The talk I wanna give today is basically trying to kind of explain why we did this weird thing that we did as a firm, which is we adopted a deeply obscure language and build almost everything we do in that sort of weird enough you're putting language and I kinda want wanna talk about a little bit about why we need a decision and how we think it's turned out and you know, what were the kind of motivations there and what are the both both technical motivations and social motivations for doing it. So just talk if it's something a little more pretty organized. And ladies, we'll talk because I gave a version of this talk at Google or something. So I made some actual slides, but don't let the slides put you off from like raising your hand and asking questions and interrupting and totally happy for this to be a conversation rather than me lecturing for a while.

And in fact, I think I've given a talk a bunch of times. I don't ever give exactly the same way cause there's a lot of different paths you can go. So you guys already have a sense of who James street is, right? So I'm not going to talk a lot about that. But I do want to talk about Kendall and kind of what kind of language that he has and how it fits into the space of programming languages you might've used. I know you guys all have some program experience. Like I feel like it's hard to be kind of a technically ready person without being had forced at some points. You a little bit of programming. So let me get a sense of from like how many of you have programmed in Java, right? How about Python? How about working in a spreadsheet?

Also a program actually a form of programming, right? Any people it's not, it's a, it's a very different from programming and it's, it's kind of an interesting duality or any way we program, right? If you program in a sales of a spreadsheet, it's the reverse of your normal experience programming, programming languages you see text about your code. That's the primary thing. And the data kind of hovers in visibly underneath and in a spreadsheet. The relationships, it's reversed, right? That the data is what's visible in the code is what's hard to see. It counts in various ways. It actually ties in interesting ways to accountable. I won't talk much about that here. So if we're thinking about where okay, well fit into the languages, you know, so actually let's get a little, how many of you have programmed in scheme Java script hassle? You? Okay, so did this, there's like a bunch of, of diversity here in language.

People see nothing. It's worth thinking about how the programming language world breaks out and how camel fits in. And I'm going to kind of divide the world down to different dimensions. One dimension is the one across the top imperative versus emotional. And the other one is dynamic versus static. And here, I mean dynamically Titan statically typed, and I'll talk a little bit about what those mean. So let's kind of go quadrant by quadrant. So any imperative dynamic corner, you have Python and Perl and Ruby and JavaScript and PHP and languages like that. Cause it wasn't like she was like, what characterizes this class? Like, what do you call these scripting languages, right? And what and, and what are they like, what's going to bad programming [inaudible] language.

But like functionally like that's, is that good or bad with anything? How does that affect like a day to day experience programming in them? Easy to write little programs, right? They also tend to be pretty lightweights and tactically, right? Not a lot of typing. It needs to be done. Anything else? Any other functional properties? If he's like, we just slow in some cases like crushingly horribly, insanely slow, like a hundred times slower. That compiled language, right? Like if you're probably, you put like Ruby or Python or both kind of amazingly so languages. Javascript actually shockingly good for a dynamic language because the bodies of many, many PhD students have been hurled on the big mountains to optimize JavaScript with life. Much lipstick has been put on that pig. So now let's go, let's go do a different one. It's a static languages, C, C sharp Java people plus Fortran four times. By the way, I'm on the very oldest program, limited and feeling active. He's by, this is shocking me. Like [inaudible]. So what's your advice to these guys? How many different from the quarterbacks slowed it down in a second. Let's write what else? Compile. They have compilers. Any else, other properties are different.

You're mapping the machine and fast, right? That's also, which aren't necessarily exactly saying that, but to go ahead and hand them in many cases. So and, and when people say slower than all the cycle floating on the second part, because there's this whole compiler thing you have to deal with. And also more for boasts, right? It's kind of just more that has to be written down. Right. You cannot have it kind of deeply obscure piece of jargon stand for redeveloped print loop. And what it means is like a little little interpreter if you can like type of expressions and see how they evaluate and you can have them for languages like this and that's how they're playing with you like this. Cause you can have like a big heavyweight, like compile everything and I'd run it kind of model. So what's the dimension in which these guys different?

They did differently. Dimension weather of how the type system works. So first of just say what is a type system for what types. So when you think about programming, there are all these objects floating around in your program values you're dealing with and you can categorize these into classes. And sometimes you can call those types, right? Different types of values. There's another thing of your program which is a little more abstract, which is the idea of a variable, right? What's a variable verbal hanging name that points to some value, right? It's not a very concrete thing necessarily. The points, your concrete spot in memory and unique way. All that is, it's just a name in your program, a logical name for things which is similar in many ways to like a name and an equation, right? You know, you name your variables in the equation, you name your variables, your program.

Those things are, are in many ways very similar to each other. So anything you're likely type language, very low values rather have types because there are different kinds of things you like with dealer deals with. But variables don't in a standard B type language, the variables themselves also have types, which is to say at the time your program is written, the compiler can do is kind of stack analysis, can look at your code, say, Oh, this variable always refers to things like certain type, right? And so why is that important? So they're really two different reasons. The importance, one reason is importance because of speed. And the reason that is for speed is if you know in advance what are the kinds of things are being manipulated, then you can write extremely efficient code for doing that. So any compiled language, if you add together two integers, what that turns into at runtime is effectively a single machine instruction.

It says, you know, add the content of this, register to the content of that register and put in the other register, right? A single instruction [inaudible] language like Rudy, this is a longer conversation. It's like, Oh, you want to add things together? Well, what are the things, right? Maybe there's strings and I should concatenate and maybe they're integers and then I should add them together. One of your floating point numbers I need different between riding together where maybe their tuition all added together and now they don't fit anymore in what kind of normal machine integer is. And so I need to use a big numb and cause all sorts of complicated things. It can happen in dynamic language and it's in that gap of figuring out what you need to do. That's with a factor of a hundred Ken's to come in, you think machine code, it looks like that.

I just want to see about all of this categorizations. None of it is precise, right? So for capital even just kind of run time decisions about what you're going to do. That's the dominant way that things work in dynamic languages. But there are parts of static languages that operate way. For example, lots of statically typed languages, job included offer forms of introspection, which is like a language feature that let you kind of ask questions about the type and make decisions about it. And it's also crushingly slow in a way that dynamic language is getting applied even slower than most dynamic languages are. So yeah, there are thinking about it that don't quite work and, and it, we'll see actually more that as we move to the right hand side. I guess another thing to note about this whole column of languages, they're all fairly popular, right?

So we're in a second, we're going to the unpopular side of the street. Oh, I thought you said performance. I said performance much talks about types. There's also correctness. So types were really born for performance. You go back to the fifties when people were first, and I think we're going to limit because there were really two major languages. There was Lisbon, it was Fortran. Right? And there was a few other things and getting back in, but like the only things they had, any staying power with those two languages and lists we'll see was actually any functional dynamic corner. And fortunate it was any static imperative corner. And I think the key reason why Fortran was static was because they needed to be really fast because they were running big job, lots of matrix operations for competing various fitting simulations and to the type where necessary for that.

But techs are also useful for making your code more correct for helping you nail down properties of your program. More library. I'm going to talk a little bit more about how that works, but it's just worth knowing. It's coming back around it. That's an important part of the sanction between these two classes. Okay. So this functional dynamic languages and if you see we have like gone off the cliff and popularity already a lisp scheme, racket closure how do you could you can send to JavaScript and kind of switching over to the functional side. This goes to like the funny separation between kinds of languages. In many cases whether something is a functional language is some mix of what language features does it have and how does it tend to be used. So now that I've listened to languages, maybe it's worth saying what, what does it mean when you say functional versus imperative?

And they're kind of two things that come up in a, the one part of being a functional language is having good support for programming with functions and the kind of lightweight and simple extraction. One thing that you get in all of these language on the right hand side, our functions are things that are easy and lightweight to create. And you can deal with them as if they were data, but you can take a function and [inaudible] function and how they were turning out to function and take a function and store in the table functions like you can in a very easy way kind of flinging functions around. And that's one willing to care guys in class. The other thing is that functional languages give you good support for programming in a what's called a pure style programming that the hold that comes from purity to self, any sort of a little contentious.

It's like, but it must be good. It's pure, right? The pure needs something kind of with less inherent values in it. Then the name sounds like pure means programming that doesn't involve mutation. It doesn't involve changing the values of things as your program evolves. And I think to somebody who hasn't programmed in a functional environment, that seems kind of weird, like the how, how do you compute something, you always can keep things by, you have some data and you can keep some things and you modify what you had before. You do that over and over and over until you get the right value and you're done right? This is sort of the ordinary way that you compute. And it turns out that that's not the only way to express a computation. It's not the only way to do it. To just give a kind of concrete sense of that. Let's get to the terrible part of the talk where I actually bring up code.

So it's a really simple example. So here is a simple function for selling a lift in Python and this like lots of programs that you've come up, you've done in pair to style is pretty simple, right? So we want to sell everything and listen to me and it's always a variable total to zero. And then we have a full, if it goes over all the owners and lists and increments, the total by that number and then we return the total. Right. So then my question is, this is meant to be you press record. Okay. So now I guess a touch smaller. Yeah.

So here is the corresponding function in Ocala, which is a function languages has as we'll see. So the syntax is going to be somewhat foreign for those you haven't [inaudible] high school but or similar lines. But let me kind of walk through. So this, this whole let req some of lists just means declare a function that has a single argument list. The rec here means the function's going to be recursive. You know how she's allowed to call it self. And we look at the list and we see what case we're in and we say, if we're the case manager, let's, we're gonna return zero. And if we're in a case, I have a list that had at least one element called first. And some remedial is called rest. Then we're going to return first plus the sum of every other rest. So I just want you questions. Which of these seems more natural? Intuitive? You made your hand if the Python one seems like a more natural one.

Really only that, not everyone, I feel like even to me who's like about as like addled by the languages as possible. This was still, it seemed a little bit easier to follow, right? Like it's what it's like the, like the, the recursive definition, right? This is kind of, you were doing axiomatic description of what it means to somebody. I on to listen. You might write it this way, it was kind of a current relation, but it's a little bit of an odd way to write down a summation at least. At least I think. But he does have a certain cleanup medical flavor to it and it's worth noting it doesn't have any notion of mutation. Right? No worries there a variable which has evaluated it and now we're going to change the value. So there's a thing that often confuses people about function languages, which is in function languages.

You have these things called variables, but they don't exactly vary in a way that you might expect them to write. You don't modify this within it. So why do you call them variables and the reasons you call it able for the same reasons you call variables, equations, variables, not that it varies over the course of executing the equation, right? It's because the equation can be applied to too many different inputs, right? There's some part of the equation which is like static and some part of it that you can think of as varying over different possible applications. And that's what's going on here, right? We can call some on many different lists. So any need, some calls itself. So over the course of the many different executions of, there's some function first will mean different things at different times, but we were able to write it down in a way that doesn't explicitly do anything that looks like mutation.

So I think what our question about this is who the hell cares? Right? Why does the distinction between two different styles matter and I think the reason it matters is because mutation is more than just a way of getting a computation done. I tell him he teaches years. Sometimes it's very simple and important. Very local. I did his little tiny function, which means a new middle variable, some very small scope and it never escapes anywhere else. Like I run for a while and finishes and like, let me teach this all kind of thinking, a little tiny box, but in general when you have a mutable stating your program, when you have stuff that can be mutated that that mutable state, that changeable state is more than just a way of competing things. It's a communication channel. It is a way for giving part of your program to talk to each other and the reason you want to avoid that when you can.

You said having a program that is clearly separable, it's different pieces that don't talk to each other. It's really valuable for having a program that's easy to reason about as you can pose it together. Right. It's nice to be able to have different parts of a program that may be running a kind of together as part of one massive system that your building and to be able to think about the program that you built by assembling it together as a kind of simple composition of the pieces that you started with and the more mutations you have in your program, the harder that is to do because that mutation had gives different back channels for dinner party parents talk to each other. Whereas when you're in a social setting, you're not mutating anything and the only form of variation is like the functions being called on different values.

Then the functions are really explicit about what their inputs are and what their outputs are and it's easy to understand how the pieces connect to each other. So it sometimes thinking about these examples in the small give you the wrong intuition because it's not the examples and the small they make avoiding side effects useful. It's also worth saying that it's not, there's nothing wrong with side effects. I feel you want to try to fix it by another sort of another technical term for mutation and other similar things. There's nothing actually wrong with doing any problem. In fact, all interesting programs have side effects, right? You write your program cause you want to have some effect on the world. I don't, I just kind of boring thing to do. Right and dry. She programs can be thought of as just like pure computations and like you know you give them some input, any pretty spun out but that's all they do.

Programs, read things from file systems and look at the clock and write stuff to the network. Right. In the end, almost all real world programming, it's about having an effect on the world so you could judge your functional programming language and any programming language. Not so much on how much it lets you programming artifacts but how much it helps you program with the facts and the reason why it's good to have the ability language to program in this way without affecting because that part of the program probably doesn't effect. It's simpler to reason it out to think about and maximizing the amount of your friend that he's the reason that was good but not you're trying to Ben, to me your program. That's what at the heart of what you want to do. There's a lot of really abstract stuff here. They don't want to ask any questions about this part please. But it does. So I wrote, I wrote this program, I wrote this little function in a simplest way possible, which has this property of you know those, you know a little bit more about how programs program ladies work. Yeah, like it's one stack frame per got a list, but it is a small transformation you can do to have a recursive function. It doesn't have that property. But I wanted, I wanted to maximize my example for readability rather than performance.

So it's different in a lot of ways, but. There are two or thought it will change things. You can do your program like does it make the conversation easier? What does it have? Good forms of abstraction and objects gives you a form of abstraction. I think not quite as good as the one you get in language. Like I know that's a more complicated story that probably isn't outside the scope of this stock, but it gives you that. But having that in your language has communication channels to get out of there in a fundamental way. Right. And like you don't have to trust me. Like John Carmack, right. Again, we can hear John Carmack is like your head if you played quake or rage or doing whatever heard of those games. Oh wow. Is this been a while? I guess so. John Carmack is an excellent C plus plus programmer, one of the like kind of fairly legendary guy in the games industry.

The lots of, lots of, lots of hacking in people's bald head. If you listen to the way he talked about programming, he argues very strongly in favor of immutability, avoiding offense as a way of making it easier to decompose. This is, I think, fairly standard advice for people building programs. Know what they're like. Go read the book. Effective Java by Josh block was like chief Java engineer. It's not right. Immutability, simplify things because you had to have some missing catalytic object. But like if this part of the program modified an object and it's program reads from the object, like it's a, that you put an abstraction barrier on it, but these two things now are still having this kind of communication channel between each other, which is not explicit anywhere in the code, right? Like this guy doesn't go to, this guy exists and has no way of knowing about it.

So if composability is about more than just avoiding mutation, let me tell you, it does give you a bunch of value back. And another example, think about concurrency, right? It races are really hard to think about in a context of mutable data and basically trivial and the absence of it, right? Because of the lack of mutation, he is no interference when things are actually each other like floods, right? So now, now we've got into the last, I think, I think this used to be the most unpopular corner of his lot and I think over time it's got a little more equalized with a, with a one directly above it. So Campbell's is the one we use. Scala is a pretty well known language in the same rough family that runs on a JDN. Haskell is many ways quite similar to a camel. F sharp isn't a camel clone that runs on a.net, which SML is a language is really, really similar to a camel that kind of comes out of the same kind of historical line of research in which in which I'm looking.

So this adds to its basic functional set of I guess fucked up. We're going to add the static types that you, that you are used to from like CEC sharpen Java. But interestingly, the type systems that these language in the bottom right hand corner have are actually a lot better and more powerful than the types of things you tend to have on the systems on the left. And I think it in some sense accidental, like I don't think it had to be the fact, the fact that the ones on the right are function languages and having more sophisticated type systems wasn't unnecessary property. I think one of the reasons why it happened this way is the ones that the ones, at least I can be upgraded languages and I'll be going to, it's actually a lot harder to get right in terms of thinking out what the way mapping onto a good type system.

It's a lot more complicated and and so sometimes the problems were simpler over here and so people were able to build nicer and fancier type systems. But one of the key trade offs that you had between Diana can static over here, you don't have when going to a functional static site because functional static languages tend to be fast, fast, and about the same level as the imperative languages, but you don't have the same syntactic overhead. If I can go back to that example, we just looked at it and like, so she programmed in Java think how long the Java version of this would look like. Like it would look a lot longer, right? And here this is gotta be tight. I said, right, but why, why should you believe me? Like, well, I can ask that. It just comes off really small, unfortunately.

But I went over there and said, ask the editor, tell me what I did was I went to you, did your list arrow and so forth and take scenarios and returns, right? I can't do that in the Python because Python doesn't know, right? Doesn't wait, but accountable knows the types of things by a process called type difference, which is to say it kind of looked at the program. I says, well, what type does it seem like it has to be? So I said, well, if you mashed a list with this pattern that kind of turns out to only apply to lifts. So this thing that you would call listed also have type list with something on the inside. What might be on the inside? Well then I've plucked things together. Plus operator, I need to drill. So it must be interesting. Now you know that you put an easier list and because we were turned zero here, so this thing must return.

We're turning into your list as well. Or was it mr to integer. So basically typing for is a process which takes the little kinks that you get from the program as to what things are doing and in first the overall types. And there's a trade off here. It gives you a lot of extra concision. Things are much are, are much tighter and it's a tactic level. It's also a little more complicated to understand, especially when it goes wrong. Like if my tires are not super hot in all cases, but it can, it can be confusing. But overall I think this is a pretty big weight and in fact if you look overall at how program languages are evolving, type inference is becoming more and more popular. Language like C sharp are getting local tech and friends as a kind of addition to the language.

Okay. So that was a lot of really kind of high level stuff about programming languages and ready to hit in. Let me talk a little bit about why we care about this. So we ended up switching to a camel a long time ago about, about, around a decade ago was when we started using it for the kind of primary Delta might be for billing systems. And there's a bunch of interesting reasons that when you, that someone actually had to do with non technical things like how easy was hire people. And it turned out was easier for the hire people from the empty set of Ocala programmers, then from the very large set of Java programmers say, and that's probably because even though this set of programs is empty, the people in F that are pretty good and guess instead of measure zero more precisely.

So, right, so what technically, why do we think these languages are good? And I think if you think about what distinguishes the business we do, one of the key properties is it's terrifying, right? And for all of us, you have programs, you are of course terrible at it mean we all are, humans are really bad at programming. Like you write programs and if full of bugs and come out very well. And one of the things you learn after having done it for a long time, you don't get that much better. I mean you get better but you're still like full of errors and mistakes all the time. And going and writing programs that you then like let loose on the market and have access to your wallet. Like that's kinda terrifying and it's not terrifying cause you haven't thought about it enough. And so we care a lot about having things that help us write programs that don't do the wrong thing to do what we expect.

That behavior, ways that we can understand. And there's lots of ways of getting a testing of course an important way of getting confidence about the poems that you write. But there are things about the financial markets and the testing. Not like the only story you want to tell one of them is that financial distributions can be tail heavy, which is to say things happen rarely but less rarely than you might think that are really dramatic. Right. Thanks. Sort of these kind of things happen kind of in a normal attribution things far out or like decreasing? Probably very, very fast. And so like I can pretty quickly cover the territory, want to cover like just random sampling and like, you know, for even for, for complex programs, the large spaces of possible things can happen. Even that's not a very good story. But like in the financial markets it's especially bad because things can vary, you know, by orders of magnitude more than you might expect if you thought they were normally distributed.

So that's, that's been one kind of sort of high level reason why testing isn't quite the full story you want. The other thing is there's an adversary, right? I think to remember about trading is like overall trading is a positive expectancy business, like for the people participating in it and for the world, right? When you treat your axes intermediary, other people's transactions are cheaper and easier to do because you're there being on the other side. But any individual trade, right? You turn the dial to the right, you know, I'm saying I'm making more and more money. Right? Somewhere else did the dabbing turned to the left. Right? And someone might not want you to turn that dial, right? And someone like want to turn the dial the other way on you, right? So like you can test against what you think is going on, but you kind of want to, you want.

And sometimes you want the power of the upside down a right, universal cosmic codification for all cases. You want to know that properties for all cases and testing is a way of approximating that. But you can get some of these, some of that universal guarantees in other ways. It's really valuable and one way is by reading and understanding the code. But we're all very bad at that, right? Like it's, it's an important thing to do, but there's lots of lots mistakes that humans make. And having tools that help catch those mistakes for you is worth a lot and type systems turn out to be a very good tool for catching a surprising number of mistakes. So it's very hard to really give you a flavor for this kind of, in a context of a short talk, right? I can give you a few devils and there's only kind of limited you can, we have limited kind of convincing powers.

I can give you a few examples, but if you really want to understand this, well, you have to actually go and use it, right? Does it, I know stories of, of people who like Peter for years turned be kind of how useful the Catalan NL type system is. Like, yeah, yeah, it's fine. I guess I'm just, someone helped let me go and use it, be like, Oh no, it's really true, right? Like, you know, it used to be, it used to be no, yet I'd write a program. There's been a lot of anybody and now I got to put them in like a whole class. I figured like just aren't there. It's a surprisingly dramatic effect. So I'm not gonna be like kind of show that effect in full force. But let me try and show you a little bit of how this kind of works out.

Let's see. So let's start with this example. So here's a small piece of Python code. So the idea is you're supposed to do, this is a function that is supposed to take as input to dictionaries where the character mappings from key to data and find all the keys that are in both dictionaries, whose, whose data differs. Right? So look at a piece of code for a second as you can tell anything interesting about it, right? So the key is so, so if with that exception it turns out, but like I loo, I overturned no, right, it depend exactly what I'm hearing, but he put it on a frozen section. But through the blog, not a very complicated blog, pretty simple out here. But if the kind of bug that she would make all the time, right, this is again, if someone asks is that an exception or no?

So it could be either one. So you know, you can get your points and you're putting me in a crash with an exception that's generated by this look up or it's going to crash with a null pointer exception. But like a difference isn't all that I tell you in both cases is going to crash when you run it. And this is of course the good outcome. The bad outcome is if your language where digital key returns. No. And then this just does the wrong thing, right? Because he tried to say God just checks against whether it's Nolan, my sort of not if he, if it's like a case of handle automatically, you weren't really thinking about it. And maybe I'll handle the wrong way. Maybe I held the right way. It's kind of ugly. So sorry. So now let's look at what the same thing might look like in a camel.

So there's a bunch of like obscure characters here that you won't recognize. You let me come to mind. Walking to the party really matters. Give or take. So these things that you called the, if I thought we would call it more or less the same thing, maps, you know, cattle here we take an app and convert it to a sequence, which just means like we're gonna think about the set of key value pairs. And then we're gonna call this function filter map, which is going to iterate over everything in the dictionary and it will allow you to return. I do none or some of something. So these, this is a thing that's even a type single camera which just isn't in the type system in Java or C sharp or C plus plus called a variance with are two different cases and it lets you kind of returning to one day does have these two different possibilities and then if you were turned down then it will drop that thing from the return sequence.

And if you try to sum up something, a little return the object inside the Sontag. So let's splice a little obscure. Let's, let's walk a little more here. So we can, we call this function map that fine of map too. And the key is remember the key, like the key up here, this is the key from the first dictionary. Slash map. And again this is the key from the first one because we've filter mapped over. Yeah we filled them out over the data and this and that one. So key comes from Avalon and then we're going to look up, you think it's mapped that line function in map two and then do we, we didn't match it because there are two possible cases either none of where some, so we looked on case which sort of corresponds to know everything about before. Well it returns.

We're going to return on meaning we're going to drop it in the case that it's not in the second list. And if there is some data, there was a, if the data is different, if the didn't in the second map, I going to do the first map, then we'll, we'll keep the key in the return list. And otherwise we want. And so one thing you might look at this code and say, well why is this different from the Python code? Right? Maybe you could have explicitly handled the case of the thing not being a second out there too. And then it would have also been correct. But the difference is the way we've written like our libraries, the wait not that find works because it returns this explicit none or some type is what's called an option type. You actually can't ignore it.

Meaning if you try and ignore it, the compiler will just yell at you and say like, no it doesn't work. Well it's not. It'll just fail to compile. The competitor essentially forces you to do the case analysis and it's just a question before like we're talking about like what's different? Are you typing in on type language and other language? You can kinda do whatever you want and in the type language or as much constraints. And this is an example because we were able to design our math library in a way to impose it obligations on the users of the library to do case analysis and those kinds of obligations are really valuable for improving the quality of your code base or they help you track down bugs and make sure that things are handled in the way that they should be handled. We can first encounter type systems in the seafood languages with type system language without them.

I think their attitude is often one of like maybe it's good, maybe it's not likely, but the types of stuff may catch some bugs for me and that's nice and it also stops me from doing some things that I want to do and maybe that's not so nice. And so you sort of think of like a type of thing, like a negative hovers on top of your code and slaps your hand when you do the wrong thing. But this is what an impoverished view of what types of useful for, because really type should be part of your design process because you want your program to be right or you want to be correct and you need to do a lot of work. Whatever programming systems you're using, use a lot of work to try and ensure that your program is right. Maybe it's writing a lot of tests, maybe it's, you know, writing a coach.

It's pretty simple for humans to read. Maybe it's using a type system, but these are all tools that you use actively and designed to make them less error prone. And if you just think of pipes is like a negative thing on the outside and catches when you pass an injury or when you hit passing a string, that's kind of all we'll do for you. But if you actively cause on your program with types of line, you can dry the air right down a lot more. Another, another little example about types. This is sort of an unfair example. But I wanted to give a sense of why how type systems affect how easy it is to write code. I think type systems kind of get a bad name in some sense because of languages like Java, where the type system and post a lot of verbosity.

So just to kind of give a concrete example here is it's a little bit of a devil cause right in the wheelhouse of what a candle does really well. But here's a very simple piece of code for expressing a binary kind of bleeding expression language, right? So imagine you want, you want to like have say a filter that you don't think about. Like the Gmail filters you write when you say like, I want all that to do this, to have no have this and this and that or the other thing, but not this, right? Do you want like some set of Boolean operators you want smash together for the 10 of us at the bottom? Some sort of base predicates like you know, this regular expression shows up in this email or the front line. It is from this person. So did he hear this?

This typing juice? Here's an alpha expression, meaning that that alpha is what's called the type yard and pick a right? Why am I coming in Greek, right? Then they take a, there is a a type variable which made it, this is a general class of types. So we're gonna be able to have these kinds of looking expressions used for filters from emails or we could use them for filters for our traffic viewers upstairs or whatever. You can use it for lots of different things. It's generic, does it actually that today is like the generic GC in Java. So you're like the angle bracket, a angle bracket. It's the same basic concept, if that helps. So here this is, there's a little like nun in some case, there are a few different possibilities for the type of true false hand or not and in a lot of ways base.

And that corresponds to these base predicates. They were going to kind of weeding together and said these expressions. And then you, I have a little function whose job is to evaluate an expression. So this is like super simple. So there's this email function that takes as an argument, a function called email base. And so this is the thing that evaluates base put against and it takes the expression, you wanna evaluate and then all this code is going to start is the, is the guts of that evaluation. And [inaudible] he hired an ed first class. He's a functions, you're low, right? You know, basically they function. They just happened to email. I mean, in fact we can ask what the type of that function is. And it's alpha to bull. And in fact, the overall title II Val, if it takes a function that takes alpha, which is going to hit the base part again and to true or false and in alpha expression and returns a single true or false for the overall expression. Your question about so far, I'm gonna go into this code, I wanna make sure people have some understanding.

Oh yeah. This is a terrible bit of sort of syntactic Nelson's just we have a collection. There was like this, think of it that they function with two arguments where this is the pick of the first document and it's is good thing. The second argument and in the first time I can get itself a function and there's some interesting technical reason why I've written in an odd way. It's not actually a good idea. Turns out that that's how it works. It's like, do you have a question?

Let me see if I can get the right assumption is if you don't understand it, like 50, 60% of them also don't understand it. So what I mean is if I wanna be able to say, you know, I want all the messages that are from microsoft.com and have the subject line virus, right? Well you can think of that as as a conjunction, like, and there's a lot of people, you know, this and that, right? So [inaudible] math, right? There's like Boolean logic, right? But it doesn't tell you what the subject matter is. It just says, Oh, this is what an end isn't. This is what an aura is and it's what implication means. But again, having useful system out of this, you have to have facts about the world and if the base predicates are the facts about the world, right? So the idea is basically like a little language for pulling together a bunch of facts that you know about the world and constructing expression that talks about how this actually lied to each other.

All right, so, so then, so, so what did this go do? Can we actually define another function called [inaudible] prime? Which is basically just a specialized version of evil that already knows about you all base. It doesn't make sense. Don't worry. I think it was, it was just calling you now and then we do a pattern match. We look at the expression, say what are the possibilities? If it's the case we call true, then we return the blend value. True. We always want to call false, which I believe to be false. If it's a base predicate, we call it email base on that predicate. If it's an if two things, well then we eat all the first one and get an ordinary logical conjunction with the email of the second one. If it's or we do a destruction and if it's not, we do have integration in your language. We just go here very straight for the translated. All these things are a little language into the corresponding operations. You know, camels, libraries, right? So the code is sometimes more than anything else, really boring, right? If like [inaudible] or the orange who is true and false is false, there's almost nothing there, right? Which kind of how it should be.

Here's a Java version, it's probably not worth going into great detail on this codex. Exactly what it say. Like you see it has the same like I tried to map the structure pretty closely. So I try getting to fit. So that like that like angle bracket, the angle bracket does that same parameterization over what the input is. But I think to know about this is the reason it's so much more for Bose is there's some like accidental details of Java at the time, which was not so great that make it more verbose than it needs to be. But a big part of why for [inaudible] it's missing something in the pipe systems. It doesn't, this whole thing was like a very bit about different cases and you want to kind of match on them and make sure you cover all of them and all they're like, it's just not naturally expressing Java. Like it just kind of hard thing to say concisely. And so this goes to the other reason why the type systems in mean is very useful. One is I said before, like it helps you catch lots of bugs, but the other days it's more precise, it's more expressive. You can see more of what you want to say about your programs and the types of sending you can in more traditional languages. I know there's lots of examples. This is somewhat like other, they're like the evils.

Is this the same father? This is like, I guess a little bit of an unfair example, but it's not that unfair. In fact, I think in many cases like a three to one compression ratio between John and Elyssa. Not that unusual. And a lot of it has to do with the details of the tax system. Does it make sense? People need other questions about it.

So there'll a lot more to say about types and why they're valuable. I'll say a little bit more about that along the way, but I think what do another thing about, and that I think turns out to be very useful, which has to be what I call the dynamic range of language. So this is a common thing you'll hear programmer saying, which is you should pick the right tool for the job, right? If you're doing a little scripting job, you should write in bash or Python or Ruby. If you're doing, you know, a big complicated enterprise program, you should write in something like Java. If it's close to the middle, there's actually writing C plus plus. And then whenever people have a pretty strong, you should use different tools for different kinds of applications, different kinds of plums we're trying to solve. And this is in some sense, obviously true.

You should of course pick the right tool for the job. Who are you going to recommend to pick the wrong tool for the job? Right? But there's something behind this suggestion, which is that two things, you pick the right tool for the job thing. You need like a really big tool set with lots of different choices. They handle all the different kinds of jobs you have. And I think to some degree, that perspective comes from the fact that people's junk, people's normal tools suck, right? They don't have good dynamic rates. They're not good for a wide range of problems and this goes back to him and he's getting nice about this class of languages that the camel sits in. Is it a kind of sweet spot in the design space? It is concise, like critical, dynamic languages are and efficient on a scale that traditional compiled languages are.

You have the type system that catches a lot of errors for you and it doesn't require a lot of typing in place and is like relatively easy to work with like there are, I think that if you look at, it's not a program language, there's lots of things that you can do to get even stronger guarantees and even more precise kind of help from the compiler and catching bugs. But as you push more and more in that direction, it gets kind of more and more complicated to use and okay, we'll have just going to sit in a place where we're both really good at catching bugs and and relatively simple to understand what it's doing to people were living in for the first time. It maybe doesn't feel enormously come to understand that they gave you that just like a person. And, and I think this is reflected a lot in our own use of language.

We use a camel for everything from little script that we're running. Just automate small tasks to kind of lightweight kind of mini languages for when you can pick files all the way to kind of big trading systems that we, you know, tried to build a dollar a day through. And one of the advantages in an organization, like I said, having a single tool that for a wide variety of applications, if you get to share a lot like wanting what's notable about offices, we're a pretty small organization. I think we have, you know, somewhere North of 400 employees now, something like 70 full time developers, maybe another 50 or a hundred people who spend a significant faction of every other day programming in one form or another. But like I know Twitter has like 2000 programmers. I'm like, that's probably just Twitter, right? It's not like Google is, you have enormous scale and a lot of different things.

But like, you know, even for a product gets feel so you have to like a relatively small thing. Like there's a lot of humans working on it. We have a really small set of people working on a pretty big and complicated infrastructure and you really want to maximize people's need to do a productivity or their ability to do things and get these done in wherever they are and also be able to switch to some other area like one terrible thing that can happen to you. It's like you use some obscure tool or the rest of the company don't use something something or you have to build something and then nobody else wants to touch it ever and you're kind of changing it until that, till the project kind of dies away this way. And the fact that we have one tool that we use for a very broad set of things, make it easier to kind of share ideas and share code and share people across that. And it's all very valuable for making people more productive.

You skipped a slide. Yes. I want to talk about how can we do these? Interesting, but super rung at a time. And this point so teaching, so one of the questions that you can, it comes up when you think about getting some weird, fewer academically, well like programming languages. Like can you actually effectively teach people, and this is not a rare concern when thinking about programming, let me just, and there's a great quote by Rob pike who's at Google and he is the author of go, which is a program language you guys may have heard of. Which is a new, one of the few new languages at Google. Google is actually very restrictive in what languages? That's what you, they have like basically three blessed languages and CC plus plus Java, Java and Python. Sorry, I've got to feed those languages. And [inaudible] kinda includes Caesar sub language.

And in addition, they, they now have go with language the people use, we're doing some production stuff. And so pictures are the key points. You're gonna programs are Googlers, they're not researchers. They tend to be fairly young, fresh out of school. Probably their job maybe like C or C plus plus probably the Python. They're not capable of understanding brilliant language, but we want to use them to build good software. So the language we give them has to be easy to understand, easy to adopt. It's about to be a great slide to give it. I gave a talk to Google. So pipe freezing it a little kind of sending an unfortunate to the people at Google, but the point is super reasonable, right? Like when he says, you know, there are people just getting a brilliant language. That's not a compliment, right? Like there's no brilliant in a bad way and you have to be willing to understand it, right?

This is not a property you want of your tools, right? So it makes it a question of like, you know, camel too hard is a brilliant language in the bad sense of the word brilliance. And it's hard to definitively answer the question sometimes from our turtle perspective, clearly not in a sense that we have lots of people here who successfully put him in a kennel. But there's lots of selection involved there. Right? You know, one of the ways we attract people here is people who have an unhealthy have and factual programming are more likely to come to want to work here. And so maybe we're really bad at telling whether or not this is the kind of language it's really too hard for general purpose use. My intuition is that it's not. And I think that there are some bits of evidence that lead me in a direction of thinking.

It's not, it's not really too hard to understand. One is we actually have all the traders learn how to program. You know, Kendall, some you know, Khalil bootcamp where people do, so this is largely for the traders. Some of the people from that part of the firm complex operations often get as well, but the bulk of people have to put on the training side. And he did a month long program where people can, are all off hitting with each other, learning the unit shell and INAX and kennel and our build system and all our libraries and how to use the monad. A complicated thing of own. And I want like this class and it's not too bad. Like I feel like about half the people get over the finish line where they're like, they are then able to go and do useful things and the other half the people I didn't come out like a little more informed and like less afraid of versus the tools but not actually going to use it for useful stuff.

But it's only trouble. We can make a pretty broad class of people in a kind of stunningly short period of time and get them to a level of reasonable competence without any, without enormous mental difficulty. Another experience we've had is we have an intern program where we had people come in and we have like for basically mostly MIT and Harvard sun has Brown have winter break that are aligned nicely. So we can have people come to do a short internship here and it's really short. It's like, think for the Harvard students, it's three weeks and then it seems it's four weeks and usually everything Harvard, MIT students is harvesting. And I'll learn the Catalan because it turns out it's part of that kind of standard curriculum there. There's intro CF class, part of that sequence is in the camel and the MIT students don't, anybody ends with that extra week.

They're all kind of able to like put together a simple project. That'll give me some hope on the question of whether it is, you know, it brilliant language and that's nice. But again, it's hard. You, there's lots of ways in which the experience here is limited. But at least my feeling coming to experience it that I think it's not, it's not too, too painfully hard. So I had an interesting that hiring, which I mention this earlier, and to talk how hiring turns out to work pretty well. And I did this amazing quote from an essay that Paul Graham wrote a years ago. So Paul Graham, you heard him, he's the guy behind Y Combinator, which is this like startup angel, like startup camp, right? So he wrote a couple times on paradox about how picking an esoteric language allows you to attract better programmers. It isn't actually that he wrote this more than a decade ago, that Python where like [inaudible] you can breathe.

But other than that it allows, yeah, I think this has been a quite useful effect for us. And there are two parts of it. What effect is there is selecting people who are good to come work here. And that definitely the fact that I think, I think like, you know, the average Ocala program was pretty good even though there aren't any of them. Like that totally worked out for us. Like one of our early rounds of hiring, I said one [inaudible] I got 15 responses, which 12 seeing worth responding of which five were ripping and onsite at which three main offers been hired and they were really good. Right? And that is a shockingly good sort of worked to reward ratio, but over time to actually affect the value effect has shifted almost entirely the other way, which is we now talk to many more people and look a lot more resumes.

And now I think, Oh, the overwhelming effect is not that it helps us find people are good, but it causes people who are good to have a reason to come work here. I think that, you know, if you, if you come out with your size, you're getting, what am I going to do with my life? It's like, well, where do other computer science people go with not so much to finance? Right? It's like, you know, how can I think deliberate ads or something, right? It's, it's been decided it's another area of barrier. We're gonna decide to go for reasons that some of these gave me. But having, you know, usual technical choice that you've made it as attractive to us. Also, people turn out to be really effective for attracting will. There are lots of people who would never have spoken to us, but we got higher than like [inaudible] giving up.

He is the author of like the world's kind of fastest compiler, or maybe the world's slowest in Oakland pilot generate the fastest code. But anyway, and there's absolutely no way he would work here, but for the choice of the technology and he's really good and there's a lot of people like that who we have here only because of this. So I think that there's lots of discounting essentially. Whether you think, you know, picking some technology will kind of box you out, make it hard to hire people. But sometimes it goes the other way despite the way the numbers were down. Oh, so this is, this is the thing we're saying, it's basically like a real bad side of using a minority mind. It's like not so much about the language, but about just using it, keeping a popular language and giving a potty language, which is the tools that are good and there aren't that many libraries.

And I think from our perspective it wasn't so bad initially because we had lots of things we need to build. They were completely custom. Like if we were like doing web development, I think it would be a very different story. Like you read the lots of like, Oh my God, you don't want to reinvent all of those wheels. There's a lot of wheels involved in like getting all the web browsers tool that you're willing to wait an incubator well and hooking up the database. And I'm like, there's lots of things. You don't have to reinvent the wheel. You didn't have as much of that issue, but it still hurts. And I think it's an ongoing problem when we've tried to work on by trying various ways to help kind of cattle in the CUNY to get more, more popular ends, help us support in ways monitoring others things, tool to make things better.

There's actually been a huge amount of progress in this in this way is a package management system. Pro camel, which like all the other one [inaudible] it doesn't completely stop actually pretty good. There's more language, actually a Freddie singer. There's one of me off other than tomography who also, who works here. I though, because I kind of kind of like IntelliSense like auto-completion and looking at the types and things like that, which is awesome. Oh, so you can get another small tool for automating the annotation of your code. Kind of like go for net. If you ever heard of that. Kodak is a new doctor and he's still coming out extension points in lateral aliases or new language features that have made a lot of this tooling natural and easier and all that. And so there's kind of a mix, which is to say it's truly hurts, right?

It's nice to get visual studio this, you'll see this pretty good. Like, you know, lots and lots of work has gone into making videos for your effective clips. I had lots and lots of work in it, although it's still on my socks, but still with a lot of benefit you get from the, Melanie told him from the library and all that. And I think one of the things that you inherit when you use it, but minority languages, you break it, you bought it, right? Like you, you go on and use some weird technology that we're using and you were really successful and like you are now kind of responsible like for your own self interest in maintaining this and helping you grow. And that's the position we were. And I think all in, I'm very happy with a trade. But it's, it's not, it's not a trivial one. So basically at a time I'm like, I'm going to over let me like, entertain like a question or two and and then people can hang out after [inaudible] after that.

So it's a good question. I think one background piece information is that permitting languages for real, I don't totally understand in general, move incredibly slowly internationally. So a great example is garbage collection. That's a pretty good idea, right? Coddling the single best idea for improving Porter programmer productivity. In the history of, you mentioned the industry languages. How old is that idea? It was invented in 1958. It hit the mainstream in 95 with Java. Why? I don't know like, but yeah, it took a while to get efficient enough. There's something, a lot of things I had to work out, but mostly people seem to act in people irrational ways and picking a tools. Right. And I don't totally know how to explain it. Some of it, some of it is no doubt network effects that create inertia and organizations. Like if you showed me a new language, it was gonna be a catalyst switch to, he would have me better by a lot.

Like we have several million blind Dakota cattle and lots of understanding about the details of language likes what you do, anything. It's really hard and it has to be better by lap. And actually that's one dairy or the cheerleader. I think also people get deeply emotional and somewhat irrational and other tools, I'm sure I do it as well, but the tools that you use in some ways, like they kind of formed the way you think about his approach problems and the get almost offended when they're asked to use some tool. It's not their regular tool and they get like more offended the longer they've been using some other thing. So I think that that also, I think it has something to do with it, but it's, it's a complicated question. Probably the program language is really like shocking me. So again, garbage collectors, a good 35 years.

Generics, we showed up in Java also summertime tech in France. Similar time be like throw a landing like C sharp. Now I look up type in France, yay. You know, 35 years late. It's hard to get a little hard to figure out. And I, you can, I don't even know. These are kind of small. So on that I don't, I don't think that's true, but it's incredibly hard to study. That's another [inaudible] research study showing, showing, you know, how, how much language, the technical program, it's a great, do you want to get a group randomized to these and COBOL for the next 20 years of your career and you need a long time because like the only experimental tool you have and university, he's like sophomores. I don't think I can have a hand. Can you start to study the software. You cannot construct the pilot sophomores high enough to get anywhere on this problem because permanent languages are professional tools and what matters about how they're fucking and how they're effective. You know, over a long time. Like you've got a 23 year career doing this stuff. Like you know, your first two weeks doesn't matter. But that's all they can measure. Right? And it doesn't matter what you do in like the 69 program, you're right, it matters in the million line project, you collaborate with a lot of people. I'm like, they can't, like, they're just in any way of running regional studies about this. So it's like hard convince people and it's so much emotion in the topic that it's not clear what to do.

So I don't have any other front it use it kind of in the like both hands and both feet all the way. Wait, did we do it? Facebook uses a camera these days they wrote a compiler called hacking, another one called flow, which are compilers, which the point of the completed [inaudible] add ad type system. Two languages like PHP and JavaScript respectively and a certain kind of general kind of training, computer science, a lot of energy going into big lifting on tags and the PHQ tickets had a lot of lipstick from Facebook. But that is a big system, a fairly significant team working on it for years. So developers all know camel if very high performing system I'm using like so this goes to like, so in Facebook's kinda developer kind of organization, you can't get people to use tools like telling them you have to convince people they want to use the tools.

And so these programs aren't used to compilers so you had to get the world, it felt like it was no compiler. So they wrote a really high speed reactive compiler that like traces down dependencies down to the function level and watch the file system for changes and you can like use it to like get and you would get rebates to changes, thousands of files and it probably just noticing everything running along and a compilation is finished 20 milliseconds after getting it done, which means a competitor is wanting to get, can be fascinating. You can change the file system. So an interesting bit of technology. Bloomberg uses a camel for a bunch of stuff. The guy to use scholar, which is a closely related language actually consider using a CamelBak side of the JVM was too valuable for [inaudible] to talk about. There are some smaller places using kind of [inaudible], which is another related languages for something to the infrastructure.

So there are different places in the world, other, other places use it. And there are lots of financial firms that use Haskell for fairly different den assistance when they were in. I'd have sharp another, some language shows up, but like I can point to lots of other examples, but you know, they're kind of, you know, grains of sand in a large sea of not using it. I still think that functional static text, actual language is like candle are increasingly talked about, increasingly taught, but they are more talking about than used. Okay. So let's break and feel and go and I'm happy to sit around and chat to people, have other things I want to ask them.