**language-creators-edit-480p.mov**

[00:00:00] You guys ready some language creator. I have further to introduce Carol Willey. She is a python during counseling Project Jupiter Steering Council member. She blends the strings of pythonic. Jupiter notebook's to improve access to learning and education due to the worldwide foundation of the Pyfrom language. The hard work of the Jupiter team users worldwide. Carroll was awarded the Twenty Seventeen ACM Software System Award, recognizing the lasting influence of Jupiter, a broad collaboration to develop its open source tools for interactive computing with a language agnostic design.

[00:00:38] Thank you.

[00:00:47] Welcome, everybody, for being here. It is my absolute pleasure to share this evening. You and some of the people that actually influenced my career in software development.

[00:01:01] So let's get through this. And for those of you that are watching us on the lifespan. Hello and welcome and hope you have a wonderful evening with us. We haven't really distinguished panel of language creators. And I'd like to invite them to the stage as we read their bios. I'm going to keep the bios really short because I think most of them need no introduction whatsoever. And that'll save us some time more. Do you see language design stuff? OK. So first we have been Rassam to develop the Python language that touches pretty much every corner of society today in some way. And it's.

[00:01:51] He's the creator of Java, which is UK-EU, used across a broad variety of devices and deployments at scale. Well done.

[00:02:00] And please get a.

[00:02:07] Childcare has architected a number of languages over the past several decades, including Turbo Pascal, Delphi, C-sharp, and so you might know this little language typescript.

[00:02:26] Definitely not these.

[00:02:27] Larry Wall to combined his unique perspective on text and computing with his strong background in linguistics to Craig Pearl and its sister sibling Pearl 6. Welcome.

[00:02:51] So just a couple of housekeeping things about this evening's format. You're in for an engaging and hopefully very lively discussion. More likely than we were at dinner. I think that's possible, right?

[00:03:04] OK. The format. More beer. OK. We can do that. Keep it coming.

[00:03:13] OK. So the format will be two sessions with a fifteen minute intermission. What I'll do is I will start by asking one of the panelists a question, and hopefully a response will be like one to two minutes and then kind of throw it open for other panelists to respond and hopefully get a really good discussion going.

[00:03:34] And the more you discuss, the less I have to ask questions and the more in-depth we can go on the really cool stuff after the 15 minute intermission. We're going to take questions from the audience over a winner Twitter.

[00:03:54] OK. So tweet out your questions. Hashtag is happy bbf out. And if you can get those in by 8:00, we'll kind of collect them and have some really great audience questions about language creation and language design. So I'm going to invite you to sit back, enjoy a cup of strong Java Sea, start offering responses and insights, learned pearls of wisdom and embrace. The Python program makes it.

[00:04:33] We'll start the celebration with a question for Kito about principles of language design. You mentioned before, perhaps somewhat jokingly, the Harry Potter theory of language design. What is this theory and what do you see as the key principles of language design? I think you're good to me. Am I good? OK.

[00:05:01] The Harry Potter theory of language design was close to. I do know that I cannot believe that every detail that J.K. Rowling puts on the first Harry Potter book was intended as.

[00:05:20] Some important Wolf point game part 6 or 7. And even if J.K. Rowling was such a genius that he hadn't actually owned land, that and that was who I think we found out the FUTHI was.

[00:05:44] But who won? It was just like a temporary diversion during the train ride. And otherwise, you didn't hear much about the rap with fingerprints in languages.

[00:05:55] I often. That's exactly how things go. You you choose some detail because you have to commit. You sort of you have to pick keywords. You have to choose a style of coding like you maybe you choose indentation instead of curly braces or maybe you don't. You're a special case, but whatever you do in some way, you're stuck with them and you find new uses of that. That detail that you picked before you knew how important that would be and the sort of the craft of designing the language is on one hand pick your initial set of choices so that there are a lot of possible continuations of the story. The other half of the of the art of language design is going back to your story and inventing creative ways of continuing it in a way that you had not.

[00:07:15] So I think each of you have languages that span multiple decades.

[00:07:19] So what are some of the principles that made, you know, so so Javal would sort of on in the. It didn't come out of like a personal passion project or something. It was. From trying to build a prototype, you know, I mean, we were trying to understand it because the. A lot of time talking to people who build software or embedded systems.

[00:07:59] And you know, what was it in the whole process? How did their systems fit into the universe? It wasn't because I was trying to worry when we were worrying about things outside of data centers.

[00:08:22] This was a project that had about a dozen people on it and my little. Yeah, but it was my my little piece was what it was was sort of like.

[00:08:36] You know, we we we we sort of figured out that part of the problem was that. See has issues. And and so I started out so. So out of this like large pie of a project, my slice was.

[00:09:01] To make things a little easier from a programing language point of view and fix the pain points that came from the programing language part.

[00:09:10] And it started out as kind of do a better see. And then it and it got out of control. The rest of the project really ended up just providing context. You know, the only thing out of that project that survived was was Java.

[00:09:31] But but but the fact that it was sort of directed at a set of pain points. But it happened to be it, you know, about people who were living outside of data centers and people who were. Getting shredded by problems with networking and security and reliability.

[00:09:56] They had to build things that ran in hostile environments like in the homes, which in any home with a child in it is a it is a hostile environment. Speaking of hostile environments, feral. What are some of the guiding principles for designing for a business? I love you, Larry.

[00:10:40] More and more coming out of this. A computer scientist. So I sort of actively ignored some of the computer science literature of the time and said, well, what can we just talk together? One part of the brain will work more like a natural language instead of. But instead of putting on a campus university campus and deciding world, the walkways would go, we're just going to see where people want to walk. And they put shortcuts in all those places and know that more as a network, not as a terribly orthogonal or computer science or mathematical thing.

[00:11:20] And then that turned out to be in the right place at the right time for bootstrapping a lot of the web and.

[00:11:31] It also got used a lot for system administration and so various principles that relate to trying to provide API eyes to everything. I'm trying to be both a good text processing language linguistically, but also a quick Google language. Here's why it was. Kind of very useful, was hasty and there was text and, you know, database seconds that's that needs. And so it was in the right place at the right time. We were very fortunate to have that time because in the 90s it became very stable, a lot of a lot of widespread use. Lot of people were you know, the language itself was stabilizing in the form that it was. But there were a lot of issues and salat. And so in the year 2000, we we took a step back and and and basically said, we're gonna break everything he's breaking.

[00:12:33] So we kind of did the same thing. You know, it's done the pipeline to the step, except instead of breaking a few things, we decided to break everything here. So we came up with a whole raft of design principles and 50 years since then. By the way, forensics did come up two years ago. It's getting faster. And it's suddenly at the beginning we said we were going to do something impossible and and fail at it could be a very big failure. And so far, this proved me to be that. But in the course of the fail. Yeah. In the course of that, we came up some really interesting list of about 60 different design principles.

[00:13:19] When read, even though I think many of them have already, you know, you know, kill two birds with one stone, pick the right people, you know.

[00:13:29] It's possible like that. But we came up with some cute ones, like, you think that's cute today?

[00:13:38] Oh, oh, know.

[00:13:43] Conserve your brackets because SFE, even Unicode does not have enough brackets that don't reinvent object orientation queerly, which arguably provided no substitute strong features for weak buttons, late binding sometimes oposite programs be. But some of the major ones are, you know, a lot of the stuff is just perfect to screw over time. And, you know, there were a lot of weird magical variables I'm going to. And so. A great deal of the redesign was to say, OK, what is the right peg to hang everything on? Is it off to Korea? Is it something in the works for school or in a larger school? Or what is the right to have each piece of information on? And if we don't have that thing, how do we create it? Great question.

[00:14:48] And since it's such a great question, Anderson, the spot and Terrence Parr had a quiz about while programmers value simplicity, they more often value powerful functionality.

[00:15:02] An amazing one liners incurring the cost of complexity. So for express. Yes. So what are your thoughts about simplicity versus complexity and some of the principles that might guide you, as you know? Really?

[00:15:22] Suppose I. I was always late in doing the language, saying that I've done. Always tried to make it so that there was only one way.

[00:15:37] So a lot of languages have four different ways of doing something and you pick the wrong one. And then only later you realize you went down the wrong branch and now you've got it back up.

[00:15:49] And so. So it's not it's not always easy. And I think there's like often we're guilty of creating what I call this thing I call simplex. Take something from blacks, you grab a single wrapper on top of it. That's make the complexity go away, but you're really creating is simplicity. You know, it's just like a bad abstraction on top of another bad abstraction that is simpler. So I don't know. I mean, it's it's we think that as picking up correct default overly complex thing. Yeah. But but I mean, the thing about that, about language design is like like any any decision you have to live with forever.

[00:16:38] I mean that and in languages you you can always add, but you can never take away. And so you've got to actually, as a language sign up to be very judicious. About reasoning over not so much what to put in, but what not to put it, an army because people come to you all the time. Wouldn't it be great if you could do this with that? Are you going to? Well, yeah, but you can't fundamentally change the nature of the beast. If you created an empirical and an imperative programing language, you can't turn it into a functional program. Show me you can you can borrow from functional programing. But it is what it is. And you gotta stay true to the nature of the beast, or you gotta create a new beast so-to-speak. Right.

[00:17:27] Any other thoughts? Excuse me? Can we interrupt you or you may have to interrupt any time. So.

[00:17:38] This is sort of the point about what what do you leave out? And the what I remember in the early days of Python's design, there were so many people who thought they had a good idea. Is sufficiently critical to say no.

[00:18:03] And so there are a few things that didn't work out at the time, I I haven't learned to say no. And I quickly enough did learn to say no. And I sort of felt like several lines of defense. Like the first line of defense is, hey, you think you need a new language feature, but actually you can already write that typo if you needed the locked right to function or a module. Well, if they say, yeah, but everybody needs that. Well nowadays we say put it on the package repository in those days. I said well maybe you can propose a new standard library module. That's a lot cheaper for the language design than the new language feature. And another line of defense was, well, you can actually write extensions in C or Fortran if you really care to. And you can help yourself that you can modify the behavior of the language in ways that aren't accessible when you just write the pure language. But you can still sort of you can extend it in a way that doesn't fundamentally alter the core language. And if you've tried all those things and and you failed, then maybe you could argue for, well, you have to change something in the core language. Hi, I'm interrupting.

[00:19:39] I find that whenever I get feedback on whatever whatever it is, program makers, I've been working on it. It's people come to you with a with a particular instance of a problem. And it's your job as a language assignment to tease out the class problem they're talking about and then try to understand whether there is a feature there underneath that you could put in place that is broadly applicable, not just to that one crawl. You're putting in place a class, not an instance suit. So that damn slide. And if all you're talking about is an instance, then you're probably better off. Save your money for power.

[00:20:32] Actually found this. In the early design of the Perl 6, we asked our CS I expected. Since we got 361, so we have this in spades lately. And, you know, it's completely overwhelming. Until I hit upon this principle, which Anderson alluded to, which is basically a lot of the proposed solution. Yeah, but there is a pain point underneath.

[00:21:01] And if you look at more enough of these RFD and they haven't seen pain point, there is something you should address. There's some fundamental unification you should do underneath or something that is is clunky that could be fixed. And especially if you're doing a complete redesign that you can think about the sort of. To fix it. How do you know? There were a lot of problems where. Difference that just won a victory with themselves. You get like 40 different.

[00:22:35] Released. It was really clear in the way that this was like a big issue and you feel dry and I came about as close to spilling blood on this topic as I've got. And, you know, I would much rather. Do something much, rather leave something out there, do something stupid. One of our principals planned to visit Margaret later. Yeah. Well, well, what that means is that that means the opening answer is no. No. Right.

[00:23:14] And one of the things that happened with like a whole bunch of different outfits and their job was that they kind of turned into competitions and generics and closer to work both. Probably the most hard fought competition. People were writing pretty steep pieces on the topic. And I was like trying them all out. I mean, some had no shortage of smart people at the time, but it's really hard to be a planet full of grad students, even grad students. At some point, I think most kids who were in grad school. That's correct. And in linguistics and linguistics. But it's.

[00:24:00] Oh, probably 20, 30 years ago to say I'm going to write my own language. And people are very different. Back then, the Internet was sort of starting. Maybe not that they're single. I think we have computers. We have cell phones.

[00:24:20] Nobody cared, but nobody cared. What's all that about?

[00:24:24] Nobody cared what you did with the eighth that.

[00:24:31] Decades. But these are languages that persisted. Many changes and they're on different hardware.

[00:24:45] When you started designing your language, what was the key goal that you were trying to meet versus using some other extinguishment? Well, I mean, for me, it's never been people. And one of the secrets of success is try to solve as many problems at once as you can. And, you know, if there's just one problem, there's probably an easier way.

[00:25:17] Right. Right. You think that way you'll never have a new program.

[00:25:25] Well, there's always a better solution in the 300. Well, it depends on how much you pay. Chasing down memory, corruption.

[00:25:34] But the three virtues of the program are. Blasi This impatience and cupids takes Kupers.

[00:25:46] The world needs another program like. Is that a code? But the thing about programing language is that I think a lot of people don't realize that every programing language is about 90 percent the same and maybe 10 percent if you're lucky.

[00:26:06] And a lot of people get very focused on the 10 percent new and forget about the 90 percent of the same. That makes it at great length. Right.

[00:26:14] I mean, there's a lot of hard or you work in creating programing language, loss, semantics that it's got to worry about.

[00:26:21] And like everyone and loves to talk about syntax and syntax is the easy part. Semantics is the hard part.

[00:26:30] You know, like how do the types of work and one of one only dogbone supported promotions and one of the different kinds of type constructors that the language, as etc. said are these are the hard things to do to design. But, but but not the things that people people love to bike shed on the syntax, you know, shouldn't be a colon or or comma.

[00:26:52] You know, it's like, oh my God, it's have a long thread about that. So talking about people having opinions and syntax and typing.

[00:27:08] These languages don't all take the same approach to typing. Maybe we'll start with veto and talk about typing in Python and then kind of work our way around.

[00:27:30] I mean, I just remember that when Pyfrom first happened, it was not a class. It was a little conversion function. There was an internal object type, which was really just kind of a Zite table that represented integer objects. And there was a built in function if you needed to convert the string to an integer and that was we had a bunch of those functions and we realized that we we had made a mistake.

[00:28:11] We had given users classes that were different from the built in object types. And for a long time, it was like, oh, well, like the real stuff is implemented in C and the user writes a little bit of blue top of that. And when when I found out that we had 80 different competing web frameworks, it was sort of time to to realize that people were writing much larger programs. And that's a different approach. Two types was needed. And that's where we sort of reinvented the whole approach to types in Python. And there was a bunch of cleanup that didn't happen until 5 3, actually. But one of the things we introduced and we were lucky that there weren't actually despite those 80 web frameworks, there weren't enough users that we were completely stymied by backwards compatibility. Yet like we are now. So we just one day we changed the function into a designated for the class. And sort of followed that calling, the class would be constructing an instance of the class so that if you had simple code that wrote in left brand some expression right. And would work exactly the same way that we have the same effect. But the workings inside were completely different. There was one particular file that's sort of implemented the that's sort of the basic functionality of types in the language and it in for one to it was like 50 lines of code. And at some point I rewrote it and it was five thousand by.

[00:30:16] That's great. What about Titus in Dublin? I got this this this one history of caring about things like. Building robot software often that that that comes out about, you know, I'm much more worried about what it takes to build production quality software than about what it takes, just like you like a quick thing and chuckles not a great language for.

[00:30:59] Things, but it's it's OK, it's not for me. One of the one of the things that I love to do and maybe I'm weird, but being able to do automatic or improving on hunks of cup and and. Type system is a really great place to be, one of the foundations there begin they're improving it for. Four months of code turns out to be really useful for things like building optimized. And doing it ahead of time. To be able to hear improvable way as many things as possible, so satellite like like, you know, one of the not well-known things about about Java is that, you know, in the jobless pack, you know, it says Ouray, subscript checking is always on. But, you know, it's only conceptually all of his all right. But the truth is that there are there's more than on my books or the compiler to improve away almost all index checks. And same thing with like null pointer checks and all kinds of things that look like they're heavy weight, but they're really cheap, you know, so it's a one of the things that I like. I really cared about it.

[00:32:37] The time was that A plus B should almost always be at most one instruction bonus points for zero instructions. What about after? Well, the thing about zero instructions is often you can like fold them into some weird addressing mode. Right.

[00:32:59] And. And so in kind of the universe, I'm right, I tend to live. Being able to look at A plus B and realize it's that destruction that all feeds off of the type system. And sometimes you can you can do this with sort of optimistic just-in-time coupon compilation, depending on how far you want to push that. So like the the JavaScript engine in Chrome is is absolutely astonishing for the the kind of core that they go through too optimistically. JavaScript code, but it's also very hard to do those kinds of tricks if you're trying to get into smaller devices. So, you know, there are there are Java compilers that work for machines that only have like. You know, to do that kind of compact distillation, you need every kind of hope that gives you every little every last drop of information. And the earlier, you know, what better job you can do.

[00:34:20] Right. So speaking of lots of information that one gets, typescript gives you a lot of flexibility and power. What's your general view that typing?

[00:34:34] Well, let me actually it's funny that you mention cycles and counting. How many? And whenever I do, I remember when I wrote Turbo Pascal, which was all written in C-A-T assembly code.

[00:34:45] And back in those days, you could literally look up the intel manual on the sidewalk or whatever, you know, and see how many clock cycles every construction took. And actually it would work out just like that.

[00:34:56] I remember now everything takes zero clock cycles except when it takes a thousand clocks. The memory and it is absolutely impossible. The reason about what I've seen you use execute your code today. That that's just one.

[00:35:13] Well, it's not impossible, but it's much harder. Yeah, it is a complete pain in the ass.

[00:35:21] And they don't give you manuals. Not good at all. You don't want to actually understand that. If you need to get the chip diagrams. But but with respect to types, I've always worked on programing languages that have type systems.

[00:35:35] But it's interesting how it's gone from being type systems far for the code generators sake or type systems for four for, you know, generating errors too. Now I almost view type systems as as a tooling feature, and that's really sort of the thing that has been interesting. High automate for typescript. It is, you know, first of all, starting with a dynamically typed programing language like JavaScript and then trying to add a type system on top of it that that faithfully reflects the semantics of the programing language. And the reason for doing it is actually not because we think type systems are interesting, but because if you think about what it is that powers programmer productivity today, like everyone loves their I.T. like. Whatever you're using, I hope it's Visual Studio code.

[00:36:29] But putting that if it's not, you know, I am sure you're all like accustomed to things like state completion and refactoring code navigation and go to definition and so forth. And if you think about it, the things that the thing that powers that is semantic knowledge of your code. And the thing that provides the semantic knowledge of your code is a compiler with a type system. And once you add types, you can actually dramatically increase productivity, which in some ways seems counter counter intuitive. Right.

[00:37:03] I thought like dynamic languages were easier to to to approach because you got rid of the types which was like a bother all the time.

[00:37:11] Well, look, when it turns out that you can actually be more productive by by adding types if you do it in a non-intrusive manner and if you work hard on doing good type inference and so forth.

[00:37:23] So, so, so, so, so, so. So I want to jump in here. And then I take over.

[00:37:34] So I really, really, really believe in that point in the in the in the power tools for power geeks.

[00:37:44] And one of the things that drives me nuts is the real man usvi movement. And, you know, it's it's really I just want to punch people who are.

[00:37:57] But I'm really inside state. I I'm a real developer because I use v.i. And it's like, you know, I do it the hard way. I think.

[00:38:07] I think I make language developers lazy.

[00:38:20] IBRD Let me get a lot more done a lot faster, OK? I mean, I'm not I I mean, I'm really not into proving my manhood. I'm good at getting things done.

[00:38:36] Case in point, but I take the devil's advocate here first said he meant something like.

[00:38:43] Yeah, but you their notebooks, a lot of science people. Data scientists get a lot done.

[00:38:51] It's actually a pretty simplistic I need a dynamic language. Most of the time.

[00:39:00] So you could get a lot more than that. I don't know. No, but I think typeset. I mean, it's not just a type system can help you.

[00:39:12] First of all, if you're uninitiated and you want to know here, here's my food and now I say food dot and then the ivy can show you. What can I type next? Right. As opposed to I gotta go read manuals and figure it out or know it. All right. I mean, your original developer of whatever piece of code might know it all, but then people move on and new people come in.

[00:39:34] You know, there's here's this project. There was there was no documentation written about it. And if you think about it, types or documentation to write, I mean, and then there just there's so many things about like like adding them that down the line give you increased productivity.

[00:39:54] So there's productivity and then that's that's maintainability.

[00:40:00] Oh, well both. And maybe any different thoughts about how you get to talk about types. Well, talk.

[00:40:09] The colonel types out. Well, that story is very different for profile purposes. As you know, Girlfight, to sort of rural over time and the whole idea was sort of particularly idea.

[00:40:22] Everything you can pretend everything is a straight even if it's a number or floating point internally, it's stalled, you know, interchangeable. And that works out for bootstrapping people into a language. They're quite nicely. And it's it's a feature that we wanted to keep in Perl 6. But what we discovered in Perl 5 is part of the redesign was it's it's fine if the user is confused about the interchangeability to tell more for the Muslim to take the care of your standard.

[00:40:55] Tikes But it's not so good if the computer is confused about.

[00:41:01] You know, for all the girl wanted, it was written way back in the dark ages. You had a schuberg, a bunch of stuff in, and it cheated a lot. So part of the redesign when we did girls school Perl 6 thing we wanted to do object oriented programing that is that these languages and we wanted to be functional programing that are the most functional programing languages. To do that you have to have a fundamental very sound type system of a sound object model underneath. And you really have to take seriously these slogans like everything is an object. Everything is like closure. Every feel the process, even blocks are closures. And he has got to optimize. Hard to get up, get them away from there. But but I I also agree with this idea that the tikes also are cultural. I talked about Pei's hanging things on pace. They're one of the one of the picks. We didn't have something to hang on the prettified. And now we we can Pangle. But all of that information and then just run a little program there. And then you have to have an idea. They say, well, this object. You know what methods.

[00:42:15] But what you know, they could do their own introspection. The whole thing. And people just you know, I'm baffled by the way we do also have an idea.

[00:42:27] Good points all around. So most of the time programmers actually spend maintaining versus writing new code out in the wild and.

[00:42:41] What things or elements of a programing language make it easier to maintain? And maybe we'll start with Fito.

[00:42:52] Well, I found that. TypeScript is actually incredibly useful, and so we're adding a very similar idea to Python, adding it in a slightly good way because. We are taxed.

[00:43:28] All right. Go. Anyway, we have time. I sort of learned a painful lesson of that before school programs. Seafair dynamic typing is great for large programs. You have to have a more disciplined approach and it helps if the language actually gives you that. It's rather than telling you, well, you can do whatever you want.

[00:44:02] Yeah, yeah. That was part of our scale up idea for Perl 6, the typist would help with programing in large because we didn't notice those limitations.

[00:44:16] One year ago, I started working really heavily on Unbury Factoring In. And, you know, being able to do like large scale refactoring is unlike millions of lines of code at once, that really changes the way you think about maintaining code.

[00:44:37] Because, you know, the world is filled with libraries that have things like stupid variable names, stupid method names, because, you know, when you started out, it meant one thing. But now you realize that you really didn't really understand what you were doing in the beginning. And so this this method name really should be different. But nobody ever Reding's methods because it's really hard to go over a piece of code and rename. Exactly. That's the right variable. But if you've got a good refactoring engine, you just you just type, you know, medic control are typing the new name, you know, hundreds and hundreds of files later, which takes about, you know, maybe 30 seconds if it's a lot of files. You're done. And so you basically be right.

[00:45:34] Seven-part series, I hope this is this is actually like you're describing, that's what we what.

[00:45:43] What was the genesis of the typescript project, which was these enormous JavaScript code basis that we were seeing customers. Right. And then in-house projects that they were getting bigger and bigger because JavaScript engines were getting good enough for you to write really big programs.

[00:46:00] But maintaining them was impossible because it turned into write only code. You know, you do dare not touch it. Once you've written it because it worked then and then. But, you know, there's this property called text.

[00:46:13] And I really want to change it to be like some other name, except there is like a million other properties called text. And if I'd just like to do a global search and replace for tax, then oh my God, it's like at all. So I can't change anything. But if you have semantic understanding up like this property call text is different from that other property over here.

[00:46:35] Call text. And if you want to rename this one, that doesn't mean you want to rename that one. But but that takes for something like a type system to be in place. And, you know, once you start adding that, you add documentation to the code and you add these capabilities where we're now seeing people like with regularity, refactor one hundreds of thousands of lines of JavaScript code, you know, we'd like ten minutes and end it and it just works afterwards. And people are like amazed that it's possible.

[00:47:06] But but but it's true. Said it takes a bit more work to begin with. And it's not maybe not right. If you're writing five lines of code that maybe more bother than you really care to. But but, you know, it started to pay off pretty quickly.

[00:47:21] There are different types of really good lexical scoping helps with refactoring. It gets back into this. What's the right pick to hang the sign up and off and it's slope. It's a lexically skulked or dynamically scale. So now we're doing those right and not interfering with threading.

[00:47:43] So I have one more question now before the break in three minutes and we made sort of a reference to Donald New, one of the turning winners in this last section of conversation. And he wrote, Premature optimization is the root of all evil. So what's your approach to optimizing code and gaining efficiency, saying Python?

[00:48:12] Well, I I'm terrible at that. So I leave it to others.

[00:48:17] That is a very wise language creator and they've lived up to the task.

[00:48:26] Piceance hash table implementation has been rewritten so many times.

[00:48:32] I don't understand any part of it, but it is so much better than the thing I actually copied out of truth 30 years ago. Happy to have a clue.

[00:48:46] Okey doke. I kind of 50 percent believe that because they're, you know, the premature optimization in terms of algorithms and code. That part of it, I believe. But people often don't really think about the role of data structures in optimization and interpret data structures exposed.

[00:49:16] You have to know and you decide that, you know, this algorithm is a problem. But, you know, the reason that it's a problem is because, you know, this data structure was just slapped together and they thought, oh, they'd only ever be like 10 items in the list or 10 items in the set. And then and then you start, you know, having, you know, million eight million items sets all the time. And all of a sudden, you know, you're a little linkedlist is. And that is not gonna happen. Right.

[00:49:50] Ex-directory or a unit or. Yeah. And and so and so if you can you know, it's like you know hyd early hide.

[00:50:01] Often. Right. Never tell the truth about what your data structures are.

[00:50:09] And on that note, we're going to take a quick look.

[00:50:14] I just want to say that I think like with anything, the important thing is to make sure that you actually have the right data before you you start optimizing. Right. By too often you have a hunch that, oh, I think finally I need to do this.

[00:50:32] And I need to change this data structure from being a linkedlist to being overweight or whatever.

[00:50:36] But but if you measure, then you discover it doesn't matter. And but really, there's this other thing that you haven't even thought about that matters enormously. And maybe you should go look over there. So get perf., get it, use a profile or figure out where it is, the time actually spent. I am continually surprised and I write compilers for a living and I'm always surprised by where time is spent. All right.

[00:51:01] And that is that we're out of time for this section. We're going to take a quick 15 minute break.

[00:51:09] And when we come back, the really hard questions are going to come out because the audience questions that have been submitted are going to be the ones that believe in you. So thank you.

[00:51:55] And teach in every one of you has a community that is formed around the languages that you've created.

[00:52:00] I'm curious if you were talk two ways in which you've seen the design choices you've made for your language, so shaping the communities that are formed around.

[00:52:11] It really helps. Well, good. It really helps if you're trying to make everybody look what's here with your slogan here.

[00:52:24] Everybody comfortable? We're doing that right now. They think they can make all of all Americans, I heard.

[00:52:37] You know, feel like comfortable programing. You see it for all. Yes. To make sense of it. I guess you know what?

[00:52:48] The sign was a clue.

[00:52:51] It's not see us for all of us, just Americans. No. There are many underserved populations in the world.

[00:52:57] And what we've really discovered that it really helps build international camaraderie and community to have world class literally support for unico. Yes. I would encourage any language designers out there. Don't do this halfway stuff with, you know, code points. Go all the graphics.

[00:53:23] I completely agree. Kita, you had something you were going to say.

[00:53:29] I was trying to say.

[00:53:33] I how my design choices necessary to make choices about how to use energy certainly have. Did I? I was somehow a dude with the importance of user centered based something because by comparison. That was ABC long lost, its authors had a of things right, and one of those was listen to the users asking users what they think. And then again, you making a new one of their suggested solutions. And you first figure out what nobody spends. You gave users a voice. And I said, take that back and let the users help. And now they're raising of.

[00:54:36] I was going to say I always felt like it's important as a language society that you don't create unnecessary work for your community because it's it's very easy to in the name of purity or whatever. Go change something or or strive for the perfect language. And I've always said, like, I mean, show me the perfect language and I'll show you your language with no use or shoot over. Every every language has its functions. Has them because. Because it has evolved. And the hard part of language aside is actually not version one. It's like the older versions that come after you try to sneak things in without actually causing extra burdens on people who don't necessarily care about the things that you're trying to sneak in. Right. And so.

[00:55:27] So that's the hard part of language design, I think is is the evolution of the language and keeping your community with you as you have all.

[00:55:40] Yeah. And so and so with with Jarbawi like always followed that we've maybe been a little bit excessive about it. I mean, I've got binaries that I compiled like 20 years ago, but still run. And and it's it's, you know, on machines that were never invented when the code was compiled. And you only get there with with crazy discipline and and a a tolerance for living with your mistakes. But but it but it makes the you. It also help. It also ends up selecting who your users are because in the in the Java universe, pretty much everybody is is really disciplined because, you know, it's kind of like like mountain climbing. You know, you you don't get there, get sloppy with your gear when you're mountain climbing because it has a clear price. And, you know, when you you said something about Larry, you said it said something about Unicode Java's had Unicode since ninety two. So, you know, it's, you know, trying to be inclusive. Yeah. And and that really made a difference. I mean lots of folks thought it was like the lamest thing ever. But that was that that was during my my, you know, Lord of Java phase which. Yeah. Yeah. But that that phase of my life ended in about ninety five.

[00:57:25] I'd like to say something possibilitythat.

[00:57:30] And that is we did this perfect try to perfect compatibility. All the way through. I didn't really think.

[00:57:45] That eventually runs into the problem. But, you know, you can't I can't fix your mistakes. So one of the mistakes that we made was was kind of a mistake and that was assuming that you had to always have that kind of stability and not change anything.

[00:58:04] How would it be to design a language where the language itself could be evolved from within by the community and the language we want?

[00:58:17] Twenty five or 50 years from now, this ties into what Polygram was talking about, a hundred year language. We kind of took that seriously. What is it that prevents languages from evolving, not being not having things still thrive?

[00:58:34] But fundamentally, the Perl 6 compiler is written in Perl 6. Most the runtime system is written in Perl 6. The users can extend it. Perl 6 itself does not care whether things are built in or not. It all works the same. So most of the buildings are written as if they were used or code.

[00:58:55] So users, even if we say no, we don't want that language right now. But put it in a module you can do. You can turn classes in two monitors with about that much code. You can put into implement actor model classes with about about how much code you can you take. Adding operators is just trivial. When it does, it does everything perfectly one pass and drops into the sub compiler. And with the end, it comes back out. So there's never, ever any ambiguity as to what language you are in.

[00:59:32] And so because this scoping of the exact language you're in is always like if we scoped any lexical scope can say use whatever language I want a feature version of Perl 6. You could say use COBOL, use Python. If someone actually people have written Python parsers in April 6, I don't think anybody's going to call for sure yet, but we'd like to think that, you know, a lot of language stuff like languages and Perl 6.

[01:00:03] So I think that I think the third module of semantics would likely. But I think there is a way us. I think there is a way forward to get the stability and rehabilitation.

[01:00:19] And I think.

[01:00:23] Awesome. Do we want to take another audience question?

[01:00:28] All right. Thanks.

[01:00:36] Can you talk into the mike like you're eating the mike? Hi. Yeah. Can you hear me? Yeah.

[01:00:42] There's been a lot of talk tonight about the evolve ability of languages and the trouble with implementing things that might not be backwards compatible. What's something that you wish you could have implemented with your language? But for that, or maybe another reason that wasn't possible.

[01:01:02] A word, right? As all the words or these or other forms of naming is hard.

[01:01:31] Well, I'll I'll I'll go. I've. I mean I my favorite is always like d that the two billion dollar mistake of the billion dollar mistake of having Nall in the language. And since JavaScript has both null and undefined, it's the two billion dollar mistake.

[01:01:48] But I mean if it did.

[01:01:52] And I mean what's done is done right. And and now now, you know, I spend a significant portion of my time actually working on ways to to make code null safe to and who's never who in here has ever had a null pointer exception.

[01:02:12] Right. There you go. It's it is by far the most problematic part of language design. It it's a single value that that if only that wasn't there. Imagine all the problems we wouldn't have. Right. If if type systems were designed that way and some type systems are and some type systems are getting there. But boy, trying to retrofit that on top of a type system that has normal in the first place is quite an undertaking.

[01:02:41] The features I wanted to add were negative features. I think all of us as language designers have borrowed things. You know, we all steal from each other's languages all the time.

[01:02:55] And and it's often we steal good things.

[01:03:00] But for some reason, we also steal bad things. You know, like what? Like regular expression syntax.

[01:03:08] Oh, yeah. I did that one. Like the C precedence table. OK. Another.

[01:03:15] OK. These are things that I could not fix in four or five. And we did fix it. Burleson's. Awesome.

[01:03:25] All right. We do tend to be doing pretty well with the audience. So how would another audience question?

[01:03:36] Hi. Hello there. So, well, what I wanted to ask is you can sort of know what is these sort of pendulum effect in regards to tech and programing, throw it out. Time to Vinick's several different paradigms. There's a certain time where people were like, are we going to do things imperatively? Are we going to go object oriented or functional? And for example, right now there's a whole bunch of language used are sort of like very aggressively taking that standpoint of being very pro, being friendly with concurrency or being bery regardless of what immutability with memory. And I think that that kind of pendulum effect happens because technology has actually been evolving throughout time. Our machines are like beefier and we have more memory now. So that waste on we design programing languages now are probably a lot different than there were 20 or 30 years ago. So considering that, where do you think or how do you think the programing languages of 10 years into the future are going to look like? Or in your opinion, where where is language you design headed to in the future?

[01:04:53] So. So. So my favorite candidate for that is that, you know, these, you know, major breaks and things like that always happen because, you know, something major happens in the underlying technology.

[01:05:07] And, you know, one of the areas of underlying technology that is kind of like a computer that I think is really underserved is is writing code for GP use. Right. I mean, right now there are there are no languages worth that crap.

[01:05:30] And that's a technical for trademark that work well on GP use. And, you know, maybe because there's like a limited number of algorithms that people are willing to are really interested in, but they're really important ones. Or maybe it's because they're they're inherently all mathematical. And the number of people who care about writing numerical code is small. I'm one of those, though.

[01:06:02] And you know, and so every now and then, I spend time thinking about, you know, what would you do to make GPE you programing really easy.

[01:06:17] Yeah, but but but it's more than just vector operators. I mean that that's kind of my my problem with the whole thing is that most of these things are just libraries of hand coded assembly that do vector operators. And and surely there's something more clever than that.

[01:06:38] Maybe I'll just add one with language design. You know, one of one of the things that's interesting, you look at like all of us old geezers sitting up here and there they were, we're proof positive that languages move slowly.

[01:06:49] And I mean, it's not you know, like a lot of people make the mistake of thinking that languages move at the same speed as like hardware or like all of the other technologies that that that we live with. But but languages are much more like math and much more like the human brain. And, you know, and they all evolve slowly. And we're still programing and languages that were invented 50 years ago, like functional. But all of the principles of functional programing were thought of more than 50 years ago. I do think one of the things that that is luckily happening is that like as Larry said, everyone's boring from everyone and languages are becoming more multi paradigm.

[01:07:29] Yeah, I think it's wrong to talk about oh, I only like object oriented programing languages or I only like imperative programing or functional programing.

[01:07:38] I mean, it's important to look at where's the research or where the new where's the new thinking and where where are new paradigms that are interesting and then try to try to incorporate them. But do so tastefully in a sense and and work them in to work them into whatever is is there already.

[01:07:56] And and I think we're all learning a lot from functional programing languages these days. I certainly feel like I am because a lot of interesting research has happened there.

[01:08:06] But functional programing is imperfect and no one writes pure functional programs. I mean because they they don't exist. So it's all about like how can you tastefully sneak in mutation in ways that you can better reason about as opposed to mutation and free threading for everyone, you know?

[01:08:26] And that's like just a recipe for disaster. Right.

[01:08:30] So, yes.

[01:08:40] All languages are imperfect. Let's just settle it right there. You know.

[01:09:04] Yes, we've been thinking a lot about the the the parallelism concurrency issue. This is one of those issues where we were waiting to be smarter. Uh, we we did borrow. I mean, steal a bunch of ideas from Haskell in terms of lazy lists and things like that. But in terms of how you actually are going to deal with the end of Moore's Law or if not the end of Moore's Law, at least, you know, multi, multi, multiple C.P.S., many, many cores. When you have a thousand cores, what are you gonna do with them?

[01:09:44] And so a lot of our early design on Perl 6 was we didn't understand how we wanted to do it yet, but we knew very we knew very deeply that we didn't want to do anything. It would prevent it. So a lot of the early design decisions were just saying, well, we have to keep this open. Then when the right smart people came along, they said, look, what you really have to look at is things that are composed. All threads are not composed of, well, we can't take two threads and turn it into a third thread. What you want is things like we stole from go from. We stole promises and channels from C-sharp. We stole functional reaction will programing.

[01:10:29] And with these sort of primitives which end up being duels of your regular list processing kind of primitives, they just happened to be working on events. And you're you can have loops, event loops that were just like regular loops that they're running on and same control flow. So that is easy to learn. There are ways of sneaking these things into a language, at least in the ways that we understand right now that I heard.

[01:10:59] Was it. Who is talking about adding fibers to their runtime?

[01:11:04] I think you'd think it was a Java news item, but like lightweight threads that can you, as Erlang has done, has shown, you know, you can run a hundred thousand threads in your process.

[01:11:15] That's another language that's been around for a long. Oh yeah. And we stole from them too. Yeah. Should we. We like their pattern matching. But one of the many therms is that all of that language design is theft of course.

[01:11:28] Yeah. So yeah. I mean we're thinking about it.

[01:11:30] We don't know have all the answers yet but I think all these languages, you know, long term tend to converge on similar solutions.

[01:11:39] So very sadly we are at the last question. I know you went so quickly and unfortunately all good things must end except for maybe good languages. So I'm going to start and maybe quickly wrap up as you look back over the long lives of the languages that you've written.

[01:12:06] What have you found most rewarding?

[01:12:10] The people?

[01:12:11] The people by far after the.

[01:12:16] Oh, yeah. Yeah. I mean, that that's what keeps me coming back, that to work everyday is like the incredible excitement that the community shows. You and I I am so grateful for all the people who who have used that. I mean, it there's nothing more rewarding than have people just like on Twitter or wherever it is, go, oh, my God, this is so great. And it's like, yeah, that makes you want to just keep doing more of it. I mean.

[01:12:46] Oh, yeah. You know, somebody comes up to me in the street somewhere and says, you know, thanks for giving me a career.

[01:12:54] Can I have a selfie?

[01:12:57] You know, that's like this is kind of like, you know, light and dark thing.

[01:13:02] But being this is really wonderful. This is all great. I mean, I have to be in Silicon Valley at a movie theater, the tire shop.

[01:13:14] Yeah. Well, what's really weird is when you're when your kids are with you when that happens.

[01:13:20] But but, yes, it's. It's when the when people say, you gave me a career, you get my livelihood. You fed my family. Yeah.

[01:13:29] That's the best. It's. And you know, and and not just.

[01:13:36] And we're not talking about a star relationship here. Each of us to those individuals, we're talking about the second order, Community 2.0 kind of effects.

[01:13:47] It's it's one thing to see them, you know, getting help from you. It's another thing to see them helping each other.

[01:13:56] That's a whole whole level above even the individual relationships. I feel to see a community that is learning how to love, you know, kind of building a little bit of heaven on earth. There's just nothing like it.

[01:14:13] Completely agree. Kita, you get the last word.

[01:14:22] And we love learning from you as well.

[01:14:24] And with that, I want to say we've all learned so much from all of our wonderful panelists tonight.

[01:14:33] I want to thank you for allowing me to share and celebrate this moment with you. The little fifth grade kid in me that was plinking away on basic at Bell Labs who later watch the P.C. cell phone, Internet, Web and cloud computing and change our world. Never dreamed that I would be chatting with four individuals who profoundly impacted our world by doing what they love, creating and building languages. Please join me in a huge round of applause to thank Anders for inspiring this evening.

[01:15:15] Thank you, Carol. OK. OK.

[01:15:19] Was that epic? So, Marina, I have an idea.

[01:15:24] If you guys could step down right here, we're going to get a giant picture of everybody. You guys stand right there. We're going to get up here with your camera. All right. We're getting a picture of everybody in that room. All right. I need some excitement.

[01:15:47] Oh, my God. I met my idol. She can stand on the chair. You can use my shoulder. OK. All right. Show some excitement, people. Let's see it.

[01:16:01] Monahan's. Three times it's kinda weird and lumpy.

[01:16:11] OK, thank you, everyone, for coming. Thank you to all of our speakers. Thank you to our organizers. Let's do something equally awesome next year.

[01:16:23] And if you want a robot there, 100 bucks. So come see me.