

A human view of programming languages

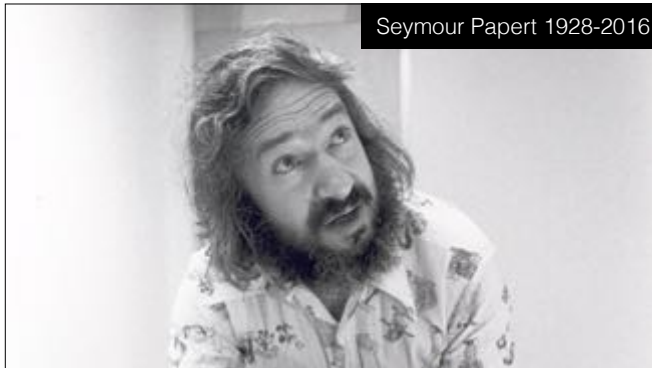
Andrew J. Ko, Ph.D.

SPLASH 2016 UNIVERSITY OF WASHINGTON



- It's been a great week at SPLASH
- One of the things I've enjoyed most is learning about the many valuable, diverse **perspectives** on PL research.
- These **perspectives** are actually what I want to talk about today.
- In particular, I want to talk about how our perspectives on PL shape the PL research we do, and in specifically the research we **don't** do.
- I'd like to start on a somber note.

Seymour Papert 1928-2016



- Seymour Papert died three months ago.

Computers as thinking tools

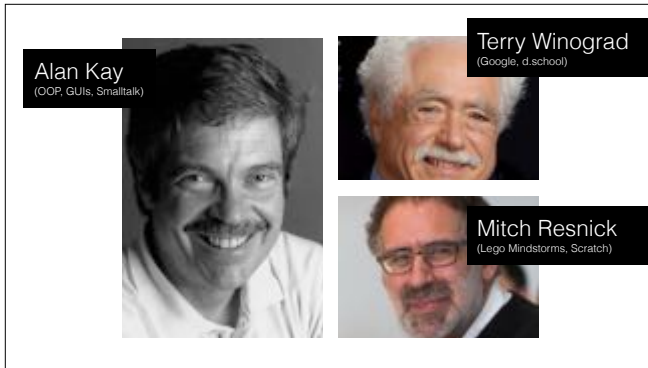
Constructionism

Neural networks

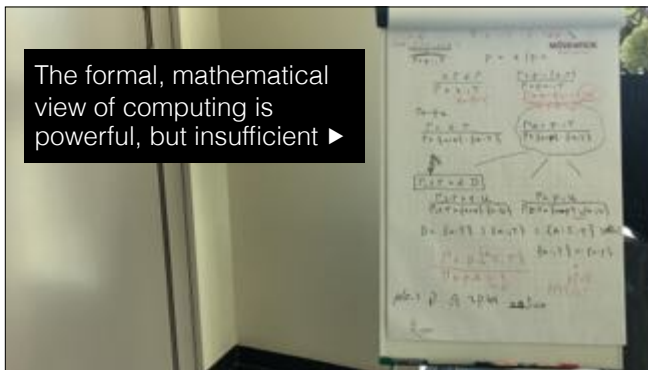
One Laptop Per Child



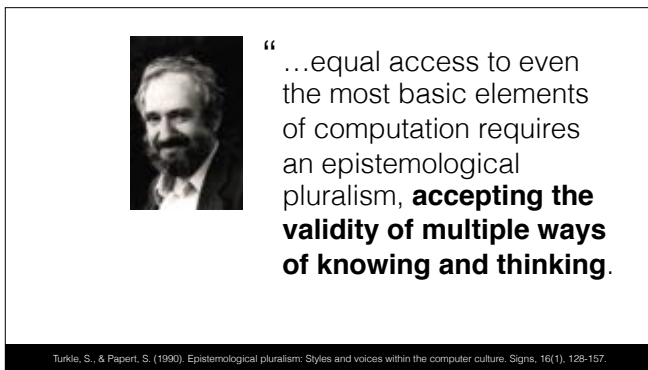
- Did many incredible things for computing
- (Read)



- He also impacted generations of impactful researchers, including
 - Alan Kay of Smalltalk and the Dynabook
 - Terry Winograd who helped spawn Google
 - Mitch Resnick who helped create Scratch, used by millions to learn to code
- We have to step back and ask: **where did all of this incredible impact come from?**,
- Papert wrote about this in the early nineties in his discussion of **epistemological pluralism** with Sherry Turkle.
- His claim was as follows.



- The formal, mathematical view of computing is powerful and necessary
- But it's not sufficient



- (Read)
- If we want to involve the world in computing, we have to accept the multiple ways of knowing and thinking about computing

What are these other views? ?

? ? ? ?

Computing
is math

Computing is a
thinking tool

? ? ? ? ? ?

- In this talk, I want to talk about these **other ways**
- Most of us in this room take a formal view of computing
- Papert took a learning view of computing
- What are all of these other ways?

Today's talk ? ? ?

? ? ? ?

PL is math

PL is a thinking tool

? ? ? ? ? ?

- I'm not going to take on all of computing in this talk
- Instead, I'm going to take the slightly smaller topic of PL
- I want to support the following claims
- (Read claims)
- To discover these conceptions of PL, I'm going to do what Papert would do

Today's talk ? ? ?

There are multiple ways to view
programming languages (PL)

These views have a vast
potential for discovery

- I'm not going to take on all of computing in this talk
- Instead, I'm going to take the slightly smaller topic of PL
- I want to support the following claims
- (Read claims)
- To discover these conceptions of PL, I'm going to do what Papert would do

Approach

? ? ? ?

Mine my experiences for
different views of PL

Explore these views as research
agendas

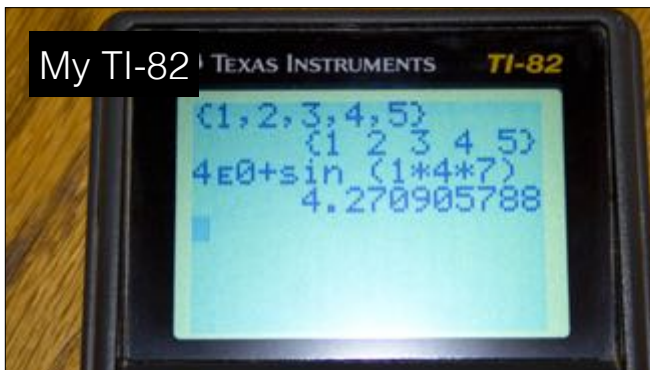
- (Should be at 5 minutes)

Middle school 1992

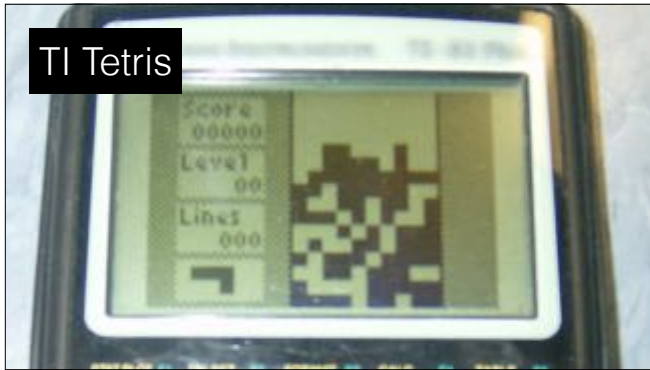


- To begin, I want to talk about the first program I ever encountered.
- I was 12 years old
- At the time, I was a typical adolescent: **no status, no identity, no control**
- My only passion was video games, because they gave me purpose
- I encountered my first program in a pre-algebra class

My TI-82



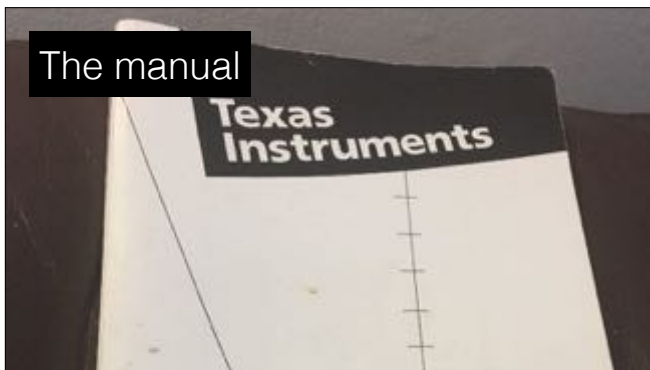
- Our teacher had required TI-82 graphing calculators
- He showed very little about how to program it other than giving us some trigonometric functions to copy into the program editor



- I had a classmate with an older brother who knew all about how to program the thing
- He showed us a version of Tetris that rendered on the graph
- Tetris was one of my favorite games
- I cut out the card to mail order a link cable so I could transfer the program to my device
- A month later, I got the cable and the program, and it was a beautiful, functional replica of the Game Boy version that I adored
- But, IT WAS UNBEARABLY SLOW



- Disappointed that I couldn't play the game in class, I realized that it was a program like the ones my teacher had shown us. I pressed edit, and was faced with tens of thousands of lines of conditionals, gotos, variables, all written in TI BASIC.
- Here was my first computer program, written in a programming language, but rather than viewing it as gibberish, in this context, it was something very different: it was a form of **power** that, if I acquired, would allow me to bring my favorite game to school in disguise.



- I got the instruction manual, read it back to back, and learned the language, so I could understand how it expressed the rules of the a game
- A month later, I had removed the rules that slowly rendered on the plot and instead rendered vertically on the text console, which was much faster.
- I had a playable version of Tetris that I immediately shared with everyone in my class, bringing me fame, respect, and glory (at least in math class)
- (And a scolding from the teacher, who suddenly had to outlaw graphing calculators from our class).



- TI BASIC was a way for me to exert control over the game, to tell it to behave differently.
- Also a way to exert control over the classroom, shifting my peers' attention to a game instead of algebra.
- Throughout middle school, I saw PL as a form of power over my class, my peers, my teachers, and my experiences at school.



- If great power comes great responsibility, what responsibility does knowing a PL bring?
- For example
 - Why aren't software developers in the U.S. responsible for the failures they cause?



- If power corrupts, how does PL corrupt?
- For example, Mark Zuckerberg amassed great power at Facebook by harnessing the power of programming languages to prototype social experiences.
- He used this to try to reform Newark, New Jersey public schools
- Widely considered to have failed miserably, wasting \$100 millions by ignoring experts
- Perhaps PL provides so much power it blinds people



- (Read)
- This is President Obama doing an hour of code in D.C.
- He announced the CS For All initiative this year, which is attempting to bring computing education to every public school in the U.S.
- Should we be doing this? If so, how?

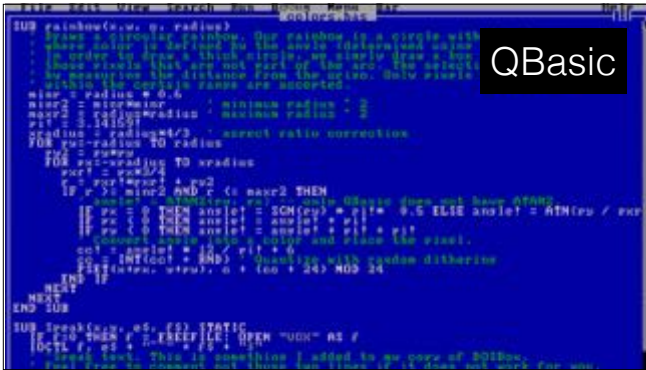
Rest of middle school was playing with code
(Unflattering photo warning)



- Wow, look at that horrible posture!
- As I finished middle school, I learned, I primarily used my coding skills to create.
- And this was my playground.



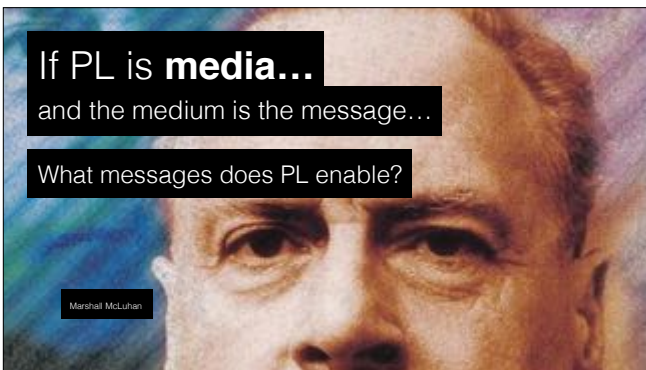
- I created animations
- I created interactive stories
- I created games with my friends
- I created games for myself



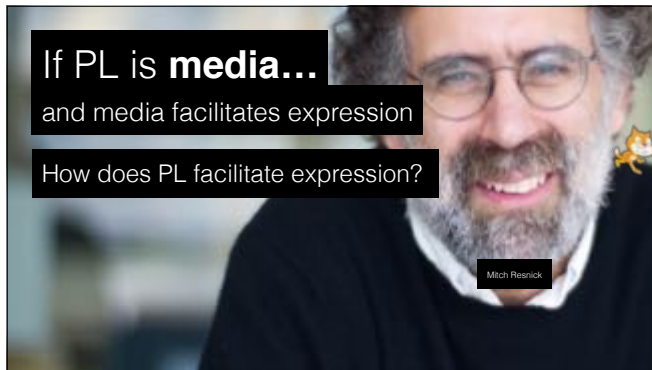
- I wrote on my calculator
- Also Quick Basic on my PC
- But the language was not the subject of interest



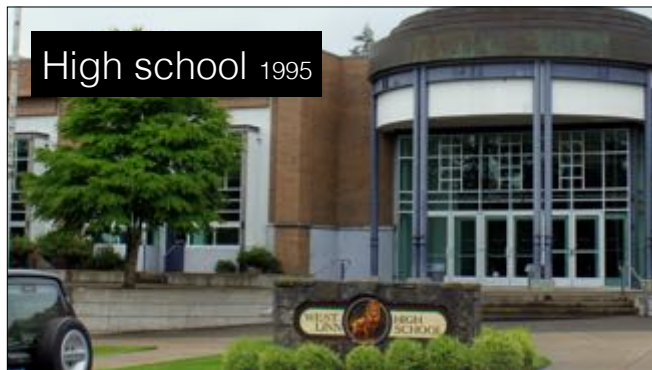
- PL was a way for me to express myself, my ideas, and share them with my friends.
- It was a **medium**.



- McLuhan taught us that “the medium is the message”
- If this is true, PL shape, bias, and even warp the kinds of programs we create
- How can we reason about these biases on expression?
- Do these biases go beyond *how* programs are written to influence what programs do in society?



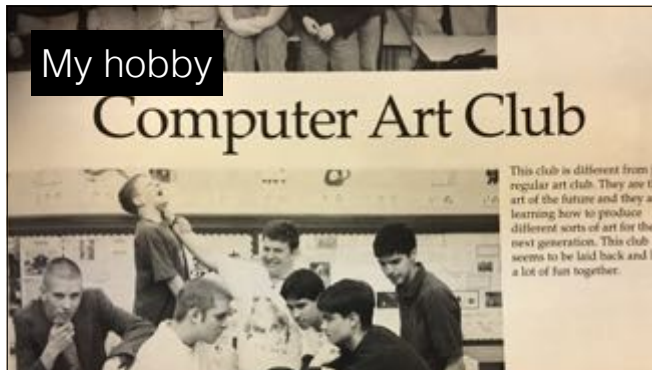
- If media determines what can be expressed, how does PL design affect expression?
- This is Resnick, who created Scratch
- Alan Kay, who talked about computing as “Personal Dynamic Media” back in 1976
- He’s spent the past decades trying to understand how PL fosters creativity, but also limits it.



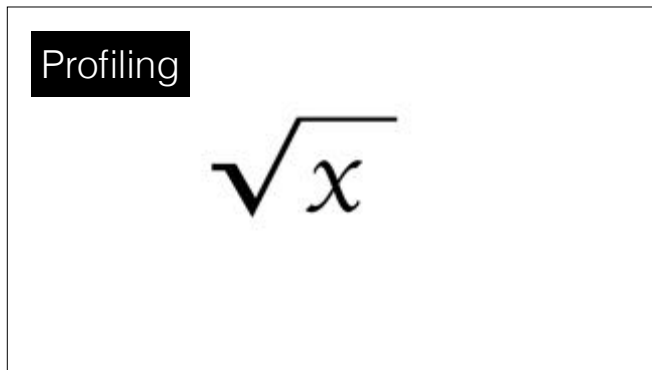
- PL was power in middle school
- Then it was an expressive medium.
- In high school, it transformed again.



- This was my computer lab at West Linn High School.
- I signed up for a zero period computer science class that started at **7 am**.
- (Yes, that’s how passionate I was).
- There were 8 of us in there, 7 who just came early to play MUDs
- The class was taught by a student from the local community college.
- He brought his homework in, challenged us to solve it, and then he would submit it in his class.
- We were using Pascal, because that's what the 386 PC's had installed



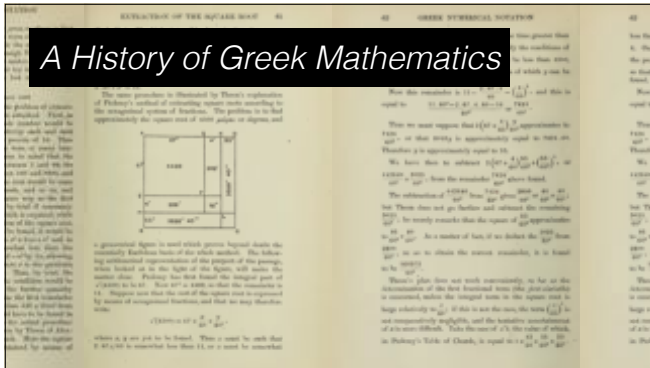
- This was my passion: the “Computer Art Club”.
- These were my friends, artists, musicians, and other weirdos who liked to do strange things with computers.
- At the time, I was obsessed with 3D rendering and all of the geometry it entails
- I had read about a game that was trying to render scenes with ellipses instead of polygons and I was curious if I could do the same.



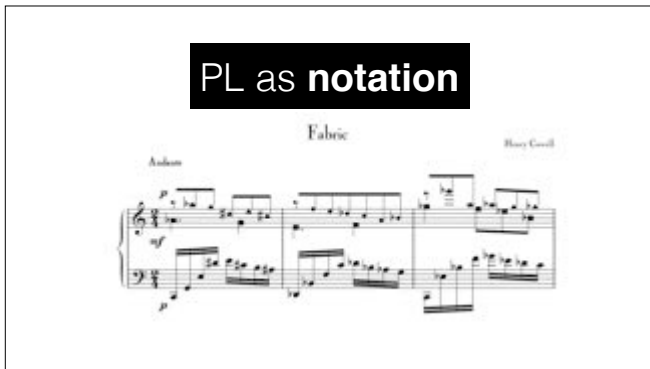
- My teacher asked what I was up to and I explained that the ellipses were rendering too slowly to do anything interactive
- He suggested I profile it and we found out it my rendering algorithm was spending 95% of its time on square roots
- He suggested that I talk to my math teacher to find out if she knew of any ways of computing square roots faster, that maybe I could use instead of the built in Pascal library



- Now, my teacher, Mrs. Hudson, was no regular teacher
- She had a Ph.D. in Math from Texas
- She didn't know any algorithms, but she did dig up a fascinating book on the history of Greek methods for computation from 1920's
- Through interlibrary loan, the little book showed up, she gave it to me bursting with excitement, and I dashed home to read it.



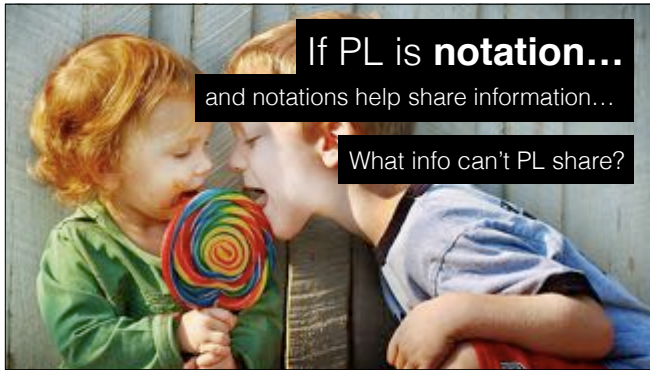
- Inside was everything I wanted: square roots, cube roots, nth roots, a whole bunch of algorithms for optimizing root computation.
- But the notation in this 1920's book was far from Pascal, and far from even math. It was its own notation for computation, a mixture of math, natural language, and other invented symbols.
- Translating from the book's programming language to Pascal required me to learn a new language to a level of depth that I could understand its semantics.
- I eventually translated the algorithm and greatly accelerated my ellipse rendering.



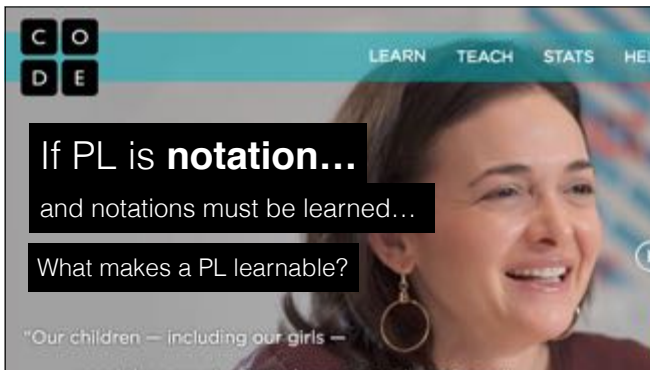
- Notations for modeling abstract ideas, like the rules of Tetris, or the concept of roots
- This view of PL as notations for modeling has many implications



- What can't our PL model?
- For example, how close can abstract logic get to representing the dynamics of trade?
- Do we need a different type of logic to model this?



- Are there kinds of information that simply can't reduced to 1's and 0's?



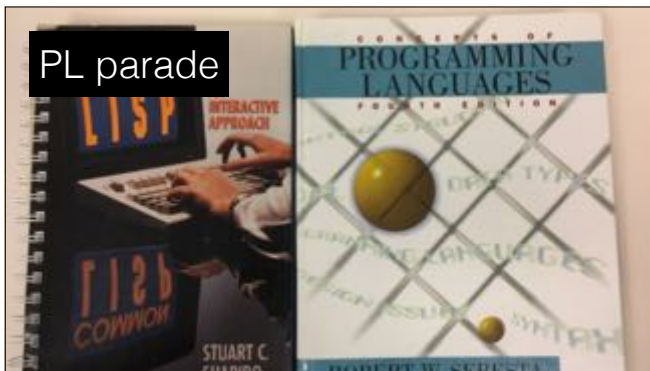
- There's nothing about syntax and semantics that is "natural"
- What about these language design choices affect how we learn these notations?



- After five years of obsession with code and art
- I found myself in college as a CS major.
- (I was also a Psychology major)
- PL was power, it was was media, it was notation
- I was eager to discover what else PL might be



- My mentor Margaret Burnett was the one who taught me.
- I met Margaret after seeing a flyer for an undergraduate RA position in the hallway
- It paid (almost) as much as the other summer job, and made a good pitch for the life of a professor
- I was sold.
- Not only did she taught me how to do research about PL
- But she also taught my PL course



- * She framed it as a tour through a dozen languages
- * Dissecting the tradeoffs of all of the different design choices they make.
- * We wrote the same programs a dozen times over, understanding the design space
- * She helped me see that PLs come from people. They design them.



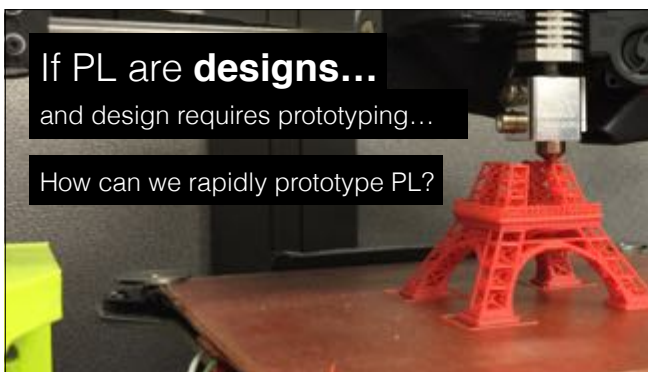
- And as designs, each design has
 - tradeoffs
 - flaws
 - principles
 - values
 - priorities
 - A whole process and set of people behind these decisions.



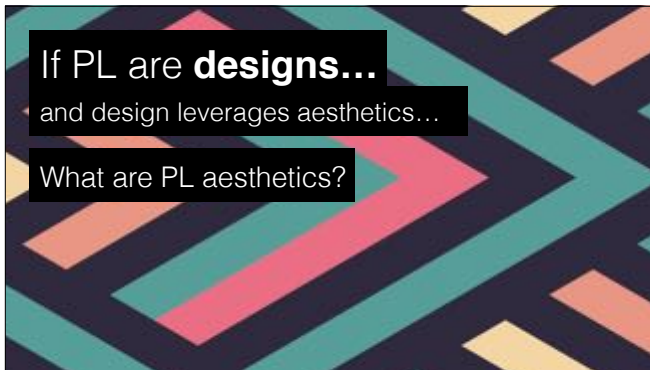
- How do we model this tradeoff space?
- Which qualities trade off with one another?
- What parts of the design space have we not explored?



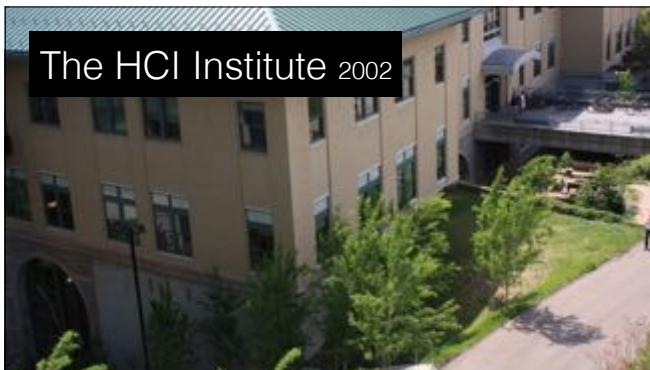
- This is a picture of David Kelly teaching a class the d.school at Stanford
- He teaches a process
- When we teach our graduate students to design programming languages, what process do we teach them?



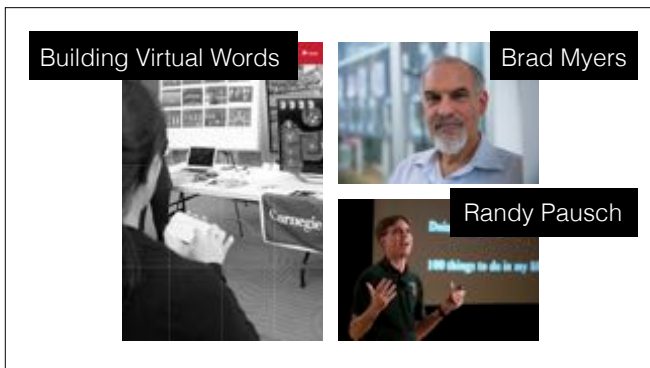
- Is it enough to jot down a sketch of a program in a syntax?
- Or do we need to be able to execute sketches?
- If it was trivial to prototype a PL, how would that transform our discoveries of new PL?



- Why don't we have a formal semantics of beauty, parsimony, brevity?



- My time at Oregon State with Margaret introduced me to HCI, and the PL.
- But my passions in design and human behavior were equally strong.
- The HCI Institute at Carnegie Mellon was a natural place to get my Ph.D.
- I brought my curiosity about design, expression, notation, and power, and hoped to find yet more ways of seeing PL.



- And I did, thanks to several mentors.
- I was advised by Brad Myers, who had been working on the foundations of user interface toolkits
- And I also spent time with the late Randy Pausch, who had worked on Alice and ran an incredibly fun interdisciplinary course, Building Virtual Worlds
- Brad had encouraged me to study the programming happening in the course, to find interesting problems to solve



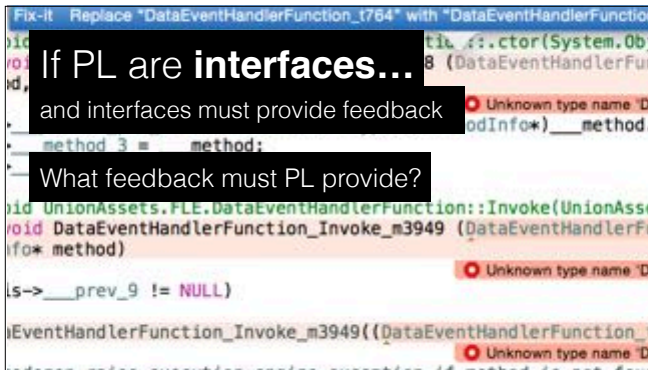
- While I expected to see expression, most of what I saw was usability breakdown after usability breakdown
- Students expressed what they wanted, but never got it quite right, leading to hours and hours of unproductive tweaking and debugging
- From my HCI-lens, the students' lack of understanding of the semantics of the language, and their inability to see those semantics execute, were fundamental user interface problems.



- This view was inevitable, and not only because I was an institute primarily concerned with interface.
- It was because computers, and programming languages in particular, were the *first* interface to computers. It's how we operated them.
- This view of PL as interface has many implications.



- Andy Stefik has recently been doing some fantastic work in this area
- For example, he's found that
 - Statically typed languages reduce debugging time
 - Transactional memory prevents synchronization bugs
 - Notations used greatly impact novice learning
 - Inheritance depth doesn't impact maintenance effort
- He's embodied these and other discoveries into the design of Quorum, a language that aims to be the most accessible and usable language ever invented.



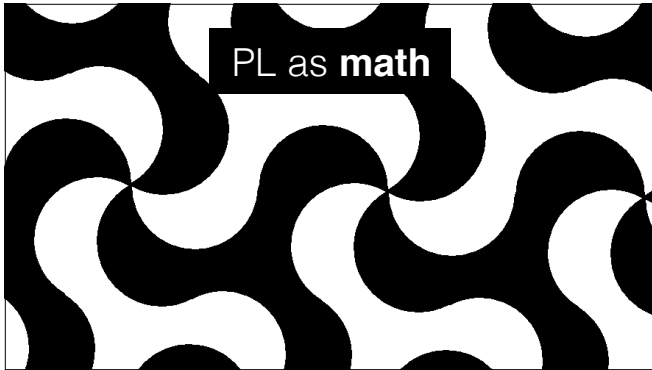
- What makes a good error message?
- One of my Ph.D. students did a great study a few years ago that found that just by using personal pronouns like “I” and “We” made novices read error messages more closely.



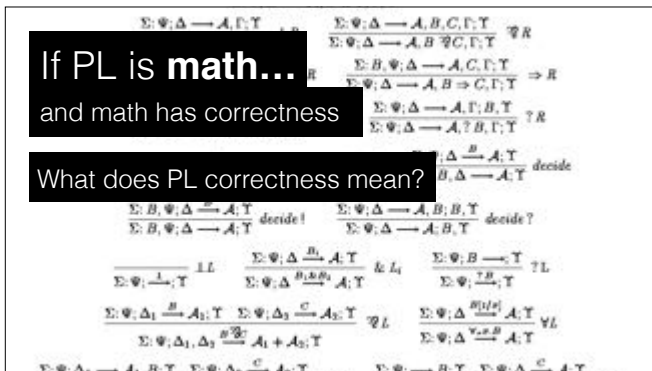
- Is Stack Overflow really the best we can do?
- Are a bunch of examples really good enough?
- Why can't we provide much richer, more robust articulations of what's possible with a programming language?
- But also what's not possible



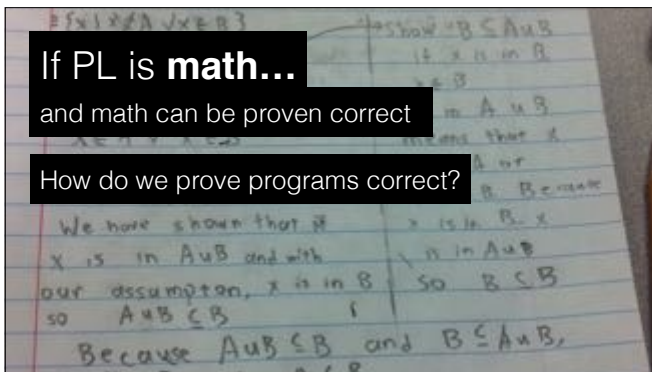
- Now, you might be wondering: why haven't I talked about the formal, logical view of PL that we all use in this room?
- That's because I didn't encounter it until I started coming to conferences like SPLASH, ICSE, and FSE.
- This is me at my first ICSE, four years in to my Ph.D., after several years of publishing exclusively at HCI conferences.
- I must say, this was a shock. I'd spent my whole life thinking of PL as interface, as media, as designs, as notation, and as power.
- And here was a community, our community here in fact, that viewed PL as math.



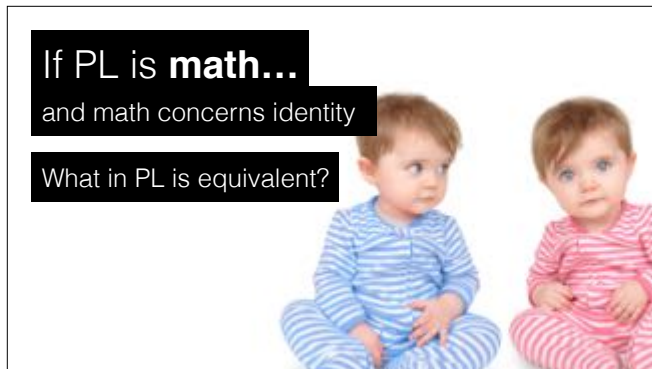
- Now, I realize ICSE is not *really* the home of PL as math (that would be PLDI or POPL)
- But back in 2006, a large proportion of the work was formal methods, with only a tiny fraction of empirical work.
- In most of these papers, programs weren't expression, they were propositions
- This was the “computing culture” that Papert had talked above.
- And what I learned was that PL as math had many implications that most people in the room already know.



- This is basically much of the history of PL and software engineering research.



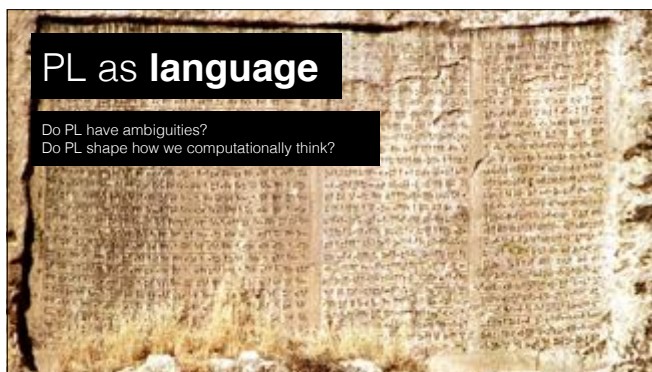
- This is expressed in research on proof assistants like Coq, which leverage the Curry-Howard correspondence



- This is the history of theory of computation, with its focus on complexity equivalence, problem equivalence



- My last stop was the University of Washington Information School
- It is an exceptionally interdisciplinary place
- Nothing more has stretched my epistemological stance, my understanding of knowledge, and my perspectives on PL.
- Here are some of the many perspectives I've encountered.



- What if programming languages are like natural languages, which are about **exchange**?
- This is, after all, why we use words such as “syntax”, “grammar”, and “semantics” to specify their rules.
- Do they follow the same evolutionary patterns as natural languages?
- What kind of ambiguities do PL have?
- Sapir-Whorf hypothesis: does PL determine how we computationally think?



- We "tell" a computer what to do with PL
- So what if PL is communication?
- Communication, which is about **understanding** and **common ground**
- What should PL do to verify developer intent?
- What should PL do to express it's intent to developers?



- Four years ago I co-founded a startup called AnswerDash and spent 3 years as its CTO and lead engineer.
- In my time as a developer, it was hard to think of PL as anything more than glue between APIs, frameworks, libraries, and platforms.
- Languages just weren't that useful for routine web applications for which there was already so much built. We didn't use much more than function calls and conditionals.
- So perhaps PL is just glue, which is about **connection**.
- What makes PL a good adhesive?
- What materials do certain PLs adhere to better?



- In software engineering, programs are often agreements between parties about what an application must, should, and must not do.
- If programs are contracts, than programming languages "legalese", the formal and technical language of legal documents, which are essentially about promises.
- If this is true
 - Who should legally interpret code?
 - Are programmers lawyers?



- We can view programs as conduits for information, funneling, filtering, and distributing data that ultimately has meaning
- If we build roads with concrete and nails
- We build information systems with programming languages
- If programs do the transmission, than programming *languages* are **information infrastructure**, creating the shared systems that allow society to operate
- How do PL decay?
- How should we maintain PL?
- Is PL a public good?



- Here's my most recent perspective.
- I taught a class of 11 south Seattle high school students this past summer while I was on sabbatical.
- Few of them had ever encountered code, or had any interest in learning to code
- But they were in my web design course nonetheless, because many of them would rather take that than ballroom dance in their summer college prep curriculum.



- I asked them, what is a programming language?
- One of them said, "A way out of poverty"
- Now, that definition is clearly pretty far from the ones we've discussed so far, but let's consider it.



- In this way, programming languages are a path from poverty
- This notion of PL was entirely reasonable
- In her world,
 - Surrounded by Seattle software companies,
 - By wealthy Bay Area entrepreneurs in the news
 - By constant non-profit advocacy as learning to code
 - And as teens with parents who did not finish high school
- In fact, this view may actually be the *most* dominant view of PL in the world today
- And given this, there are many important implications:



- The world is beginning to answer these questions
- Rise of MOOCs that only reach the privileged
- Coding bootcamps that again only reach the privileged
- The White House's CS for All
- Britain's CS For All for K-12 CS education



- Should it be the obligation of PL designers to make PL accessible?
- If governments fund PL as infrastructure, should this funding be contingent on accessibility, as is law in United States government IT projects?



- In today's world, we don't really choose
 - We let our culture determine interest
 - We let our ideas exclude
- How do we choose?
 - Is their innate aptitude to code?
 - Can everyone learn?
 - By what principles to we selectively encourage, especially in the context of universal access to computing education.
- Or, perhaps *everyone* can be on this path, because *everyone* will be doing some kind of programming.

Definition
PL is math
PL is interface
PL is design
PL is notation
PL is media
PL is power
PL is language
PL is communication
PL is glue
PL is legalese
PL is infrastructure
PL is path

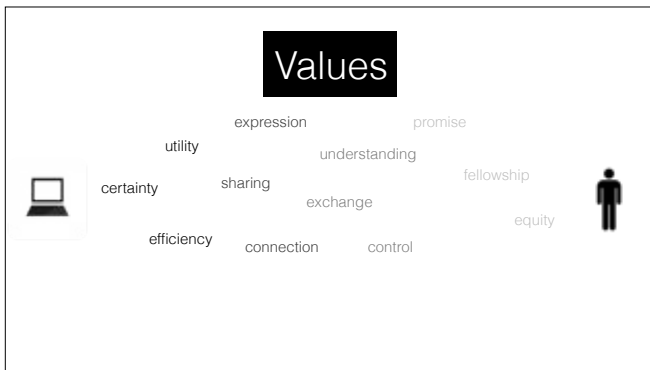
- Across the past 17 years of research, it is clear to me now that all of these views of PL have value.
- They were productive in my life
- They were productive in my research.
- But also that each view had **values**

Definition	→	Value
PL is math	→	certainty
PL is interface	→	efficiency
PL is design	→	utility
PL is notation	→	sharing
PL is media	→	expression
PL is power	→	control
PL is language	→	exchange
PL is communication	→	understanding
PL is glue	→	connection
PL is legalese	→	promise
PL is infrastructure	→	fellowship
PL is path	→	equity

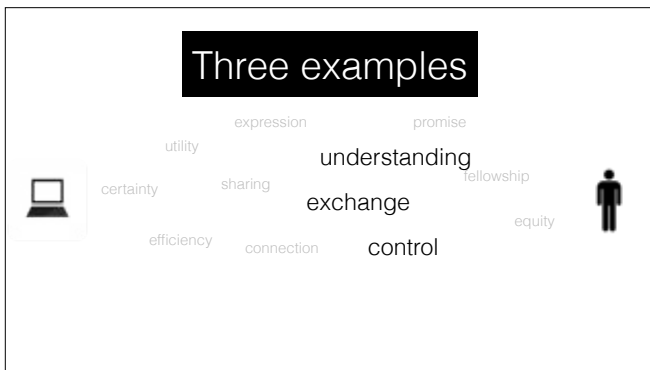
- Each of these fundamentally prioritizes different human values
- (Read examples)
- Inside each of these values, however, is also a research agenda.
- And each of these research agendas therefore expresses a different value

Definition	→ Value	# Discoveries (my impression)
PL is math	→ certainty	████████████████████
PL is interface	→ efficiency	██████████████
PL is design	→ utility	████
PL is notation	→ sharing	██
PL is media	→ expression	██
PL is power	→ control	█
PL is language	→ exchange	█
PL is communication	→ understanding	█
PL is glue	→ connection	█
PL is legalese	→ promise	█
PL is infrastructure	→ fellowship	█
PL is path	→ equity	█

- **This is my impression** of the number of discoveries that explore these perspectives
- Some of these agendas are already deeply explored
- Others, particular those that probe the human, social, societal, and ethical dimensions of programming languages, are hardly explored at all.
- If we look at these as a space of values...



- We can see that some are about **computing** and some are about **people**
- And we've spent most of our time understanding computing
- How do we investigate all of these other values?



- I'll give three examples from my own work to illustrate

PL as communication → understanding

- I'll start with one of my earlier attempts from my dissertation work
- I started with the idea that programming languages are ways that **we tell computers what to do.**
- But after they do something, we also inquire why they did what they do

PL as communication

How can we **interrogate** program behavior?

- Research question: how can we interrogate program behavior?

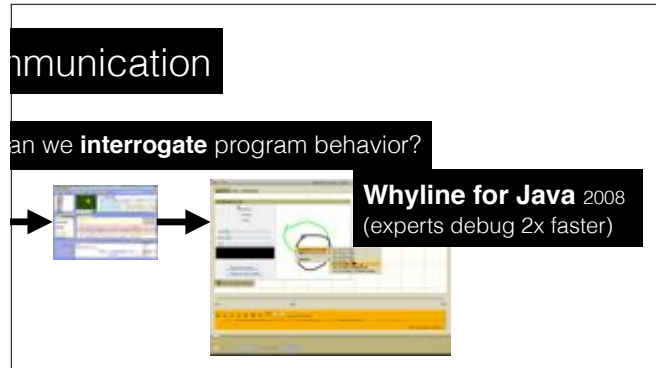
PL as communication

How can we **interrogate** program behavior?

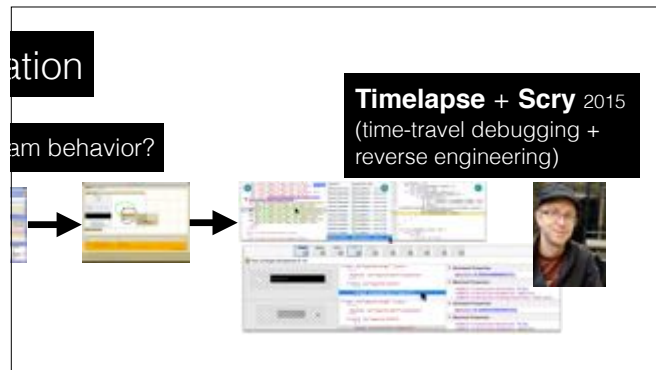


Whyline for Alice 2004
(novices debug 8x faster)

- This led to my early work on the Whyline for Alice, where novices could ask “why” questions about program behavior
- Dramatic acceleration of debugging times



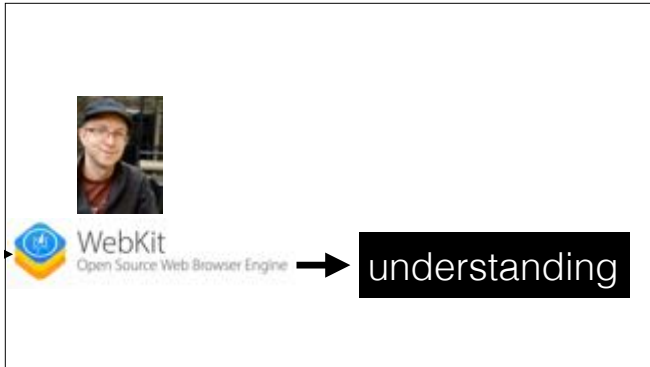
- And then Whyline for Java, which showed how the work could scale to large Java programs and accelerate expert developers' debugging



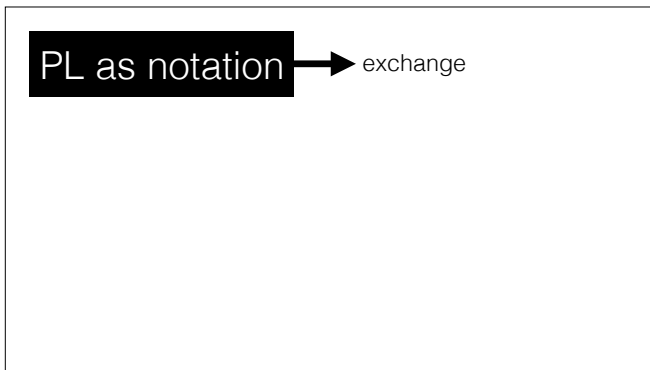
- This led to my Ph.D. student Brian Burg's work on Timelapse and Scry
- Time-travel debugging and reverse engineering



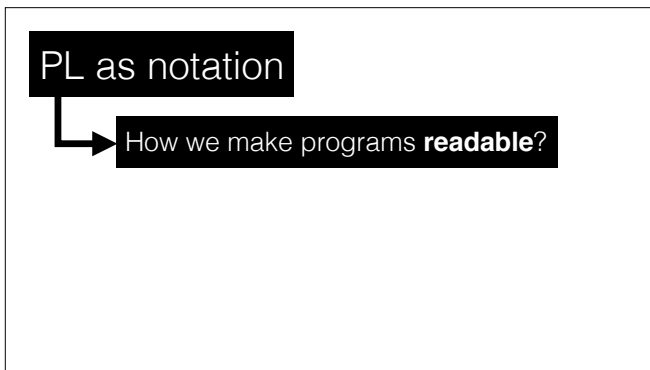
- He joined Apple and has begun upstreaming much of this to WebKit
- Bringing this back to values



- This means millions of developers will better understand how browsers are executing code.




- Another perspective, this time from the perspective of **notation**, which focuses on the value of exchange



- The question I investigated was ultimately about readability

s notation

How we make programs **readable**?



Barista 2006
Code as rich text

- One project about 10 years ago looked at how to render code to be more readable

program



Jasper 2006
Views on concerns



- Another project at the same time with Michael Coblentz (who's now a Ph.D. student at Carnegie Mellon) looked at how to make cross-cutting concerns readable

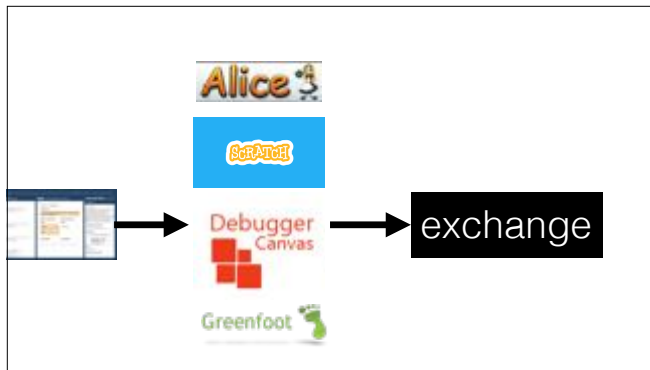
able?



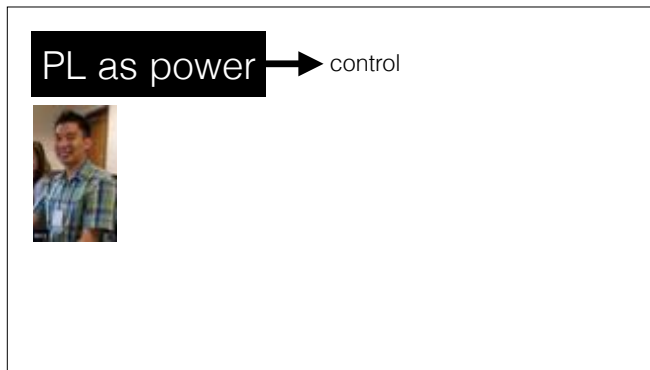
PLTutor 2016
Learn to read a PL
in 3-5 hours in hours



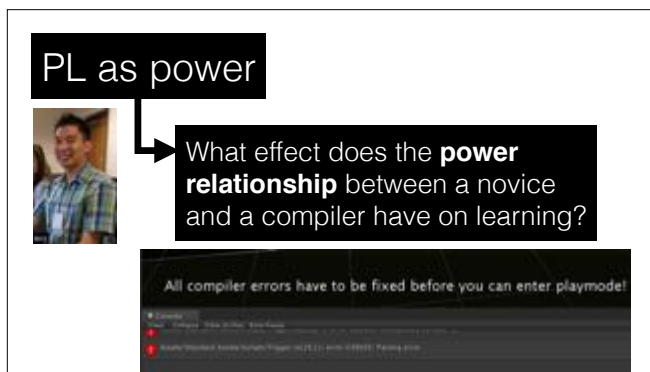
- Most recently, my students Greg Nelson and Benji Xie are looking at how to teach people how to **read** programming language notations
- Idea is to rapidly teach the operational semantics of a PL, ensuring learners can reliably predict the behavior of each of its execution rules
- In preliminary work, can teach a PL in 3-5 hours to a rank novice



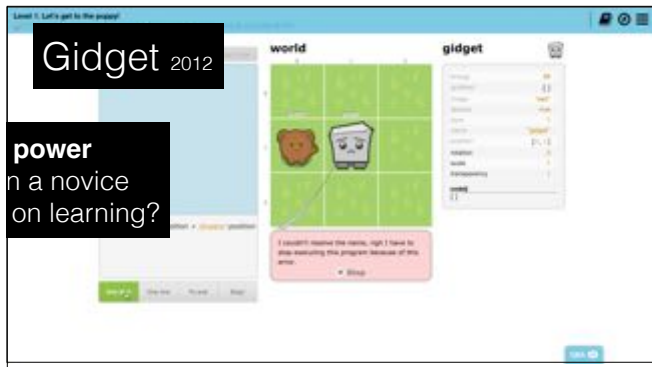
- Our work on programming language tutors is only just beginning.
- Many of these ideas about reading code have made it into widely used programming and debugging tools
- This transfer has led to each of these environments embracing a value of **exchange**, ensuring that programming is framed not just as a writing task, but a reading task as well



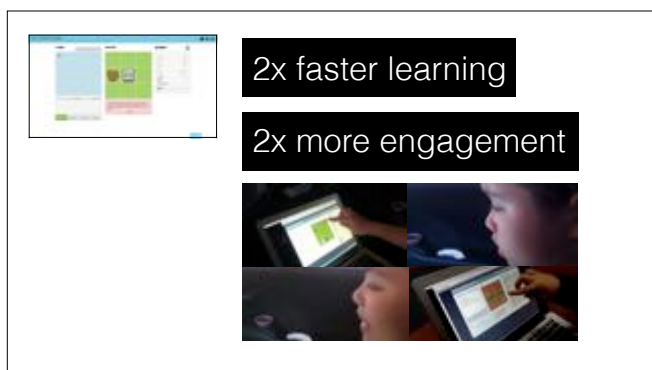
- Another perspective, this time from the perspective of **power**
- This was led by my student Michael Lee



- Mike wondered: what effect does the power relationship have?
- Most compilers frame an **authoritative** relationship, telling the learner what they've done wrong
- Creates a dynamic of constant failure



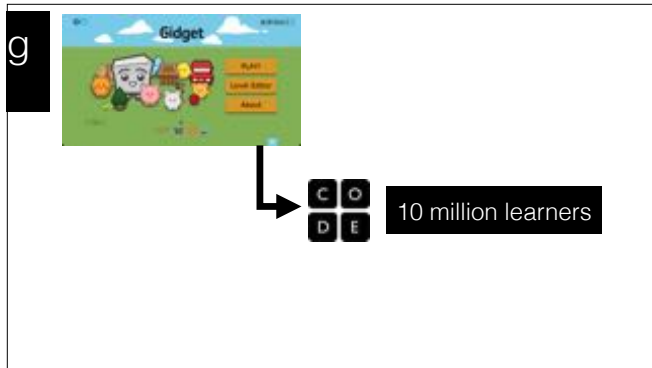
- This led to a game called **Gidget**, which framed the compiler as a **reliable but unintelligent collaborator**, placing the learner in the role as creative problem solver.



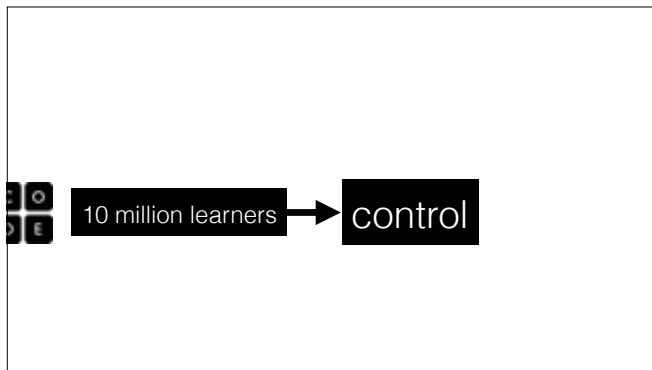
- This led to a series of studies that showed that very subtle design choices about error messages, data representation, and assessment have very powerful effects on learning and engagement.



- Mike deployed the game, reaching more than 10,000 learners



- Code.org incorporated these design principles



- And, in turn, these 10 million learners have a deeper sense of control over their ability to shape computer behavior though code.

Definition	→ Value	# Discoveries
PL is math	→ certainty	████████████████████
PL is interface	→ efficiency	████████████████
PL is design	→ utility	██████████
PL is notation	→ sharing	████████
PL is media	→ expression	████████
PL is power	→ control	██████████
PL as language	→ exchange	██████████
PL as communication	→ understanding	██████████
PL as glue	→ connection	████████
PL as legalese	→ promise	████
PL as infrastructure	→ fellowship	███
PL as path	→ equity	██

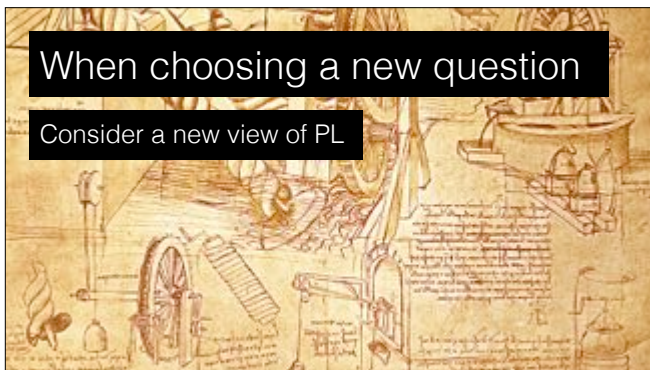
- Work like this is changing the distribution of what we know
- But it's also changing the values that the work disseminates to the world
- And this new values are engaging new people in computing, computer science, and computer science research
- This, in turn, is changing the values of our research community
- These are good things, but it begs several important questions

Definition	→	Value	
PL is math	→	certainty	Communication
PL is interface	→	efficiency	CS Psychology
PL is design	→	utility	
PL is notation	→	sharing	
PL is media	→	expression	Sociology
PL is power	→	control	
PL as language	→	exchange	Economics
PL as communication	→	understanding	
PL as glue	→	connection	Information Science
PL as legalese	→	promise	
PL as infrastructure	→	fellowship	Design
PL as path	→	equity	

- What **should** the distribution be?
- Who **should** be doing this research?
- Will this research **happen** if we don't explicitly embrace it as a community?
- Here's where we have these conversations

<ul style="list-style-type: none"> • As a discipline, CS doesn't have to explore all of these views. 	Communication CS Psychology
<ul style="list-style-type: none"> • But CS should encourage the exploration of these views, supporting and collaborating with other disciplines. 	? Sociology Economics
<ul style="list-style-type: none"> • And to equitably engage everyone, CS does have to embrace these views, showing that CS is more than logic. 	Information Science Design

- **Here's what I think**
- (Read)
- How can we do this?



- Are their views that draw upon expertise and skills that you have?

When writing a paper

Be explicit about your views of PL



- What epistemology are you using?
- Perhaps you're using multiple distinct epistemologies?

When reviewing a paper

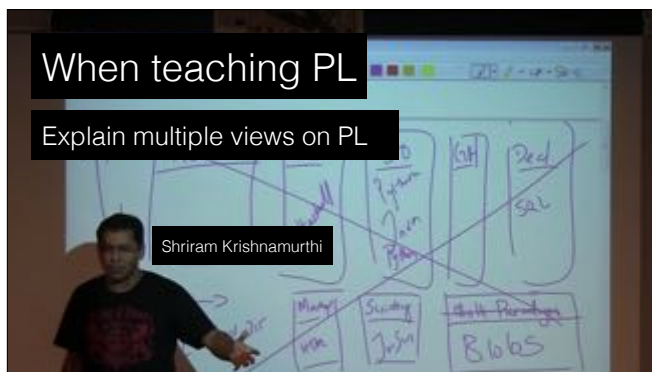
Evaluate it against the paper's view of PL, not yours



- Recognize that each view demands different methods, different theories, and ultimately, different epistemologies

When teaching PL

Explain multiple views on PL



- Be pluralistic in your claims about truth
- Expose students to these views
- Be explicit about which views you subscribe to



- If we do these things, we'll not only approach the inclusive **computing culture** that Papert dreamt of
 - But we'll also create a more inclusive world in general, in which everyone's **perspectives**, everyone's **values** on computation is reflected in our work, and in our society.
 - **Thank you.**
-