ProgrammingParadigms-Lecture19

Instructor (Jerry Cain):Hey, everyone. Welcome. I have a glorious set of handouts for you today. I have a midterm solution and I have the first two handouts and a stream of a few handouts I'm going to be giving you on our next paradigm and our next programming language. You'll see that I do have the midterm solution in there. I actually distribute that, obviously, so you can check your answers, but also so that we're somewhat transparent in how we actually grade these things.

Turns out that these things amount to like 25 percent of your grade, so I like you to know what criteria we're using to consistently grade everybody so that if you see something that is so clearly wrong in terms of the way we graded it, you can confirm that by looking at the grading criteria and then coming to me if you have problems.

Usually when there's a one-point error that's not listed specifically on a criteria, that doesn't mean I'm gonna give it back to you. It means that you made a mistake that we didn't anticipate anybody making and we didn't put it on the criteria. But if you have a total of five points taken off for a problem out of ten, and it's not clear why you have that many points taken off, that's a difference scenario, and so consult the answer key, and if there really is disparity, come and talk to me about it.

You should come and talk to me about it if you're worried about there being a discrepancy because this thing does just end up counting a lot. So don't feel shy about coming back and asking for clarity as to why we took points off that we did.

So what I want to do today is I want to introduce a new programming language, and I want to first illustrate a new paradigm, one that you've certainly not seen before unless you've coded in other classes at Stanford or coded prior to Stanford. We have spent a lot of time talking about these two paradigms, imperative. You'll also hear me call it procedural.

They're not necessarily the same paradigm, but the language we're using to illustrate both of them is the same. We focus on C as the representative of those two paradigms. We also have the object oriented paradigm UC++, and even though we know that CSC++ ultimately translate to the same type of assembly language that you kind of think about the problem differently or we think about our solution to the problem differently when we take a pure C approach versus a pure C++ approach.

The reason this is called imperative or procedural is that they're focused around the verbs that come up in the paragraph description of a solution. Think about what a main function looks like in a typical C program. You declare all your data, and then you make a series of call to these very high level functions like initialize, do it, terminate, and then return zero or something like that. Do you understand what I mean when I say that?

Okay, so the first thing you see associated with any C statement is usually or very often the name of a function that gets invoked to kind of take control of 10 percent of the

program or some small percent of the program just to get something done. As far as object oriented is concerned, you're used to something like this. I'll write the C equivalent in a second, but C++, over here you declare vector V.

You do something like vector new where you pass in the data and a few other parameters. You do things like vector insert ampersand of V, and vector sort of ampersand of V with additional arguments. Those aren't prototypes. That just means I don't feel like spelling out the rest of the call.

In C++, you declare a vector maybe of Nths called V, and you do something like V dot push back of 4 or maybe you do something like V dot erase of V dot begin to remove the front element. I know you haven't dealt with that specifically. Don't worry about the fact that you haven't necessarily used all those methods, but clearly, in this exactly right here, you're looking at the verbs first, okay.

It is oriented around the procedure, so I'll just go ahead and say that it is procedure oriented whereas right here you declare this object up front. And the first thing that comes up in a statement of any particular – comes up in any one particular statement, usually the first thing you look at is the piece of data that's being manipulated.

In this case, V is the data. It's also called an object. Because this is up front, it looks like each statement is oriented around the object, which is why it's called object oriented as opposed to procedurally oriented, okay. Does that make sense?

So you may think, "Well, I don't understand how I could possibly program it in any different manner." Well, even sequential versus concurrent programming, there's a little bit of a paradigm shift. You have to think a little bit differently about the problems that are gonna come up when you program in a threading environment versus a non-threading environment, okay.

Usually when – this is a little bit of a caricature, but this is really how I feel. Whenever you're coding up a normal program like programs one through four, you have this very linear way of thinking. You have a series of tasks you want to get through, and it's almost like you're inside the computer like typing things out one by one. But when you're programming in Assignment 6, it's at least not all of it has to do with execution logic. A lot of the hard stuff is like figuring out how all the threads are gonna interact.

And so you're thinking about multiple things at a time. And I'm actually like standing up a little bit more because I actually think with the back of my head when I'm programming concurrently because I'm trying to imagine all of these different scenarios of thread interaction that I have to worry about that have nothing to do with code, right? I actually have to see all these little players in like some thought cloud and how they might be interacting and how race conditions might come into being.

And so concurrent programming and multithreading is it's own paradigm that isn't really tied to any one particular language. Object orientation isn't tied to C++ any more than it's

tied to Java or Python or any other LON, which you might know. And even though C is probably the only procedural language you've really dealt with, there's Fortran, there's Pascal. Those things really do exist, not because a lot of people are writing new code in them, but there are Legacy systems from 20 years ago that still exist, and even if they're not adding features to that code base, they're certainly maintaining it and fixing bugs that crop up, things like that.

The amount of energy that was invested in fixing COBOL code bases back in like the final three months of 1999 was outrageous because everyone was totally petrified of the Y2K threat, that because we weren't storing years with enough information that everything was gonna go back and jump back to like year zero or 1900 or however they actually started it.

It turned out to not be nearly as big of a problem as they thought it was gonna be, but everybody was working in a procedural language called COBOL for a good amount of 1999, not everybody, but a good number of companies were, okay.

What I want to do now is I want to stop talking about procedural and object oriented for a while and go back to sequential programming for the most part and start talking about what the functional paradigm is.

Now functional and procedural sound similar, but procedure, if you're a purist about the definition, it is a block of code that gets called where you're not concerned about a return value, okay. Does that make sense to people?

Like you have to think about a procedure as a function that has void as a return value. When I talk about functional, I'm talking about procedures and functions again, but I really am oriented around the return value, okay.

We're gonna study a language, I think it's a very fun language to learn, called Scheme. There are aspects of Scheme that are interesting. I want you to invest a little bit more energy in understanding the paradigm than the language because the paradigm is – features of the paradigm are interesting takeaway points from a class like this if you're not gonna program in Scheme again, which probably will be 90 percent of you.

But nonetheless, the functional paradigm is very much oriented around the return value of functions. So let me just do an aside right here and just think of pure algebra, nothing sophisticated. But if you had a function like this right there, don't even think about code. Just think mathematical function. That looks like – it's the name of some function that takes two real numbers or two complex numbers or whatever, two pieces of data, and in traditional math, you know that it just returns or evaluates to something, okay.

So it may be the case that in a mathematical setting it's X cubed plus Y squared plus 7, okay. And in a pure math setting, you know exactly how to evaluate that if it stops being parameterized on X and Y, and you actually pass in 5 and 11, okay.

If I do something like this, then you know exactly what I mean when I write that down, okay. It turns out that the definition of G of X involves the definition of X where it takes this one parameter and splits it out into two parameters so it can call F. And then it incidentally adds eight to whatever is returned there. Does that make sense? Okay. And if I go so far as to do this, maybe it's the case that H of X, Y, and Z is actually equal to F of X and Z times G of X plus Y, and that's just the associations that are in place, okay.

Now this isn't legal code. This is just math. What the functional paradigm approach does is it assumes you have lots of little helper functions that are interested in synthesizing one large result. So maybe it's the case that I'm interested in the result of H where it gets a one, two, and a three.

And I happen to decompose it this way. I could actually inline the definitions of all of those things, and I could frame H in terms of just X and Y and Z, and not have any helper function calls whatsoever, right. But for reasons of decomposition, it's usually easier to frame things in terms of helper functions, and that's kind of what I'm doing right here.

What Scheme and what the functional paradigm tries to emphasize is that you give a collection of data to the master function that's supposed to do everything for you. It does whatever is needed in place to synthesize the results, and that answer is returned via the return value, and that's all you're interested in, okay.

Maybe it's the case when this is been one, one, and four. I have no idea what the numbers are. Maybe it returns 96. I have no idea. And I'm only interested in the 96 because that is the material product that I'm trying to get out of the program.

What a functional paradigm approach would take is that it would just say, and it would associate something like, how do I want to say this? It would do this, and it would associate it with F of XZ times G of X plus Y, or it might actually prefer to write it this way, is equal to X cubed plus Y squared plus 7 times F of X plus Y, X plus Y plus 1. And that's plus 8, okay.

But you would actually write it this way and really expect F and G as functions to themselves return values that contribute to the synthesis of a larger result. Does that make sense to people? Okay, question in the back.

Student: Why isn't the Y squared replaced with Z?

Instructor (Jerry Cain):I just – I didn't go that far. Where did I do? I'm sorry. I just messed up.

Student: Yeah.

Instructor (Jerry Cain):Yeah, sorry. Thanks. So rather than actually trying to do this in terms of pure math, let me just give you an example of what a Scheme function looks

like. I'm not even gonna try and explain what the syntax is. You're just gonna have to intuit what it probably does as a function.

I'll get to the pure syntax later on, but you can kind of gander what that as a keyword is probably gonna do. And I'm just going to do this. Let's say Celsius to Fahrenheit. It takes the temp, okay. And I just do this. I'm going from something at temperature, so what I want to do is I want to multiply it by 1.8 and add 32. This is how you would do this, okay.

Now you're not sure what the syntax is, you can kind of see that the right numbers come up in the conversion of Celsius to Fahrenheit, okay. So I want to scale 0 degrees or 100 degree by 1.8, okay, and then actually add 32 to it. And that's how I got 32 or 212 out of it.

So I'll go over syntax later, but what's really happening here is that in a Scheme environment, which is an example of a functional language, it associates this is a symbol, and the actual dash and the greater than sign forming an arrow, that's actually a legal part of a token in Scheme. They want you to be as expressive as you could possibly be using the full alphabetic or a full [inaudible] set, pretty much the full [inaudible] set to name all of your symbols.

It's framed in terms of this one parameter, and as a function call, it's equated with this expression right here where whatever value of tenth is supplied replaces that right there. So if I go ahead and I type this in to the shell, to the actual Scheme environment, it's supposed to somehow pop out a 212, and it succeeds in doing that because it takes this as a recipe, stops – there's a template on the tenth variable, actually figures out what it would evaluate to if ten became a value – came down to 100.

As an expression, it evaluates the 212. And so this as an expression is equated with this expression, and it comes back with a 212, okay. Does that make sense to people? Okay. Don't worry about the mechanics. Just think about the actual description of what a functional language is trying to do here.

Now let me actually just describe what the Scheme environment is like. We're using an open source product that I happen to – well, I didn't work on it, but I used it quite a bit a few years ago for a consulting job. It is a product called Kawaa. And I don't want to say that it's standard, but it happens to work fairly well and I just wanted to use it and I did because it's free and it's open source and I can just install it in 107 space, and then nobody has to – I don't have to bother anybody in trying to get support for it.

When you launch this thing called Kawaa, you more or less launch an environment that functions much like the shell where you type LS and make – and CD and things like that. It just happens to not speak the batch language or the TSCH language or TCSH language. It's actually a little bit more elaborate that something like Bash or SH or something like that where you actually go ahead and you get a prompt, and it always expects you to type in something that can be evaluated.

Very often – not very often, but it can be the case that you type in things that are very, very simple to evaluate. If you type in the number four, then in its little functional way, it says, "I have to let that evaluate to something." It's gonna do it. It's gonna have a very easy time doing it, and it's gonna come back with A4, okay.

If you go ahead and you type in the string hello, then it, itself is also considered to be atomic strings, or more or less atomic types in Scheme, or at least we can just pretend that they are. So it will print out hello because that's what the hello string evaluates to.

If I want to deal with Booleans, it turns out that pound F is the Boolean constant for false. It'll actually print this out for you. If I want to print out true, I can do that. It'll print out true. If I want to deal with floating point numbers, I can continue up here. You may think that it's going to be very clever about things like this, but if I type in 11/5. That looks like it's a request to do division.

It's not. You happen to type in a number in the rational number domain, and so what it's gonna come back is oh, that's just 11/5. Thanks for typing that in, okay.

If you go ahead and you type in 22 over 4, it will go ahead and reduce it for you, okay. But it usually stays with – it preserves type information as much as possible in going from the original expression to whatever it evaluates to, okay. Does that make sense? Okay.

The one composite data structure that is more or less central to Scheme programming, at least how we learn it, is the list. There are a couple of things that can be said about the list, but let me just put a list up on the board. If I do this, then technically, what I'm doing is I'm typing in a list. It happens to be framed in such a way that I ask it as a list to invoke the plus function against all of the arguments that follow it, okay. Does that make sense?

So the list is the central data structure in List and Scheme. We happen to be dealing with a dialect of Lisp called Scheme. Lisp would be a better name because that's short for List processing, but we're having to use an earlier version of Lisp called Scheme that was invented by John McCarthy, who's at Stanford now, but like some 50 years ago when he was at MIT as a – just as a untenured faculty professor at the time.

He just wanted to illustrate how closely tied mathematics and programming languages can be made to be by coming up with a programmatic implementation of something called the lambda calculus, which is basically some very fancy phrase for coming up with a theory on functions and how they evaluate, and not necessarily restricting them to real numbers and fractions and things like that, to let functions arbitrarily deal with floating points and Booleans and strings and characters and lists and hashes and things like that, okay.

If I – obviously this would put another six. If I do this times plus four, four, plus five, five, I do that right there, I'm dealing with nested lists where what were previously

housed by simple [inaudible] before are now – those possessions are now occupied by things that are themselves lists that should be recursively evaluated, okay.

So whereas this evaluated to one and a two and a three much like this evaluated to a four and a hello and a false constant. These evaluate to themselves, and then they participate in the larger function evaluation that uses plus to kind of guide the computation, okay. And not surprising, you'd get a six there. That four would evaluate to a four. That four would evaluate to a four. This as an expression would evaluate to an eight.

Five would evaluate to a five. Five would evaluate to a five. This entire thing would evaluate to a ten. The overall thing would evaluate to an 80, and that's how you get an 80. It's not the math that's interesting. It's actually the manner in which things get done, I think that's the most informative here, okay.

So I'm willing to buy that even these are function calls, I don't like using the word function call, okay. I mean I think function call is fine. I just don't like to speak of the return value of a function call because that's a very imperative procedural way of thinking about it. I like to think of this as evaluating to a 6 or an 80. Does that make sense to people? Okay. Okay.

So there's that. It turns out that plus and asterisk are built-ins. All the mathematical operators you expect to be built-ins are, in fact, built-ins. If I'm curious as to whether or not the number four is greater than the number two, I can ask. Isn't the case that four and two ordered that way actually to respect that greater than sign as a function. They still didn't return a number of a string. It returns a Boolean.

This would come back with something like that right there, okay. If I did this, less than – is ten less than five? That would come back with a false, okay. If I – this is just the prompt. That doesn't agree with that sign.

If I did something like this, it would assemble things in the way you'd expect, okay. It actually even does short circuit evaluation. So this one right here would be evaluated. This four as an expression evaluates to a four. This two evaluates to a two. This over all thing evaluates to a two. This ten evaluates to a ten. Five evaluates to a five. This overall thing fails, and it's at this point that the conjunction that you would just expect to be in place with ends would overall evaluate to a false, okay.

Now I am assuming you're gleaning the fact that the zero parameter or the zero position in a list the way I'm using them right here always identifies some form of functionality. There's functionality associated with this symbol right here, okay, and it knows how to take these two arguments and produce other true or false.

The same thing can be said right there and right there, okay. But the actual symbols that are attached Old Testament functions always occupy the zero place. It has this very prefix oriented way of dealing with function calls, okay. Does that make sense?

One of the most complicated things about Assignment 7, no joke, is actually getting the parenthesis right. You're so used to typing it at the end of a function, and then typing an open paren after it that that's what you'll type out, and then there's so many parenthesis surrounding you anyway when you're typing this stuff up that it's very easy to miss it.

So you have to be very – and just balancing the parenthesis isn't enough. You have to make sure that you get into this habit of just opening up a parenthesis, thinking like you have this entire list of things that help express some kind of function call, and just know that that's the type of thing that's really hard to get right when you write your very first function in Scheme, okay.

Now there is – there are a couple of things about lists that I want to go over before I start defining my own functions. I told you that Lisp, even though we're doing it with Scheme, we're really doing it with Lisp. It's called List processing for a reason. Everything, including function calls, come in list form. The only exceptions are things like for's and hello's and things like that, the atoms of the data types.

But normally anything interesting is bundled in a list. We don't really have – they do have structs in Scheme. They do have classes in our version of Scheme. We're gonna pretend like those just don't exist. Everything that's an aggregate data type is just gonna be packaged as a list. And we're gonna know that the 0th item in the list stores like the name. And the 4th – I'm sorry, the 1th slot in the list stores the GPA or the address or the phone number or something like that, okay.

What I want to do is I want to go over a few fundamental operations that are technically functions in Scheme that allow you to dissect and build up new lists. You're not gonna always want to return a 212 or a hello or an 80. A lot of times you're gonna want to return a list of information, or a list of lists, or a list of list of lists, or whatever happens to be needed in order to present the overall result to you.

There is a function called CAR. There's another one called CDUR, and there's one calls CONS. I'll go over why they're called this is a second. It's not really important. It's sort of interesting, but then it stops becoming interesting after a few seconds.

Car and CDUR are basically list dissectors, okay. If at the prompt, I type in CAR, I'll explain what the quote is in a second, one, two, three, four, five. This returns the number one, okay. If you just think about these things as link lists, they kind of are link lists behind the scenes. Car is associated with the data that's housed in the very first node, okay. It always is the evaluation of whatever is occupying the zero slot of a list, which is why this one comes back, okay.

If I ask for the CDUR of the very same list, it basically covers everything that the CAR does not, so it returns the rest, which in this case would be two, three, four, five, okay. Does that make sense to people?

If I do this and I nest them, I've asked for the CAR of the CDUR of the CDUR of one, two, three, four, five. It does recursive, that evaluation, in this bottom up strategy, comes here and identifies two, three, four five as a list, three, four, five as a list, and then the CAR of that is the first element of what was produced by this, which was produced by this. So this would come back with K3, okay. Does that make sense?

Now what's this quote all about? If these quotes weren't here, Scheme, and it may seem weird to you at the moment, but this is actually a much simpler language than CSC++, and I'll have several defenses of that in just two minutes. But if I take this quote away, then this right here is supposed to be treated just like this and this right here. Do you understand what I mean when I say that?

So it would actually look for a function called one, okay. And when it doesn't find it, it's gonna be like, "Whoa, I can't apply the one function to two, three, four, and five." So it would issue an error right there, okay. In our Kawaa environment, it'll throw a Java exception to advertise the fact that it's implemented in Java, but nonetheless, it will break, okay.

So you don't want to take the CAR of something that doesn't actually evaluate by putting the quote right there. It just is an instruction to suppress evaluation, that the list that is being presented after that quote is really raw data that should just be taken verbatim without any kind of recursive evaluation, okay.

It's actually shorthand. Whenever you see something like "One, two," and I could even do this like that right there, the quote just says basically to the parser, "Stop evaluating." From everything from the parenthesis that I'm looking at to the parenthesis that matches it, okay. It's technically shorthand for this right here.

It matches that, matches that, matches that, matches that, and quote is just this metafunction in place that doesn't actually – it kind of evaluates its arguments, but as part of the recipe for this quote function, it just doesn't evaluate its arguments. It just takes them verbatim, okay. Does that make sense to people?

You're gonna type it this way. You're not gonna use the quote function. There are all kinds of nifty variations on the straight, flat quote. I might go over it in a section handout, but it's so ridiculous. There are actually variations on this where you can actually use the back quote and the forward quote and the comma, which are variations of this right here, to suppress evaluation temporarily and turn it back on internally, okay.

But I just want to have this one thing where everything recursively is not evaluated, okay, and not deal with these variations. You can read about them if you want to, but you won't have to use them for anything that we do in this class.

Okay, so that is the way to suppress evaluation. That's gonna be very good because if we're gonna want to express all of our data in list form, we don't want to be penalized because we're using Lisp, that we always have to have some function evaluated in our

data, okay. We might just want to present our data as these bland lists, okay, and package them in a way that we just deal with consistently, okay.

So CAR is like synonymous with first. In fact, some dialects of Lisp actually first defined as a function. Coulder is synonymous with rest. It's like whatever you get by doing – following the next pointer behind the scenes, okay, whatever list you arrive at after the first element.

And you can use these in any clever way you want to to get to the third element or the last element or this element right here expresses a list, okay. Whatever you need to do to package – to get to your answers. You can package CAR and CDUR in some – whatever clever way you want to, okay.

Now why are they called CAR and CDUR? It's really not a very interesting reason, but they – I mean it's kind of interesting. There was a – the original implementation of either Scheme or Lisp, I'm not sure, was just on an architecture that had two exposed registers. One was called the address register, emphasis on the A, and one was called the data register.

And the CAR and the CDUR that were most recently dealt with the addresses that implemented them were stored in the address register and the data register, okay, and that's where the AR and the DUR come from, address register and data register. Does that make sense? I don't know where the C came from, something related to the letter C, I'm sure. I just don't know, okay.

So that's why they're there. Our system doesn't define any synonyms to these. Some versions of the language define first, second, third, fourth, all the way up to tenth I've seek, okay. But ours doesn't, so you really have to deal with the raw CAR and CDUR calls, okay.

These two functions take lists and break them down into their constituent parts. CONS is kind of the opposite. If I, at the prompt, do this, CONS, and I say one, which evaluates to itself on the list two, three, four, five, CONS is short for construct. It actually synthesizes a new list for you, and it would return this. So CONS is always supposed to be – take two arguments.

The first argument can pretty much be anything. The second one is supposed to be a list because what more or less happens is that it takes this element right here. It pulls this – it effectively pulls this parenthesis in like it's on a spring or something and drops the one in front. And whatever you get as a result of that is your resulting list. Does that make sense to people? Yes, no? Yeah.

Student:[Inaudible] two, three, then [inaudible] four five? So you can put one in between three and four?

Instructor (Jerry Cain):You could, but you – I'm sorry. So tell me what you want me to write. You want a CONS call right up front? And then what?

Student:[Inaudible].

Instructor (Jerry Cain):Write it like that?

Student: Yes. [Inaudible].

Instructor (Jerry Cain):Four, five?

Student: Yeah.

Instructor (Jerry Cain):Yeah. I mean they're – actually I know what you're trying to do now. That would not work. CONS really has to have two arguments, and the second one has to be a list, okay. If you wanted to do – let me just – in two minutes, I'll revisit this example and at least just show you the code as to how you would assemble this.

What I do want to emphasize – let me erase this since it is syntactically a little off. I want to emphasize the fact that it's very literal about how it takes the first piece of data and puts it into the front of the list. If I do this, CONS of one, two, three, and I try to CONS it onto the front before five, I actually will get from that another list where four, five is its CDUR, okay, but I will get this out, okay.

It's very literal in the fact that it takes this one element, which happens to be a list one, two, three, and it kind of prepends it to the front of everything that resides in the second list. So this emphasizes a point. I haven't formally said this yet, but lists in Scheme or any dialect of Lisp for that matter, can be heterogeneous, okay.

Right now I've – almost all the lists I've done up to this point except for one of them, I guess I erased it. All of the lists have been homogeneous in that they've all stored integers or they've all stored Booleans or strings or something like that. That isn't a requirement.

So there are a couple of features so far about this that I think are pretty interesting. There's very little type-checking going on, okay. There's a little bit, but there's not nearly as much of a compile time element to Scheme as there is in CSC++, okay. It just lets you type in whatever you type in, and it's only as it evaluates things that if it sees a typed mismatch, because you try to say add a float or a double to a string, that it'll say, "You know what? I can't do that, okay."

But it's actually at run time when it does the required type analysis to figure out whether or not something will work out. Does that make sense? Okay.

As far as what you wanted to do, there is a way to do it. I'll just introduce it because I can introduce a function pretty quickly. If I really wanted the list one, two, three, four, five

out of this, I don't have to use CONS. I can use a built-in called append. And that's not CONS. It actually does effectively remove that paren and that paren right there and build one big sequential list out of everything, okay. So that would give me the one, the two, the three, the four, and the five.

Append, unlike CONS, can take an arbitrary number of arguments. You can even take one list if you want to, but if I gave it this, that would return what you'd expect. It would actually figure out how to return one, two, three, four, five, six, seven, eight. And we'll be able to implement our own version of append in a little bit, okay. But it basically just threads everything together. It's like it removes all intervening parenthesis and whatever list is left in place is the return value. Yeah.

Student: Would it work in three [inaudible] list?

Instructor (Jerry Cain):It would not, which is the next example because that's what – the way he fed arguments to the example he was announcing didn't have one of them as a list, so I will fix that problem right now. If I really wanted to put a one in between a two and a three and a four and a five, I could do this, append – let me put the arguments this way. I will just say two, three.

I will write it incorrectly. I'll say one, and then I'll put the list four, five. And let's, of course, pretend that my goal is to get the list two, three, one, four, five out of that. Append doesn't like this. It wants to see parenthesis around all of its data points, okay. You could actually create a little list around the piece of data by calling this other built-in, and then all of the sudden, that just temporarily, or not even temporarily, wraps parenthesis around it and creates a singleton list so that it can actually participate in an append call, okay.

Now I'm just breezing through all these functions. I will be honest. I've probably talked about half of the functions you're gonna need to learn for the Scheme segment of the course, okay, and none of them really are that surprising. Like list and append, that's not rocket science.

It may be interesting how they work behind the scenes, but it's not like they're obscurely named, okay. CAR and CDUR, yes, they are – and CONS, they are obscurely named, but those are probably the only three that really need to kind of think and remember what they do, but even then that's pretty easy, I think, okay. You guys get the gist of all the mechanics here? Okay. Yep.

Student:[Inaudible].

Instructor (Jerry Cain):[Inaudible] are cool. In fact, they're used a lot. I should emphasize that if you type in something like, let's say, CDUR of the list four right there, what follows the four is nothing, but it still has to be expressed as a list, so this would return that right there, okay. It's fine, and actually, the empty list is kind of the basis point

for forming all lists. When I talk about how CONS is implemented, you'll understand that the empty list is kind of like the base case of a recursive call.

I should say that if you do this, that's a no no. Now some implementations will just – whenever you try to take the CDUR of an empty list and try and remove a car that isn't there, that some implementations will just return the empty list for you. I don't want you to program that way.

I want you to assume that either a CAR or a CDUR levied against the empty list is actually an error, okay. And I actually am forgetting right now was Kawaa does because I never try to exploit the feature if it is one. I just assume that this is gonna be [inaudible] in there, okay. So no, it would print no, don't do that, okay. Does that make sense?

So I dealt with every – more or less I've dealt with all data that's been a constant or a list constant or something like that. That's not the way it is. You really do define functions in Scheme as well, or else you wouldn't be able to build scalable systems that can be parameterized in terms of arbitrary datasets.

So I already gave you an example of one function over there, but let me start even a little bit easier. If you go ahead and use the define keyword, define has its own purpose. It's occupying the slot where you normally see pluses or times or divisions, okay. What happens next, if I just do this, add, okay, does that make sense? And I just pass an X and Y, there's no comma separation between the arguments. The space is the delineator.

And I equate this functionality like that, okay. Then you type that in. It actually comes back and says, "Oh, I just defined add. Thank you very much, okay." It actually prints out add, not because it evaluated to add, just because it's the define keyword. It just wanted to remind you what function just got defined.

Now this is the very first example, and this is an obscure point, but I kind of want to revisit this a few times later on. This is the first example of any kind of Scheme expression we've dealt with so far that has some side effect associated with it. And the way you hear that, you may be like, "Well, why is that interesting?"

This purely synthetic approach where it takes the data and it synthesizes a return values so that the overall expression evaluates to it, it does all of that without printing to the screen or updating memory in any way that we know about. You're not passing around raw pointers anywhere. Do you understand what I mean when I say that? Okay.

Even the lists themselves are being synthesized on your behalf. If you were trying to do the equivalent things in CSC++ you would have to declare your list data structures, okay, or worse yet, you'd have to actually define a node type, and you'd actually have to thread together link lists using Malak or New and free or delete or whatever you have to do. You'd have to manage the memory allocation by yourself.

Scheme is so much more of a higher-level language and it's smaller and it tries to do less that it's easier for it to do – take what it does and do it very, very well. The list, as opposed to C or even technically C++, the list is a built-in data structure that's core to the language. So in the same way that we breed string constants and integer constants to C, you can actually express list constants. I don't have any up here. Yeah, I do.

This right here, this knows how to build a data structure to represent the list, one, two, three, four, five behind the scenes, okay. You don't have to manage any of that. In purely functional languages, and we're gonna strive for this in the [inaudible] scheme we're gonna learn, you try to program without side effect, okay.

Only to the extent necessary do you update variables by reference. I've certainly not done any of that yet, okay. I've always just relied on what it evaluated to. Technically, there's a side effect associated with this right here in that in the global name space it associates this add keyword, okay, to be associated with this functionality right here, so that from this point on add, the way I've defined it, it actually is a built-in. It behaves more or less like a built-in just like CONS and CAR and CDUR and list and append all are, okay.

They really are peers. It's almost as if there's a map, a global map of symbols mapping to actual functions, okay, where the functions themselves are expressed as lists, and that map is prepopulated with code for CAR and CDUR and CONS. And then you can add to it programmatically by associating this keyword with this list right here, which knows how to behave like a function. Does that make sense to people?

Okay, so when I do this, add 10 and 7, it comes back with a 17 because it somehow knew how to take this 10 and the 7, crawl this list right here to figure out how to deal with the 10 and the 7 that were passed in, and whatever it evaluates to is what add evaluates to. So it's like you equate this symbol parameterized by these two arguments with this Scheme expression, okay. Yep.

Student: Does case matter? Like why does it give you add?

Instructor (Jerry Cain):That's just – actually, I shouldn't have emphasized that. Case does matter when you're typing these things out yourself. For some reason, and this may not even be the case with Kawaa, I just remember the Scheme interpreter I used in this class forever capitalized everything for reasons that weren't clear to me.

But you should be sensitive to case, but just because I print something out in uppercase doesn't mean anything, okay. I like de-emphasize this. Pretend it's just – don't even worry about it, okay. Yep.

Student: Can you use ellipses and say like add XY and –

Instructor (Jerry Cain):Yeah, you actually – I'll talk about that the last day of the Scheme segment when I talk about these equivalent features to CSC++. You don't do

that. You actually use a special parameter that catches everything beyond a certain point into a list.

And when we implement, you'll see a little bit in like two or three lectures what the equivalent of the dot dot dot from CSC++ are. I just don't want to go over it quite yet, okay. I mean I just defined my first function ever and it's add. You can see I'm just not that far yet. Yep.

Student:[Inaudible] can you redefine it later?

Instructor (Jerry Cain):Yeah, absolutely. You can redefine it. Some systems will let you redefine CAR and CDUR if you want to. I'm not recommending it, but if you want to like displace the built-in functionality that's associated with CAR and CDUR and list and append, some implementations will let you, okay. I'm not sure whether Kawaa does or not because I haven't tried, but I just know in the spirit of Scheme and how it's implemented, it's certainly possible to do that, okay.

Now this isn't very interesting. What I will do is I will write a function that just deals with a list as data. Notice that I have not actually typed X and Y here at all. So if I want to do this, there's no problem with the definition itself, but if I try to do this, it's only when it tries to evaluate this expression right here that it says, "Well, I don't like levying a plus against two string constants." And only there will it issue a runtime error. Does that make sense? Okay.

Think about the CSC++ equivalent. You would have had to attach data types to this right here, and you would have had to script this call up in the same file or some other file and compiled it so that at compile time it could detect that this isn't gonna work out. There is really very little compile time element to a Scheme interpreter. It just – when it parses the list you type in, that's technically compilation, but it also evaluates it at the same time.

So there's really very little separation between compile time and run time in Scheme, and because it's an active interpreter system, we just call it the run time, okay. So if I type this in, this would error out.

Okay, so would we all agree that length lists are recursive data structures? Okay, more often than not, if it's linear recursion, you would probably just implement it iteratively. In Scheme, we're gonna take this purely functional approach and we're not gonna do any in place iteration whatsoever. If I wanted to – oh, get a clean board.

Here's a better function that illustrates how compact and dense. In many ways, it's a bad thing, but it's just a feature of the language, how dense Scheme code can be. I have two minutes to write this. I can certain do it.

What I want to do is I want to write a function that knows how to add up all the integers that are supposed to be in a list, okay. So I'm gonna assume that it's a number list. And so if I give you – let's just say sum of – and that's not a minus sign. It's actually a

hyphen, so it's one token. And I give you this right here. You know it's supposed to be ten, I think, yeah, ten.

And the way that the Scheme functionality is gonna realize this is it's gonna say, "Oh, I have a non-empty list. That means I'm gonna add the CAR to whatever I get by recursively applying the same function to the CDUR." So the ten isn't so much a ten as it is a one plus a nine. The nine isn't so much a nine as it is a two plus a seven. Do you understand what I mean when I say that?

Okay, here's the implementation of this function. Define sum of – oops, sorry I did it, sum of – and I'm just gonna call it – I don't want to call it list. I don't want to get in the habit of naming my variables the same as built-in functions. So I'll call it num list just like that, okay. Does that make sense?

And what I'm gonna do is I'm gonna employ a couple of built-in tests. If it's the case that null question mark num list, then return zero. The if is exactly what you'd expect. It needs three parts to follow it, a test, an if portion, and an else portion. The else portion is technically optional, but I don't want you to pretend – I want you to pretend it's not optional.

Null actually comes back with true if and only if this thing evaluates the empty list. If it is empty, then trivially it's the case of all the numbers in this empty list add up to zero. Otherwise, what I want to do is I want to equate the original sum of call with the value that you get by levying plus against the CAR of the num list and the call to sum of the CDUR of the num list, okay. The headache really is just matching all the parenthesis, okay. But conceptually, this is the recur one that's in place to get this done.

Now you don't have to implement this recursively, but we are at the moment, okay. And we're always gonna opt for recursion over iteration in the Scheme segment of the course just to emphasize the functional aspects of the language. Do you understand how this is working?

It is just basic recursion, which is with the new syntax, okay. Synthesize the recursive result, get the nine back, add the one to it, and that's what this overall thing needs to evaluate to. Does that make sense? Okay.

So you have that. As long as I feed it one, two, three, four, it doesn't have a problem. If I feed it one, two, three, and then four is a list, it'll actually succeed in making three recursive calls, but only when it tries to levy a plus of the four is an empty list against a zero that it'll actually have problems, okay.

So it just does on an as needed basis the type of type analysis that is needed to confirm that the addition can be done between the CAR of the list and whatever was returned recursively, okay. Make sense? Okay.

I want to write a lot more of these come – today's Wednesday, yeah, come Friday. I'll write a lot more of these things. And then I'll start talking about language constructs that are equivalent to the types of things we've seen in our C++ work and also in Assignment 3 and 4. Okay, have a good night.

[End of Audio]

Duration: 52 minutes