ProgrammingParadigms-Lecture23

Instructor (Jerry Cain):Hey, everyone, welcome. I don't have any handouts for you today. A couple of clarifications. Apparently, I goofed and I made the assignment due on some nonexistent day next week? It is due Wednesday evening, which is the 28th, not the 29th. That's ample amount of time to actually get the assignment done so I'm not worried that you're all going to be all of a sudden flustered, but it was always intended to be due Wednesday night. I think I even said Wednesday night in class; I'm sure I did.

But otherwise, that is due Wednesday evening. You will have one more assignment going out on Wednesday. It'll be just a single problem. And the language we'll probably start next Wednesday, if not in the last ten minutes of today, next Wednesday, where I start talking about Python.

Once I get to the Scheme segment, we're going to transition and lighten a little bit and we're not going to be so concerned about trying to master all these new features of language, because I don't want to say we've covered all of paradigms, that's not true. But we've covered a good number of paradigms, and when I teach Python next, I'm going to really just highlight individual things in the language that will get you up and running very quickly and point out where it's imperative and where it's object-oriented and where it's functional, okay?

So that's what we'll do throughout the rest of the quarter. I'll probably introduce you to a few of the languages that I happen to know, just so you can see them and really just trust that just because you don't know them that it's actually pretty easy to learn them.

Today I want to focus on Scheme. I probably will not keep you the entire time. I have two things I want to do and then I will probably just let you go and go on this virtual five-day weekend. Okay?

When I left you last time I was spending a lot of energy talking about the memory model, and I just want to review that for five minutes so that you understand exactly how you could very easily implement a Scheme interpreter.

When you type this in, this is really an instruction to just go ahead and synthesize a link list behind the scenes and equate it with the address of the leading note.

So I drew it very carefully last time, but this is going to be returned in response to typing that in and hitting enter, and then some print functionality is levied against this address behind the scenes, and it arrives at something loosely drawn, like this right here. And I drew it as if it were a nil object.

And I did this a little bit differently last time, but conceptually, that's what's there. And the print function can actually figure out, either because it's overloaded by strong type or it could actually look at this thing and determine that it's a list node, it can use that to figure out whether or not it's printing a standalone number or a standalone string, or an open param before it actually prints the leading list and things like that, okay?

When you do something like this, I've downplayed this. Let's just call it SEQ. Then very much of the same thing happens. Something like this would be constructed on behalf of that statement right there, and because the defined statement is one of the few things we've learned – in fact, it's really the only thing we've learned that has a side effect, it would associate in some global symbol table this thing called SEQ, and it would be associated and just put as an entry in some kind of global map.

So that any time you reference SEQ and they note that it's not a local variable, it looks in this global map to figure out what data is associated with it, okay? Does that sit well with everybody? Okay.

So let me just make it clear how lists are shared, and then because we don't allow lists to be updated, the aliasing problem is never dangerous like it is in C. If I go ahead and I ask for the car of SEQ, well, it evaluates to this thing right there, and the car is just basically a request to return whatever that is and it levies a print statement against that arrow or the base of that arrow, it detects that it's an integer type, it has nothing to do with lists, so it just prints a standalone one.

When I do this [inaudible] of sequence, it's just a request to go and grab that, okay, which itself is a list just like the leading list is; it just happens to have fewer nodes involved. So that's why it prints out two, three, okay?

The cons instructions are more interesting. If I do something like this, cons, and I'll do this – one, two, three, and I'll cons it onto the front of the list four, five, six, I will be exhaustive in the drawing here, what happens is this is synthesized, points to a nil – that's how I draw it, with a four and a five and a six, okay?

This is also synthesized. Points to another nil, okay? And then has this as a structure, and the leading arrows of these are just the product of the evaluation and the construction of those two lists right there, and then the con statement is actually just a request to build a new one of these to put the address of the first thing there, whether it's a one or a list.

Put the address of the second thing right there. And so when the address of this is produced by the evaluation of the entire statement, it does have the side effect of constructing this node right here, okay? It prints out this for pretty obvious reasons. It prints out the car, which happens to be a list, so that’s why it prints that way. And then it prints out the [inaudible] in the form of a list like that, okay? Does that make sense to people? Okay, question?

Student: How do you know that the [inaudible] points to a list and not to, like, like –

Instructor (Jerry Cain):Well, the [inaudible] is supposed to always, at least in our subset of Scheme, it always points to a list. Now that actually isn’t true in real Scheme.

You can actually construct what are called dotted pairs. There's no reason to do this. I'm only telling you because you asked. But if you wanted to do something like this, that would break the list paradigm that I've been pushing for the last three days. Turns out that that is legal, okay?

This would produce what's called a dotted pair, and the dot is in there to emphasize that the five isn't part of a list, that it is a standalone element. It's actually an integer that's in the [inaudible] field, okay? Without this, it means that the [inaudible] has a list, which points to five.

I'm not saying that this isn't important; it's just that I'm only talking about Scheme for five lectures, not 12. Some of the data structures that do exist in Scheme that I'm not talking about, they have the notion of associative maps. They actually store everything, all of their pairs behind the scenes, as dotted pairs.

And if this is incidentally, the values are lists, then the actual serialization and printing of it would look like a normal list. But if you mapped integers to integers, then you'd have all of these types of things being stored by the map, okay? Does that make sense?

Scheme actually has vectors, which are really the equivalent of arrays, and it actually also has these things called maps, okay, which [inaudible] a little bit of hatching behind the scenes, okay? Make sense? Okay, awesome.

So let me just emphasize a few things. I want to construct this list two different ways. Okay, I'm going to do one the obvious way. Right here, cons, one, two, onto the front of one, two, and this is an example, it's actually pretty much coincident with whatever's happening up there. But I want to draw the memory diagram out just to emphasize how the memory diagram changes when I write this a second way.

I would have this point to this right here, which points to – mm, oops. Yeah, that's right. Would point to a one, would point to a two, and this would point to an independent list with a one and a two. Sorry, I didn't mean to smush that, but it just kind of happened.

Okay? But you get that there actually are four-linked list – I'm sorry, there's five linked list nodes combined, okay?

You understand that. This right here, this is trickier. Here's the one, here's the two, and there's the nil. If I serialize and print out what's accessible from the tail of this arrow, I certainly print that out right there, okay? The printout rhythm is so blind to memory management that it doesn't actually care that there's some internal aliasing going on, okay?

The instruction for printing this out as the car and putting that out as the [inaudible], it's exactly the same. And the fact that they happen to reach the same nodes in memory is kind of just incidental. It would still publish this way. Does that make sense?

Now the reason I get this and this as separate is because I actually put down the two list constants, okay, like that. If I wanted to actually construct this, I'd have to be a little bit more clever. I'm actually going to go with the more clever of the two ways I'm thinking about it. What you could do is make sure you only construct one list from a list constant. That's going to create just this right here.

And then do this – lambda of X cons X on X, and make sure the argument is a list so it can be the second argument of the cons that's inside that lambda, okay? So I construct one list and I actually bind it to one variable that's used in two different places inside the definition, okay?

So this really is an instruction. Don't think about it from a data standalone. Understand that X and X are both associated with references to structures, and it's saying to populate the car and the [inaudible] field of some new con cell with exactly the same tail, okay? That's how you could do this, okay? Make sense? Okay, good.

So what I want to do, before I talk about garbage collection, which will be kind of high level and just a lot of drawing – it's not really difficult, but it's actually pretty interesting to understand, I wanna go over what somebody reference in a question on Monday, the equivalent of the dot, dot, dot, okay?

There's not much reason to learn the dot, dot, dot in a course that teaches Scheme over a course of five lectures, except for the fact that it is a paradigm class and we're trying to compare and contrast features of languages. It also allows me to implement the full map operation, okay?

We've been pretending, from an implementation standalone, that map is always unary map, and I wrote that last time, I'll write it again in a second. I actually wanna write generic map. That takes some function object and either one or two or three or four lists, depending on how many arguments the actual function object can deal with.

So we're pretending – we've been pretending that something like this – car, one, two, three, four, five, six, seven, was the only type of way you could use map. Using it as a unary function – car is certainly a unary function – and this would produce one, three – oops – five, and all the [inaudible] would be ignored. But again, just consistent with the idea of a map, it transforms one list of length three into another list of length three.

If I do this, I want it to just deal. That's right. I type that in, I want it to generate 111 and 422, okay? To emphasize the fact that this doesn’t have to choose whether or not it's being used as a three-argument function or a four-argument function, I could do this. I could map times over the list one and the list two and the list three and the list four and the list five.

I'm not saying that this is the way you'd want to realize that [inaudible] but nonetheless, this is supposed to trivially collect all of the cars of the list, seven of the peers of arguments, apply this to that set of arguments, and come back with, as a list, the number 20 – 25, where am I getting that – 120, okay? Does that make sense?

I have a ton of examples on the very last page of I think handout 32 about how map can be used as the generic map, okay? Not a ton, but just enough to illustrate the idea.

When we wrote unary map, I think on Monday, maybe early Wednesday, I really only handled this case, okay? I called it unary map specifically to emphasize that this was covered but these were not, okay?

I want to do two things. I want to reproduce – define, I'm going to call it unary map to remind you that it's unary – function, and then I'll call it list. No, I don't wanna call it that, I lied. Call it sequence. And if it's the case that I have the null sequence, go ahead and return null. Otherwise cons onto the front of some recursive result, whatever you get by applying fn, the car of the sequence, like so, to the recursive result. Unary map of the same function, to the list excluding the [inaudible]. [Inaudible] unary cons if [inaudible]. Okay?

Now you can see from the calls up there that I have to accommodate either one or three or five lists. I am going to force map to deal with at least one list, because it's really kind of silly to support the notion of a map over no list whatsoever, okay? But in order to accommodate a variable number of arguments, you have to do the equivalent of dot, dot, dot, but in Scheme.

As an aside right here, let me just talk about what happens if you define – I'll just call it some dumb name like bar – and I'll give it these arguments – A, B, C. That means that any call to the bar function I'm defining right here has to take three arguments exactly at the moment, unless I do this. And that's fine.

The dot in this context just means that everything that comes after argument three, like arguments four and five and six, if there are that many, that they should all be collected and put into this thing as a list, called D, okay? Does that make sense to people?

So if I do this, if I just define bar to be this, to be the listification of A, B, C, and D, list just basically takes all of these elements and wraps an extra bookend of parentheses around it, okay? So if I call the bar function on one, two, three, four, five, six, the output of this – it's a little weird, but it listifies these four elements – one, two, three, and D is then equal to the list that contains these three elements.

That's how one additional argument can accommodate everything in the dot, dot, dot sense. Does that make sense? Okay.

So the product of this thing right here is [inaudible] would be that right there. Okay, now that's not all that sexy of an example, but it motivates the dot and the way of catching an optional number of arguments for the implementation of generic map, which I'm going to do in a second.

I should emphasize that if you do this – bar one, two, three, you're actually gonna get one, two, three, empty list, okay? And if you try to pass fewer than three arguments to bar, it's gonna choke for the same reasons it would choke if there weren't a dot there, okay? You have to have at least as many arguments as are required.

So the implementation of map I actually think is very, very cool but it's very, very dense and it uses apply and it uses mapping, it uses unary map, and it uses function objects and things like that. It's a great function, and this is going to be the prototype for it.

The reason I wrote unary map is because I actually use the implementation of unary map in the implementation of generic map. Define – I'm just gonna call it map, we'll pretend it's not a built-in. We've just discovered the idea of mapping and Scheme and we're implementing it for everybody.

You can take map, you have to take something that evaluates to a lambda or function object. I'll take something I'll call first list, which is required, and then I will actually have a final argument called other lists, which because of the use of the dot – that's supposed to be all one line – because of the use of the dot, a second and third and fourth and fifth list will themselves be bundled in an additional list. So they actually are carried around by one argument called other lists, okay? Does that make sense?

I'm going to assume for simplicity that all the lists are of equal length. It turns out they real implementation doesn’t need that, it just uses the smallest of all the lengths to determine how long the products would be, and if there are any extra arguments in some lists, they just ignore them.

But I'm just going to – for simplicity, I'm just going to assume that the length of first list is the same as the length of any list inside or all lists inside other lists, okay?

If it is the case that regardless of what the function is – null, first list – that I'm just going to assume that there's no mapping to be done. That for whatever reason I was given, maybe a ternary function or a function that takes ten arguments, but I was handed five or ten empty lists, or three or ten empty lists, okay? In which case the product should just be the nothing list, okay? That make sense?

Otherwise, what has to happen is I have to cons on the result, I have to figure out how to collect – in this case, this number was produced out of those three numbers. That make sense? Okay.

Obviously, I'm going to plus these. These two lists, in the context of this implementation so far, first list would be bound to that, right? And other lists would be bound to this, okay? So there's a little bit of asymmetry in how the data's handled, but that's just what you have to do, okay?

What I wanna do is I want to take the car of this, I wanna cons it onto the front of whatever I get by unary mapping the car function over this thing right here – clever that it does that, okay? Does that make sense to people, though?

I needed somehow to get the ten and the 100 out as peers in order, in a standalone list, so that one can be smushed onto the front of it so that I can apply the fn function, whatever it is, to the list that's been collected for me, okay?

So I wanna cons whatever I get by applying – I'm gonna need – I'm gonna run out of space. Apply fn to the consing of the car of first list onto whatever I get by calling the unary map this – whoops – right there is a helper function right here that knows that the list it's getting is – it's just getting one list. That ends the unary map, that ends the cons, that ends the apply, that ends the cons. Oops, sorry, I don't wanna do that. Oh, not that one, yeah. Sorry. Okay.

So do you have faith as to why this is gonna work? You may ask why I need the apply right there, why don't I just invoke fn like the function it really is? The problem is that fn, look at the plus. Plus doesn’t take a list of integers, it takes a sequence of integers that follows it with no intervening parentheses, okay? Does that make sense? Okay.

So the apply has to basically get the plus or the car or the cons or the multiply onto the front of the list that's assembled by this thing right here. That make sense? Okay.

Then what has to happen is I have to cons this onto the front of what I get by recursively mapping this function over all of the lists where they’ve all been replaced by their [inaudible]. Does that make sense to people?

So I have to somehow get – let me bring – did I erase it? [Inaudible] Even though I'm handed the data this way right here, I have to somehow assemble the data for the recursive call to look like that and that and that, okay? I have to make them all peers again for the recursive call to map to work out. Make sense? Okay.

So the second argument to this cons, I'm actually going to have to indent over here, so just really assume this right here is just exploding over here so I have space to write everything, okay?

I certainly have to unary map [inaudible] over other lists, okay? That at least gives me the list with the 20 singleton and the 400 singleton inside, okay? What I can do then is I can cons the [inaudible] of first list onto the front of that. That at least gets all the [inaudible] to be peers, okay? Does that make sense to people?

So there's that. Then I also want to cons fn onto the front of that. So what I'm basically doing is I'm setting up – I have this asymmetry with the lists, I made the argument, the [inaudible], symmetric, and then I have to get fn onto the front of it, because fn isn’t actually what's being invoked right here, it's map that's being invoked.

So what I can do here is I can apply this map routine that I'm writing to this right here, okay? Does that make sense? So in the context, let me just – I can actually trace to this, I think it'll be pretty good if I do it. Let's concern ourselves with just enough of a trace to convince yourself that this is gonna work, when I do map – and I'll just do – I'll map the list function over one, ten, 100, two, 20, 200, and that – I should do one more – three, 30, 300.

This is supposed to give me three lists back, the list one, two, three, the list ten, 20, 30, and the list 100, 200, 300, okay? Make sense? Okay.

So the initial call actually has to take this one and the – I'm sorry, the initial call actually takes the first piece of data and binds it to the thing called first list. Everything else is bundled in a second list. So those are my two arguments to this function, okay? What this does right here is it takes that and conses it onto the front of this, so that unary map call actually produces a list two, three out of this.

This one is the car. The first innermost cons call actually gets the car onto the front, okay? And then when I apply fn, I get the fn on the front, which is listed in this case. And that's how I get the list one, two, three out of it, okay? Does that make sense? That's the easier of the two to follow.

What has to happen for the recursive call to map is that this has to have [inaudible] mapped over it, like that, okay? That's what the unary map does, the second unary map thing does, okay? I have to cons the [inaudible] of this onto the front, okay, and then I have to actually put the function object, whatever it is – in this case it's a list – onto the front of that so that all of the arguments that are peers at this level are once again peers down here, okay?

And then I can use apply of map, so that the map is effectively put onto the front and it's invoked as if we typed it in that way, okay? This is dense, but I think you're all getting it, okay, because I see nods. Okay, so you're all with me?

Okay, that is more or less the implementation of the real map function. The only thing that I'm not handling is what I alluded to earlier is the variable length lists, but that's not really interesting. That just means that there's more base cases right here, and this is more involved. It has to check whether or not the first list is null or any one of the lists in the others lists is the null list, okay?

So it's just another unary map, basically, to figure out whether or not there are any falses or any things in there that pass the null question mark test, okay? Question?

Student:[Inaudible]

Instructor (Jerry Cain):Oh, a left bracket. Which board?

Student:[Inaudible]

Instructor (Jerry Cain):This one. What did I – a list bracket – you mean just more parentheses right here?

Student: In the middle. After the first [inaudible].

Instructor (Jerry Cain):No, no, no, no. These right here, this list is consed onto the front. So this used to be the list, the product of the last of all the cons calls right there. Does that make sense? And then I consed whatever fn was in this example of list onto the front, okay? And the four arguments that were passed to the initial call, one, two, three, four, are sort of peers again in this list right here, okay?

They've all lost – well, these three have lost their cars. This doesn't – it's not supposed to lose anything because it's supposed to be passed through verbatim, okay? And then when I apply the map, this effectively takes this onto the front and the map says okay, there's the function object and there are three peer lists after it, okay? And it invokes it like the original call was made.

Student:[Inaudible]

Instructor (Jerry Cain):No, no, no, no, no. This right here, because I consed the [inaudible] of the first list onto the front of it, it made – took the asymmetry where this was in a list of lists and this was just a standalone list, and it put this list onto the front of this one, so they're all peers. So there shouldn't be any extra parentheses in front of the 20, okay?

There was, but then I pulled that parentheses in like it's on a spring and I smushed in the list ten, 100, okay? Does that make sense? That's kind of what I think about cons doing.

Okay, you all happy? Now, the one thing you're hypersensitive to from assignment one through assignment six but not at all sensitive to in assignment seven is dotted memory allocation and deallocation. The only places where you allocate memory in Scheme, it's not on your behalf. It's when you type in list constants and it builds lists behind the scenes.

Or technically, you can say that you're actually dynamically allocating memory when you use the cons function, because you know that that actually creates a cons cell and populates its two fields with two pieces of data, okay? Does that make sense to everybody?

What about deallocation, right? That's actually more involved. Well, very often when you just type in expressions – when you do something like this, when you do let's say cons, hello, onto the front of the list there, you don't actually store the result of that list anywhere in memory. You just for whatever reason had decided that this is how you want to construct the list hello there just long enough for it to be printed so that you can see them as peer words in order so that it reads "hello there," okay?

Behind the scenes, this is constructed on your behalf. The hello and the there are a little bit more involved, but conceptually you can just assume that what they address are Scheme strings. You're returned this right here, okay? And then after it uses the tail of that arrow to figure out how to print the list hello, space, there, it actually, in principle, could deallocate the list for you right then and there, okay?

It can recognize from the statement as simple as this one that there are no side effects whatsoever. Like the result of this list isn't being attached to some variable via define. It's not being passed to another function. It's just this little hiccup of a statement right there that actually prints something, and then after that happens, the memory can be reclaimed. Okay, we're all in agreement?

Well, actually, reclaiming it, that's algorithmically difficult when you learn about length lists, but eventually you get very good at it and you know exactly how to do this breadth-first reversal of the list and free all the nodes along the way, okay? I mean, if you wanna think about it, you could do this – you could just say to free anything is to just free that, free that recursively, and then free the cell. It's as simple as that, okay?

It's hard for you to do that in 106b when you first learn it, but eventually you get very good at these things and you just kind of do it, and once it's done once, you just forget about it because it really basically covers the deallocation of everything.

There are situations, even in the Scheme that we're learning, where the length lists that are built up actually do have to persist for a little while, but it also wants to be able to detect later on whether or not a list has been orphaned so that it can actually free it once it does detect that it's been orphaned.

So let me just forget about functions for a minute. They're their own beast. Let me just use define and just define some global variables, which comes up more often than Scheme – no, I don't wanna say more often, but certainly comes up and we use a few globals in where am I, but not very many.

We use globals but we define constants, which is what you're supposed to do when you're dealing with globals. But just imagine the scenario – and I'm only interested not in the elegance of the coding solution but how memory management works – if I do this, then you know that there's a link list that's formed in memory and the tail of the leading arrow is bound to and associated with the symbol called X, and it's stored, permanently or semi-permanently, in a map.

And I say semi-permanently because obviously the map comes down when you quit [inaudible]. And it also can change if I were to redefine X somewhere later on, okay?

But on behalf of this, X is associated with some list one, two, there, period, okay? Do you understand how the node that holds the one and the node that holds the two and the node that holds the three, that they all can't be reclaimed right now? Because I can refer to X anywhere down the line until [inaudible] quits, okay?

Let's say I do this: define Y to be the [inaudible] of X. All that does, because of what [inaudible] X evaluates to, is it basically points to that. I know it's obvious to you pictorially, but I'm going to depend on this observation is that one is addressed by just one thing, okay? The two and the three are effectively reachable from two different symbols, okay?

So I go on and I include a few statements that have no side effects that are of interest to the drawing, and I go ahead and say you know what? I don't like the fact that X is defined that way. I want to associate it with the list four, five. So X stops pointing there, and it points to this list four, five. I can even see the one, it's like over, like, the edge of a cliff there, the way that the drawing works out, okay?

So if whatever Scheme environment you're dealing with is detecting that memory is dear and it actually would like to clean up orphan memory for you, deallocate the memory, do what's called garbage collection – that's the terminology that's used in all modern languages, really.

It really should understand or have enough of a structure in place to not touch the four and the five, and not to touch the two and the three, but to reclaim the one if it wanted to, okay? Does that make sense to people?

So just in this drawing, 80 percent of the con cells are still actively – are still reachable from some symbol, okay? One of them, 20 percent, is not reachable, okay?

The system could just leave it orphaned for a while, but eventually, when maybe it's waiting for IO or it's waiting for a network connection or doing something where it actually doesn’t have anything else to do, it might say okay, well, since I'm not doing anything meaningful, I'll go and do garbage collection.

So you might ask well, how could it actually do that? Well, one technique is that some systems might actually go ahead and decorate these nodes with what are called reference counts. Those aren’t little commas and dots, they're actually zeros and ones, okay? Before X was reassigned up there and it was pointing to this, I would have had one and two and two there. Does that make sense, okay?

I'm sorry, I would have had one and two and one, because this would be two because the one and Y would be pointing to it. But this would still be one because only the three is pointing to it. Only the two is pointing to it, rather, okay?

Student: The problem with reference counting is that the real version of Scheme actually allows you to update the cells of list nodes or cons – I'm sorry, update the current [inaudible] fields of con cells in place. So you actually can program with side effect, and that basically means that if you want to you can create a circularly linked list, okay?

And that would mean everything in that list would have a mutual reference count of at least one, even though it might be this orphaned ring of friends, okay? Does that make sense? So reference counting doesn’t actually work if it wants to be able to reclaim everything. So what happens is that this is very high level, but I think conceptually, this is the better takeaway point, and it's not important that you know how to implement it down to the semicolon.

Every single time you call cons, either directly or it's called on your behalf because you’ve just typed in a list constant or you even use a define – define a factorial as a function – it actually stores the code in list form.

But you can imagine there being something of a master con set, okay, where every single con cell that has been created but yet to be deallocated is just catalogued somehow behind the scenes, okay? And maybe this actually points to a one, and maybe this points to a two. I'm sorry, maybe this points to a two and that points to a three, and this doesn’t point to anything.

Oh, I'm sorry – this points to two, because it's a one, but nothing's pointed to the one. So I'm being a little bit more elaborate in my drawings of that over there, does that make sense?

This could be a four, this could be a five. This could point to nil, I should do this the same way. When I talk about, from a data structure standalone, this master con set, it actually is something like a hash table or a binary search tree, so that it can actually find any single one of those in a breadth first reversal or a depth-first traversal or just a clear mapping over that set data structure.

When we talk about this symbol table, or this symbol map, where things like X are associated with that right there, okay? And things like Y are associated with that right there, okay?

So I'm just being a little bit more structured in my drawing there of the two lists that are relevant to that rightmost board over there, okay?

Now how would this as a system know that this con cell can be reclaimed? The fact that it's pointing to two is of no interest to anybody whatsoever except this orphaned one field, okay? And it's not like just because it has a hook on this two field, but anyone can discover it. It is really gone, okay?

But this and this and this and this all have to be preserved, okay, they cannot be freed by some garbage collection process that's running in some low priority thread in the background. Well, the algorithm that works beautifully in the Scheme setting, okay, and it actually could technically work in any setting if you can actually control how assignments are done and who actually gets – how pointers are handed out, what could happen is the low priority garbage collection thread could say okay, I'm about to go and clean up the garbage, okay?

But in order to do that, I'm gonna go through this three-stage process. I'm gonna go and I'm going to assume that every single one of these things can be freed. Okay? And it goes through and it marks every single node. Attached to some node is some bullion or some little bit of one or zero that's set to zero, meaning that if I ever see you again, or I see you in the near future again, and that bit is set to the founding face, then I'm going to assume that you can be freed because no one cares about you, okay? Does that make sense?

But then, before it actually goes and sweeps the garbage, it comes and says okay, symbol table, but here's your last chance to go and save whatever con cells you think are important. So from this standalone it says okay, I'm going to do a depth-first traversal of everything that's accessible from any node in the symbol table here, and this is gonna traverse and say you know what? No, you gotta keep that and you gotta keep that and you gotta keep the four and the five that are associated with it, okay?

Does the same thing for the two and the three, and in the process, because one is truly orphaned, it's not reachable from anything that's in X or Y, so the third of these three steps, master con set says okay, I gave you your chance I hope you were good about it and accurate about it, because I really am going to go and commit to deallocating – operator deleting or freeing, however – or letting garbage – Java's garbage collection do whatever it does to go and reclaim that right there, okay?

So the actual con cell set, that's not real terminology. I'm just making that up so I can call it something. It's only slightly more bloated than it ever needs to be in terms of overallocation, but it always has the option, if it feels that it's too bloated, to go through and clean exactly what isn’t needed anymore and spare everything that is needed at that point. Does that make sense? Yup.

Student:[Inaudible] the master con set, instead of just marking everything that needs – that [inaudible]?

Instructor (Jerry Cain):Well, the problem is is that – actually, you technically don't need the first pass if every single time you create a new con cell it's automatically marked with the frowny face, right? I'm just worried about the situation where something like this might be created during the sweep, okay? So it's probably better to actually, when this thing is allocated, it's allocated for a reason, okay? And that the smiley face bit, or the don't sweep me bit, is set to be high, okay?

Because at least initially you're optimistic in saying this is probably going to contribute to something for a while, okay? In that case, if you just can't trust what these values were ahead of time, then the initial sweep has to go through and say I'm marking all of you for deletion, okay? But there are certainly ways of optimizing it.

I haven't thought all the way through because I haven’t actually implemented something like this yet, okay? Does that make sense? Okay. Yeah, go ahead.

Student:[Inaudible] not necessarily work because [inaudible] all the false – first [inaudible] that you created them, and then you do a sweep then at that time that it's being marked by another, like by a variable? And then so you set it to say no, don't clean me up, but then it gets changed before the next sweep, so it would say oh, no, don't clean me up, even though it's still not being [inaudible]?

Instructor (Jerry Cain):Well, no [inaudible] happens in this binary [inaudible] critical region series, so that con cells would not actually be created during the sweep process. Now that's counter to the defense I was giving as to why by default I would associate this with a smiley face or whatever. That just might be a heuristic, to say I'm assuming it's in use because I'm actually defining it right here.

But I mean, basically you're about a [inaudible] condition threat, and it could just solve that using concurrency, like we learned a couple weeks ago, okay? You guys are good?

Okay, so [inaudible] curious, I know a lot of you are just doing nothing at all this weekend because you're going to be so bored, so if you want to learn about other languages that are purely functional languages, at least most of the purely functional, just go to Wikipedia and look up – I'm just throwing these at you; I'll actually talk about one or two of these in the last few days of the course – look up STEAM, look up ML, look up something called Haskell, which I'm still planning on bringing someone in to talk about.

And then there's this other language which is absolutely hilarious called Erlang, which actually I use – well, I don't use but a lot of people at work use because not only does it have really good functional characteristics to it, it also happens to deal with distributed systems very nicely. Kind of a weird mix of features, but that's why people use it.

Scheme you know a lot about. ML, short for Metal Lizard – don't ask. Actually, it's short for Meta Language, but I always say it's short for Metal Lizard, because that's the name of some interpreter where I learned ML. ML is a strongly typed version of Scheme. It has a different syntax, it uses square brackets instead of parentheses in a lot of places, it has a different notion of cons and car and [inaudible] in how it actually dissects lists and builds them up again.

But as opposed to Scheme, which is, like, I don't care about types, it's only if I'm actually trying to do something where I depend on the types that I'm actually going to holler with some kind of runtime error, ML actually has a little bit – even though it's purely functional, it has a little bit more of a compile time element, because when you define functions, as opposed to Scheme, it'll infer type characteristics about everything that's making up the function.

So if you do something like X plus two, it can see that two as a token is an integer, which actually constrains X to be an integer if it's going to be compatible with the plus function that deals with the integer domain. Does that make sense?

And then it can detect if the function itself is one that takes one integer and maps it to another integer, and that is itself a data type. It's kind of like a function pointer type that knows how to take two void stars and return it into another void star, or something like that.

So Scheme is functional, weakly typed; ML is a functional language, and it's strongly typed. I don't know very much about either of these two languages. Some of my TAs do. I just know that they are more the rage recently. I don't know whether it's because it's fun or they think they're great languages, but these are other buzz languages that come up in programming language circles these days.

I don't know any – I'm sure that there are practical uses of Haskell or else it wouldn't exist. I don't know what the motivating factor for designing it was, I just know that it is a functional language. It's very Pythonesque in its functional language characteristics. We'll learn about Python come Wednesday.

And Erlang I actually want to just do some research and tell you what that is about if we have the time. But nonetheless you have this little bullet list of things you can look up over the weekend if you're curious about what other functional languages there are out there, okay?

So there you have that. There is not going to be any discussion section on Tuesday, okay? I mean, we're done with Scheme, you've actually seen the set of very difficult problems in Tuesday's discussion section, and you have the assignment due on Wednesday night. So I'm not gonna have section on Tuesday because there's really nothing to cover. I haven't talked about Python yet. We will have a discussion section during dead week, okay?

So have a good weekend I will talk to you later.

[End of Audio]

Duration: 50 minutes