mull over. When you type this in not surprisingly it returns this. That means that the thing is returned is part of the read, develop, print loop has to be the address of the leading nodule list and the list has to know it's a list, so it knows how to print itself out.

This is constructed on your behalf. I'm just drawing a linked list. I'm gonna draw this a little bit differently. It's not required to do it this way, but this is just the way I like to do it. This is associated with a one, this is associated with a two, and this is associated with a three. Okay? All I did there was draw a linked list. Thing is that's interesting is that when scheme digests this right here it knows how to programmatically – like, it's almost, like, it's reading a data file where you've happened to express your sequence of numbers using scheme like syntax and that behind the scenes every time it hits an open paren it knows it's gonna be building a list for you behind the scenes.

This type of drawing is consistent with the way I drew this four up here. These things right here, they're intended to be the node to the linked list. They're actually called cons cells. Okay? And it's not a coincidence that we have a cons function as well. This right here is understood to be the car field. This is understood to be the cdr field. Okay? When you levy a car against this list right here it actually constructs this thing right here and the car operation is just instruction to go in and return that right there. Okay? And then print it out. It's looking at a one that knows it's an end so print it as a one. When you get that right there, because you've levied a cdr against the list 1, 2, 3, it gets back the list to three. It's self-typed as a list, so it knows how to print itself out and it involves parentheses and it involves the two and the three. Okay? Does that make sense?

Stand-alone just returns this. Each of those cons cells is actually tagged with like the cons word, just like texting into there. Okay? Make sense? What this really is is this. It's functionally equivalent to the following. When I type this in it's really just as if you did this. Cons one onto the front of cons two. What you get by consing three. You prefer this for obvious reasons. But this can just be taken as a syntactic sugar for this right here. Okay? You can think about this being framed in terms of lots of elements and the base list, which is right here. Okay?

Cons as a scheme symbol is attached with code. It's native to the interpreter that knows how to basically mallic or operator new these thing right here. Does that make sense? And after it does that it has to figure out what to put there and what to put there. Oh, it just puts that in the car field and that in the cdr field. Does that make sense? Okay. And this entire thing evaluates the same thing that that does. Both of these, no matter how I type it in, comes back with 1, 2, 3. Okay?

So obviously there's a lot of implementation detail that's being left out, but if I basically charged you with the task as a final project, so just go and write a very miniature scheme interpreter. Okay? Then you could do it based on what you know. You could write it in Java or C++ or C if you want to use the vector. I wouldn't recommend it, but you could. You have all the rudimentary understanding of how the memory model of backing these things actually works. You don't know how function call and function evaluation works. How the code that's attached to a symbol is used to instruct how to crawl over all the remaining arguments and synthesize an answer. That's difficult. That's why every time I think about the beginning of the quarter I think about actually giving a final assignment where you write a scheme interpreter in scheme or you write a scheme interpreter in Python or something like that and I always revisit the function evaluation part and I'm like, ah, that's too much work.

But it's very easy to understand that it's possible to do it. Everything is framed in terms of a list behind the scenes. Okay? I have more to talk about memory management next time. In particular, I want to talk about how things are freed and how garbage collection, much like Java garbage collection, except it uses a slightly different model, how garbage collection works and I also want to talk about the equivalent of a dot, dot, dot from C and C++. I'll write some more code for that and we'll implement the generic map car of the generic map function. Okay? Have a good night.

[End of Audio]

Duration: 53 minutes