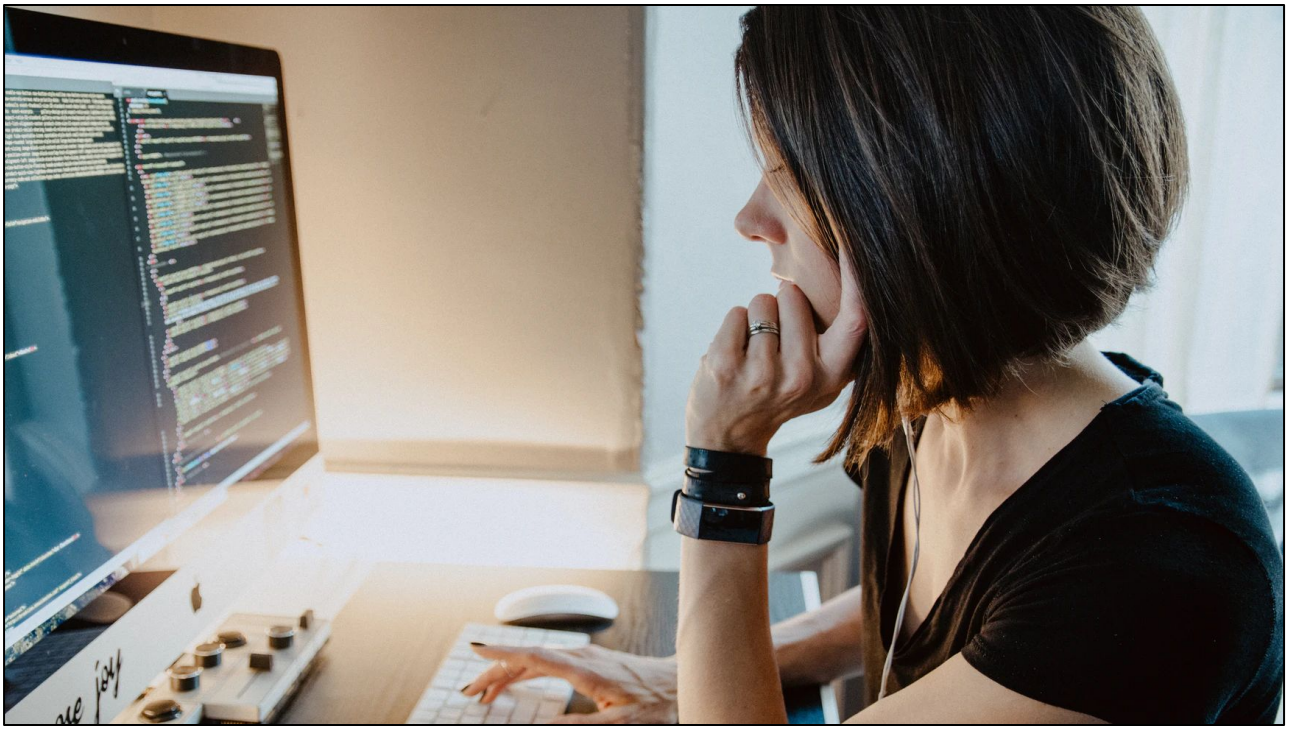
Programming as Cognition, Programming as Politics

Amy J. Ko, Ph.D.
Professor
The Information School
University of Washington, Seattle

- Thank you so much for the invitation to speak; it's such a pleasure to get to stay connected during this pandemic, even if it is through the clumsy and brittle infrastructure of the web.
- I've given a lot of talks recently, and many of them have been quite personal, telling my story as a transgender person in computer science, and connecting those experiences to the intersections of computing and justice.
- Today, I'm going to continue that theme of computing and justice, but instead of centering on myself, I want to indulge in a more conventional talk that centers my research.
- But I want to do this in an unconventional way. Rather than share a deep summary of 2-3 discoveries and get into every nuance of their methods, I want to share with you nearly *all* of my discoveries from the past 20 years about the central focus of my work: programming.
- I'll share this in three parts: a rapid tour through a synthesis of everything I've learned about programming by studying it as a cognitive act, and then everything I've learned about programming by studying it as a social and political act. I'll close by trying to bridge the two lenses, showing us how these different analytical frames aren't as far apart as we think.



- So why study programming?
- If it were still the 1950's, it'd be a harder case to make. Computers filled rooms, they were largely programmed by low-wage women mathematicians in science and government, and while they played pivotal roles in wars and business, they were still niche.
- Now, of course, computing is everywhere: in our pockets, in our homes, in our cars, trains, and planes, on our bodies, in our bodies, and even in the invisible infrastructure we rarely consider: energy, trade, law, justice, and politics.
- All of these systems exist because of one activity: programming, which involves conceiving of some computational behavior we want a computer to perform, and then carefully, and sometimes not so carefully, trying to translate that behavior into logic, calculation, structure, and process.
- *How* and *what* we program has never been more central in shaping society.

- Who does it, how quickly, how carefully, how thoughtfully, and how inclusively, no longer just shapes government and science, but nearly every dimension of our lives.
- It determines who gets loans,
- Who goes to jail and for how long
- Who eats, who thrives, who dies.
- And so understanding how people do it, why it is hard, why we continue to make the same mistakes after 70 years of progress, is of great consequence.

Computer Art Club

This club is different from just regular art club. They are the art of the future and they are learning how to produce different sorts of art for the next generation. This club seems to be laid back and has a lot of fun together.

Top Row: Matt Teske, J B Wallick, Bryan Ko. Middle Row : Jacob DeGraw, Mike Teske, Andy Ko, Will Nelson-Romick, Evan Hayashi.

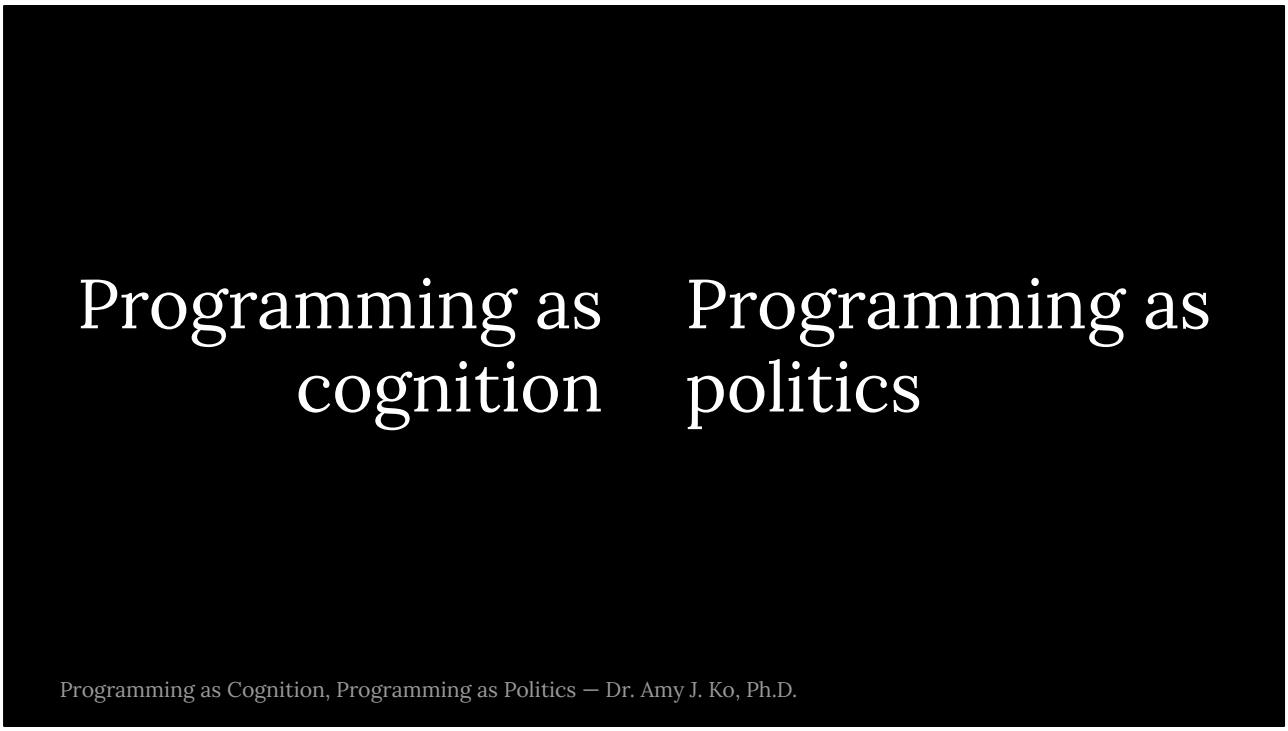
Clubs 69

- My own curiosity about programming began when I was 13, in middle school.
- I first learned about programming it in 7th grade pre-algebra, when our teacher taught us how to write simple programs on our TI-82 graphing calculators.
- I found ways to create games and procedural art, and found it a incredibly fun, hard, satisfying, and social way to express myself. This is a photograph of my friends and I in our computer art club, which was just as much about art and computers as it was using code to create art.
- When I got to college, though, I got more interested about what programming is and how people do it, and stumbled into research.

- I first studied people programming spreadsheets for finance, simulation, and grading, focusing on how they test and debug their formulas, or more often don't.
- I expanded my focus to professional software developers working alone and in teams, at companies big and small.
- And to creative professionals harnessing obscure languages to create interactive games and digital music
- I studied students learning to code in and out of school.
- And now I even study primary, secondary, and post-secondary teachers learning to program, and learning to teach programming, in their classrooms.

- Throughout I've examined it from largely a distributed cognition lens, considering not only what is happening in people's minds, but what is happening on their screens, on the websites they visit, in the tools they use, and how all of these interact to produce programs, and defects in programs.

- But over the past several years, I've also begun to examine it from a social cognition lens, investigating how people who make software think and reason about who they are making it for, and how they reason about the broader social impacts of what they are making, as well as how they project and signal who deserves to program and who is welcome in their communities.

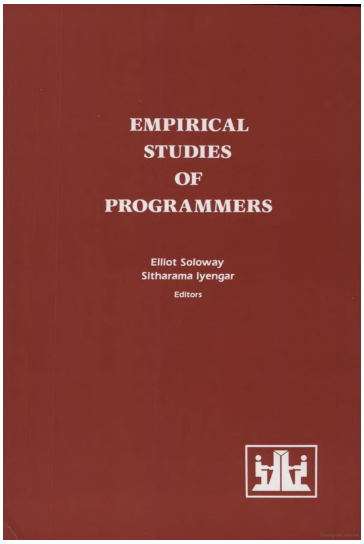


- [5 minute mark]
- So that brings me to the dichotomy in the title of this talk
- Programming as cognition
- Programming as politics
- These are the two lenses that dominate how I think about programming, and that I hope to bridge by the end of this talk.
- But before we dive in, a few disclaimers...

Prior work disclaimer

Research on the psychology of programming began in about 1980, when I was zero years old! Everything I'm about to present builds upon a rich history of studies and theories.

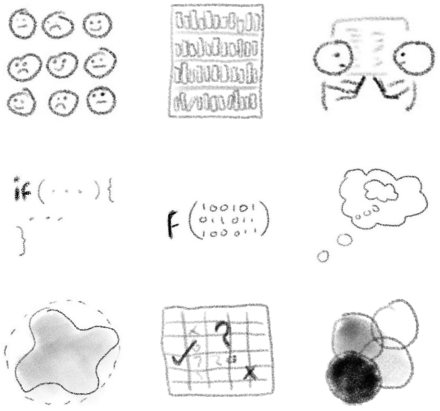
However, much of this work stopped in the mid 90's when everyone started studying the web; I started studying programming around 2000, and helped reboot discourse with new methods, theories, and perspectives.



- First, I'm not the only one who's studied programming.
- This work began in the early 1980's, when psychologists and computer scientists came together to study the dominant way that people interacted with computers in the late 1970's and early 1980's: by writing programs
- [Read]
- I'll focus on my own work in this talk, but it builds directly upon these foundations.

Methods disclaimer

This is going to be a *broad* talk. I won't spend much time detailing methods, sample sizes, measurements, or other details. Everything I present is peer reviewed, and many have been successfully replicated.



I've used mixed methods lab and field experiments that combine qualitative observations with log and artifact analysis.

Programming as Cognition, Programming as Politics — Dr. Amy J. Ko, Ph.D.

- Second…
- [Read]

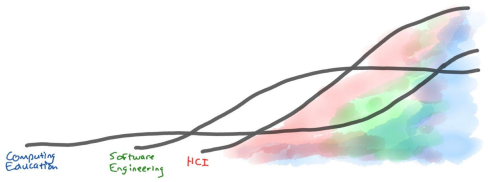
Collaboration disclaimer

I've done all of this work with an amazing group of collaborators, doctoral students, undergraduates, and practitioners. They deserve as much credit as I do for these discoveries.

- Third…
- [Read]

Narrative disclaimer

I won't talk about my work in chronological order. My theories and hypotheses and programming span two decades of interleaved ideas, and they did not happen in a tidy order. I'll simplify this history to make theory salient over sequence.



- Finally…
- [Read]

Programming as cognition

- [~6 minutes]
- With that, let's start with what I've learned from a cognitive perspective.

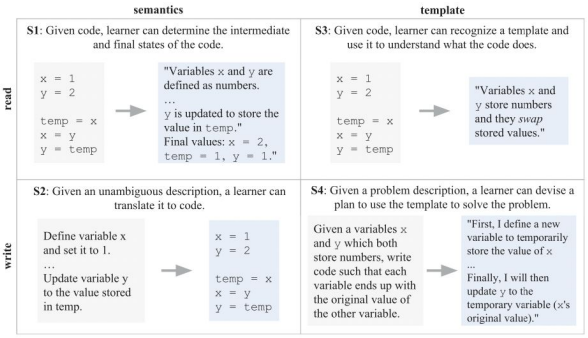
Reading and **writing** code are distinct skills

Most people learn to code by writing a lot of code and learning from failure. However, it's not clear that this is the most efficient way to learn.

We experimentally examined two ways of sequencing learning:

1. *Write lots of programs with feedback*
2. *Learn to read programs with feedback, then learn to write them with feedback.*

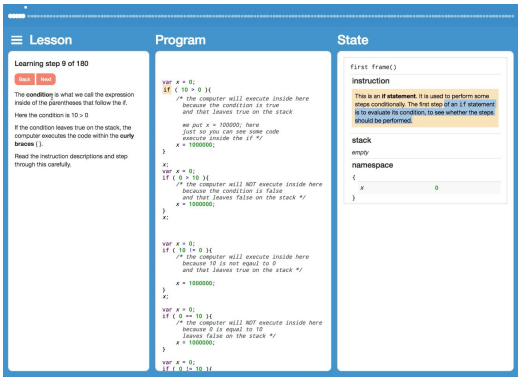
Learning to read first significantly improved the quality of practice, depth of understanding, and error rates.



Benjamin Xie, et al., (2019). A Theory of Instruction for Introductory Programming Skills. *Computer Science Education*, https://doi.org/10.1080/08993408.2019.1565235

- [Read]
- In doing this, we distinguished between tasks that involve reading the line by line behavior of programs, reading the larger templates that achieve some computational goal, and between writing these two different levels of code.
- This 2x2 table summarizes these four distinct skills.
- We found that sequencing learning by these four skills, reading semantics, writing semantics, reading templates, writing templates, significantly improves learning.

Language learning requires **granular interactivity**



Greg Nelson, Benjamin Xie, Amy J. Ko (2017). Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER. https://doi.org/10.1145/3105726.3106178

Many people learn a programming language by slowly infer its semantics through trial and error. But many struggle, and develop brittle, inaccurate language understandings, and often quit out of frustration.

We've tried teaching languages with **granular**, **interactive** examples that map syntax to semantics, and then formatively assess knowledge of that mapping.

Rank novice learning in **3-4 hours** produced better learning outcomes when compared to an entire 6 weeks of classroom learning.

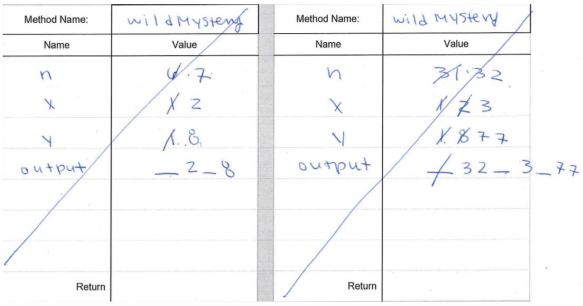
- When we drilled down into that first quadrant of reading semantics, we learned something else:
- [Read]
- This image on the left shows our intervention, PLTutor, a kind of interactive book that offers this granular interactive learning.
- This work suggests that difficulties with reading code, and learning to code in general, are more dominated by inaccessible insights about how programs execute, and the precise ways in which we explain and visualize semantics, than anything intrinsically difficult about programming languages themselves.
- In other words, we've just been teaching them poorly.

Effective code reading is **active and distributed**

Even when separating reading from writing, learners vary in the effectiveness of their reading.

We tried scaffolding reading by providing an explicit step-by-step reading process and structured format for externalizing program state.

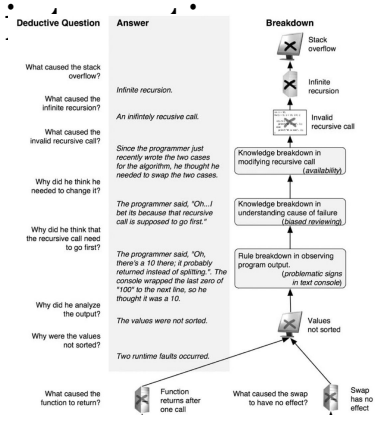
In an experiment, students in the treatment group were more systematic, made fewer errors, and scored a grade level higher on their midterm.



Benjamin Xie, Greg Nelson, Amy J. Ko (2018). An Explicit Strategy to Scaffold Novice Program Tracing. ACM SIGCSE, Research Track, 344-349. https://doi.org/10.1145/3159450.3159527

- That close look at program reading also revealed another dimension
- [Read]
- You can see an example of the externalization we encouraged here, showing variable names and values, and annotations for changing values.
- This insight, combined with the previous two studies, suggests that program reading is not only an essential foundation for all program writing, but also one that depends on external memory and self-regulation aids.
- When we turn our attention to writing, we see very different skills...

Errors emerge from **skill**, **bias**, and **tool**



Amy J. Ko, Brad A. Myers (2005). A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. *Journal of Visual Languages and Computing*. http://dx.doi.org/10.1016/j.jvlc.2004.08.003

I adapted research on human error to programming, and empirically examined error production across 100's of hours of programming, finding root causes of mistakes.

- Whether and when people **notice** their errors is a function of their tool environment and verification practices, which often fail to reveal errors.
- Error diagnosis (i.e., debugging), suffers from a **recency bias** (e.g., "*It must be something I just wrote*"), and exacerbated by **action bias** (e.g., diagnosing by implementing fixes that introduce more defects).

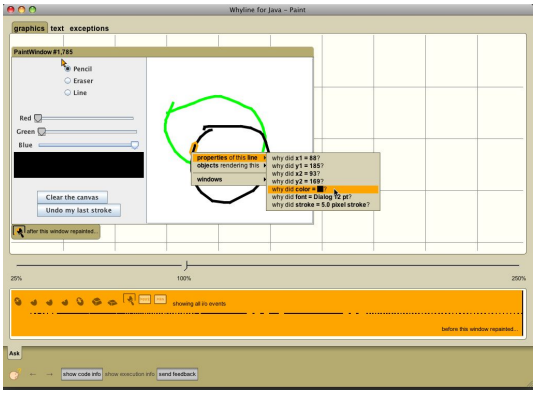
- [Read]
- This diagram on the left shows an example of a root cause analysis of errors, where a series of false assumptions lead to a misinterpretation of a program's output, which lead to a false hypothesis, which led to an incorrect change, with led to another error, and further confusion.
- This insight about false hypotheses led us to examine debugging more closely...

Debugging is driven by **hypothesis testing**

To find errors, most people haphazardly generate hypotheses about errors based on superficial features of the failure, and then spend most of their time testing and rejecting false hypotheses.

We predicted that we could circumvent this vicious cycle by inverting this process, having people **work backwards** systematically from the failure to its root cause, rather than guessing forwards.

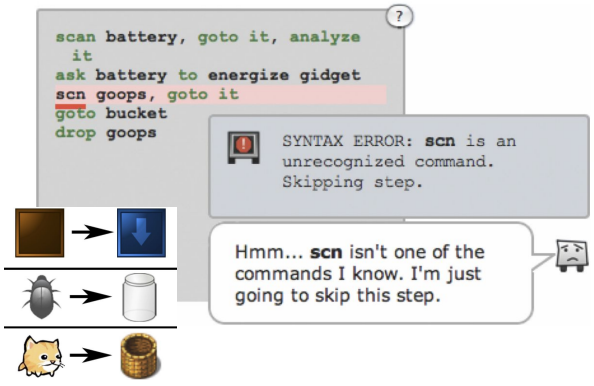
Across a range of tasks and experience levels, this approach reduced debugging time by a **factor of 2-10** by eliminating fruitless hypotheses.



Amy J. Ko, Brad A. Myers (2009). Finding Causes of Program Output with the Java Whyline. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI), 1569-1578. https://doi.org/10.1145/1518701.1518942

- What we found is that...
- [Read]
- This figure on the right is an example of the tools we invented to make this possible: they gather a log about a program's execution, and then let people select the output that was wrong and ask questions about it directly. The system then gives an answer as a chain of events that caused the wrong output.
- One of the more striking qualitative observations from this work was how shocked some of the more experienced programmers were at its simplicity, saying things like "I can't ever go back to the way I used to debug, but I have to, because these tools aren't on the market yet!'
- But cognitively, it was also striking that this basic reversal fundamentally changed the nature of debugging: it was no longer a game of guess and check, but rather a systematic examination of evidence.

Tool efficacy is mediated by **attention**



Michael J. Lee, Amy J. Ko (2011). Personifying Programming Tool Feedback Improves Novice Programmers' Learning. ACM International Computing Education Research Conference (ICER), 109-116. https://doi.org/10.1145/2016911.2016934

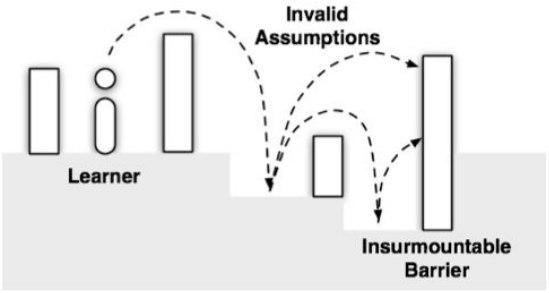
It seemed to be that how these interventions presented information greatly determined how closely people attended to the information, and thus how much the information influenced their behavior.

We compared conventional tool messages (e.g., impersonal, technical language) with personal messages (e.g., anthropomorphized). Giving compilers **eyes**, having compilers use **personal pronouns**, and representing data with **vertebrates** significantly increases learning outcomes and decreases debugging time over inanimate, impersonal forms by guiding attention to details in error messages.

- Throughout all of these efforts to examine program reading and comprehension, we observed something else
- [Read]
- These images on the left show some of the simple changes we made that led to dramatic changes how closely people read messages, and thus how well they learned and performed on tasks.

Modern programming is driven by **code reuse**

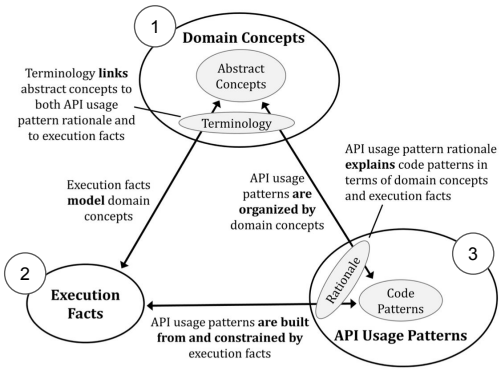
Through a 15-week longitudinal study of application development, we found that what made programming hard was more than just learning languages and debugging. It was also **finding**, **learning**, **evaluating**, and **coordinating code** that *others* had built — in libraries, application programming interfaces (APIs), and other projects — and that a lack of information about this this reusable code often poses insurmountable learning barriers.



Amy J. Ko, Brad A. Myers, Htet Htet Aung (2004). Six Learning Barriers in End-User Programming Systems. IEEE Symposium on Visual Languages and Human-Centered Computing (VL/HCC), 199-206. https://doi.org/10.1109/VLHCC.2004.47

- While we found that reading was largely dominated by people's ability to carefully and precisely attend to information about program behavior, our examination of writing revealed something quite different.
- One of my earliest studies, for example, found that writing was much less a process of careful reasoning and much more a process of reusing what others have written
- [Read]
- This diagram on the right shows the common pattern we observed in this work: someone would try to use some code they found, they would make a series of assumptions about its behavior, and then reach an insurmountable barrier to progress caused by that assumption.
- The only thing that helped them overcome it was correcting that assumption, but it was quite rare that the tools, documentation, or other resources ever helped reveal and check assumptions. In fact, these resources tended to make numerous assumptions about the expertise of the person reusing the code.

Reuse is about comprehending **models**



Kyle Thayer, Sarah Chasins, Amy J. Ko (2021). A Theory of Robust API Knowledge. ACM Transactions on Computing Education, 21(1), Article 8. https://doi.org/10.1145/3444945

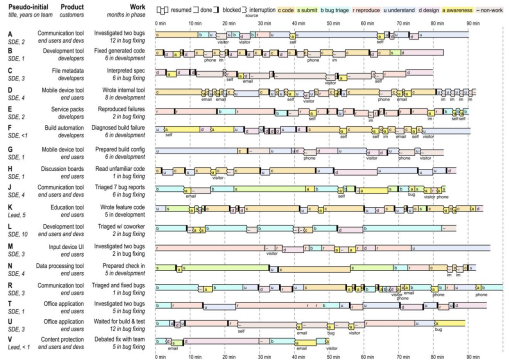
Learning application programming interfaces (APIs) requires different knowledge than learning programming languages:

1. Concepts in the **domain** that an API models and how it (imperfectly) models them
2. Declarative facts about the **execution semantics** of the API
3. The (ever growing) space of possible **usage patterns** that the API supports

Documentation rarely makes 1) or 2) visible; in an experiment, we showed that making all visible improved task success, but overwhelmed developers with learning.

- When we considered this learning more directly, found that most of reuse is about comprehending models that others had designed.
- We developed and tested a theory that argued that learning APIs reduces to three essential types of knowledge, as portrayed in the figure on the left.
- [Read]
- In essence, learning to use others' code seems to mirror some of the challenges of understanding programming language's and one's *own* code — they both require a careful, precise understanding of how a program executes — but when a precise understanding isn't available, it requires people to build a more coarse mental model of that behavior using other concepts.
- This is a recipe for mistakes, misunderstanding, and frustration.

Reuse is about **information seeking**



Amy J. Ko, Rob DeLine, Gina Venolia (2007). Information Needs in Collocated Software Development Teams. ACM/IEEE International Conference on Software Engineering (ICSE), 344-353. http://dx.doi.org/10.1109/ICSE.2007.45

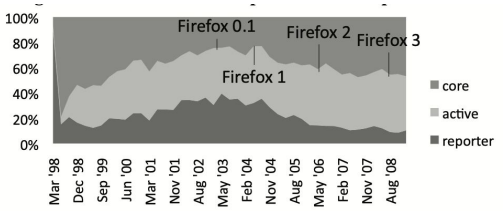
Through a 3-month field study observing professional software developers, we found that *most* of their time was seeking information from resources — sometimes documentation, but mostly other developers — and that if they could not find it, they often had to abandon some task.

And yet, few of the strategies that developers used were effective because most of the systems they used were not designed for information retrieval and most of the people they consulted did not have well-organized archives or could not remember the necessary details.

- And this is true not just in learning contexts, but in professional software engineering as well, where we found that...
- [Read]
- This figure on the left shows about 2-hours of work by 17 different professional software developers; their entire activity was structured by information seeking. In fact, I don't think there was a single person we observed write a line of code: all of it was gathering enough information to make a decision about what single line of code to modify in some very careful, surgical way.
- And so this study showed that whether novice or expert, the problems are the same: understanding program behavior is central to fixing or changing it. Experts just had more careful, systematic strategies for doing it, and had to rely on others for information because of the scale of their projects.

Information from the **crowd** is of limited value

Through a field study of hundreds of thousands of bug reports on the open source Mozilla Firefox web browser, we found that extracting meaningful signal from community contributions — if there is a signal at all — is often so time consuming that most of it is **ignored**. Instead, most of the meaningful information came from core developers on the project who had deep knowledge of how the browser was built. This created outrage amongst outsiders trying to report and resolve problems, who felt ignored.

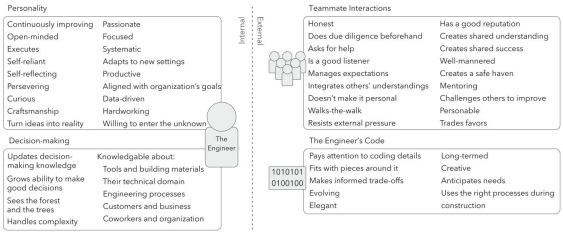


"Over two months ago I gave complete information on when and how I got the error AND spent a great deal of time isolating the messages that caused it... Which part of that is just saying "me too"? For crying out loud, I'm a nursing student, not a programmer. Do you do your own x-rays before going to the doctor?"

Amy J. Ko, Parmit K. Chilana (2010). How Power Users Help and Hinder Open Bug Reporting. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI), 1665-1674. https://doi.org/10.1145/1753326.1753576

- We considered some types of information seeking more closely.
- For example, one common type of information gathered by software engineering teams is from users themselves, reporting defects and problems with programs.
- But we found that…
- [Read]
- Here we see 10 years of information gathering from different groups of contributors; most contributions were from active contributors to the project, and the very few that came from users were mostly ignored.
- This quote conveys some of the challenges in sharing meaningful information about program behavior
- [Read]
- And so talking about code and its behavior was not straightforward at all.

Expertise is also **social** and **organizational**



Paul Luo Li, Amy J. Ko, Jiamin Zhu (2015). What Makes a Great Software Engineer? ACM/IEEE International Conference on Software Engineering (ICSE), 700-710. https://doi.org/10.1109/ICSE.2015.335

In a study of more than 3,000 developers' perceptions of programming expertise, we found that programming was about more than just languages, tools, debugging, reuse, and information. In fact, we found more than 50 attributes that developers viewed as essential, including:

- Personality characteristics
- Decision making skills
- Interpersonal teamwork skills
- Product management skills

- As we considered information seeking more broadly, we found that much of the need for information emerged not from inherent complexities in programs themselves, but from the social and organizational context of the teams that maintained them.
- [Read]
- This figure on the left shows the 53 attributes that experts identified
- That small box in the lower right is about programming. Everything else is about higher level decision making skills and how they interact with the broader social context of teams and organizations.

Self-regulation is a central programming skill

In a survey and interview of 700+ developers, we further investigated how these skills ranked in their significance to expertise, finding that most of the "must have" skills were **self-regulation skills**:

- Systematic attention to detail in code
- Metacognition about skills and context
- Proactive efforts to correct conceptions
- Awareness of when to analyze vs. act
- Ability to manage time, tasks, resources

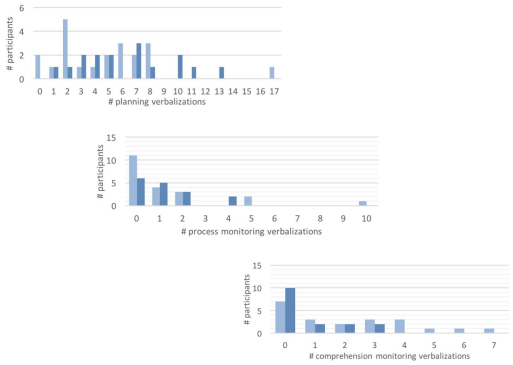
Table 1 Attributes of great software engineers, along a scale of *must have*, *should have*, *nice to have*, *irrelevant*, and *should not have*

# Higher	Mode	Attribute and description
53	Must	Pays attention to coding details, such as error handling, memory, performance, style
52	Must	Mentally capable of handling complexity; can comprehend multiple interacting software components
49	Must	Continuously improving: improves themselves, their product, or their surroundings
49	Must	Honest: provide credible information and feedback that others can act on
49	Must	Open-minded: lets new information change their thinking
46	Must	Executes: knows when to stop thinking and to start doing
45	Must	Self-reliant: gets things done independently and does not get blocked easily
45	Must	Self-reflecting: recognizes when things are going wrong with a plan and pivots
43	Must	Persevering: not dissuaded by setbacks and failures
41	Must	Fits together with other pieces around it: code accounts for surrounding constraints and products.
41	Must	Knowledgeable about their technical domain, including product, platform, and competitors
39	Must	Makes informed trade-offs: code is responsive to time to market goals, critical needs of the business
39	Must	Updates their decision making knowledge: does not let their understanding stagnate
36	Must	Curious: desires to know why things happen and how things work
36	Must	Evolving: code is structured to be effectively built, delivered, and updated incrementally

Paul Luo Li, Amy J. Ko, Andrew Begel (2019). What Distinguishes Great Software Engineers? Empirical Software Engineering, 1-31. https://doi.org/10.1007/s10664-019-09773-y

- We followed up this work by asking hundreds of developers to rank the importance of these different skills.
- We expected programming skills to be central, and they were number one.
- But to our surprise, most of the "must have" skills were self-regulation skills.
- [Read bullets]
- This painted a picture of programming as an act of cognitive self-control: always monitoring what you need, how problem solving is proceeding, and when information and comprehension is sufficient to take action.

Self-regulation is **rare**



We empirically examined the *planning*, *process monitoring*, *comprehension monitoring*, *reflection on cognition*, and *self-explanation* of several novice programmers, and found **planning**, **process monitoring,** and **comprehension monitoring** activities to be strongly anti-correlated with errors and learning outcomes.

In general, all activities were infrequent, and this frequency explained most of the variation in programming skill.

Dastyni Loksa, Amy J. Ko (2016). The Role of Self-Regulation in Programming Problem Solving Process and Success. ACM International Computing Education Research Conference (ICER), 83-91. https://doi.org/10.1145/2960310.2960334

- But when we examined self-regulation more closely, we found that it was quite rare.
- [Read]
- So the professionals' instincts were right: self-control in problem solving was one of the single most important factors in success.

Self-regulation is **hard**

We had several dozen students attempt to purposefully self-regulate in work diaries, but nearly all reported significant difficulties:

- Some were completely unaware of their process or decisions
- Some struggled to integrate reflection into their process
- Some found reflection distracting, perceiving that it interfered with progress

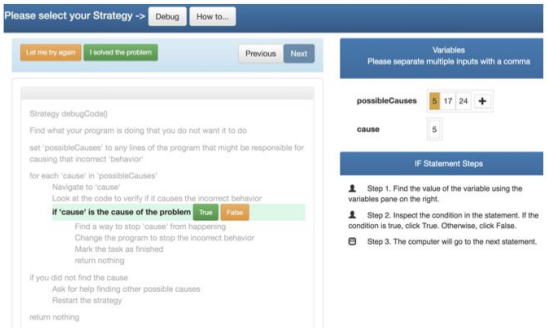
"*When I hit really difficult bugs, I don't want to reflect on them or journal, I just want to look at my code and chase them down.*"

"*It was really difficult to remove myself from my workflow and constantly having to switch between my journal and my code; it broke my workflow and made me work slower.*"

Dastyni Loksa, Benjamin Xie, Harrison Kwik, Amy J. Ko (2020). Investigating Novices' In Situ Reflections on Their Programming Process. ACM Technical Symposium on Computer Science Education (SIGCSE), Research Track, 149-155. https://doi.org/10.1145/3328778.3366846

- Why is it so rare?
- [Read]
- These findings suggest that it may not be that core metacognitive skills were the barrier, but rather the immersive, absorbing nature of programming, the way that it fully consumes all of working memory, crowds out the need for self-regulation — at least until the lack of it leads to enough failures that they eventually make room for it, as we saw with professionals.

Self-regulation requires **impulse control**



Amy J. Ko, Thomas D. LaToza, Stephen Hull, Ellen Ko, William Kwok, Jane Quichocho, Harshitha Akkaraju, Rishin Pandit (2019). Teaching Explicit Programming Strategies to Adolescents. ACM Technical Symposium on Computer Science Education (SIGCSE), Research Track, 469-475. https://doi.org/10.1145/3287324.3287371

In a classroom of adolescent novice programmers, we offered a set of step-by-step programming strategy scaffolds to help with some of the harder tasks in programming (e.g., planning an algorithm, debugging a defect).

Students reported:

1. Believing that the strategies were helpful,
2. Trying to follow the strategies, but
3. **Succumbing to the inability to control their impulses**
4. Regretting following their impulses, and eventually having to be systematic

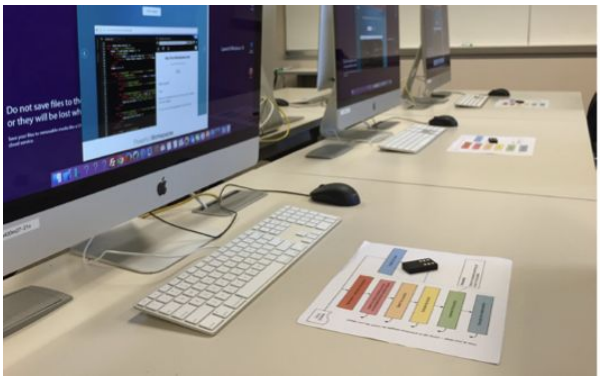
- One of our attempts to help scaffold self-regulation was promising.
- [Read]
- This screenshot on the left shows the tool we gave them to help track and regulate their problem solving progress.
- And while these strategies we gave them we reliably successful, and these externalizations did help them track their problem solving, students only used them after slowly and begrudgingly accepting that their impulsive, uninformed, haphazard strategies were just not working.

Self-regulation can be **learned**

In another intervention, we instead tried scaffolding self-regulation by promoting reflection during help seeking. Each time a student asked for help, we asked them:

- What are you doing?
- Why are you doing it?
- Is it helping?
- What could you do differently?

Over two weeks, they eventually started asking these questions of themselves, and were more **successful**, **independent**, and **confident** than students in a control who did not receive these reflection prompts.



Dastyni Loksa, Amy J. Ko, William Jernigan, Alannah Oleson, Chris Mendez, Margaret M. Burnett (2016). Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI), 1449-1461. https://doi.org/10.1145/2858036.2858252

- Another attempt at scaffolding self-regulation was more successful.
- [Read]
- What seemed to work better about this approach was actually a psychosocial effect: students quickly gained a sense of independence and confidence when they no longer needed to ask for help, and that promoted self-regulation practice, which developed skill, which was self-reinforcing.
- This was in contrast to the attempt on the previous slide, which only seemed to shame them for being impulsive.

Programming is dominated by **strategy**

"*I don't typically do the due diligence of reading all of the variable names and function names when I'm dealing with this sort of thing. And it seemed pretty clear to me that this is maybe a really good idea. Because one thing I noticed was that* ***my initial instinct was to try to really close[ly] read the flow of the program****. Then, when I remembered that the task was actually just to read the variable names and function names, I was able to get through it much much faster. I still had a pretty good idea of actually how it worked without getting quite as in detail with the rest of the flow of the program...*"

Thomas D. LaToza, Maryam Arab, Dastyni Loksa, Amy J. Ko (2020). Explicit Programming Strategies. Empirical Software Engineering, 2416–2449. https://doi.org/10.1007/s10664-020-09810-1

While self-regulation is clearly critical, *how* someone approaches a problem appears to be even more critical, and not wholly dependent on expertise.

We experimented with a set of explicit expert programming strategies with professional software developers, and found that when they used the strategies instead of their own skills, they were significantly more successful, **independent** of their skills. In fact, *novices who used the strategy did better than experts who didn't.*

- Despite all of this work that demonstrated the importance of self-regulation, one thing became clear
- No matter how carefully someone reads or writes code, or thoughtfully regulates their attention, the dominant factor in success was more strategic than cognitive
- [Read]
- Here is a quote from one of our participants
- [Read]
- When they followed strategies known to be effective, they were more successful; self-regulation was an essential skill for following those strategies. But strong self-regulation on bad strategies did not help.

What is programming, cognitively?

A social, distributed, and cognitively immersive form of surgical "sculpting" with logic, structure, and data that requires frequent learning, reasoning, externalization, about program execution; immense persistence, patience, precision, and growth mindset; and a robust capacity for self-regulation and metacognition, cognitively, socially, and organizationally.

No wonder it's so hard to learn and teach!

- So that was 20 years of research on programming.
- To summarize it one sentence, I would argue that the activity, at least cognitively, is
- [Read]

Programming as politics

Programming as Cognition, Programming as Politics — Dr. Amy J. Ko, Ph.D.

[25 minutes]

I'm proud on of our work to explaining programming. The work has been highly impactful, shaping developer tools, curriculum, software engineering methods, and even hiring practices in the software industry. I think it's good work.

But there was always something that bothered me about it; it never really considered *what* people program, or *why* they program it. These things matter immensely, even in low level decisions about how programs are constructed, and yet too often, they are afterthoughts when people write code.

This brings is to my second topic: programming as politics.

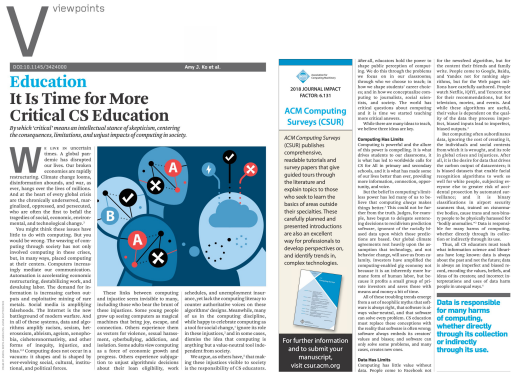
I first started thinking about this more intentionally when when I finally accepted that I was transgender about five years ago. My every attempt to live in a digital world started breaking: databases were designed in ways that prevented me from changing my name, doctors made medical errors because data schemas couldn't accurately encode my anatomy and physiology, automated marketing platforms deadnamed me a dozen times a day and gave me no power to stop them. It was hard to see software as anything but a tool of capitalism designed for the cisgender majority, at my expense.

And so I began to ask: how is it that this fascinating skill of programming so often leads to the oppression of me, of my communities, and so many other marginalized

groups?

Building on countless works from those that came before me, here’s what I’ve learned so far.

CS education is usually apolitical


viewpoints

Education
It Is Time for More Critical CS Education

By which 'critical' means an intellectual stance of skepticism, centering the consequences, limitations, and unjust impacts of computing in society.

ACM Computing Surveys (CSUR)

I wrote a critique of CS education, arguing that it's apolitical stance is harmful, because it ignores the complicity of algorithms, data, and software developers in amplifying systems of oppression, and even in creating new ones.

I called for educators and researchers to begin to examine the limits of data and computation and the responsibility of programmers, and to explore how to teach these limits and responsibilities to future generations of engineers.

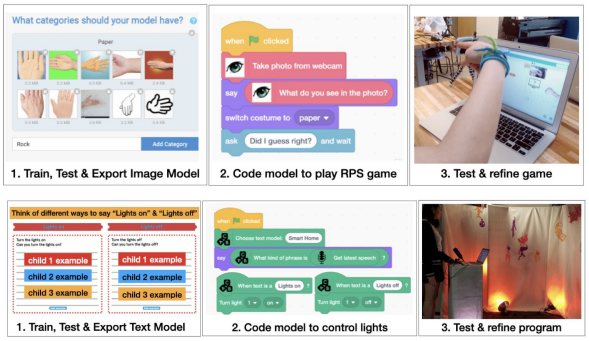
Amy J. Ko, Alannah Oleson, Mara Kirdani-Ryan, Yim Register, Benjamin Xie, Mina Tari, Matt Davidson, Stefania Druga, Dastyni Loksa, Greg Nelson (2020). It's Time for More Critical CS Education. Communications of the ACM (CACM), 31-33. https://doi.org/10.1145/3424000

- I started with observing that
- [Read]
- And then I got to work trying to answer my call.

Overcoming misplaced faith in the authority of code requires **exposure**

Many people, especially children, perceive computers and software as magical and authoritative, willfully granting them power over their lives and communities. Why?

But when we engaged several dozen youth in creative applications of simple machine learned programs over the course of several weeks, we found that their perceptions of machine intelligence rapidly shifted from **unquestioned faith** in the computers as authoritative to **skepticism** in their severe limits.



Stefania Druga, Amy J. Ko (2021). How Do Children's Perceptions of Machine Intelligence Change when Training & Coding Smart Programs? ACM Interaction Design for Children, 49-61. https://doi.org/10.1145/3459990.3460712

- One thing we examined was why people have such faith in computing to be correct and "smart"
- It turns out that...
- [Read]
- So this faith ubiquitous but is brittle: even a little bit of programming breaks the spell quite quickly.

Comprehension is mediated by **egocentricity**

We experimentally compared learning basic machine learning concepts from 1) concrete *impersonal* datasets, and 2) concrete *personal* datasets, against their impact on learners' ability to advocate for or against machine learning in social contexts in technical terms.

Personal data sets were superior at promoting not only learning of the concepts, but also **near transfer** to model analysis tasks, and **far transfer** in machine learning advocacy tasks.

Yim Register, Amy J. Ko (2020). Learning Machine Learning with Personal Data Helps Stakeholders Ground Advocacy Arguments in Model Mechanics. ACM International Computing Education Research Conference (ICER), 67–78. https://doi.org/10.1145/3372782.3406252

- But how that exposure happens appears to matter too.
- For example, we examined machine learning in the context of political advocacy.
- The tutorial shown here varied only in whether learners were using generic data sets or data from their own lives.
- [Read]
- And so centering people's own lived experiences when learning about computation appears to not only lead to better learning, but more forceful and convincing advocacy against harmful applications of machine learning.

Design and **development** are different

Much of the public and the software industry conflate these two things:

- **Design**. Deciding *what* to build.
- **Development**. Deciding *how* to build it.

For example, we analyzed several national curricula, and found that most conflated these these two skills, and framed programming *as* design, but only taught development.

Professional software developers are often portrayed as doing both, and often *do* both in reality because people give them that power.

	Problem-Space Design					**Program-Space Design**				
	UtC	*CI*	*ES*	*II*	*Comm*	*UtC*	*CI*	*ES*	*II*	*Comm*
Code.org CSD										
Problem Solving	✓	✓	✓	✓	✓	✓	✓			
Web Development	✓	✓				✓	✓	✓	✓	✓
Interactive Animation and Games	✓	✓	✓		✓	✓	✓	✓	✓	✓
The Design Process	✓	✓	✓	✓	✓	✓	✓		✓	✓
Data and Society	✓	✓	✓	✓	✓		✓		✓	
Physical Computing	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
AP CS A										
Primitive Types								✓	✓	
Using Objects							✓	✓		
Boolean Expressions and "if" Statements							✓	✓	✓	✓
Iteration							✓	✓		
Writing Classes								✓		
Array							✓		✓	✓
ArrayList						✓	✓	✓		
2D Arrays							✓	✓		
Inheritance							✓	✓		
Recursion								✓		✓
Exploring Computer Science										
Human Computer Interaction	✓	✓	✓		✓					
Problem Solving	✓						✓	✓		✓
Web Design	✓	✓	✓	✓	✓		✓	✓	✓	
Introduction to Programming	✓	✓	✓		✓	✓	✓	✓	✓	✓
Computing and Data Analysis	✓		✓	✓	✓					
Robotics	✓	✓	✓		✓		✓	✓	✓	✓

Alannah Oleson, Amy J. Ko, Brett Wortzman (2020). On the Role of Design in K-12 Computing Education. ACM Transactions on Computing Education, Article 2. https://doi.org/10.1145/3427594

- Dispelling myths about programming is one thing; helping people understand what programming is and isn't is also critical.
- In particular, we observed that...
- [Read]
- In away, this conflation is an irresponsible power grab: if programming *also* design, but then we don't teach design purposefully and thoughtfully to ensure that people don't design harmful, exclusionary, oppressive things, then when will anyone ever learn to do otherwise?

Who learns is shaped by **stereotypes, stigma**

"This carried me through high school into college where my love of programming has been brutally murdered by out of control CS Monsters. I said earlier that my love of the subject matter was inspired through socialization. Well, many of the people I have met in the CS major have grated on my nerves like a cheese grater. They are possibly the most proud people I have ever met."

Amy J. Ko (2009). Attitudes and Self-Efficacy in Young Adults' Computing Autobiographies. IEEE Symposium on Visual Languages and Human-Centered Computing (VL/HCC), 67-74. http://dx.doi.org/10.1109/VLHCC.2009.5295297

I solicited several dozen "code autobiographies" from students who did and did not succeed in learning to code. Their lifetime of experiences revealed that:

- First encounters are often inaccessible, unsupported, stigmatized
- Mentorship was a critical factor in building resilience to programming difficulties
- Toxic computer science cultures could quickly erase an entire lifetime of positive experiences.

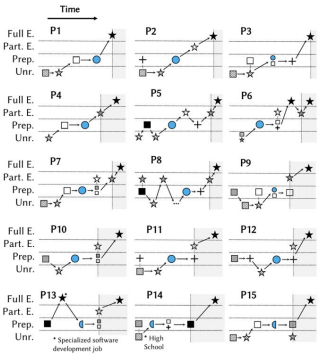
- So not only are most people not learning about programming, not learning about it effectively, not learning about it correctly
- But who is learning about it is highly skewed towards those at the top of our social hierarchies.
- I found that… [Read]

Second chances are rife with similar barriers

We interviewed dozens of adults, mostly women of color, who pursued coding bootcamps to pivot into software development careers. They reported similar issues as students entering higher education CS learning contexts, but also faced:

- Stigma from family
- Heavy financial burdens
- Lost relationships due to toxic work/life culture in bootcamps and industry
- Humiliation from instructors and peers

Most quit and failed to get a job.



"There was this one time where my database wouldn't work because I hadn't capitalized a letter and I asked one of the assistant teachers about that and he thought it was ridiculous that I made a mistake about this capital letter." (Black woman)

Kyle Thayer, Amy J. Ko (2017). Barriers Faced by Coding Bootcamp Students. ACM International Computing Education Research Conference (ICER), 245-253. https://doi.org/10.1145/3105726.3106176

- Industry perpetuates these toxic cultures in retraining efforts.
- For example, [Read]
- Some of these negative encounters are structural barriers imposed by corporations and capitalism, which have committed to myths of meritocracy tied to proxy indicators of programming skill, like puzzles, gender, or where someone graduated.
- [Read quote]

Structural inequities limit learning

"*I feel like there is a big culture of people working together to understand material. In some ways, I feel like that's a good thing, and in other ways I feel like that's not really fair to a lot of students. If you work by yourself, you won't understand as much as if you work with other students. If you commute, you can't work with other students.*"

We studied hundreds of students who transferred from 2-year colleges to our university to study computer science. Most struggled to earn comparable grades to their "native" peers, but attributed those differences to:

- Longer commutes
- Greater caregiving responsibilities
- Being excluded from study groups due to age, availability
- Lack of faculty awareness and accommodation of inequities.

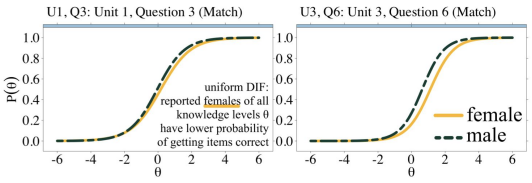
Harrison Kwik, Benjamin Xie, Amy J. Ko (2018). Experiences of Computer Science Transfer Students. ACM International Computing Education Research Conference (ICER), 115-123. https://doi.org/10.1145/3230977.3231004

- We saw similar disregard for students of color transferring to our own institution to study computer science, where...
- [Read]
- And so the broader context of learning disregards differences in resources and capacity to learn, only supporting students who have everything they need.

Assessments embed **gender, racial bias**

In a series of studies, including one on a corpus of ~140,000 responses from 20,000 students across the U.S., we've used psychometric techniques to examine the bias in programming assessments, finding that:

- Many items have systematic bias against women and marginalized racial groups
- Which items have bias is unpredictable and often unexplainable by item designers
- Psychometric analysis require insurmountable expertise for designers to interpret, limiting their actionability.



Matt Davidson, Amy J. Ko, Brett Wortzman (2021). Investigating Item Bias in a CS1 exam with Differential Item Functioning. ACM Technical Symposium on Computer Science Education (SIGCSE), Research Track, 1142-1148. https://doi.org/10.1145/3408877.3432397

Benjamin Xie, Matt Davidson, Baker Franke, Emily McLeod, Min Li, Amy J. Ko (2021). Domain Experts' Interpretations of Assessment Bias in a Scaled, Online Computer Science Curriculum. ACM Learning at Scale, 77-89. https://doi.org/10.1145/3430895.3460141

- We've found the same disregard for diversity appear in assessments
- [Read]
- This plots on the right show some of the gender differences in some assessments that we've found
- The designers of these don't even know what their doing to create these differences; they seem to just be deeply implicit stereotypes that shape the choice of problems, and then because those assessments are used in summative ways as a supposedly objective measure of "meritocracy", they end up excluding in ways consistent with gender stereotypes.

Mentorship helps, but doesn't fix oppressive structures

"*My neighbor Laura who did APCS and really enjoyed it and introduced me to it.*" (SY, Hispanic female, 15)

"*My dad is a software engineer and he frequently talks to me about his job. He has enrolled me in several classes and in our free time, he often teaches me.*" (SY, Asian female, 14)

Amy J. Ko, Katie Davis (2017). Computing Mentorship in a Software Boomtown: Relationships to Adolescent Interest and Beliefs. ACM International Computing Education Research Conference (ICER), 236-244. https://doi.org/10.1145/3105726.3106177

Amy J. Ko, Leanne Hwa, Katie Davis, Jason Yip (2018). Informal Mentoring of Adolescents about Computing: Relationships, Roles, Qualities, and Impact. ACM Technical Symposium on Computer Science Education (SIGCSE), Research Track, 236-244. https://doi.org/10.1145/3159450.3159475

Across two summer high school programming courses, we examined the social networks of students who had learned to code, finding a rich diversity of informal mentoring relationships.

- Mentors were teachers, friends, siblings.
- Having an informal mentor strongly mediated increases in interest in programming after the course.
- Students who shared the identity of the instructor had higher interest in code after the course, viewing her as a role model.
- Having a mentor was not enough to help students persist when they encountered exclusionary, toxic learning communities.

- And so teaching ignores politics at the expense of learning, it demphasizes impact at the expense of marginalized groups, and it systematically excludes those who are harmed through disregard and ignorance.
- Does anything help?
- Mentorship, but only a little.
- [Read]

Teaching programming as a **political** act grows communities of support

We taught a summer course to marginalized youth of color and rather than focus on purely technical aspects of programming, we focused on the **sociopolitical** aspects of programming: how it is used to create and reinforce systems of oppression in broader society.

Students reported leaving the class with an entire community of peers with a shared mission of harnessing code for justice.

"*i think a key to reducing inequalities of like,* AI, *is by reducing inequalities everywhere else first/cause ultimately its humans designing all of these digital systems and basing all of their datasets and machine learning off of existing human systems/so without first breaking down the human systems that cause inequality we'll always be producing machines that reinforce that*" – student

Jayne Everson, Megumi Kivuva, Amy J. Ko (2022). "A key to reducing inequities in like, AI, is by reducing inequities everywhere first": Emerging critical consciousness in a co-constructed secondary CS classroom. ACM SIGCSE, to appear.

- But centering politics in programming can actually help a lot
- [Read]

Dominant groups resist learning programming sociopolitically

"*[as a] CS-minded person who believes efficiency more than anything, this unit alters my mind...*" (Asian woman)

"*Everything I'm learning in this course is excellent, but the socio-technical content is boring [and] unnecessary.*" (White man)

We revised a required CS course on **computer architecture**, linking the history of decisions underlying computers and operating systems to world wars, wars on drugs, anti-Black policing projects, corporate monopoly, and neoliberalism.

- All students expressed surprising and newfound awareness of the historical social context of programming
- Marginalized students in the class resonated with links to oppression, justice
- Some students from dominant groups found it distracting and irrelevant; others worked hard to integrate it into their sense of self

Mara Kirdani-Ryan, Amy J. Ko (2022). The House of Computing: Integrating Counternarratives into Computer Systems Education. ACM SIGCSE, to appear.

- Unfortunately, when you bring the white kids into the picture, the resistance is strong
- [Read]

What is programming, politically?

Mostly White and Asian men making harmful design choices about how our digital world should work, often from a place of ignorance and disregard of the diversity of human values and experiences, and with a commitment to exploitative, extractive, normative, capitalist, meritocratic goals of efficiency, convenience, and profit.

Faculty, students, and professionals from these dominant groups systematically exclude others and resist any change in culture or curriculum that might threaten the status quo.

Programming as Cognition, Programming as Politics — Dr. Amy J. Ko, Ph.D.

Reconciliation

- [~40 minutes]
- It's easy to see these two threads of scholarship as irreconcilable.
- After all, one paints a picture of a very hard cognitive task, completely agnostic to who is doing it. That account of programming might view anyone who succeeds at learning to program as demonstrating great merit and overcoming great odds. And this is generally how scientists have treated it.
- But lurking beneath that narrative is a often a darker assumption: that there is something inherent, intrinsic, or natural about a predisposition to program successfully, that it is determined by personality. Taking only a cognitive view, even a socially distributed cognitive view of programming, ignores the broader cultural and political forces at play in this increasingly critical and contested skill.
- The sociopolitical view of programming simply examines and acknowledges these forces. It accepts that programming is hard, but also says that it is made much harder for marginalized groups by both explicit and implicit choices by dominant groups to exclude them. And it is made harder by refusing to allow for programming to be viewed as anything but a strictly cognitive, technical activity.
- And so reconciliation then, to me, does not seem like such a difficult intellectual task.
- Here's what I think it takes.

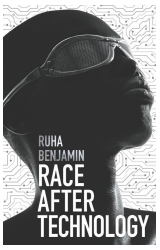
1. Accept that programming is **cognitive** *and* **political**

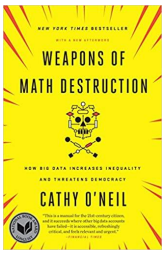
Treating them as a dichotomy is unhelpful and incorrect. Programming is both at the same time, and can and should be taught, discussed, and performed accordingly.

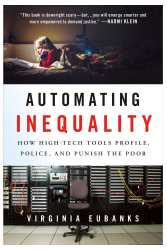
- [Read]

2. Examine the **interactions** between the cognitive and political in programming

That means examining ideas from critical race theory, which imply that racist (and sexist, ableist, and transphobic) ideas and outcomes are encoded into computer programs just as they are in law.



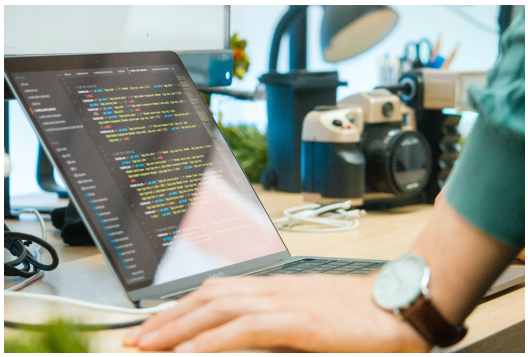




- [Read]

3. Begin to examine "**political cognition**" as a central part of programming skill

When someone is writing a line of code, how do we help them reason about it in **political terms**, and weave that into the other more strictly technical and cognitive challenges in programming?



- [Read]

Twenty chapters that weave together foundations of computing, foundations of social justice, and methods for teaching at this intersection in secondary and post-secondary settings.

Programming as Cognition, Programming as Politics — Dr. Amy J. Ko,

- My lab is just beginning to explore these three in a new book, *Critically Conscious Computing*, which we will release next month
- It's primary audience is secondary and post-secondary CS educators
- It tries to do three things:
    - It makes the case for CS education not just as a pathway to good paying jobs, or a means to personal expression, but as one of the most important fronts in preserving democratic norms and institutions throughout society
    - It teaches foundations of CS in sociotechnical and sociopolitical terms. For example, it doesn't just explain the syntax and semantics of an if statement, but also the social and political consequences of if statements when deployed into the world.
    - It offers new teaching methods for teaching CS in these terms, building on Paulo Freire's notion of dialogic teaching, which centers discourse aimed at helping students recognizing their limiting situations in society, and their power to organize and act against their oppressors. Code, after all, isn't just a tool for those with power, but also a tool for the powerless.
- We launch online on December 6th. The book is free, built for the web, and will be a living document that evolves with community feedback.

Thank you!

Key ideas:

- Programming is a social, distributed, and immersive "sculpting" with logic that requires learning, reasoning, externalization, persistence, patience, precision, and self-regulation.
- Programming is mostly White and Asian men making harmful design choices about our world, often from a place of ignorance and disregard of the diversity of human values and experiences, and prioritizing exploitative, extractive, normative capitalist goals of efficiency, convenience, and profit.
- We must accept that programming is both of these things, examine how these two dimensions interact, and deepen our understanding of the "political cognition" at play in programming to imagine more just applications of computing in the world.

Programming as Cognition, Programming as Politics — Dr. Amy J. Ko, Ph.D.

This material is based upon work supported by the National Science Foundation, and Google, Microsoft, Adobe, Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

- [~40 min]
- Thank you for your time and attention
- Here are the key ideas from the talk
- I'm happy to take any questions