Beginner Focus Group

ME

[00:00:00] Hello guys, thank you very much for coming. This is the focus group for my dissertation project, which is called SFL Explorer. SFL stands for Simple Functional Language. This is just a very, very basic functional programming language. So I've asked you guys to come because I wanted people who knew a little bit about programming and potentially maths, which is great because there's functions in maths, so functional programming might be hopefully a new concept to you and hopefully an interesting one. And then I wanted to kind of get Amos here to teach you a bit about functional programming using this tool, and then I'll give you a little interview afterwards about what you thought about the tool for my evaluation of my dissertation. Do you just want to introduce yourself?

AMOS

[00:00:52] Yes, Amos. Hi there. I'm a fourth-year computer science student here at the university. I have taught functional programming units before. I've taught a first-year unit and a third-year unit before, so that's why I'm here. I've taught things similar to this, not exactly this, but things similar to this before, and hopefully that will be something. I've taught a previous, we did another version of this with people that had done functional programming before. But only like a little bit. I'd only done a little bit, so it's sort of, we've got different levels. Hopefully this session will as well. That would be nice to be able to do.

ME

[00:01:24] Cool. So, as I mentioned, I forgot to bring the consent forms. The information on the consent forms would have said that all of this data will be anonymized at source. It's a transcript. I'll get you to say your names at the beginning, just so I can assign you like person one, person two, person three, person four. But it's all anonymized at source. If you want to withdraw your data from this, you will have to do it before I anonymize it, but you can do it after, you know, you can come speak to me after this session, or you can send me a message if you don't want your data to be involved in this process. Yeah, it's anonymized at source, and you can't withdraw after I've anonymized it. I think that's all the, yeah, that's everything. It should have had that on the form, but anyway. Is everyone happy to go ahead? Cool. Yeah, do you want to go? Yeah. Okay, I'll start. First of all. Can everyone see everything okay? Yeah, does everyone have a view of what? Yeah. What I'm doing with my screen, blah, blah, blah, all this. You can zoom in a little bit, maybe.

ME

[00:02:28] Yeah. How does it look to everyone? You can do light mode, maybe.

SPEAKER_6

[00:02:31] Yeah, what if I move it?

ME

[00:02:33] Settings, toggle theme.

SPEAKER_6

[00:02:35] Is that any easier to read? I'll do it like this.

ME

[00:02:39] What would you prefer, light mode, dark mode? Do you want it zoomed in a bit? Can you see it?

SPEAKER_6

[00:02:42] Yeah, that looks good.

ME

[00:02:43] Yeah, I can see it.

AMOS

[00:02:45] If you want to see it, just leave it as it is. If my eyes hurt, I might move to dark mode. Okay. Okay, I'll start off with introductions. So, I guess we could go around and just say names just to get introductions so I can know who I'm with today. Hi, I'm SPEAKER_3. Nice.

SPEAKER_4

[00:03:03] I'm a third-year math student. Hi, I'm SPEAKER_4. I'm a third-year engineering student. I'm SPEAKER_5. I'm a third-year maths student, and I don't really know much about functional programming, but okay, great.

SPEAKER_6

[00:03:15] I'm SPEAKER_6. I'm a second-year engineering student. Cool. I thought you were maths. I don't know why.

AMOS

[00:03:21] Okay, so we have SPEAKER_3, SPEAKER_4, SPEAKER_5, SPEAKER_6. Correct. Okay, what is our... okay, I'll start off just to, I guess, get a ground level. Have we all programmed before? Have you used... what languages have people programmed before?

SPEAKER_4

[00:03:38] Python and R, mainly. Python and R? Yeah. C++, Python.

AMOS

[00:03:43] C++, Python. C, I guess. C, I guess. Yeah. You have all programmed before, and all the languages that you have given to me are imperative. And imperative languages is generally a language where you give the computer a list of instructions, and it does those things in order. And the way it evaluates the program is it sort of just, it compiles it to the assembly, but essentially it is just going through the list of instructions, doing them one by one. So it's like, do this, do that, do this, and it will evaluate them by going in order through the instructions. Functional programming languages, instead, are evaluated like actual mathematical functions. So when you

define a function, there is no, like, the order of execution is not decided by what line, like, by the lines they're on. It is, if the function is called, then essentially you just directly substitute variables into that function and then evaluate whatever you get out the other end. Which is sort of mathematically impossible. It's a bit cleaner, more sort of closer to how, like, how things are described mathematically. And also has some nice, like, abstractions that can make programming a little bit, maybe make a little bit more sense, or sometimes make less sense. So, for example, like, the basics of a, basics of, like, a functional programming language is you can define labels, and you can say, like, oh, x is equal to 5. This, in a traditional language, like in Python, when you say x equals to 5, that means that x is now always equal to 5. Like, it will run that line, and x will be equal to 5 whenever you reference x later. That isn't quite how, I should turn this untitled.

ME

[00:05:24] Turn the time checker off, yeah.

AMOS

[00:05:27] Before I continue. So, in a functional language, if I then did main equals x, instead of it being that x is just equal to 5 now, when we see x, we look for our definition of x, and we substitute it in. Which is slightly different. Which is slightly different. To how it works. There is no state. We're not writing down, for example, somewhere. We're not sort of, like, setting the value of x in some, like, big array of variables to 5. We are just saying that when we see x, we replace it with whatever's on the other end here. And we can put anything on the other end here. But if we, when we evaluate this, so main is our sort of main function that we execute. When we evaluate it, which we can do lazily as the normal way, I'll talk about that a bit more later, we see that main, we just get x. And the way it evaluates is it just substitutes x for whatever is over here. So it does that, and we get a 5. And so, we're evaluating instead of going through this program line by line. It sort of looks at what main is. It substitutes in whatever it needs to substitute in. And it finishes evaluating. Does that kind of make sense how that evaluation works? Do you have any questions about why it's different to imperative? Is there like, oh, this doesn't actually sound any different? Or do we feel, how do we feel about it? Does it feel like it makes some sense?

SPEAKER_4

[00:06:44] Yeah, I don't really get why that's different to like how you define a variable in Python, for instance.

AMOS

[00:06:51] I see that, yeah. The effect of it, especially at first, it doesn't seem very different, does it? Because you are just replacing it. Under the hood, it's doing a very different thing. But hopefully we'll see later how what we're doing is quite different. Because we can't, if I couldn't write like, you know, several lines of, for example, like if I did x equals 5, y equals 2, I couldn't then do another line that says like, oh, y equals. I guess I could do that. But if I wanted to. You couldn't do that. I couldn't do that. I guess it's already defined, yeah. So we can only define variables, labels once is one thing. So if I did, imagine if they did y equals x plus 2.

ME

[00:07:27] So these aren't really variables. They're more, I mean, they are, but they're. Are they more like definitions? They're definitions. They're definitions. So you cannot change it. It's just saying, you're not saying, oh, x is 5 for now. Yeah. We're saying that x is literally 5.

AMOS

[00:07:43] So we cannot mutate these, for example. We couldn't update x. We could assign something else to be x plus 1. We say, OK, that's what, that's x plus 1. But we cannot change what x is. X is 5 now, definitionally. And so we can do things like, so we've defined x to be 5 and y to be 2. We also have like arithmetic. So we can do x plus 2, x plus y. And we would expect this to result 7. And if we evaluate it, the evaluator is going to look. It's going to see x plus y. It can't currently resolve this because, you know, plus doesn't know anything about x or y. They're just letters right now. So it has to perform substitutions first. So we substitute in 5. And we say, OK, this isn't quite, we still can't evaluate this. Then we substitute 2. And only then do we apply our function plus. Because plus is essentially a special function. It's an inbuilt function. We can define our own functions. But plus is sort of, it's inbuilt. It comes in the box. And it can only apply when it sort of knows that its inputs are, like, it can only apply when it can work with its inputs, essentially. So if we apply this, then we get 7 out the other end. So we can sort of define things in this way. We could define any variables we wanted. We could say, like, our label is like 7. And we could do multiplication. We could do various operations with it. But right now you're probably thinking this is a bit, you know, we can't do much right now. Because we can only, you know, we can only add things together. We don't really have much expressive power. Because, like, we can't loop right now. We don't have, like, you know, we can't, like, define subroutines. It kind of feels a bit annoying. We do have access to functions, however. And the way that I think we're mostly going to define them in this session, different languages have different ways of doing it. But I think this way is a little bit more direct. Is functions can be defined with lambda expressions. So to define a function, you can, so in the same way that we sort of are just, like, saying that x is 5, we can say that some label is a function. And the way we define it, we can write our function is with lambdas. So we use a backslash to be lambda. And then whatever we do, like, for example, lambda x dot x. And this is saying if we, we can, and we can apply functions. So this function. When applied to some value will, essentially, this is, we've bounded this variable. So if we applied this to 5, we would substitute 5 in for x here. And then that, and that would be how it evaluates. And this x is not the same as this x in here. This is an x bound within the function. It only exists to be the function definition. It's not referring to x we've defined elsewhere. It is just used for this function. It might have some problems if I, I don't know. Would it have problems? Or would it be fine to just evaluate like this? It's not going to have a problem with the fact that I've used x. I've bound x here and I have x there. It will have a problem with that, yeah. It will have a problem with that? Okay. Parse error. Okay. The parser wouldn't have a problem with that. So we'll do something else. Or I'll just get rid of x, actually. Yeah. Definitionally, it's not a problem. But I guess in, like, if we define it elsewhere, yeah.

ME

[00:11:02] Because it's ambiguous.

AMOS
[00:11:03] Yeah.

ME
[00:11:03] If you have that expression, it just stops pitfalls to say if you have your expression in the lambda body. Yeah. If the expression references x, which one is it referencing? Yeah. That's a good point. Making it a, it is meaningful. Yeah. Still. It is still meaningful in kind of pure lambda calculus. But in this system, I decided to make it so that you wouldn't have that pitfall. Yeah.

AMOS
[00:11:27] Because you could have, like, the, like, precedence where the most recently bound thing is what that is was normal. But yeah. So we can define func lambda x dot x. And if we, we can apply func to values. So if we applied func to y, does anyone have a guess at what this would evaluate to? If we, so we've taken func, which is this lambda. It's, it takes x and it just returns, and it seems to just return, this is sort of what it would evaluate to, is just x. What do you think this would evaluate to if we did func y? Do you have a guess?

SPEAKER_4
[00:12:02] Yeah. Two. Two?

AMOS
[00:12:03] Yeah. Let's have a look. So let's look at how it evaluates. So first of all, func we've, is defined as a function, and so it will apply to y. And when we do that, it will just take y, substitute it into the body of this, which is just x, and then we get out a y on the other end. So if we then substitute for the label, we get two. So this is sort of how, generally how we define, this is how we're going to define functions. So we can define, we can also, for example, if I wanted to take in two things, I could say like x, z, and then do x plus z. So now this is sort of like an add function. And if I did func y, y, it would add those two things together. What?

ME
[00:12:51] Oh, because you've defined x, z, you need to put space between x and z, because you've made a variable called x, z. Makes sense. Yeah, that works.

AMOS
[00:12:59] Okay. So when we apply this to y and y, we're going to sub in y for x, and we're also going to sub in y for z. So we could have a different thing here. We could have like 0, 3, whatever, any value we want here. But in this case, I've just done y. And we apply those. So we have func y, y. We take in, substitute y here. In this evaluator, when we substitute a label. That's an interesting. The way it's written that. Yeah.

ME
[00:13:27] Yeah. That's surprising. It's replaced func y with y plus, which I mean, it's true.

AMOS

[00:13:34] Yeah, that is true. It's curried. So it's. Yeah, it's curried. Yes. I guess that's probably what I'm saying. The way that functions work when they have multiple inputs is they are curried. So essentially what that means is that instead of taking all the inputs at once, it takes in the first input, substitutes that. And then instead of having func y here, we just have func applied to y. So we have like lambda z dot. Y plus z is the middle evaluation step. It's kind of skipped that here because it's smart and it knows that it can just take two steps at once. But we can we can do sort of. We can apply a function partially, essentially to take in one input, which generates a new function that then can be applied to the next input. Essentially, you can also do multiple inputs with like pairs, which is more similar to how like an imperative language. Do you want to do const before we do pairs? Oh, yeah. I was just trying to make it. Anyway, we substitute y in both. So there are some quite common functions that we see in functional programming. So this function that we looked at before, lambda x dot x.

ME

[00:14:44] Then do you want to bring up the syntax sugar of before we do pairs so you could put idx and say run. Oh, I see. Explicit lambda.

AMOS

[00:14:54] Okay. So we're going to. Okay. Yeah. Makes sense. So defining things with lambdas is nice. It makes sense. It's how things are written in like if you're if you're reasoning about them theoretically. However, when writing programs, it's kind of annoying. So we can define. We have some special conventions for when we define functions. So instead of saying lambda x id equals lambda x dot x, we can sort of implicitly like assume this is some function, bind some variables on it and then say, okay, if we if this binds a variable x, what is the result?

ME

[00:15:27] That's exactly the same as a lambda expression. Yeah. That is what it is under the hood.

AMOS

[00:15:31] Under the hood it's doing the same thing. This is just a sugar to make it easier to write. And so another good example of sort of a basic function that we have is const. So const takes in, with const we want to take in two inputs. We can call those, we'll call them x and z since y is defined somewhere else. And we just want, given those two inputs, or actually no, sorry. Yes. Yeah. Given those two inputs, we want to just always return the first input. Based on that definition, we don't have an idea of what the body of this function would look like. So we're taking in two inputs, x and z. And regardless of what the second input is, we always want to just return the first input.

SPEAKER_6

[00:16:16] Lambda x, z, x.

ME

[00:16:19] We don't need the lambda because we've already got x, z there. Which is implicitly a lambda.

AMOS

[00:16:23] Yeah. This is implicit. And if you did that, I think you would have to apply it on the inside. But yeah, that is, it is lambda x, z, x. Yeah. So that's just, we'll always take x. And this is what, when we do this, we're sort of implicitly naming it, which is easier to write. And that's the only reason we do it. So then if we apply, say, const to seven and five.

ME

[00:16:47] Do you want to go to free choice mode for this? Yeah.

AMOS

[00:16:49] So you can do it slowly. So we have const seven, five. Free choice mode lets us look at all the different ways we could choose to evaluate this. So we can either just substitute here. We are just taking const and substituting it for its true definition, which is just lambdas and a result. So we talked before about const this just being a shorthand for lambda. This is very literally true. Const here is being substituted for lambda x dot lambda z dot x.

ME

[00:17:18] I don't have any questions about those three. I don't think we have any questions about those three options for progress. Or do they all make sense?

SPEAKER_4

[00:17:29] Sorry. I think I missed the section about ID.

AMOS

[00:17:33] Oh, ID. So the function we had before, lambda x dot x, that is just ID. I've just named it so that if we need to use it later, it's just ID x. So ID just takes in, is a function which takes in a value and just spits out that same value. Okay. It's not particularly useful, but it's useful for sort of like looking at, it's a very simple example of a lambda and you can sort of see how it behaves.

ME

[00:17:59] So do you want to do ID in free choice mode?

AMOS

[00:18:01] Yeah, we can do ID first. Let's look at how ID works in free choice mode. So let's just do ID 7 and see how that works. So ID 7, we have a couple of choices of how we can evaluate it. Does anyone have any questions about what these reductions mean? So when we reduce, we're saying we're taking a whole term and we are reducing some part of it to get closer to an evaluated form. So, for example, we can reduce ID to its definition or we can reduce ID 7, the whole thing at once. Does anyone have any questions about what these are, what they mean or

why they're valid?

SPEAKER_6
[00:18:41] When we have the, Error with the Yes.

ME
[00:18:44] So formally, the second one actually isn't a reduction. A reduction is only, the only thing a reduction is, is an application, which is where you substitute through a body or an abstraction with the thing you're applying it for. So only the first one is actually a reduction. So a reduction is a substitution into a lambda function. That's what a reduction is. The second one is just a substitution. That's actually zero steps, which is why there's a star, because a star means zero or many. Yeah, sorry, what were you going to say?

SPEAKER_4
[00:19:23] With the second one, you'd still need to sub 7 at the end. Yeah, exactly.

ME
[00:19:27] So if we did, if we did the second one. If you just do that one, then it just, it just swaps it out.

AMOS
[00:19:32] We've substituted ID with its definition.

ME
[00:19:34] That's not a reduction. That's just what it is. Like evaluating the function. Yeah, that's not even evaluating the function. That's just literally swapping it out. It's just saying this is what it is.

AMOS
[00:19:43] Like ID means lambda x.

ME
[00:19:45] So that is actually zero steps, the step we just made there. So the arrow and the star means, the arrow means reduction, and the star means zero or many.

AMOS
[00:19:55] And so you might also have like arrow plus, which means at least one step. But because a lot of the steps, especially in free choice mode, a lot of the steps that you can take are not actually steps. They're useful to see sort of like what the, what is happening. For example, we can see like ID is under the hood, a lambda. But it's not. But it's not. And we're not actually doing a reduction there. And then we actually.

ME
[00:20:20] This program is identical to what it was before. This is, we have not made progress.

Yeah. If that makes sense.

AMOS

[00:20:25] Yeah. Like this is still ID seven. You know, this is, this is just that this is ID. And then we only actually accept when we apply something. So that's where we take seven, sub it into the body of our lambda and step to seven. And if we look at const similarly. And we did like const five seven. If we look at that in free choice mode. So again, here we can take zero steps to just to reduce the const to its actual function. We can take one step where we just apply it to its first argument or we can apply both. Do these all make sense? Can you all see how these lines are varying steps along the way? So I think this is probably a good way to look at currying. So if we just evaluate it, just sub out const. Yeah.

ME

[00:21:11] That's not progress. Yeah. Then do you want to do the second one? Yeah. And then that creates a whole new function. Do you get what I mean? So you're substituting. Yeah. You're substituting. Let's go back to, I don't know how that happened. Let's just refresh the page because I think it's getting funky now. It's been running for too long. Do you want to copy paste all the, it gets funky sometimes.

AMOS

[00:21:36] It starts doing strange things.

ME

[00:21:38] There are ghosts.

AMOS

[00:21:39] It's a really complicated thing. Yeah. It's a really complicated system.

ME

[00:21:41] I have no idea. There's ghosts. Oh, you've got to turn the type checker off again. Type checker. Okay. I need to. Okay. Well, it saves the fact that you're in light mode at least. It doesn't say. Okay. So if we. Do you want to do the bottom one and then that one?

SPEAKER_6

[00:21:52] And that one.

ME

[00:21:52] So we've curried. We've applied a function and then you see the lambda expression here. Can you see the mouse pointer by the way?

SPEAKER_4

[00:22:01] Yeah.

ME

[00:22:02] The lambda expression here. You've returned the right hand side as a new function. The right hand side is now lambda z . 5 because the x is here. It's been swapped out for five. So this is literally a new function.

SPEAKER_4

[00:22:16] If that makes sense. See. So it's like taking what the x would be in the output. Yeah. In the output. Yeah. So if the output was like instead x times z, then it would do the x. Yeah. And it would still be times z.

ME

[00:22:30] Then you'd end up with five times z. Okay.

SPEAKER_4

[00:22:32] Yeah. So if that makes sense.

AMOS

[00:22:33] So.

SPEAKER_4

[00:22:33] So you're like removing the variables that you input each step.

AMOS

[00:22:37] Yeah. Yeah. But it's still a function. So we could, for example. So because. So yeah. We see we generate a new function here. And then we apply it. So under the hood, whenever we have multiple things, we really have a function. And then another function inside of that. And then however many we have functions inside of that.

SPEAKER_4

[00:22:54] Yeah. This is all just syntax sugar. So it is actually how you just do maths, right? Yeah. Yeah. I see.

ME

[00:22:59] Do you see why I wanted some mathematicians? Yeah.

AMOS

[00:23:03] Yeah. It's. It's the sort of. The thing that. There are several. Like. There are several nice things about it. The thing that. Most people that. Sort of do. Functional programming language theory as their job. We'll say. It's like. Oh. It's very. It's pure. It's mathematical. It like. You don't have weird. Like. Side effects going on. There's no side effects. Everything is immutable. Stuff like that. It's all immutable. It's all nice. It's all. You can. It's very. Like. The execution is very traceable.

ME

[00:23:29] Very, very easy to reason about.

AMOS

[00:23:31] So for example. Because we have currying. We could define a function const five. Which is. Using const. So we could say. Okay. If you want a function that just. Takes an input. And always returns five. We could. We. We could define that as this. Because. We know that. If we apply five. We get another function out of it. So we could then apply const five. Do you want to apply it to true? To.

ME

[00:23:57] True? To true. Yeah. So we've got a different type of thing. That's fair. Just. Get rid of the space in between. Oh yeah. I. I was just going to do. Const five true.

AMOS

[00:24:04] So. If we evaluate this. Um. Yeah. So we see that const five. Is going to substitute out. Its definition first. Um. Then. It's going to. If. If you. Look at const five here.

ME

[00:24:20] Did it just switch into free choice mode?

AMOS

[00:24:22] Uh. I. I wasn't pretty sure. Oh. You were in free choice mode. Yeah. Okay. Good. Um. Const five. Is going to sort of. Apply const to five. It's going to create that sort of. You know. Intermediary function. That we saw before. Um. So we can step const five. To lambda z dot five. Uh. When applying. That to true. We. Get out the other end, a five! Cause even though true is a different type, it's still... still a fine thing to...

ME

[00:24:48] yes. What's the time? Um... It is... half past-ish. Cool. How. How... how is... does this all make sense? You don't have any... cool. You want to move on to pairs.

AMOS

[00:24:58] Yeah... pairs... pairs and turn times on. Yeah... okay. So, so far... we've sort of just been having to create functions... and hope that they work... because there's no way to know that these are necessarily going to evaluate well or not... for example... if we try to pass in a true to an addition that would have no way to progress... however, uh... we have A lovely thing. Called types. That. Let us. Basically let. Uh. The program tell us. If we have. Correct. If. If the things will be correct. So. Types. Essentially. Um. Categorized. Our way to categorize terms. Essentially based on their functionality. On what you can do to them. So. For example. We categorize ints. Because we want to be able to add ints together. But we can't add an int. To a Boolean. And so. We separate them. Um. And we can also define then. Function. We can also give functions types. We can say that a function. Takes in something. And gives out something else. So. For example. Um. Our. If we wanted to say. Define a function. That did addition. Um. We would give it type. Int. Arrow. Int. Um.

And this is to say that. Add. To. To. To int. Int to int. Oh. Int to int. I always do this. Int to int to int. Yes. So add. Takes in. Two ints. And then returns an int. So this is sort of like. If you imagine. The. With currying. We're saying like. We take in this value. Here. And so. That's sort of. The arrow type. Lines up. Is sort of. Quite related to currying. In that way. We sort of. By having arrow types. Um. We have currying. So essentially. What this means. Is. This. We're saying. We give it an int. And we give it. And then. It produces an int to int function. Um. So if we. Apply to two ints. We would get an int. Which is. A different meaning. That if we say. Defined it like this. Um. Which means. We take it. We're. We're giving it an int to int function. And it gives us an int. Um. And those are. You know. Those. Uh. Mean. Different things. Yeah.

ME

[00:26:56] It's. It's implicitly. Um. The first one. Yeah. The brackets are on the right hand side. Uh. If that makes sense. It's. It's. It's. It's.

AMOS

[00:27:05] Implicit. So we have. Um. The. The. The main inbuilt types we have. Are int. Which is. Basically just numbers. And bool. Which is like a boolean. It's true or false. Uh. As you've seen those before, Uh. We can also. Uh. And we also have. Sort of like. Constructed. Types. So we can make a pair. If we wanted to say have. An int and a bool together. We could. Uh. Have a type of a pair. Which is sort of. A thing. And another thing. Uh. We only have. Pairs. We don't have tuples. Which are sort of like. Arbitrary sized. Um. Pairs. Although we could. Do that if we wanted to. Uh. We could also define. Our own types. So. If we wanted. For example. Maybe to. Work with coordinates. Let's say we have. We wanted to find a bunch of functions over coordinates. And we wanted to be clear that. The things we're working over. Are coordinates. They're pairs of ints. But we want to say. Only. Only operate over coordinates. We can define that. Um. So we could say that a co-ord type co-ord is just a pair of ints. And then we can do we can define computation using these types that we've defined. Um. So for example we could define um let's say we wanted to do a function that takes in a coordinate a scalar value and then gives out uh the coordinate the scalar value added to that coordinate so let's say we had we wanted that function so add uh scalar does anyone have an idea of what type that would have. So it's taking in an int, Uh, Co-ord, and it's giving out, Uh, another Co-ord. That, Um, is incorrect by five.

ME

[00:28:35] So what, What do you think, What do you think we'd write, Based on that information.

AMOS

[00:28:39] Yeah, So if we imagine, Add is int to int to int. This is a big step. Yeah, This is a big step.

SPEAKER_4

[00:28:46] Yeah, Um, Just trying to see if, How we're doing, Uh, Co-ord would be, Int to int, So Sorry, What was that? But then co-ord itself is int to int.

ME

[00:29:03] Co-ord is, A pair of ints. It's not int to int, Int to int, Would be, A function, That takes an int, Returns an int, It is, It's, It's a pair of ints. So co-ord is fine, Yeah, Co-ord, So int to co-ord, To co-ord. Would be the type.

AMOS

[00:29:18] Yes. Yes. Because we are producing a co-ord. On the other end. So we want to. This is sort of like. You can. You can often think of the last thing. Here. To be what we want. At the end of the function. And everything before. Is kind of like. Things that would be inputted. To the function.

ME

[00:29:31] Also. We. Even if we want. We could. We could get the last two things. As well. We could say. If you give it an int. Say. If you. If you define the body. Like. Of. Add. Yeah. Okay. The recording's still going. Um. Does that. Yeah. Yeah. Yeah. But if you do. Like. If you unpack the co-ord. Oh yes. Of course.

AMOS

[00:29:55] Um. So. Because we've used. The keyword type. Um. We haven't really said. That co-ord is a new thing. Co-ord is kind of. Just a shorthand. For. A pair of ints. So. When we define the function type. We can. Implicitly. Sort of like. Unpack. The coordinates from it. Because this is essentially. Int to int. We could sub that up for that. And when we. This is our co-ords. And we. We can. Sort of. Implicitly. Unpack the coordinates. Um. Because they are just. A pair. So. They've just. They've been constructed like this. So long as. The thing on. Over here. Matches that construction. We're okay. We could also. If we wanted to. Just have c. And this is now. This itself is a pair of ints. But, uh, because we want to access the individual components, we will, uh, sort of implicitly unpack them here. Um, so then, I believe so. Cool.

ME

[00:30:53] Um. You click outside. Or. Yeah. Yeah.

AMOS

[00:30:55] So. Um. Given this is our functional definition, don't have an idea of what our body is going to look like. So we have add scalar, so s is our scalar, and then x y are our coordinates. Um, and we need to construct some new coordinates that are the previous coordinates, what that might look like, it's another failing large step. But we're just trying to see if you've got any intuition, yep.

SPEAKER_3

[00:31:20] Added five.

AMOS

[00:31:21] Or added s.

SPEAKER_3

[00:31:22] Or are you going to. Sorry. I added s.

AMOS

[00:31:23] I keep saying five.

SPEAKER_3

[00:31:24] I don't know why. It looks like a five. Um. Open brackets. X. Plus x. Uh. S. Yep. X plus x. S. Comma. Y plus s. Yep. Yep. Exactly right. So.

AMOS

[00:31:45] We. Uh. Just do code. And we're going to do. We're going to say. We're going to add. Scalar. We're going to add. One. To. The coordinates. Uh. Two. Three. Let's say. Um. And we're going to. Just evaluate that. Basically. Recurse. Add. Issue. Uh.

ME

[00:32:04] Do I need to just. Parenthesize. Some of this. What? That's supposed to only happen. When. I have no idea.

AMOS

[00:32:15] I think. The time checker. Is not helping us today. In fact. I. I said it would be. Helpfully good. So. I'm not evaluating. That. I'm going to. Parenthesize. These. Just see if it.

ME

[00:32:28] I think it's that. Because I think it's trying to add.

AMOS

[00:32:31] Something. With the.

ME

[00:32:32] I think it's. Yeah. I think it's overflowing. And trying to evaluate. This. Um. What's that? Recurse. Add. Issue. What the hell. Is. Recurse. Add. Issue. I just. I've. Never. Seen. That. Before. I don't. Even. Remember. Writing. That. Type. What. Is. Recurse.

AMOS

[00:33:00] Add. Issue. I. Don't. Know. You. Wrote. The. Language.

ME

[00:33:13] Mean. That. Oh. Um. Let's. Just. Just. Okay. Let's. Try.

AMOS

[00:33:25] We're. To. Do. Chord.

ME

[00:33:27] We. Might. Not. Be. Able. Do this one and let's do but you oh it might be I was pretty I did have quads oh you did have four twenty what is twenty I need an indicator for that here okay let's abandon this one okay okay it's fine it's fine uh factorial factorial this should this should sense I guess yeah so this is a slightly more unique I can you okay so then we can look at the definition yeah so we got the whole thing sorry for the ad hoc thing of a jig okay here we are something completely now for something completely different so

AMOS

[00:34:32] This. Is. Our. First. Introduction. To. Choice. So. You're. Probably. Quite. Familiar. With. Having. Like. Branching. If. If. Statements. Where. Like. If. You. Do. One. Thing. Else. You. Do. The. Other. Thing. This. If. Function. Is. Essentially. Doing. A. Very. Similar. Thing. So. We. Give. It. A. And. Then. Basically. Two. Of. Anything. Else. That. Could. Be. An. Entire. Other. Function. That. Could. Be. Like. Another. Function. Applied. Or. Something. A. Here. Is. Essentially. It's. Generic. So. We're. Saying. A. Can. Be. Any. Type. It's. Just. That. These. Types. Have. To. Be. The. Same. So. A. This.

SPEAKER_3

[00:35:03] Has to be the same, so these would also have to be an int and if is essentially taking in a bool and then two other things and based on the value of that match does I think it kind of speaks for itself a little bit what do you do you see how that represents if but does it make sense why were what were doing with true and false yes so like if the condition is true then youre doing could I relate it to like a paralyzed function yeah and based on sort of what this ends up evaluating to we can take different.

AMOS

[00:36:32] Parts. So. In. This. Case. We've. Said. True. And. False. Which. Is. For. If. Makes. Sense. Right. If. True. We. Do. This. Thing. It's. False. We. This. Thing. We. Also. Use. Match. On. Any. Other. Type. Of. Thing. In. The. Thing. So. For. Most. Things. We. Can't. You. Do. It. On. Functions. But. We. Can. Do. It. On. Numbers. If. We. Define. Our. Own. Types. For. Lists. Or. Something. Else. We. Can. Essentially. Match. Different. Patterns. Of. Those. Things. And. So. Long. As. Our. Patterns. Cover. Hypothetically. Cover. All. Possibilities. Then. It. Will. Be. Covered. By. This. We. Could. For. Example. Match. Over integers and say zero do something then and do something else which is basically to say that for zero there is a specific action and for any other number you do something else which is useful and the other thing you might want to do would involve N so you can have access to N so this is useful for recursive things for example want to say recurse down to zero and then come back up which is what fact is doing under the hood so you can essentially read this as if N is less than

equal to one then one else. N Times. Fact: N Minus One. Does this definition of Factorial make sense? Does it seem like it lines up with you understand the factorials to work so we are basically saying if N is less these in the world where we so function is quite expensive and not usually the best way to do the. Them down somewhere and come back to them, we're just doing substitutions so we can just keep on substituting things and we're not creating like ridiculous resource requirements because we're just creating terms we're not like having to like remember where we were in function execution previously, we can just sub in our definition to fact here and it's as simple as that. So if we go to evaluate this um we'll hopefully see that so when we apply function fact to five we are just taking five and subbing it into this function body so n is here and we'll just put we'll sub out for this with five in place of all the ends, so we now get um if five less than one one five times fact five minus one um we see something here uh which sort of uh this says lazy. You might have wondered what that means um essentially in a functional language as we saw in free choice there are lots of choices you can make on how to evaluate a term. For example, here we could evaluate five less than or equal to one. We could evaluate 'five minus one'. Um, if this was a number, we could apply the multiplication there are lots of things that we could do and in order to have a program be able to evaluate this, a computer to be able to evaluate automatically, we need to define some sort of precedence for how we evaluate things. Functional languages do this lazily, so they will essentially apply whatever the highest level, top-level function they can apply is, and they will do that. They won't apply 'fact five minus one they won't go sort of down and figure out what fact five minus One is first, they will um see the sort of highest level thing that they can do, then see where they get to because, and this is generally efficient because there's a chance that we apply if and maybe some of this information is no longer relevant and we don't need to do it and we don't need to evaluate it so why would we?

ME

[00:40:24] You could force it to evaluate forever by constantly evaluating fact five minus one then you could have then the else branch of this, you could force it to evaluate that and it would just evaluate forever without checking whether it's less than one yeah if you don't do it kind of lazily, so if you. Deliberately, do a recursive call over and over again, you can force it to happen forever; it's lazy because we we essentially don't evaluate things unless we need to.

AMOS

[00:40:51] Um, so we don't need to evaluate five less than one now, because we can apply if so, we're not going to do it. We don't evaluate fact five minus one now, so we're not going to. We're just going to play with this, so we're just going to do that with this one and we're going to say if we do match which now we see our first sort of match in the wild; um, we have matched five less than one true false blah blah blah and this is our result on the other end. We've just substituted in um n less. or equal to n less than or equal to uh five less than one one five times fact five minus one we've just substituted that into the body of if um which is then gives us this match statement. I said before we didn't need to evaluate 5 less than or equal to 1. We now do, because this match is expecting a true or a false, and it doesn't have a true or a false right now, so it has no choice but to evaluate this less than or equal to.

ME

[00:41:40] Is everyone happy with this step that we made between 1 and 2? All the things have been swapped out in the way you'd expect.

AMOS

[00:41:49] So we've sort of applied the function all at once. So previously we've sort of looked at currying where we've done it one at a time. The computer is smart enough to see I need three things, I have three things, I'm going to do all of those right now and just step to when this function is fully applied. But if we wanted to, we could apply it partially. For example, if you wanted to make a function that was like 'if' 0, you could define that by applying 'if' to n equals 0 or whatever, or something equals 0. And then you could have an 'if', which essentially, like, you know, is it specifically, yeah, we'll always check if something is 0, we'll always do this, we'll do that, which can be useful. But right now, we've just applied it all at once. We could apply it sequentially under the hood. Again, we've got a star here. We're kind of like saying, really, what has happened is we've substituted the first thing, the second thing, then the third thing. But because we have a star, we're just saying we're doing some number of steps to get here. So we now apply less than or equal to 5 than 1, and we get out false. Which makes sense, 5 is not less than or equal to 1. And what do we think is, when we match the pattern false, what do we think it's going to be, we're going to reduce to? So the next thing is we're matching on this false. Does anyone have a guess as to what we will, what the result of that will be?

SPEAKER_4

[00:43:06] 5 times back 5 minus 1.

AMOS

[00:43:09] Yeah, exactly right. So we see false. Our false pattern is down here. And so when it sees that, it will just directly substitute for this. So we'll just, that will be what we get out the other side, match. We don't, we sort of have, take one of the forms in the same way that when we substitute, we just sort of shove our date thing into the next, shove our values into the variable slots. When we do a match, we just sort of take whatever the next thing is and we forget about the rest of it. We don't need it anymore. And then we get down to 5 times fact minus 1, which means we finally have to apply fact and we have to keep moving. And this will go on for a very long time. I guess I could just click through.

ME

[00:43:46] Yeah, you can just click through. There is more. I'll add a, is that kind of working as you'd expect? You end up with 5 times 4, because it can't do any of these multiplications until it has numbers. Yeah.

SPEAKER_4

[00:43:58] So eventually. It's just going to be expression at the end. And then it's going to do. Yeah.

ME

[00:44:02] It's going to collapse downwards in a really nice way.

AMOS

[00:44:05] Because of how we've implemented it, it's going to; you could implement fact, I think, in a different way where you would like multiply some things earlier, maybe. Or maybe you could, maybe you have to. I just wonder if there's a way you can implement fact. It doesn't matter. So eventually we've, we get to 5 times 4 times 3 times 2. You could do it using fold-downs. Yeah. Yeah. That would, about like, and you could do that, whatever. It doesn't matter. So then we do 5 times 4 times 3 times 2 times 1. And we have this, and then we will just apply all of these and get to 120 at the end.

ME

[00:44:35] I think that's really what's nice, the way it flattens like that. Yeah.

AMOS

[00:44:38] I just really like, I like functional languages, believe it or not. They're quite cool. So, like you kind of, instead of in a, like in an imperative language, when you call these functions, you'd be getting return values. You would like call into this and you'd get a return value out, and then you would apply it. Whereas here, we're kind of keeping all of the stuff in place as it was until we're done. And then we do the multiplication, which means there's less crazy stuff going on. And it means that recursion is actually efficient and effective. And that's factorial. Does that sort of make sense? If we were, let's say, you want to just quickly flip to the playing cards? Yeah, I was, I was, I was like, what do we want to do next? So we've looked at one way to define. So we, we, we, we defined quants as our own type, which is one way to find types. We can look at that and that didn't work, but that's fine. That's not a problem with functional programming. That's a problem with the Kieran's programming.

ME

[00:45:35] There's also this, this, it does work. I promise. Pairs do work. Pairs do work.

AMOS

[00:45:42] But those are product types. So you can think of those as you need to have an int and another int.

ME

[00:45:50] In terms of sets, a type is a set of what values it can take. So it's a product type because it's the product of the two sets in it. So a pair is a product of the two sets of the types. Does that make sense? Does that make, yeah.

AMOS

[00:46:10] And, and then a term that hasn't reduced will inhabit a type if it either, if it loops forever or, well, sorry, w a term inhabits a type. If, if it evaluates it, its evaluates to a value of that type. So when we reason about the types of a term, we're basically saying like, this will evaluate down to a value that inhabits the type. We could also have some types. So we've looked at product types. We're now going to look at some types. So these are.

ME

[00:46:39] These are, yeah, these are similar. Just because we got mathematicians in there, these are called algebraic data types because you can either have kind of where it's the product of the types or in this case it is the kind of addition of the set of the types because we've got it can either be hearts or clubs or spades or diamonds so that's why these are called union types.

AMOS

[00:47:09] While we read a pair as like we have this and we have we have an int and we have another int, we read this as a suit can be a heart or a club or a spades or diamonds. These are called constructors but let's just imagine them as, they're constructors and these constructors don't take in any arguments. They're like enum variants in this case. They're essentially enum variants in this case. In this case, the constructor kind of means something because it takes in another value but for example if you have a set of diamonds and you have a set of diamonds and you have a set of diamonds for example for suits we just have hearts, clubs, spades, diamonds. It's one or the other and then a card where you can view as a pair of a suit and a rank. It has one and the other. We can sort of have here sort of a closer look at what pattern matching can let us do. So let's look at the is black function. So this takes a card and it returns a ball that indicates whether or not the card is black. So you know you've got clubs and diamonds, clubs and sorry not clubs and diamonds, clubs and spades are black, diamonds and hearts are black. So this is a set of diamonds and this is a set of diamonds and hearts are red. What if we want to know which thing a card fits into? And the way that we've implemented this is we pass that we match on the cards. We take in a card which we've said is a suit and a rank and we match on it. And so obviously there are loads of forms of this right. We could have like hearts jack, hearts queen, hearts ace etc etc etc. But we basically said we don't care about what the second thing is. This underscore basically lets us say that a card will match this if it has hearts in its first position. So you can imagine this covers the cases of hearts jack, hearts queen, hearts king, hearts ace, hearts num, seven, num, eight, num, whatever.

ME

[00:48:47] Does anyone have any questions about this program? This is just like a, do you just want to run it?

AMOS

[00:48:54] Yeah let's let's let's run see what this does. So, we've also this does is black, this gets a value which does some interesting stuff with the instructors but yeah both of these are sort of more general uses of pattern matching instead of pattern matching over balls or pattern matching over an entire sort of data type that we've constructed that we've defined and based on how it's, instead of based on the values of the thing we're sort of evaluating based on the form that it has. And so if we evaluate this, we're doing get value of hearts num 10. So this we can sort of read as the 10 of hearts. It's suit hearts and it's a number card, so if we then apply that, we would go into a pattern match. So is this what you'd expect? So we have hearts num 10, we're sort of looking to see which pattern it matches and it matches this pattern because it's got a num n here and here in this case n is 10 and we don't really care about what's coming first because we're just

looking for the value.

SPEAKER_4

[00:50:00] What are those like straight up lines on the left? What are those straight up lines on the left? What's the sign of each of those things? Like what does that represent?

ME

[00:50:06] It's a case. It's just a piece of syntax. Just saying.

AMOS

[00:50:10] It's just to say like this is one case. This is another case.

ME

[00:50:14] Can I enter a remote code?

AMOS

[00:50:18] Oh yeah.

ME

[00:50:19] It doesn't work. Might not work. It doesn't have an alphabet. Ah, you can try it.

AMOS

[00:50:30] So yeah, this matches to this one. These are the colors. These bars are kind of just to indicate, yeah, it's a different case.

ME

[00:50:35] It's just to separate them, it's just a convention, it's not, it could be anything.

AMOS

[00:50:39] You could, and you could even have no thing here, and you could just have it be like it's defined by new lines, but this is a little bit easier, maybe you want to just cut the I just think it makes it look nicer. Yeah. They all get lined up, it's pleasant, etc. And so if we then do num10, it would take this case and give us back n, and if we step that, it will do just that. And if we gave it, say, let's say we had like the ace, the ace of hearts, it will match differently.

ME

[00:51:06] It will, it will apply, and we'll see it matches the ace case. Does anyone have any questions about this program?

SPEAKER_3

[00:51:15] You have to ask. I have something on the syntax. Yeah. Yeah. You currently have the straight line defined as new case, as well as or. In data, so could you use them interchangeably, or would the straight line only function as or when you have data before it?

ME

[00:51:41] Like we use a-They're not, they're not really operators, the straight lines, they're not operators. They are pieces of syntax, like a semicolon. They're not, they're not operators. Right. Yeah. They don't mean anything mathematically. They just allow the parser to be like, oh, that's one thing, that's another thing. This is one case.

AMOS

[00:52:06] This is, in a way, they're kind of similar, and like, suit has, there are several cases of being a suit, and they're separated by the bars, right?

SPEAKER_3

[00:52:14] So it's kind of division of variance. So the way that I should understand it is more so that whenever I see a new line, it's a new case, rather than-Well, whenever you see a new line, it does different things.

ME

[00:52:26] Depending on the situation. Yeah. Like, I don't know, like a colon. A colon could mean different things in different places in programs.

SPEAKER_3

[00:52:34] Yeah. So like, if I was looking at the main body of code, where it says get value card, if I collapsed line 10 onto line 9, and you still have that vertical line, it would still work. Yes, it would still work.

ME

[00:52:47] But yeah, but you wouldn't, I guess, because it doesn't look very nice. But you could.

SPEAKER_3

[00:52:52] Well.

ME

[00:52:54] Oh. Does it not work? No. Maybe it doesn't work. Oh, okay. I guess I expect a new line. I mean, I've just never done that, because it's not really, I just, I like how it looks.

SPEAKER_3

[00:53:03] There are, because it doesn't really make sense from a programming perspective, rather than a mathematical perspective. It makes sense mathematically. It's just, if you have that working as two different things in different places, then it could be a bit confusing.

ME

[00:53:23] Yeah. I see that. I think Haskell doesn't-Haskell. So the language this is based on does do this. A lot of, so this is, Haskell is a functional programming language that is kind of the convention. It was made specifically to be a functional language that's quite acceptable to everyone. It was a language made, designed by committee, literally. And they did that. That's

why I'm doing that. Because also, our first year CS module teaches Haskell, which is why I wanted people who didn't take CS, because they didn't know Haskell. So, because this, I envisioned for this to be used to teach Haskell. I could have it be some sort of different piece of syntax, like in Rust you've got a comma, for example, between cases, but that would make it seem like it's a pair somehow. I think all the, I've kind of used all the characters, I mean, I guess I could have semi colons at the end of those lines, or I don't know, but bar is nice to me. It doesn't make sense. It kind of does, because it's the separation of variants. It kind of does make sense. I could make, I could remove the restriction and make it not rely on a new line, but that's just the way I passed it.

AMOS

[00:54:44] I get what you mean. Like, if we had a, if we were doing a match on like a suit, we could end up with something that looks very similar to the definition of a suit, because we would have like a bar, bar, clubs, bar, space, bar, dot, like, it would be separated like that. But yeah, it is. I guess, yeah. I guess, yeah. It is the convention, but it is.

SPEAKER_4

[00:54:58] You could just add like an explanation that that's what it's doing, if you're willing.

ME

[00:55:01] Yeah, I'll have a, there will be a thing at some point, which will just say, these are the pieces of syntax. Yes, it's what they do, and this is why they do them.

AMOS

[00:55:09] So yeah, we have ways to define. I just can't be bothered for a dissertation project. I don't have the time. We have ways to define like product types. So we have ways to define like lots of data together. And we also have ways to find out when there's lots of different kinds of something, which will be very useful, because sometimes like it's an int or it's a bool, and you want to be able to separate out those things. You want to be able to separate out those cases, and be able to like know when it's a thing with an int, when it's a thing with a bool, and this is kind of what this is doing. And that's useful.

ME

[00:55:35] So I think we've got to wrap up the lecture side of it. So yeah, so the point of that was for you guys to learn a bit about functional programming and to see if the tool is fit for purpose, kind of. But I guess it's a bit difficult, because you guys don't know what it's like to learn without the tool. You learn through a series of lectures and slides, and it's very difficult to learn, I would say. So yeah, anyway. Cool. I mean, I guess I'll start with you. How did you find that? I mean, we've already had this conversation. Yeah, we've had this conversation, kind of.

AMOS

[00:56:24] Nothing crazy different. I think I did know a little bit about it. I think what you've noticed this time is the evaluator, the type error, but also the step-by-step evaluation is incredibly good for explaining currying. Oh, okay. Because we didn't try that last time. Well, yeah,

because everyone's doing currying. But yeah, I think it's quite good for showing what I found, and it was like, oh, I can point at this and say, look, this is what it means when it curries. Because I find it kind of hard to explain in words sometimes. Yeah, it's good because we haven't tried that yet. It's just sort of like, but you can see this is the function. It is still a function. It is still a function when we do the evaluation, and kind of show that currying is what it's doing anyway. And if you do that part way, you're just sort of getting in the middle of that evaluation. That's the only specific thing I've noticed this time.

ME

[00:57:09] Is there anything different to last time? I mean, there's the disabling the type checker. Do you think it's useful?

AMOS

[00:57:13] I think it's very useful because sometimes the type checker goes wrong, and sometimes, I don't know, if you wanted to introduce, if you want, I think it's not very useful. It's nice to be able to introduce the language and types separately because they are different concepts. They're two languages. And like, I don't know, interestingly, there are things that are hard to give types to that do evaluate fine. Like the Y-combinator.

ME

[00:57:42] Yeah. I got that as an example.

AMOS

[00:57:43] Or even just like the example where like you took, or like, there are some examples of like where like you wanted, I don't know, if some, like, there are some things like, like, where it's just nice. I'm not going to say them right now. But I think that was, I think it's nice. You can have a bit of both.

ME

[00:58:02] Yeah, Y-combinator is an option now.

AMOS

[00:58:03] It lines up maybe a bit closer with like third year units because of that. Yes. Sort of do you.

ME

[00:58:09] That's what I was hoping, yeah.

AMOS

[00:58:10] In TLC, you introduce like the language and then you introduce types afterwards. Yes. Cool.

ME

[00:58:17] SPEAKER_3, what did you, what do you think of functional programming then? What

do you think of functional programming? I think of functional programming. Oh, yeah. Yeah. Oh, by the way, be honest.

SPEAKER_0

[00:58:27] I guess right now it's less intuitive to me than imperative programming. Yeah. Just because imperative programming is what I'm more used to. Of course. I do, I mean, I do find it interesting from like a, like a mathematical perspective. Yeah, I'm interested in it. It's cool actually. I did, I did also like the, I liked the free choice evaluation. It made it, it did make it easier to understand, like when you have that function, when you like define the function which returns the first of its inputs, and it's like you evaluate it at five and seven, and oh, it's replacing it with a function that always returns five and evaluating that at seven, seeing that go step by step made it a lot, made it, yeah, it made it easier to understand what it was doing. I liked that. I found that. I, yeah, I found that interesting. That was cool.

ME

[00:59:23] Do you find the tool kind of... Nice to look at, I guess is one thing. Yeah. Yeah. Cause, cause that's not, cause that's not a thing I think about. Cool. Do you? Cause I'm a computer.

SPEAKER_0

[00:59:34] I do like it more in dark mode. Yeah, I agree. Yeah.

ME

[00:59:37] I just, I wanted to do light mode just cause I made that yesterday and I wanted to try it. No, that sucks. Yeah. Also, I made it like fine. Yeah. Slow. Anyway. I'm very proud of that. Yeah. It doesn't, it doesn't add anything at all. Fancy features. Yeah. I found, yeah. So would you say that it's kind of pleasant to look at? I guess. Yeah.

SPEAKER_0

[00:59:59] I think it's quite, I think it's good. I mean, yeah. Yeah.

ME

[01:00:02] Do you understand what each kind of thing does and why it's laid out the way it is? Would you say?

SPEAKER_0

[01:00:09] Yeah. I think so. I think I do.

ME

[01:00:12] Yeah. Yeah. Cool. Do you have any kind of user experience wise? Obviously, you haven't used it, but you are a user as somebody who's been in a lecture for it because that's kind of the desired use case. User experience-wise, do you have anything you'd like to add? I'll be asking the same questions to everyone, by the way, aren't I? Yeah. Feel free to, we can come back to this if you want. Yeah. We can come back to it. So what do you think of the language?

Language? I guess you haven't seen other functional language. Yeah.

SPEAKER_0

[01:00:55] Yeah. I mean, I guess I like it. It's interesting. Yeah. It's difficult without a point of comparison. Yeah. But I do. Yeah. I mean, I do like it. It does make, it does make like some sense to me now. It's quite. Yeah. And again, I do find it interesting as a mathematician. Yeah. Yeah. It's cool. I like it.

ME

[01:01:24] Okay. I'll just go around, but feel free to jump in with anything. I don't think we're not particularly pressed for time. SPEAKER_4, what would, so what do you think about functional programming?

SPEAKER_5

[01:01:36] It's cool. I think the tool is very useful. It's a bit hard to see how useful it would be for learning it. Yeah. I don't know what kind of the purpose of learning it is and how it's.

ME

[01:01:50] Yeah.

SPEAKER_5

[01:01:51] How that works.

ME

[01:01:52] Do you, do you have any questions? I'm just going to run through the, I'm just going to do the filter example quickly, because I think this is a nice example of something that you might be able to relate to in other forms of programming. You may have used filter or map or reduce or something like that in Python, and these are concepts that come from functional programming where you have a list, which is defined like this. It's a little bit of a confusing definition, but we can turn the prelude back on. Prelude back on. Oh, no. There we go.

AMOS

[01:02:27] My laptop is kind of broken.

ME

[01:02:29] Yeah. But if you run through this, you can see it filters a list of numbers from one to four, and you end up with four, the numbers that are even. And you end up with this, which represents a list of the numbers two and four. This kind of concept of applying a function across a list. It comes from functional programming, I guess, is an example of something that is used for outside of mathematical reasoning, which it is, which is a big part of its use. I would say, I just thought I'd add that.

SPEAKER_5

[01:03:11] I really like the coding part and the sections on the right, in terms of seeing how it

goes from one to another. I think that was really useful. In terms of learning it. Yeah. And the free choice variation. I also agree that it shows nicely what the functions are actually doing.

ME

[01:03:34] Yeah. I was thinking of removing it, so that's good to hear. I mean, that's why. That's the kind of stuff I wanted to get some people who didn't know about functional programming to say, 'oh, yeah, this is actually really useful.' Because I didn't think it was. What do you think of the kind of user experience of the, so you said you liked the input and output split.

SPEAKER_5

[01:04:01] Yes. Yeah. I really like the interface and I like the example programs. Go through it and write something yourself. I think that's really nice. Yeah. I also prefer document.

ME

[01:04:17] Yeah. Me too. Me too. But on a bad screen. Yeah. Yes? All right.

AMOS

[01:04:25] So I think you did.

ME

[01:04:34] Cool. So I think, yeah.

SPEAKER_5

[01:04:45] So I think a word in the game environment, we'd love to know, I think like the, it could have a little bit of an internet which might help you to pick up performance in like video for example, and I have an internet connection at this moment. Think of it like, well, that sort of thing. And it seems, I mean, I didn't find myself asking questions about every single thing in the language, so I think that was, that's good.

ME

[01:05:01] Okay, cool. Do you have anything else you'd like to add to, well, the talk? Anything? More questions?

SPEAKER_5

[01:05:13] No, I might, yeah.

ME

[01:05:15] Sure, yeah, feel free to jump in. SPEAKER_5, what are your thoughts on functional programming?

SPEAKER_4

[01:05:22] Yeah, it seems pretty cool. It's cool to see a language that's like, I don't know, I feel like if I was going to do some maths in my head, this is how I'd actually do it. Yeah, I don't know, it's quite cool to see, like, the process, how it actually does it, because sometimes when I'm doing,

like, Python or something, I'll write a function or, like, an inbuilt function, I don't really know how it works, I just know that it does work, but, like, I don't know, it's cool to see all the steps in it. Yeah. All the steps that it does. Nice. You know what I mean?

ME

[01:05:52] Cool. In terms of the UX, how have you found watching this as a lecture, and do you have any, do you get why everything is as it is, I guess?

SPEAKER_4

[01:06:05] Yeah, I think it's done, I think you've done a really good job of, like, making it, making it quite, like, beginner-friendly for almost, because, like, it's all, like, quite easy to, easy to read, and, like, Yeah. I don't know. I was thinking that, like, the language itself is quite intuitive as well, like, so, like, none of it is very, none of it's, like, not clear, but, like, you can quite, you can understand it quite well.

ME

[01:06:28] I aimed to, like, obfuscate as little as possible. The only things that are built in are, like, add, subtract. Yeah. That's it, in terms of functions.

SPEAKER_4

[01:06:37] Yeah, I probably like that with, I don't know, yeah, it just, it works well.

ME

[01:06:40] Cool.

SPEAKER_4

[01:06:42] I was gonna ask, actually, was this, was functional programming, like, was it quite early in the development, in the sense of, like, when languages were being created? Yes, absolutely. Because it seems like quite a simple Yeah, so, have you heard of the Turing machine?

ME

[01:06:55] Yeah. So, the Turing machine was a, kind of, thought experiment is the wrong word, but it's a theoretical tool to reason about whether something is able to be computed. So, that was in the 1950s, after, when people wanted a way to, I mean, of course, imperative came first, where, on computers, as it is, so you'd, executing a series of instructions is logical. But in terms of mathematical reasoning, Alonzo Church came up with the lambda calculus in 1936. Wow. It is very, very old, and they proved, in 1954, that if, Church and Turing, if they used the Turing, Church-Turing thesis, that they are equivalent, they are equivalently powerful. And the Turing machine is, kind of, a more traditional, yeah, list of instructions, but the lambda calculus, which is exact, this is pretty much lambda calculus. The stuff we saw at the beginning is lambda calculus. The only things I've added are types, I've added match expressions, and I've added kind of label substitution, the abstractions and application and, you know. Variables and all that, that's lambda calculus. So that all came from 1936? That is it. That all came from 1936. That's crazy. It is really, really old, and it is equivalently powerful. You can, if you can compute anything, if you

can, lambda calculus, if you can boil something down to lambda calculus, it can compute anything. Cool. I guess it's, that was, yeah, it's very, very early on. Very cool, yeah. But it's not, in terms of kind of a. Practical history, you might have heard of Erlang, big in telecoms industry. Sorry? WhatsApp uses Erlang. Yeah, WhatsApp uses Erlang. And Jane Street uses OCaml, which is why they feature it in Cambridge. But outside of that, there's not really a huge history of practical use. I wouldn't say. I don't like that. No. Does that answer your question? Yeah. Yeah, that's good. Do you have any comments on the UX, the how you've found kind of watching this, the language itself, anything? I don't know.

SPEAKER_4
[01:09:31] I don't think so. I think pretty much just what you two said, like free choice things, very good.

ME
[01:09:38] Yeah. Cool. Okay, we'll move on then. Do you have any questions? Yeah. So, we start with functional programming. What do you think of functional programming?

SPEAKER_3
[01:09:50] Functional programming is pretty cool. I can definitely see how it's really useful for mathematicians to see how things are being substituted in. Coming from kind of a programming background though, I find it easier to understand it in terms of pointers. Where you're kind of telling the codes, you know, in terms of, you know, in terms of what the code is. You have a dictionary. It's like assigning an array to a value. Yeah. And then you're telling it to, oh, just go and look for this value.

ME
[01:10:26] That's what it is out of the hood.

SPEAKER_3
[01:10:27] Yeah.

ME
[01:10:30] It's a set of definitions.

SPEAKER_3
[01:10:32] Yeah. And then you're going and just fetching the definition for it.

ME
[01:10:40] Yeah, that's reasonable. That's a reasonable intuition for it. Yeah. So, did you, did you understand kind of what we were doing and then towards the end why it was doing the things it was doing? Yes. Yeah. That was the intuition you used for substitution.

SPEAKER_3
[01:11:05] Interesting. So, it's pretty much the same. You were, I would essentially call it a

substitution. You compare it to just defining a function and then asking it to take values and then giving you something back.

ME

[01:11:23] Yes. That's exactly what it does. Yeah. That's the essence of functional programming. You are, you give it a thing and it gives you something back. There's no other, there's nothing else done.

SPEAKER_3

[01:11:33] Yeah. Yeah. So, it was pretty, really intuitive and it was really nice to see how it does things step by step. Yeah. So, it's, it's very intuitive rather than computing it straight away and then either just giving you one answer at the end and you don't know what it's actually doing in the middle. Yeah. So, that's a really, really nice part.

ME

[01:12:00] Okay. Cool. What did you think of the UX of the system? How did you find watching this as a lecture?

SPEAKER_3

[01:12:07] UX is very intuitive. Okay. Cool. Yeah.

ME

[01:12:13] Fantastic. I know. Very good. Very intuitive. Yeah. Thank you. I appreciate this. And then the language itself, you had a comment on the kind of different usages of the bar.

SPEAKER_3

[01:12:26] Yeah. And then there was a second place where I was a little bit confused about, the place where you were defining types, where it was, you were taking a type and then returning Yeah. I don't know what it was. But.

ME

[01:12:48] Was it like a type of instructor where I had. Yeah.

SPEAKER_3

[01:12:51] Where you're taking in an integer and then you were giving out something different. Yes. I couldn't really tell the difference where the arrows meant something different. Yes.

ME

[01:13:05] So, that was kind of, have you, it's quite a fancy feature in some languages. Yeah. It's called the tag that, you know, where you have this kind of data, which in the cards instance was an integer potentially, and then the other ones, they have no data. So, you have this data along with the tag of the name of the variant. So, it could be, I have something of this type. It's either a card with a number or it's a jack, queen, king, ace. Yeah. So, that's a tag enum. The, the, the word num in that case, I had a constructor that said num int. Num is, there is a constructor.

It is a function which has type int to, int to rank, where it takes in an int and returns a rank. And the rank is num int. Does that make sense? So, they are, it is different to kind of function syntax. That's a constructor. It's implicitly defined a function, I guess. It's very, very complicated to explain. If we'd have had more time, but we would have gone on to that. Okay. I think, I'm not sure I could make that clearer. Yeah. So, what's the group variable we're depth of work and what variables would mean in the time available?

SPEAKER_3

[01:14:47] I might be asking a different thing. So, it was more so, I was confused about what variables it was taking in and then what variables it was giving out, because you had multiple arrows there.

ME

[01:15:03] Well, it doesn't matter which one to, I mean, the point is that it was a label. And the label has this type, because say if it has, say if it's int int, there's a If you give it an int, it'll give you a function that goes from int to int back. If you give it two ints, it'll give you just an int. So it doesn't matter what's an input, what's an output. That's currying. It's up to you what it takes in and what it gives out. Do you get what I mean? It's right associative, so if I have, like, I don't know, add three, which is x, y, z, and then that's x plus y plus z. This has type. If I bracket it properly. So I give it one, then it returns this. If I give it two, then it returns this. Does that make sense? It's not that well defined as inputs versus outputs.

SPEAKER_3

[01:16:17] No, that's fine. I understand. It's a definition thing.

ME

[01:16:21] Yeah. So that's a functional language concept, I would say, rather than a design choice in my eyes. They are different.

SPEAKER_4

[01:16:30] So depending on how many ints does it give it, it'll either give you a function. Yeah. Which two inputs, or one input, or just a bracket.

ME

[01:16:41] Yeah, if I gave it two and three, I would get a function that adds five to things. Yeah. That makes sense. Or if I give it two, it would get a function that adds two to two things.

AMOS

[01:16:53] There are lots of libraries in other languages that steal this feature. Yes. There's, like, a lot of, I don't know, I've had this before, a lot of libraries that add partial function execution where you, like, give it one of the values and then hold that function around until you need to use it. So I think it's really cool.

ME

[01:17:09] Yeah. It's a different concept. Does that answer what you were saying? Yeah. Do you see that that's a part of functional language rather than a...?

SPEAKER_3

[01:17:19] Yeah, because I don't have an equivalent for that.

ME

[01:17:22] There is no equivalent for that. Yeah. This is a branch functional concept, which I think is really cool. Yeah. There isn't a strict line of inputs and outputs anywhere. Yeah. Does anyone have any other questions or comments on top of this or just in general?

SPEAKER_5

[01:17:42] Yeah, I was going to say, I like, on the right-hand side, you have this at the top, it said what the step was doing. Yeah. Which I think was really nice because if you're looking at it, it's kind of a self-study tool as well as just playing around with it. Yes.

ME

[01:17:58] So that's my secondary use case. Yeah. Unfortunately, it's a bit of an unused use case, not abused, but a neglected use case because this use case is the primary one, the lecturing use case, where I don't need to worry about adding help menus and stuff because you've got someone to explain it to you.

SPEAKER_5

[01:18:17] Yeah, but it was really nice to see what the step was actually.

AMOS

[01:18:20] Yeah. And also, I think from the perspective of teaching it, it means you don't have to explain it, it's a bit easier to explain, it's like we are saying this, we are writing this.

ME

[01:18:28] You don't have to explain it, you can just say, does that make sense, I guess. Yeah. I'm glad you like that.

SPEAKER_5

[01:18:33] Yeah. I think for someone with little experience with functional programming, it's hard to understand if I got it, but I think I got it. But yeah, if it's something to begin with, it's just a good path. Yeah.

ME

[01:18:49] This is kind of, yeah, hopefully this is either alongside a lecture course or with reference to, I don't know, maybe the Haskell book or something like that. It's not, it's not hugely going to be a standalone thing. Yeah. But it's useful to know that you think that that would make it more useful as a standalone thing.

SPEAKER_6

[01:19:09] Yeah.

ME

[01:19:13] Is there anything you want to add or to discuss? Well, should we leave it there then?

SPEAKER_6

[01:19:23] Yeah.

ME

[01:19:24] And then we'll call it. Oh, very good. Bang on time. Thank you. .