

No Garden of Eden

Functional Programming and the Myth of Ruining Programmers

Peter Berger

`peterb -at- gmail.com`

Hi, thanks for having me here. Today I'm going to talk about teaching kids functional programming and Haskell.

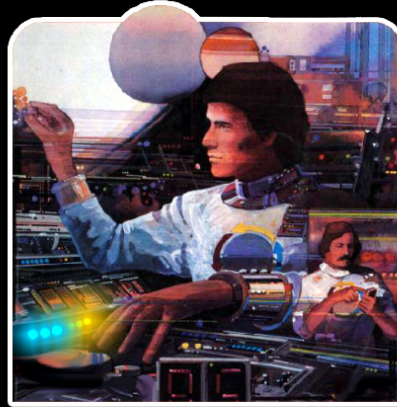
Who am I?

I'm a professional software developer, and have been for most of my adult life. My background is in networking and systems software, but for the past 15 years I've been developing consumer applications.

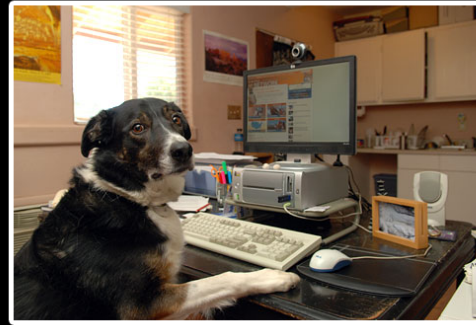
Volunteering at School

I also volunteer several times a year at public schools, talking about computers generally, and programming specifically, to children at a number of ages, but especially what we in the United States call “middle school.”

THE TWO STATES OF EVERY PROGRAMMER



I AM A GOD.



**I HAVE NO IDEA
WHAT I'M DOING.**

When I teach these classes to young people I have in the past opened with this image when talking about programming. I like the image because I'm trying to get across the idea that mistakes are a natural part of programming. But I might stop using it, and we're going to talk about why.

Edsger W. Dijkstra

“How do we tell truths that might hurt?”

ACM SIGPLAN Notices
Volume 17 Issue 5, May 1982
Pages 13 - 15

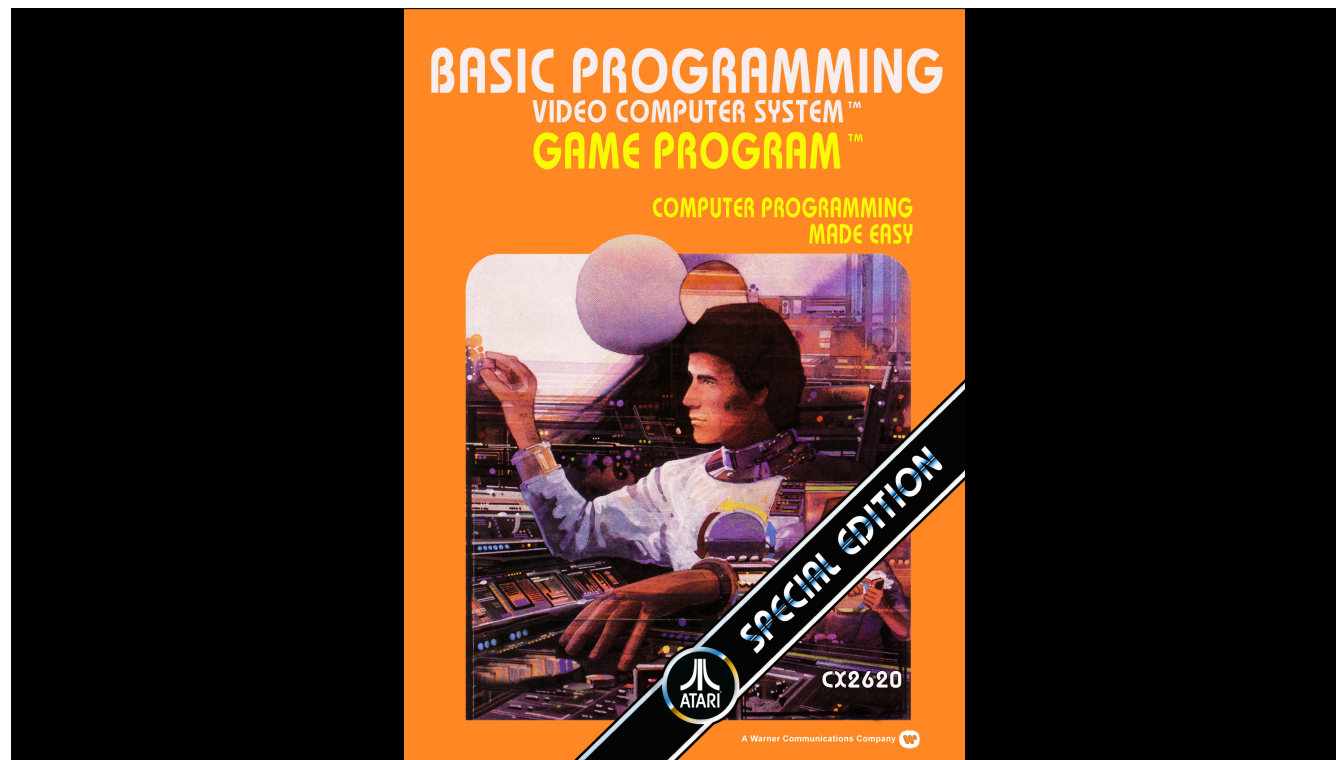
It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.

The impetus for this talk came from reading old computer science papers, a totally normal activity that I’m pretty sure everyone in this room does from time to time. In 1975, Edsger Dijkstra wrote a paper called “How Do We Tell Truths That Might Hurt”. It was mostly intended to be humorous, and was presented as a series of aphorisms

It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.

This one jumped out at me:

“It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.”



And I thought to myself “Hey, wait a minute. I started out by learning BASIC.”



Am I ruined? So at least part of this talk is me taking a shot at Dijkstra to soothe my wounded ego. But honestly, if this was just about Dijkstra's snarky paper, I would simply let it drop.

Image credit: Wikimedia. https://upload.wikimedia.org/wikipedia/commons/7/76/Fallen-ice_cream-cone.JPG

“Functional programming is **fairly intuitive** to someone who’s not (yet) a programmer.”

“People that don’t have the **baggage** of the imperative style are a lot quicker to learn functional code”

“Many programmers who were first taught imperative programming styles find it **very difficult** to learn functional programming, because it is so different.”

“Might be easier to learn functional programming before mind gets **corrupted** with imperative/OO”

“Studies have shown that if one learns a functional programming language first, it is **much easier** to learn imperative languages.”

My real motivation is that his quote represents an attitude I see echoed frequently in functional programming circles.

**“Studies have shown that if one
learns a functional programming
language first, it is *much easier* to
learn imperative languages.”**

This is a claim I've seen several times. There has been some research on this, but I think this claim is overstated. I went looking for proof of this claim, and I didn't find it.

Students learned to handle abstraction as a design tool and their skills in formal manipulation improved

Joosten, Stef & Berg, Klaas & Hoeven, Gerrit. (1993). Teaching Functional Programming to First-Year Students. *Journal of Functional Programming*. 3. 49-65

[T]eaching purely functional programming as such in freshman courses is *detrimental* to both the curriculum as well as to promoting the paradigm.

Chakravarty, Manuel & Keller, Gabriele. (2004). The risks and benefits of teaching purely functional programming in first year. *J. Funct. Program.* 14. 113-123.

“Some believe that prior knowledge of imperative programming is an obstacle to learning functional programming — perhaps inspired by Dijkstra’s comment that learning Basic mutilates the mind. *This is not my experience.*”

Hughes, John. (2008). Experiences from teaching functional programming at Chalmers. *SIGPLAN Notices*. 43. 77-80. 10.1145/1480828.1480845.

Instead, I found evidence that it’s possible to teach functional programming first, and that the results are exactly what you’d expect: it’s better in some ways, worse in others, but probably doesn’t have a life-changing impact.

“Some believe that prior knowledge of imperative programming is an obstacle to learning functional programming—perhaps inspired by Dijkstra’s comment that learning Basic mutilates the mind.

This is not my experience.”

Hughes, John. (2008). Experiences from teaching functional programming at Chalmers. SIGPLAN Notices. 43. 77-80. 10.1145/1480828.1480845.

“Such problems had led to an “I hate ML” club among the students at the time I took over the course.”

Instead, I found evidence that it’s possible to teach functional programming first, and that the results are exactly what you’d expect: it’s better in some ways, worse in others, but probably doesn’t have a life-changing impact.

Maybe it's true. Maybe it's not.

But it's not *proven* true.

It's more or less an **urban legend**.

So, this claim about teaching functional first being effective is made often. Maybe it's true, maybe it's not, but it's certainly not *proven* true. It's an urban legend. It's a way of telling ourselves what we want to hear.

Teaching Functional Programming for High School Students - Lapidot, Levy

Teaching Functional Programming and Erlang: The Galician Experience- Gullas

P.Wadler. Why no one uses functional languages?

Our experience teaching functional programming at University of Río Cuarto (Argentina) - Szpiniak, Luna

Engaging With Kids

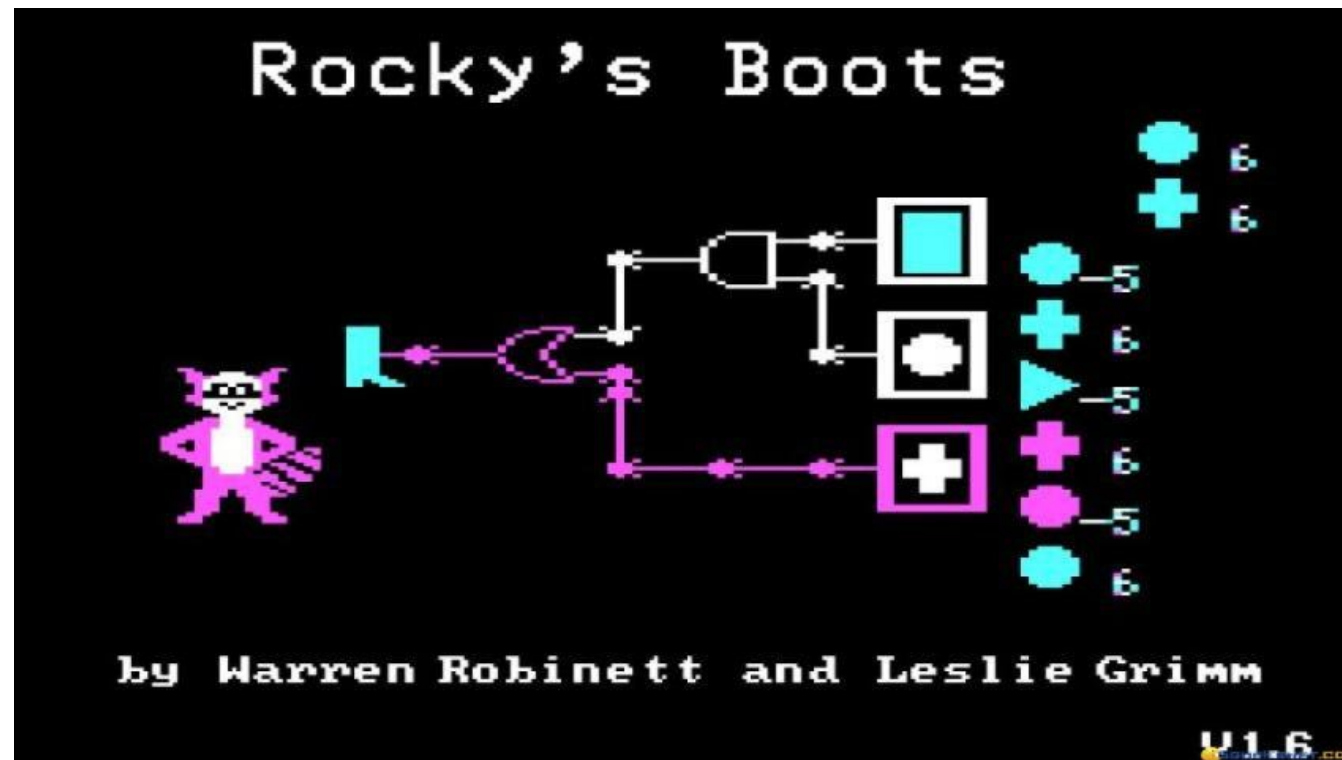
- I've worked with middle-school kids for a few days once a year for the past several years.
- Typically do 1- or 2- day practical group projects ("Make a space invaders game")
- Every class has a wide variety of skill and interest levels

I care about this idea that we can ruin people because I spend at least part of my time volunteering with kids teaching programming. I do day-long projects where we do a group project.

Languages and Environments

- JavaScript (+ Processing.js)
- Python
- Lua (+ LOVE2D)
- Elm
- Haskell (+ Code.World)

How I teach these classes depends on the grade level, what resources are available, and how long we have for the activity. We've used all of these languages at various times, and some others as well.



But sometimes, we didn't have access to computers to run programs on, and so I have to improvise. So for example, in one class of primary school kids (younger than about 10), we didn't actually have computers to run programs on. In that case I introduced the idea of logic gates and we played a live-action version of the game "Rocky's Boots", with the kids designing simple circuits on a whiteboard, and then implementing them as a human chain, with each kid serving as either a logic gate or a sensor.

This is, by the way, a hugely fun thing to do.



For older kids, who typically have better access to computers and usually more patience, my go-to move is to have them design and implement a simple video game. This, for example, is a simple space invaders game implemented in Lua/Love2D

Haskell Experiments

- 2-student group solving classic list exercises.
- Same students then created Sokoban, following Joachim Breitner's course materials.
- 10 students collaboratively designing and implementing Hangman.

When I came up with the idea of using Haskell as the language for these classes I wasn't entirely sure it would work. So I ran a few trials first, before rolling it out to a larger group. This helped me figure out what worked and what didn't.

[Ran three classes - two very small, one larger.

First small class was with two advanced students, focused around the first 10 of the 99 Lisp Problems. This was a trial run to gauge how terrible of an idea this was.

Second class was Joachim Breitner's "Create a Sokoban game" on codeworld.info.

Third class spent time on general functional thinking and decomposition, and then cooperatively designed and implemented Hangman using the repl.it live browser-based collaboration.]

**“What do you like about
Haskell?”**

At the end of the third class, I asked a few questions. I’m going to tell you what I found out about what the kids liked and what they found difficult, and then I’m going to bring it all together with the conclusions I have drawn from this. So, let’s start on a positive note. When I taught Haskell, what parts of it did the students in my class like?

“Kind of a bit different from Python.”

-Zach

“I kinda like the change of how it’s not like all the other languages.
But that can be annoying also. It’s a nice change of pace.”

-Dario

“Kind of a bit different from Python,” says Zach.

“It’s kinda cool **how much you can do** with just a few lines.”

-Samantha

“It’s kinda cool how much you can do with just a few lines.” Haskell is famously concise, and this was something the students who had experience in other programming languages noticed.

“I like how you can define a variable and just start using it without declaring it first.”

-Tyler

“I like that you can name parts of your code as specifically as you want.”

-Jack

“I like how you can define a variable and just start using it without declaring it first.” I think what Tyler was talking about here was the “where” syntax. He liked being able to write the code that referenced the variable and then only put the variable’s definition later.

**“What don’t you like about
Haskell?”**

Not all is a field of roses, however.

“My method of coding doesn’t match with Haskell very well. I’d like to know if there are any alternatives to this recursion and pattern matching.”

In fact, when I first ran the experiment with the Lisp Problems, one of the students basically asked if there was some way to solve the problems iteratively.


“I guess it’s just not exactly how you would tell a friend how to do something.”

–Callie

This desire for iteration ran pretty deep. “I guess it’s just not exactly how you would tell a friend how to do something.” Put another way, Callie is saying “Functional programming isn’t intuitive to Callie.” And let me be clear, when Callie said this, everyone in the class nodded. These students approach programming in a very straightforward manner - they are largely proceeding through analogy. The computer is a friend and the challenge of programming to them is to find the words to explain the task.

Ruined?
No.

Solving Word Problems



Step 1
Identify what the problem is asking.

Step 2
Strategise what process you need to use to solve the problem.

Step 3
Set up the problem by writing the equation.

Step 4
Solve the equation.

Step 5
Check your answer.

In fact, this is something that frustrates me about the “if only we could not teach imperative languages first.” Imperative thinking is *how we teach people to break down tasks*. Not programming tasks - *any* tasks. First do one step, then take the next step, then take a third step. When you take people into a functional mindset, you’re not just asking them to *learn a new language*. You’re actually asking them to think of a *new way of decomposing problems*. That’s not a small step!

“The loops are different from most of the languages I know”

-David

Here's a more specific complaint: “The loops are different from most of the languages I know.” David loves for loops. In fact, everyone loves for loops. For loops are the best friend of mankind and a faithful companion. For example, even after learning about map, when confronted with “change everything in this array from one thing to another,” the students wanted to reach to a for loop.

When I can use for loops again = *halo of light*

Seriously. I'm not kidding. People. Really. Love. For loops.

“I just don’t like the syntax...like all the funky little carets and stuff.
Like the backtick.”

-Jack

“I didn’t like where you had to put the parentheses. In Python you
don’t have to do that as much.”

-Tyler

Two related complaints: What these complaints are calling out here is the way functions can change from prefix to infix by use of the backtick. More generally, understanding the rules of function application was a huge problem for everyone. This is an area where one of the strengths of Haskell, the idea that any symbol can potentially be a function, is super-confusing.

Do you like Haskell or Python better?

100% Python

The Takeaway

- No one in the game-making class thought Haskell was *hard*.
- However, all of them thought it was *very weird*.

In our larger game making group, no one felt the language was hard, but they did think it was very weird. The smaller experimental group (working on more abstract problems) absolutely characterized thinking in Haskell as “hard”. The conclusion that I draw from this - which isn’t really surprising, is that what the kids really found hard was abstraction.

Abstraction is Hard

Be Concrete First

Abstraction is hard. When you've been programming for 15 years and you find something that lets you easily create a powerful abstraction, part of why we get excited is we have the context to recognize where that abstraction might be useful. Without that context, things that are super-abstract can just seem pointless. So the first challenge in teaching programming to people this age - or, I'd argue, of any age - is to find an example that is very concrete. This, more than anything, is why I use "let's make a small game" as my go-to example. It's not just pandering to the kids desire for fun, although playing games certainly is fun. It's more that it means that *they understand the victory conditions*, and as a bonus I don't have to do a lot of heavy lifting to explain the motivation either. We're going to use the computer to create a thing that didn't exist before, and that thing is going to do a thing you probably already want to do.

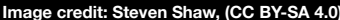
Abstraction is a tool - a very useful tool. My recommendation here is to defer introducing it until it actually helps solve a problem in a simple way. Introducing it up front is challenging.

Tools Matter

Be Focused

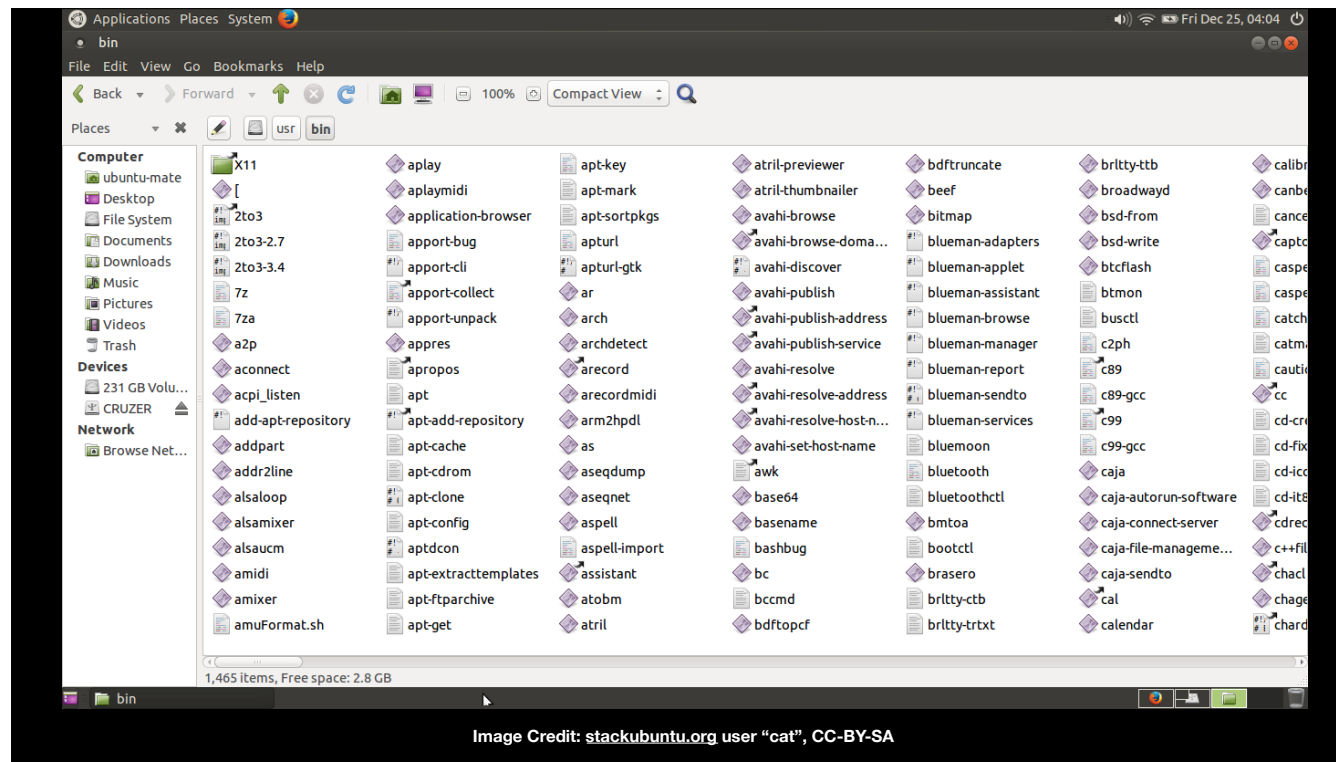
And here's my second point: I also can't overstate the importance of the tools. Given powerful enough tools, these kids created amazing things in a short period of time. But to make that work, you have to focus on what you actually want to teach, and find shortcuts to get them past roadblocks.

To use tools as a working programmer, in almost any computer language, the list of things we assume people know about is huge. In no particular order:



This also may presume you know how to or be able to *install* a text editor. If the kids' only access to a computer is at a school or library, they may very well not have the ability or permission to install programs of their own.

Image credit: © Steven Shaw 2004-2019, creative commons share-alike

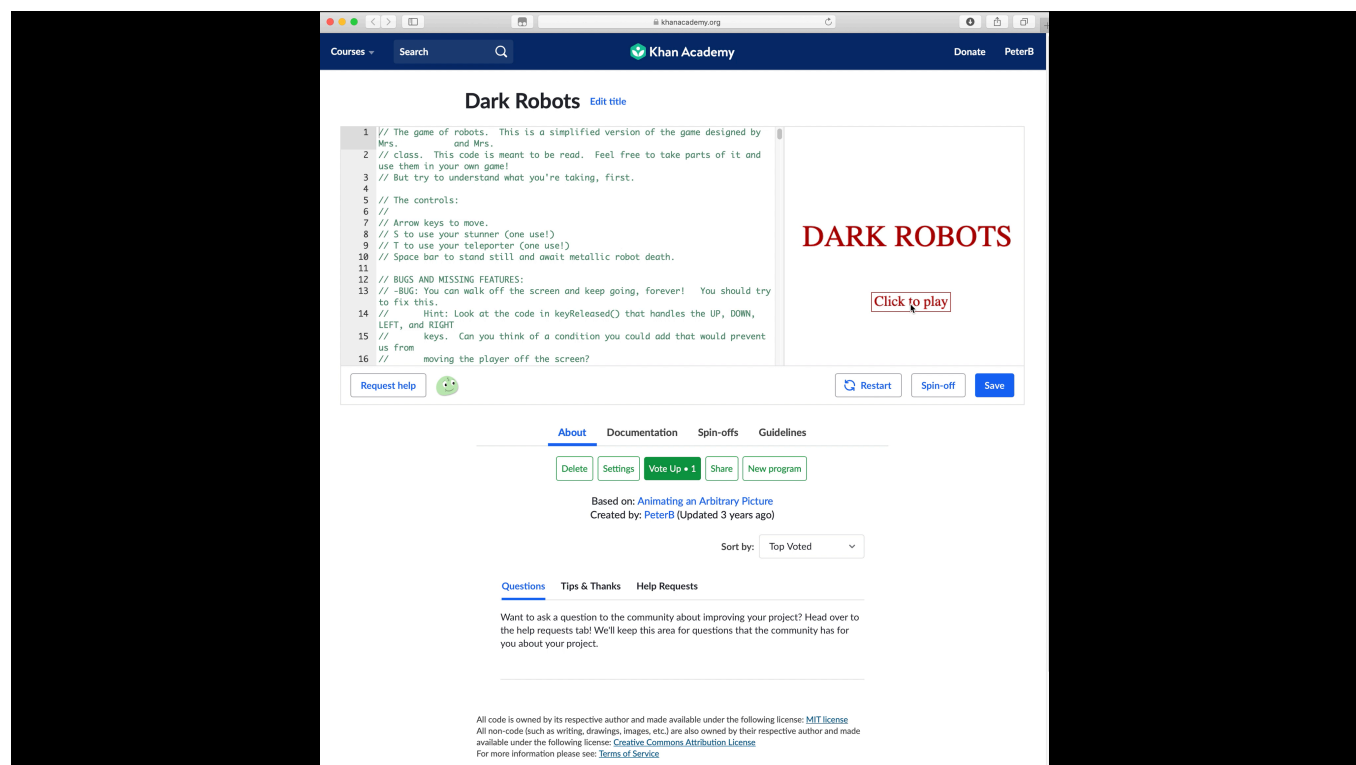


How to manage files and directories.

```
Peregrine:hangman pberger$ ~/.stack//programs/x86_64-osx/ghc-8.6.5/bin/ghc Hangman.hs
[1 of 1] Compiling Main                ( Hangman.hs, Hangman.o )

Hangman.hs:49:22: error:
• No instance for (Num String) arising from a use of 'bogus'
• In the first argument of 'gameLoop', namely
  '(bogus gs letterInput)'
In the expression: gameLoop (bogus gs letterInput)
In a stmt of a 'do' block:
    if (length letterInput == 0)
        || (not (isLetter (head letterInput))) then
        gameLoop gs
    else
        gameLoop (bogus gs letterInput)
49 |         else gameLoop (bogus gs letterInput)
   |                        ^^^^^^^^^^^^^^^^^^^^^
   |
```

Actually compiling and running the code.

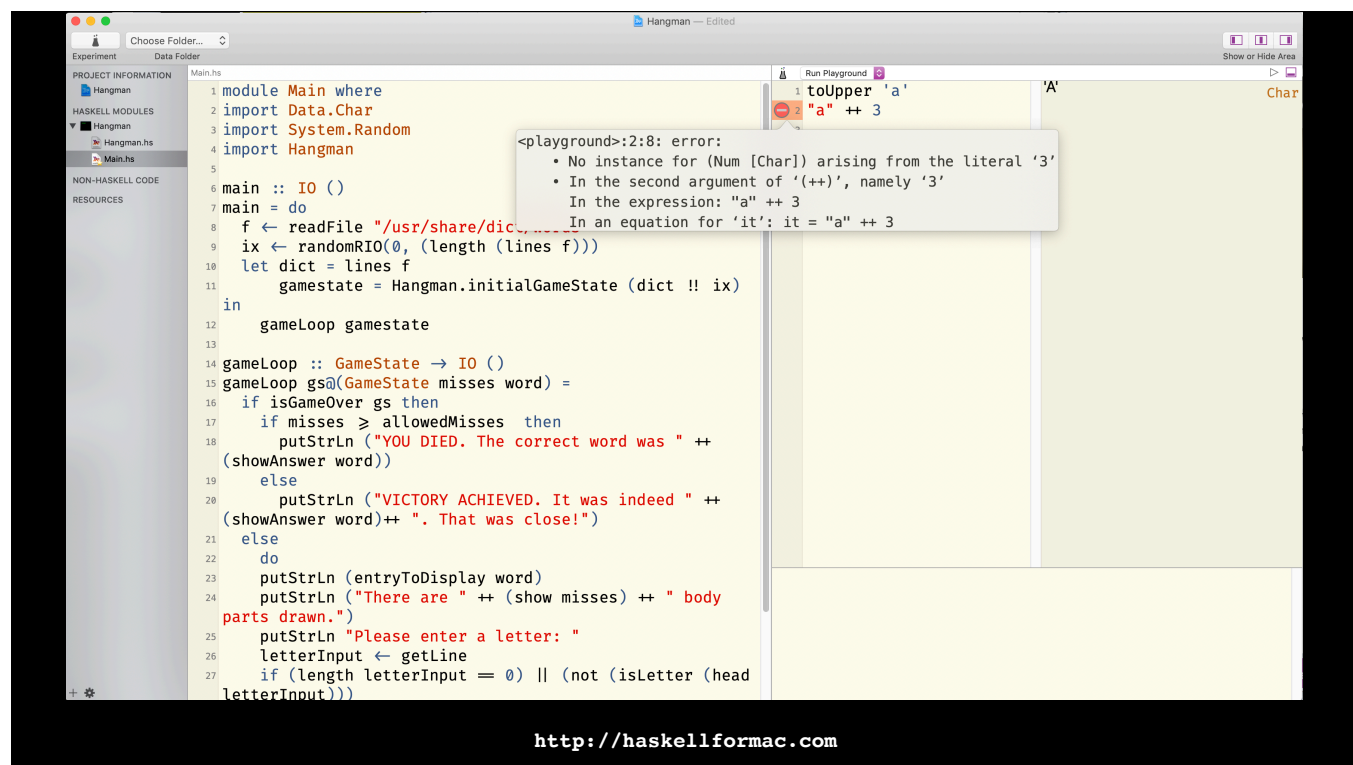


In fact, it was this choice-of-tools question that guided many of my early classes - there's a built-in incentive to use web-based tools for languages. Khan Academy's programming environment, based on processing.js, was an early winner here. This is what their programming environment looks like. Your code is in the left hand window, your program is running on the right-hand side, and file management is completely handled by the web app. Changes to the program are reflected almost immediately.

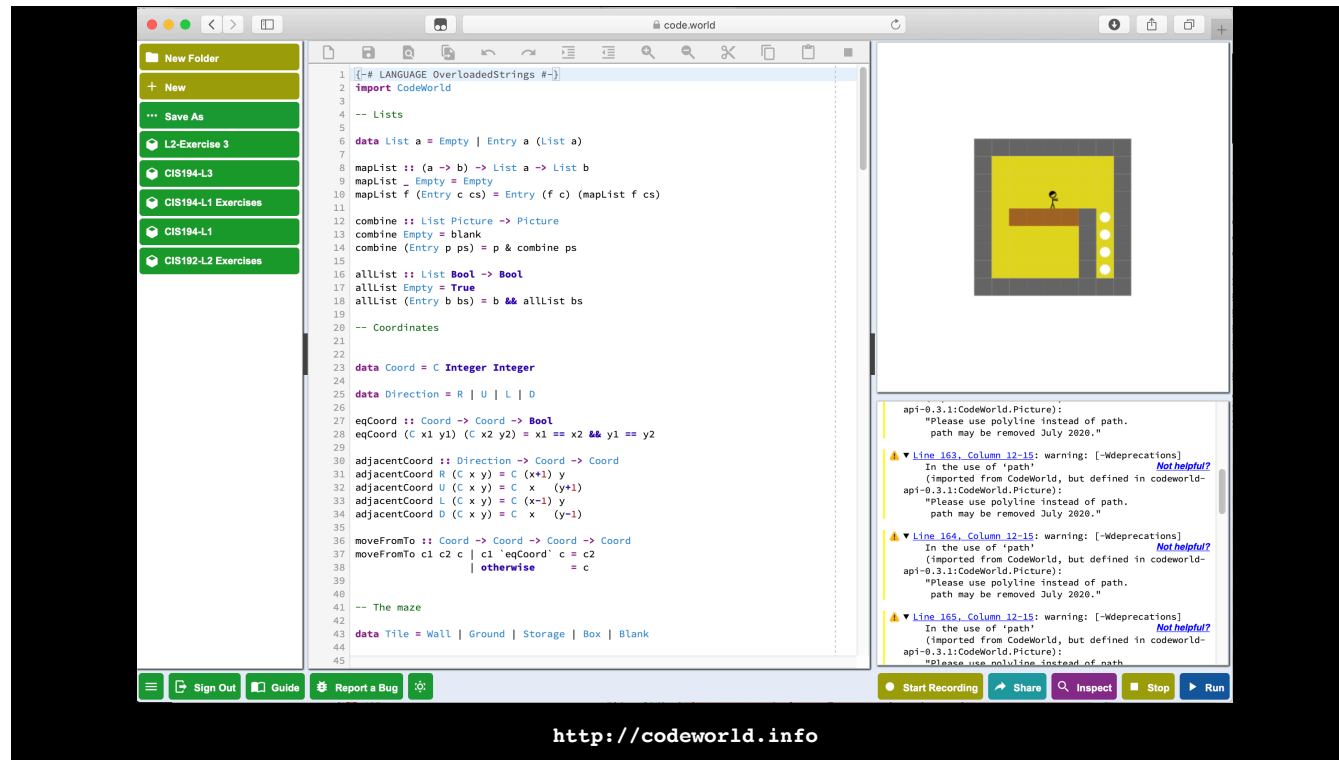
There's also an important fairness issue here: what if the kid wants to take their program with them or work on it later? If I'm using almost any native development environment, I'm making more assumptions: that the learner has access to a native platform similar to what we used in the class, that they have a USB stick or can stash the code somewhere, and so on. With a web app, literally the only thing the learner needs is an account on the platform, and access to any computer with a web browser, for instance at a public library.

Haskell Tools

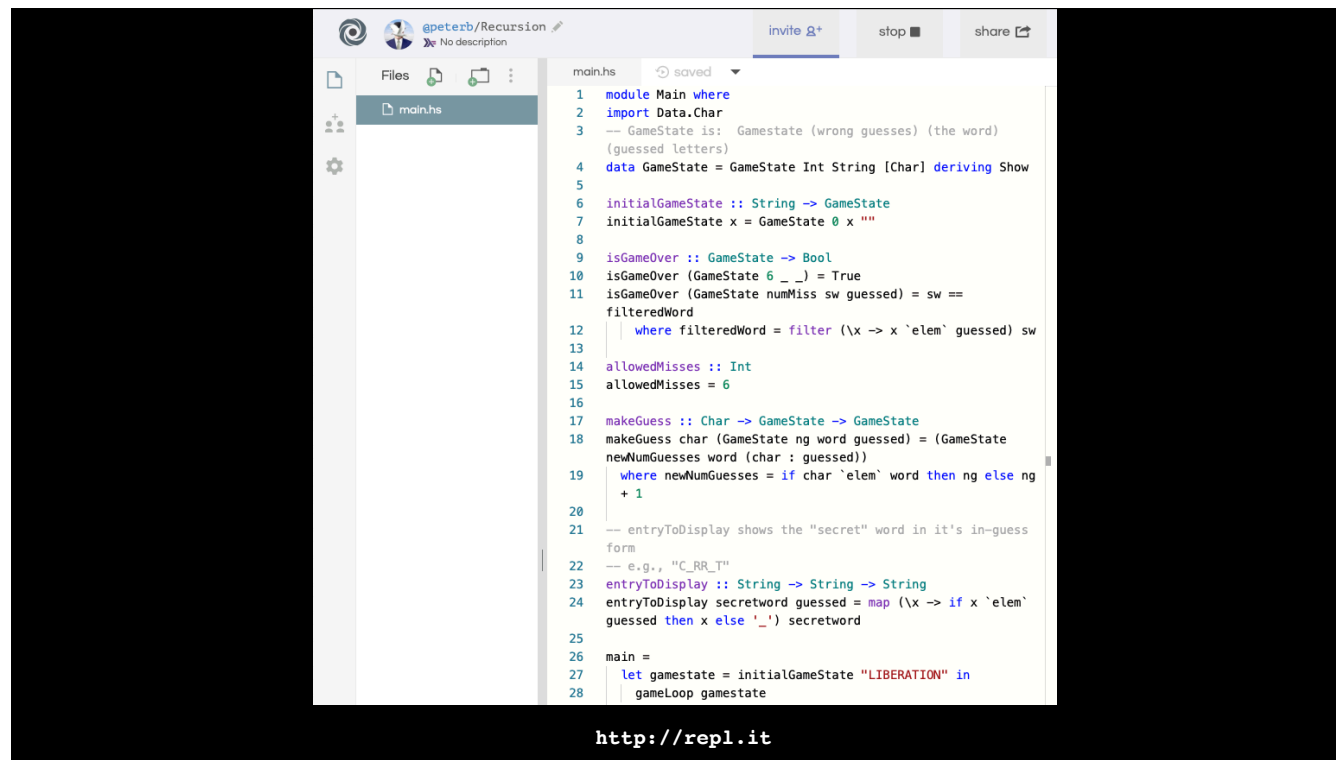
So I learned that the tools can stand between success and failure. How did I apply that lesson to teaching Haskell? I tried three solutions.



including Haskell for Mac...



Code.World...



And repl.it. I eventually settled on repl.it because of its “multiplayer” editing environment.

Some Haskell-Specific Notes

- The error messages are terrible.
- Syntactical challenges: space as the function application operator is incredibly confusing to new learners.
- It can be hard to grasp that type signatures and values live in different worlds.

So apart from the environment, what did people struggle with? A few things.

Error Message Example

```
Hangman.hs:46:32: error:
  • Couldn't match type '[Char]' with 'Char'
    Expected type: Char
    Actual type: String
  • In the first argument of 'makeGuess', namely 'letterInput'
    In the first argument of 'gameLoop', namely
      '(makeGuess letterInput gs)'
    In the expression: gameLoop (makeGuess letterInput gs)

46 |         else gameLoop (makeGuess letterInput gs)
```

With middle school students and up, I spent much less time on problem decomposition than I thought I would. The students are really good at figuring out a reasonable way to break down most games into manageable chunks, sometimes coming up with solutions that I didn't expect but which are perfectly valid. Once the code is underway, my main role becomes "Explainer Of Error Messages." In Haskell, this is the most frustrating role possible.

Here for example: I took our Hangman program and fed a string into a function that wanted a character as an argument, which would be a fairly typical mistake. Think of all the things you need to know to understand this error message: that `[Char]` and `String` are the same thing (why can't the error message pick just one of them to use?), and that the entire latter half of the error message is just trying to tell you that the actual type mismatch was this argument to `letterInput`. In other words, this error message repeats information in multiple yet inconsistent ways, doesn't order the information it's giving you by importance, and is generally hard to read. And this is probably one of the easier error messages to read!

Let's look at the same error message in Elm.

Same Error in Elm

The 1st argument to `makeGuess`` is not what I expect:

```
6| main = Html.text (makeGuess "Five")  
                        ^^^^^^
```

This argument is a string of type:

`String`

But `makeGuess`` needs the 1st argument to be:

`Char`

It's reporting essentially the same information, but I'd argue is a lot easier to read.

The Haskell error message would have been even harder to interpret if your functions were polymorphic, which is another reason I advise people teaching to keep things concrete, at least at first.

Function Application

- “Implication is not associative, although the convention is that it binds “to the right,” so that $a \rightarrow b \rightarrow c$ is read as $a \rightarrow (b \rightarrow c)$. Except for type theorists and Haskell programmers, few people ever remember this, so it is usually safest to put in the parentheses.” —James Aspnes, “Notes on Discrete Mathematics”

New learners find it hard that function application is right-associative. Avoid deep chains of composition and use parentheses liberally.

```
data A = A Int  
a :: A -> Int  
a (A a) = a
```

Namespaces are confusing, and they're made more confusing by the fact that lots of the sample code you find online uses the same or very similar bindings in different contexts. This code for example, has the uppercase A used in two completely different namespaces, or contexts, and also the lowercase a used in two different contexts. This is of course a very synthetic example, but it's not that far from code I've seen in the wild.

```
data AType = AItem Int
a :: AType -> Int
a (AItem count) = count
```

It really isn't going to hurt you to give things slightly longer names - even if those names are abstract or meaningless, you're doing two things - you're giving things different names when they're used in different contexts, and you're giving people's eyes longer, more visible handles to focus on. Maybe it's just me, but I find the latter infinitely easier to parse.

```
thing = something = other
```

The weirdest mistake I saw the kids making when writing Haskell code was this: `x = y = z`. I don't have a good explanation for what it is about the language that resulted in this pattern, but I didn't see it even one time when working in Lua, or JavaScript, or Python, but in Haskell every kid tried to do this at least once. My best guess is that something about the idea of working with what amounts to nested expressions, rather than statements, makes this idea attractive.

Gatekeeping

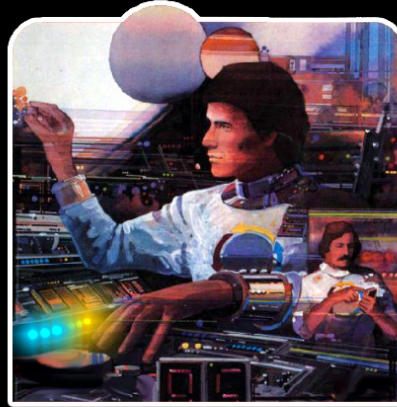
My experience introducing kids to programming has taught me that it takes a lot to ruin a mind, but it doesn't take a lot for someone to decide they aren't welcome. I think it's very easy to for us as Haskell enthusiasts to gatekeep people out of the language, both in conscious and unconscious ways.

Unconscious Bias

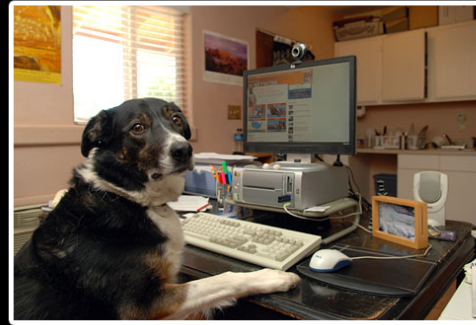
Be Open

Let's talk about unconscious ways first.

THE TWO STATES OF EVERY PROGRAMMER



I AM A GOD.

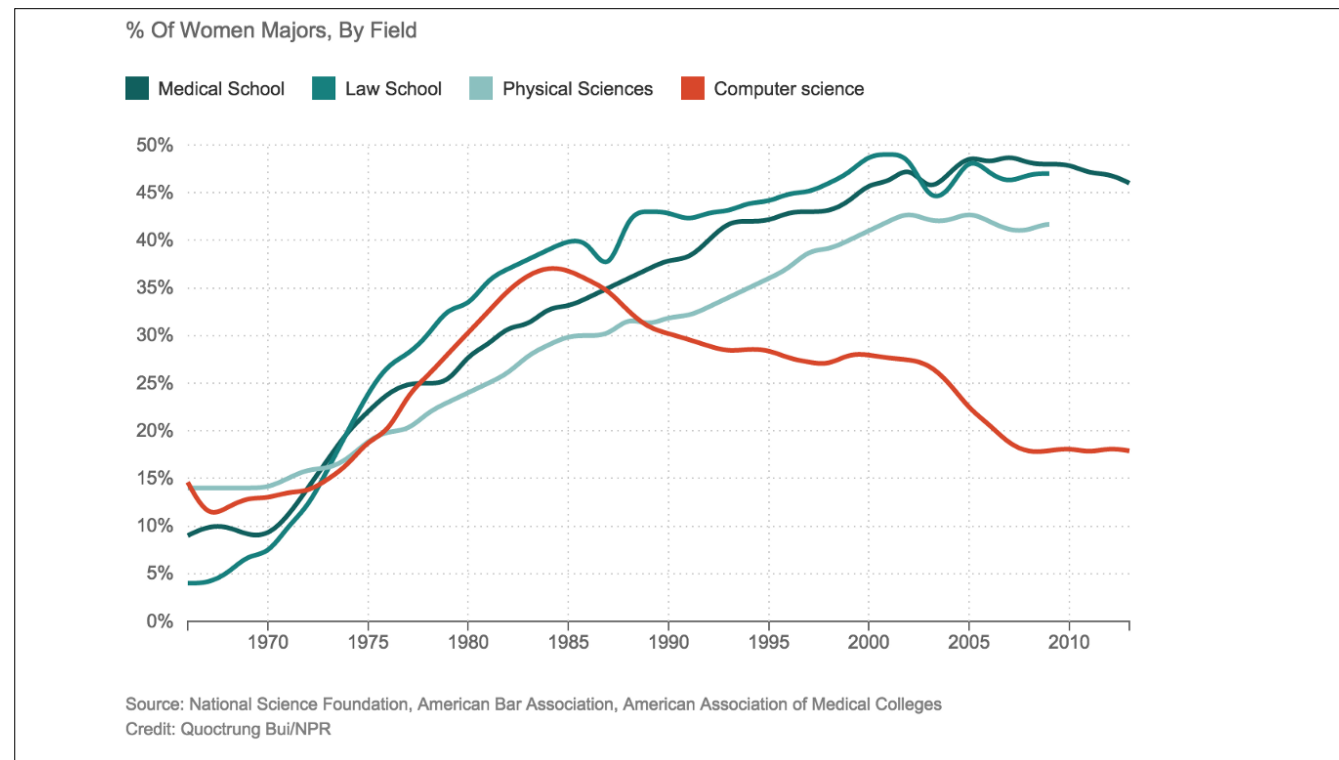


**I HAVE NO IDEA
WHAT I'M DOING.**

I want to come back to the image of our futuristic Atari programmers that we saw earlier, the image that I often used to introduce my classes.

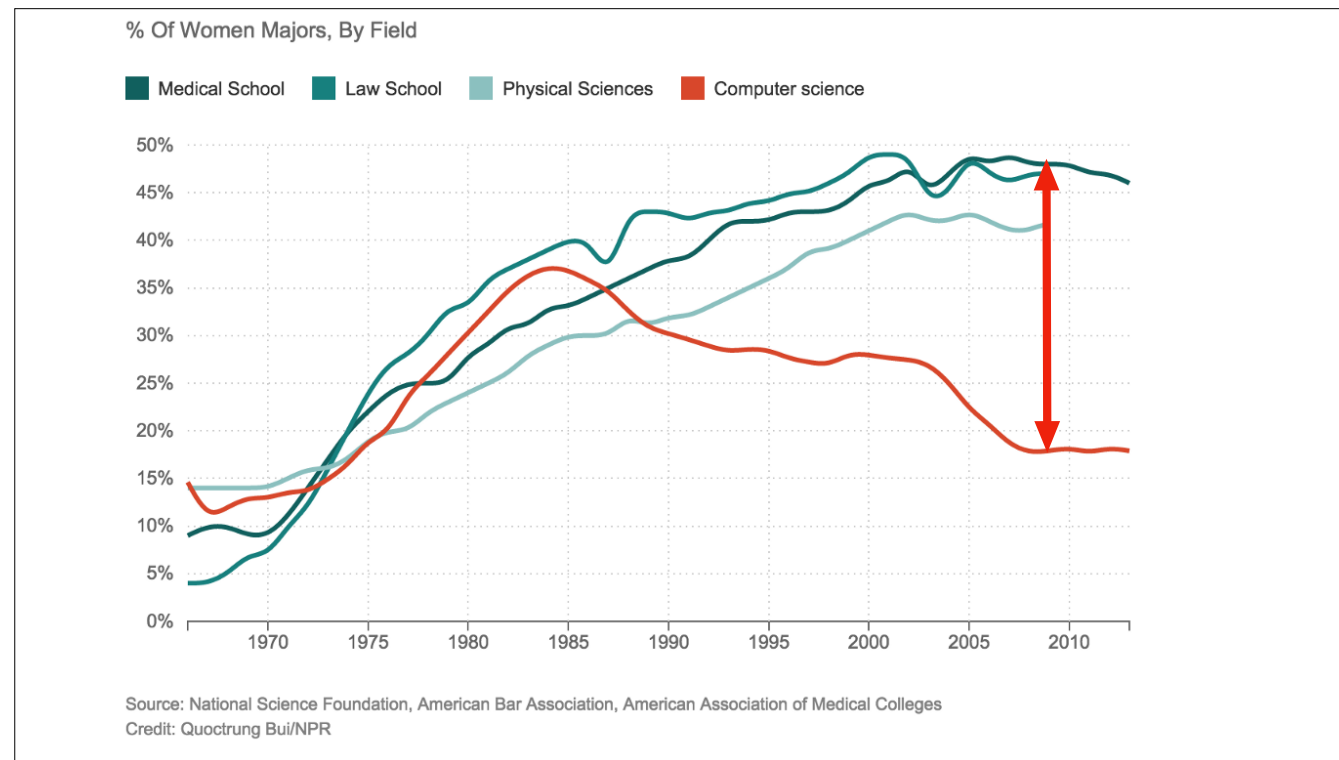


I don't want to make too much of one piece of art, or claim that this was a universal image of programming in the 1980s, but let's think about it anyway. What can we say about these dudes? Well, they're dudes. They're white. They're even dressed in white. One has a magnificent late 1970s mustache. Both are probably operating a spaceship or a time machine, they're using both hands, they're engaged in deep concentration. At least to me, they look almost like priests of technology. Of course this is just one image, made to sell a game version of a programming language for a device that didn't even have a keyboard, but I think it's fair to say there's a lot of unconscious bias reflected in the image not just about who programmers are, but also about what programming is.



Which brings me to this chart. This compares the percentage of women in a given major, by field, over time, from 1970 to roughly today. What we see is that in medicine, in law, in the physical sciences, participation of women in those majors has increased steadily since the 1970s. With Computer Science, we see that participation increased right up until about 1983, when it suddenly took a nosedive.

Women aren't the only underrepresented group in tech, but I'm talking about them as an example because I don't want to let this problem go unspoken. It's important that we acknowledge it as a problem, and that we recognize that this isn't just some weird coincidence where computer science is unlike any other field.



So part of why I'm calling this out is because it's ethically the right thing to do, but there's also a self-interested case to be made here. I think it's really easy to sit back and sort of tell yourself that this is just a problem for the underrepresented groups in tech. But if you're, like me, a privileged person, you should realize you're also harmed by this. Specifically, you're harmed by this gap. [BUILD IN ARROW]. The most important aspect of becoming a better programmer, for me, has been getting to sit next to, code with, and learn from people who were smarter than me. When I look at this gap, what I see is a huge number of women who *would* have been great developers, who *I could have learned from*, who were discouraged from entering the field. What structural bias has taken from women was much more than what was taken from men, or from me. But I do think it is to men's detriment as well. So, men, if you need a selfish motivation for doing the right thing, here it is. And of course, there are other charts to be drawn for other underrepresented groups as well.

So what do we do?

- Nurture **motivation** to reduce unconscious bias
- Build **awareness** of bias without shaming or blaming
- Reduce anxiety of outgroup interactions through **increased contact**

"Unconscious Bias in the Classroom: Evidence and Opportunities"
<http://services.google.com/fh/files/misc/unconscious-bias-in-the-classroom-report.pdf>

Google's 2017 report "Unconscious Bias in the Classroom: Evidence and Opportunities" <http://services.google.com/fh/files/misc/unconscious-bias-in-the-classroom-report.pdf> outlines some of the research around this topic. They discuss both student- and teacher-facing interventions. I wanted to highlight some of the teacher-facing interventions here:

Conscious Bias

Be Kind

Laura

Let me tell you about a young woman in one of my classes, for the sake of her privacy, I am going to refer to her as “Laura.” This was one of the Lua classes. I was giving a high-level introduction to programming and at various times I’d mention several programming languages: at some point I said “you might write this game in a programming language like Python, C, or JavaScript” and Laura, very quietly, muttered under her breath “or HTML.”

“Or HTML.”

- *Laura*

I didn't really pause, but just kept going. At some point later in the day, I did this sort of list-with-examples again, “languages like Perl, Kotlin, or Go” and Laura said, louder this time “OR HTML.” And I paused and she looked at me and said “Or HTML.

“OR HTML.”

- Laura

HTML is programming.”

I acknowledged it at the time but in my heart of hearts part of me thought “shyeah right sure it is.”



This is CrossCode, a game written in HTML5, Canvas2D, and impact.js. So this is me acknowledging that Laura was right, and I was wrong, and I was wrong in a way that PL nerds are often wrong. Those of you who have been around for a while probably remember a time when JavaScript was described as not being a real language. In the 90s there were people making the same arguments about scripting languages like Perl or Python generally as compared to C-like languages.

Bravery

The thing I want to point out here is that Laura was brave to take that stand, and I further want to point out that most young people won't be. The biases you bring to a classroom are generally not going to be challenged, so we have a responsibility to be aware of which biases we're bringing. Without even thinking about it, I sent her a message that HTML isn't real programming. That's not good enough.

BAD TAKE THEATER PRESENTS:

Knowing A Specific Programming Language Means You Are Smarter

So the conscious bias I'm pushing back against here is the idea that knowing a specific programming language means you are smart.

BAD TAKE THEATER PRESENTS:

Knowing Functional Programming Means You Are Smarter

And of course, this is a bias that lots of people have about Haskell.

BAD TAKE THEATER PRESENTS:

Knowing Haskell Means You Are Smarter Because It's So Hard

Spinning this thread out explicitly: Haskell is hard, to learn hard things you have to be smart, therefore I'm probably smart.

MOST PEOPLE DON'T THINK THIS WAY. PLEASE DON'T THINK THIS WAY

State of Haskell 2018 Survey

“[Haskell’s] reputation as a powerful tool for
superbrains can be a liability.”

“Most of the people think it is **black
magic** and stay away from it.”

“Reputed to be **hard to learn**.”

I want to be clear here that I’m not making a claim about whether Haskell is actually easy or hard, but about the perception of it in the world. Here are some representative comments from the 2018 State of Haskell survey. That survey is worth a read, especially the “suggested actions”, which include improved learning paths for developers switching to Haskell, and improvements to community moderation to ensure that newbies are treated in a welcoming style.

**“If you're not regularly using zygo-histomorphic
prepromorphisms then I'm afraid you can't
really be considered an expert.”**

**“To be fair, you have to have a very high IQ
to understand Haskell.”**

I'm a big believer in “if you want to know a community's reputation, look at what they make jokes about.” People on the Haskell Reddit make jokes about Haskell being inscrutably complex probably every day. Does that happen in the Python community?

My message here is not “Don't make jokes.” It's just that the jokes tell you a lot about the assumptions a community makes about itself - and those assumptions are going to be visible to those outside of the community.

Advice

Be Concrete First

Be Focused

Be Open

Be Kind

So summing up my advice:

Be concrete. I mean this both in terms of choosing a real, tangible goal with clear victory conditions and in terms of not pushing unnecessary abstraction.

Be focused. Decide what concept you want the class to learn and focus on clearing other obstacles out of the way before you step into the classroom.

Be open. Think about the biases you're bringing to the experience, and stay motivated to overcome them.

Be kind. Nobody wants to look bad in front of their peers. Own mistakes you made, and when a student makes a mistake, assure them that all developers make those mistakes too. Every mistake is an opportunity to learn. Someone who hasn't learned a particular thing yet is someone with an opportunity to learn.

Can Kids Learn Haskell?

- Absolutely!
- But the community needs to improve its onboarding experience.
- But Haskell's intimidating reputation is surely scaring people away.

No one Is Ruined

The Kids Are Alright

Acknowledgements

- The kids in the classes.
- Dr. D, for advice and assistance.
- Teachers everywhere.
- Lambdale organizers!

