Learning to code
why we fail, how we flourish

Andrew J. Ko, Ph.D.
Code & Cognition Lab
The Information School

UNIVERSITY of WASHINGTON
Me

• Professor for the last ~10 years at **UW Seattle**

• Ph.D. from **Carnegie Mellon**’s HCI Institute

• Background in **CS, Psychology, and Design**
Code is the most powerful, least usable interface we’ve invented

- Everyone that wants to code should be able to
- But there are immense learning barriers
- I spent the first decade trying to lower these barriers by creating more usable interactive developer tools

read    write    test    verify    debug
Skills > tools

• I spent 3 years as CTO managing ~8 developers at AnswerDash

• What I saw:
  • Tools only *amplify* skills
  • Skills come from learning
  • Learning comes from teaching
  • I spent most of my time teaching
Millions want to learn to code
Millions want to learn to code

$1.3 billion
Are people learning?

we have (some) evidence
77% of Code.org's 500 million K-12 learners complete 0-2 puzzles.
Record enrollment in AP CS, but most don’t take exam, and 60% who do, fail it (especially underrepresented minorities) College Board
After a year of intro courses, most undergrads can’t accurately predict the outcome of simple programs, or solve simple programming problems. 

McCracken et al. 2001, Lister et al. 2004 et al. 2013, Seppälä et al. 2015
In 2017, 23,000 adults in 95 U.S. coding bootcamps; 24 report dropout rates of 10-50% CourseReport.com
62% of employers view applicants to entry-level developer positions as lacking basic programming knowledge

Career Advisory Board Survey 2016
So I did some reading.

- Read seminal literature in learning science and education e.g. *How people learn: Brain, mind, experience, and school*
- Read 30 years of computing education research:
  - *ACM International Computing Education Research Conference (ICER)*
  - *ACM Transactions on Computing Education (TOCE)*
  - *SIGCSE Technical Symposium (SIGCSE)*
Why people fail to learn to code

1. People find computing boring, solitary, unwelcoming
2. People struggle to learn their first programming language
3. People struggle to solve programming problems
4. Teachers struggle to teach these things
5. Teachers blame learners for failure
6. People lose confidence and quit
My goal

1. People find computing boring, solitary, unwelcoming
2. People struggle to learn their first programming language
3. People struggle to solve programming problems

• Why are these hard?
• What are effective, equitable, and scalable ways for people to learn these skills?
This talk

• Why learning to program is hard (3 studies)
• Making programs easier to read (1 theory, 3 ideas)
• Making programs easier to write (1 theory, 1 idea)
Why is learning to program difficult?
Study 1 – 70 high school teens
High school

• Many teens lacked feedback or support about their learning from teachers and family:

  • *He do not spent much time with me to be able to understand my problem in the class or unable to help me on it… throughout the AP class I would cried myself to sleep in silent without letting my older brother know my struggle... (M, Asian, 17)*
High school

• Some teens had *informal computing mentors* who provided encouraging instruction and feedback.
  
  • Associated with stronger interest in learning to code, independent of gender, socioeconomic status.

• Teens sought teachers and mentors who:
  
  • Would not judge them for their failures
  
  • Would inspire them to learn
  
  • Had the expertise to guide them


Study 2 – 26 Bootcamps attendees
Bootcamps

- Some bootcamps were inclusive and encouraging, but many offered no instruction or feedback:
  - So they’re trying to get you into this mentality of you have to read all the documentation. They sit back in the background [to let students read the documentation], and what annoys me is that I’ve paid a lot of money so that I could have somebody there to teach it to me.
Bootcamps

• Many bootcamps offered an unwelcoming culture for learners without prior knowledge:
  • *It was divided, the class. Those with experience, I think, they were looking down at [those of us without experience] because maybe there were certain things we were supposed to know and we didn’t.*

Study 3 – 30 Coding Tutorials

Learn to code interactively, for free.
Tutorials


• Four learning science principles
  1. Connect instruction to prior knowledge
  2. Organize declarative knowledge
  3. Offer personalized feedback on practice
  4. Foster self-regulation in problem solving

• Ada completed all 30 tutorials across 100+ hours, judging every lesson against these principles
Tutorials


- Most tutorials failed to meet all them:
  1. No connection to prior knowledge
  2. No organization of declarative knowledge about programming languages
  3. No personalized feedback on program correctness or errors
  4. No instruction on how to solve programming problems.
Why is learning to program difficult?

Few of these contexts actually teach programming. There are many opportunities to read and write code, but learners receive little feedback on whether they are reading or writing correctly.
Making programs easier to read

One theory, three ideas
Extant theories about why understanding programs is hard

• Wrong programming language
  • Static typing, syntax, and errors matter, but only a little (e.g., Stefik & Siebert 2013)

• Wrong IDE
  • Relative to text, drag and drop “blocks” editors reduce dropout, but don’t improve learning (Cooper et al. 2001)

• Wrong biology
  • No evidence of “geek gene” or bimodal grade distributions (Patitsas et al. 2016)
Extant theories about why understanding programs is hard

- Wrong programming language
  - Static typing, syntax, and errors matter, but only a little (e.g., Stefik & Weber, 2013)
- Wrong IDE
  - Relative to text, drag and drop “blocks” editors reduce dropout, but don’t improve learning (Cooper et al., 2001)
- Wrong pedagogy
  - No evidence of “geek gene” or bimodal grade distributions (Patitsas et al. 2016)
A new definition of PL knowledge

Knowing a PL means:

1. Being able to reliably and accurately predict an arbitrary program’s operational semantics without the aid of a runtime environment. (Reading a program and knowing what it will do).

2. Knowing how syntax maps onto operational semantics.

Note that I’m excluding knowledge of common design patterns, architectures, tools, norms, etc. This strictly concerns the ability to accurately read programs.
An example

Knowing a JavaScript *if-statement* means knowing:

1. Condition is evaluated

2. If it’s true, all of the statements between the first set of braces are executed, and everything between the else braces are skipped.

3. Otherwise, the statements in the first set are skipped, and the statements in the second set are executed.
Knowing a entire PL

• Knowing all of JavaScript means knowing all of the semantics for JavaScript’s entire grammar

• That’s about 90 non-terminals in the grammar, each with its own semantic nuances

• Most introductory programming courses never explain any of this:
  • In UW’s CS1 course, the 1st homework is to write a Java program with function declarations, function calls, string literals. None of the lectures explain any of this, and, not surprisingly, most students fail.
Four major pedagogies

Learn formal semantics

Explain via natural language

Write code

Step through execution
Four major pedagogies

Learn formal semantics

No syntax mapping; Requires learning a notation

Explain via natural language

Write code

Step through execution
Four major pedagogies

Learn formal semantics

Explain via natural language

No syntax mapping; Requires learning a notation

Ambiguous, weak syntax mapping

Write code

Step through execution
Four major pedagogies

Learn formal semantics

Explain via natural language

No syntax mapping; Requires learning a notation

Ambiguous, weak syntax mapping

Write code

Step through execution

Requires learner to infer semantics
Four major pedagogies

Learn formal semantics

- No syntax mapping; Requires learning a notation
  - Requires learner to infer semantics

Explain via natural language

- Ambiguous, weak syntax mapping
  - Masks semantics within a line of code

Write code

Step through execution
How should we teach syntax semantics?

Teach a “notional machine” du Boulay 1989

1. Show each step of semantics and their explicit effects on the program counter, call stack, and memory

2. Map semantics to concrete syntax, creating an association between syntax and its side effects
Three ideas

**Gidget**

Learners *discover* semantics through debugging

Mike Lee, Ph.D.

**PLTutor**

Tutor explicitly *teaches* semantics

Greg Nelson

**Tracing Strategies**

Learner reminded to follow semantics

Benji Xie
• Frame coding as a **collaboration** between a person and computer

• Give learner a sequence of **debugging puzzles**

• Guide learners’ attention to **contextualized instruction on syntax and semantics** of it’s Pythonic language
Level 1. Let’s get to the puppy!

Make sure you always read the goals of the level on the bottom-left panel first, and then try running the code at least once using the buttons below the goals to see how the starting code works.
Gidget helpgidget.org  Lee et al. 2014, VL/HCC

Level 1. Let's get to the puppy!

Make sure you always read the goals of the level on the bottom-left panel first, and then try running the code at least once using the buttons below the goals to see how the starting code works.
37 levels teaching 12 semantics, including formative assessments to verify understanding
37 levels teaching 12 semantics, including formative assessments to verify understanding
Gidget \texttt{helpgidget.org} Lee et al. 2014, VL/HCC

level 20 teaches function calls

Level 20. Press the button, open the gate!

\texttt{world}
level 20 teaches function calls
All that grabbing & dropping made me remember a way to save time writing my programs though... \textit{functions}!
All that grabbing & dropping made me remember a way to save time writing my programs though... functions!
All that grabbing & dropping made me remember a way to save time writing my programs though... functions!
Gidget explains syntax and semantics
The **green line** and Gidget’s speech bubble maps syntax to semantics.
The **green line** and Gidget’s speech bubble maps syntax to semantics.
Level 20. Press the button, open the gate!

```
goto /button/
say "Let's click the button to see its function"
/goto:/openFence()  
function getPiglet() -？
goto /piglet/  
set /piglet/:nickname to "wilbur"
set /piglet/:age to 3
grab /piglet/  
getBird() -？
getThePiggy() -？
goto /basket/  
```

Let's click the button to see its function name. It has to be exact!

```
energy 97
grabbed []
image "default"
labeled true
layer 1
name "gidget"
position [6, 5]
rotation 0
scale 1
transparency 1

code[]
```
Gidget points out function definitions
Let's click the button to see its function name. It has to be exact!
Gidget explains name resolution semantics
Level 20. Press the button, open the gate!

goto /button/
say "Let's click the button to see its function /
/button/openFence()
function getPiglet() -?
  goto /piglet/
  set /piglet/:nickname to "wilbur"
  set /piglet/:age to 3
  grab /piglet/
getBird() -?
getThePiggy() -?
goto /basket/

Whoops! I don't know of any function called openFence. Did we define it correctly? I have to stop executing this program because of this error.

One step  One line  To end  Stop!
Level 20. Press the button, open the gate!

```
goto /button/
say "Let's click the button to see its function /
/button/:openFence()"
function getPiglet() - ?
goto /piglet/
set /piglet/:nickname to "wilbur"
set /piglet/:age to 3
grab /piglet/
getBird() - ?
getThePiggy() - ?
goto /basket/
```

Whoops! I don't know of any function called openFence. Did we define it correctly? I have to stop executing this program because of this error.

```
energy 94
grabbed []
image "default"
labeled true
layer 1
name "gidget"
position [6, 5]
rotation 0
scale 1
transparency 1
```
• Four online controlled experiments with over 1,000 adult learners:

  • Learning is **2x as fast** as Codecademy tutorial, **2x as much** as open-ended creative exploration Lee & Ko 2015

  • **Assessment levels** significantly increase learning efficiency Lee et al. 2013

  • Giving compiler a face and using personal pronouns (I, you, we) draws learner’s attention to semantics, **doubling learning efficiency** Lee & Ko 2011

  • Changes attitudes about difficulty of learning to code from negative to positive in **20 minutes** Charters et al. 2014
• Four online controlled experiments with over 1,000 adult learners:
  
  • Learning is **2x as fast** as Codecademy tutorial, **2x as much** as open-ended creative exploration. Lee & Ko 2015
  
  • **Assessment levels** significantly increase learning efficiency. Lee et al. 2013
  
  • Giving compiler a face and using personal pronouns (I, you, we) draws learner’s attention to semantics, **doubling learning efficiency**. Lee & Ko 2011
  
  • Changes attitudes about difficulty of learning to code from negative to positive in **20 minutes**. Charters et al. 2014
20,000+ have played via word of mouth, including Chicago Public Schools, many retirees (apparently including my mom)

Directly impacted the design of code.org's CodeStudio and Apple's Swift Playgrounds, used by 10+ million learners.
20,000+ have played via word of mouth, including Chicago Public Schools, many retirees (apparently including my mom)

Directly impacted the design of code.org's CodeStudio and Apple's Swift Playgrounds, used by 10+ million learners.
Three ideas

**Gidget**
Learners discover semantics through debugging

**PLTutor**
Tutor explicitly teaches semantics

**Tracing Strategies**
Learner reminded to follow semantics
Three ideas

**Gidget**
Learners *discover* semantics through debugging

**PLTutor**
Tutor explicitly *teaches* semantics

**Tracing Strategies**
Learner reminded to follow semantics
• Convert operational semantics into an interactive textbook to be read before learning to write programs

• Covers the entire JavaScript semantics in about 3 hours of practice

• Learner should be able to accurately predict the behavior of any JavaScript program
• Convert operational semantics into an **interactive textbook** to be read before learning to write programs

• Covers the entire *JavaScript* semantics in about **3 hours** of practice

• Learner should be able to accurately predict the behavior of *any* JavaScript program
Each chapter covers a set of semantics through a series of programs.
PLTutor

Lesson explains purpose of semantics

Program links syntax to semantics

State explains semantics

**PLTutor**


**Lesson** explains purpose of semantics

**Program** links syntax to semantics

**State** explains semantics
**Lesson**

**If statements**

Now it’s time to use what you learned about boolean values and operators!

Before this, the computer would execute all instructions created from the code.

If statements allow computers to do some set of instructions if a **condition** is true or not.

They look like this:

```
if ( condition )
{
    code goes inside the { }'s
}
```

Let’s step through one to see how it works.

**Program**

```javascript
var x = 0;
if ( 10 > 0 ){
    /* the computer will execute inside here because the condition is true and that leaves true on the stack */
    we put x = 100000; here just so you can see some code execute inside the if */
    x = 1000000;
}

x;
var x = 0;
if ( 0 > 10 ){
    /* the computer will NOT execute inside here because the condition is false and that leaves false on the stack */
    x = 1000000;
}

x;
var x = 0;
if ( 10 != 0 )
```

purpose of semantics
If statements

Now it's time to use what you learned about boolean values and operators!

Before this, the computer would execute all instructions created from the code.

If statements allow computers to do some set of instructions if a condition is true or not.

They look like this

```javascript
if ( condition )
{
    code goes inside the { }'s
}
```

Let's step through one to see how it works.

```javascript
var x = 0;
if ( 10 > 0 ){
    /* the computer will execute inside here because the condition is true and that leaves true on the stack */
    we put x = 100000; here just so you can see some code execute inside the if */
    x = 1000000;
}

x;
var x = 0;
if ( 0 > 10 ){
    /* the computer will NOT execute inside here because the condition is false and that leaves false on the stack */
    x = 1000000;
}
x;
var x = 0;
if ( 10 != 0 )
{ /* the computer will execute inside here because the condition is true and that leaves true on the stack */
    x = 1000000;
}

x;
```
Next moves through both program execution trace and instruction.
Next moves through both program execution trace and instruction.
State teaches semantics in a runtime context, Lesson generalizes back to purpose
PLTutor

State teaches semantics in a runtime context, Lesson generalizes back to purpose
Reverse execution allows learner to review instruction they didn’t understand.
Reverse execution allows learner to review instruction they didn’t understand.
PLTutor

Lesson explains the side effect of the semantics before proceeding to further examples.
Lesson explains the side effect of the semantics before proceeding to further examples.
Assessments embedded in execution trace require learners to predict side effects of semantics.
Assessments embedded in execution trace require learners to predict side effects of semantics.
• Required complete re-architecting of language stack
• Must preserve provenance of all compiler and runtime state to facilitate reversibility and embedded explanations
• Redesigned grammar to facilitate granular explanations
• Required complete re-architecting of language stack

• Must preserve provenance of all compiler and runtime state to facilitate reversibility and embedded explanations

• Redesigned grammar to facilitate granular explanations
Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.

- Compared PLTutor to Codecademy in a 4-hour controlled experiment with 40 CS1 students
- Measured learning with SCS1, a validated assessment of CS1 learning
- PLTutor had 60% higher learning gains, learning gains predicted midterm scores
PLTutor

• Compared PLTutor to Codecademy in a 4-hour controlled experiment with 40 CS1 students

• Measured learning with SCS1, a validated assessment of CS1 learning

• PLTutor had 60% higher learning gains, learning gains predicted midterm scores
Three ideas

**Gidget**

Learners *discover* semantics through debugging

Mike Lee, Ph.D.

**PLTutor**

Tutor explicitly *teaches* semantics

Greg Nelson

**Tracing Strategies**

Learner reminded to follow semantics

Benji Xie
Three ideas

**Gidget**
Learners *discover* semantics through debugging

**PLTutor**
Tutor explicitly *teaches* semantics

**Tracing Strategies**
Learner *reminded to* follow semantics

Mike Lee, Ph.D.
Greg Nelson
Benji Xie
• When learners have brittle knowledge of semantics, they often **guess** how programs will execute

• An explicit strategy for reading programs should outperform guessing
STRATEGY: Understanding the Problem;
Run the Code (like a computer).

UNDERSTAND THE PROBLEM

1. Read question: Understand what you are being asked to do. At the end of the problem instructions, write a check mark: ✓

2. **Find where the program begins executing.** At the start of that line, draw an arrow: →

RUN THE CODE

3. Execute each line according to the rules of Java:
   a. From the syntax, determine the rule for each part of the line.
   b. Follow the rules.
   c. Update memory table(s).
   d. Find the code for the next part.
   e. Repeat until the program terminates.
At the bottom of the page, write the output produced by the following program, as it would appear on the console.

```java
public class OddMystery {
    public static void main(String[] args) {
        int x = 2;
        int y = 3;
        System.out.println(x + y + "!");
        compute(y, x);
        double val = compute(x, y + 1);
        System.out.println(val);
    }
    public static double compute(int x, int y) {
        int z = y;
        y = x;
        x = z;
        System.out.println("x" + y + z);
        return Math.pow(x, y);
    }
}
```

Method: main

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>3</td>
</tr>
<tr>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>y</td>
<td>16</td>
</tr>
</tbody>
</table>

The output would be: 5!
Strategy

In a controlled experiment with **15 minutes** of practice, 12 students who learned the strategy were more systematic than 12 who didn’t, resulting in:

- **15% higher performance** on problems in the lab
- **7% higher on midterm** that was mostly writing focused
- **No midterm failures** (compared to 25% failure in control)
Making programs easier to read

Requiring learners to *directly observe* operational semantics and map them to syntax can significantly increase learning outcomes.
Making programs easier to write

One theory, one idea
Program writing

• Little prior work theorizing about what program writing skills actually are
  • Most prior work compares to expert and novices, showing that novices are unsystematic, speculative, and ineffective
  • A few papers show that the more developers “self-regulate” their problem solving, the more productive they are
Self-regulation

• From educational psychology, refers to one’s ability to reflect on, critique, and control one’s thoughts and behaviors during problem solving:
  • Explicit planning skills
  • Explicit monitoring of one’s process
  • Explicit monitoring of one’s comprehension
  • Reflection on one’s cognition
  • Self-explanation of decisions

Great engineers are highly self-regulating

- Interviewed 59 senior developers at Microsoft and surveyed 1,926 about what makes a great software engineer:

- Top attributes included:
  - Resourceful
  - Persistent
  - Self-regulating

Dastyni’s theory of program writing

- Programming involves iteration through **6 key activities**:  
  - Interpreting problems  
  - Searching for similar problems  
  - Searching for solutions  
  - Evaluating solutions  
  - Implementing solutions  
  - Evaluating implementations

- Programming requires:  
  - A **knowledge repository** of problems and solutions (in memory or elsewhere)
  - **Self-regulation skills** to help a programmer:
    - Select strategies for completing activity
    - Deciding when a strategy is failing or successful
Self-regulation is related to success

- Observed think aloud of 37 novices in CS1 and CS2 writing solutions to several programming problems.
- Most novices engaged in self-regulation, but infrequently and superficially.
- Self-regulation related to fewer errors, but only for novices with sufficient prior knowledge to solve problems.

Can we teach it?

• Taught 48 high schoolers with no prior programming experience HTML, CSS, JavaScript and React for 1 week, then had them build personal web sites for 1 more week

• Treatment group received:
  • Learned Dastyni’s theory of program writing
  • Before receiving help, required to practice self-regulation, explaining which activity they were doing, what their strategy was, and whether it was working
Yes!

More productive

Higher programming self-efficacy

More self-defined work

No growth mindset erosion

Making programs easier to write

Teaching programming self-regulation promotes independence, increased productivity, and higher self-efficacy.
What’s next?
CS1 mastery

New NSF Cyberlearning

• Prior work shows increased learning, but not mastery, which requires **personalized content and feedback**

• Human tutors can provide this, but can’t scale it

• We’re building a tutor that provides **infinite personalized practice** by applying program synthesis and our theories of programming knowledge

• **Goal**: every student masters CS1 content in 10 hours
Strategies
New NSF SHF Medium

• Self-regulation is only useful with good strategies
• Defining 1) what programming strategies are, 2) how to describe them, 3) which ones exist, 4) when they’re effective, 5) support for learning and executing them.

• **Goal**: A new science of programming strategies analogous to other disciplines’ “engineering handbooks,” which show how to solve problems in a discipline
Robust API learning

• New theory of API knowledge as **domain concepts**, **design templates**, and **API execution semantics**

• Techniques to automatically extract this knowledge from API implementations

• Building a tutor that generates on-demand API tutorials using this extracted knowledge.

• **Goal**: rapid, robust API learning at scale
Can’t do it alone…

• Many great faculty contributing to computing education research from PL, Software Engineering, and HCl. **Join us!**

• Our doctoral students need tenure-track positions to continue their work. **Hire them!**
Thanks!

Millions try to learn to code, but fail.

Explicit instruction and feedback on semantics is key.

Learning tech like Gidget and PLTutor are scalable and effective.

Pedagogies like tracing strategies and self-regulation prompting are effective and immediately adoptable.

Supported by NSF, Google, Microsoft, Adobe, the University of Washington.

Thanks to my wonderful doctoral and undergraduate students, and the hundreds of participants in our studies!