

Learning to code

why we fail, how we flourish

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



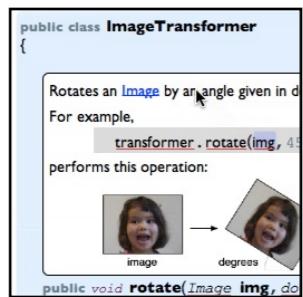
Me

- Professor for the last ~10 years at **UW Seattle**
- Ph.D. from **Carnegie Mellon's HCI Institute**
- Background in **CS, Psychology, and Design**

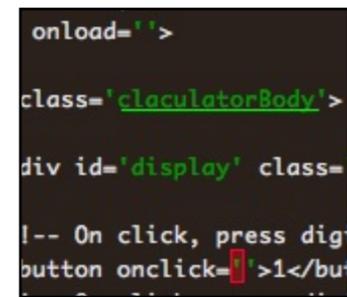


Code is the most powerful, least usable interface we've invented

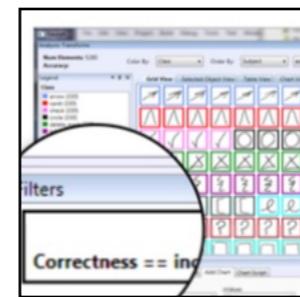
- Everyone that *wants* to code should be able to
- But there are immense learning barriers
- I spent the first decade trying to lower these barriers by creating more *usable interactive developer tools*



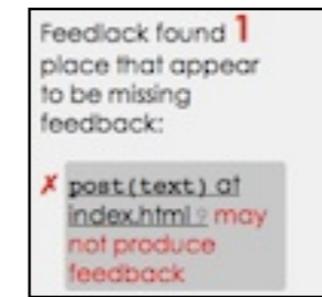
read



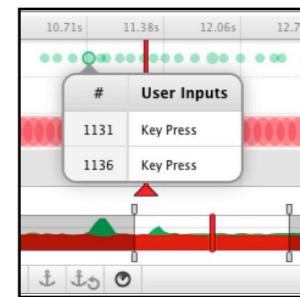
write



test



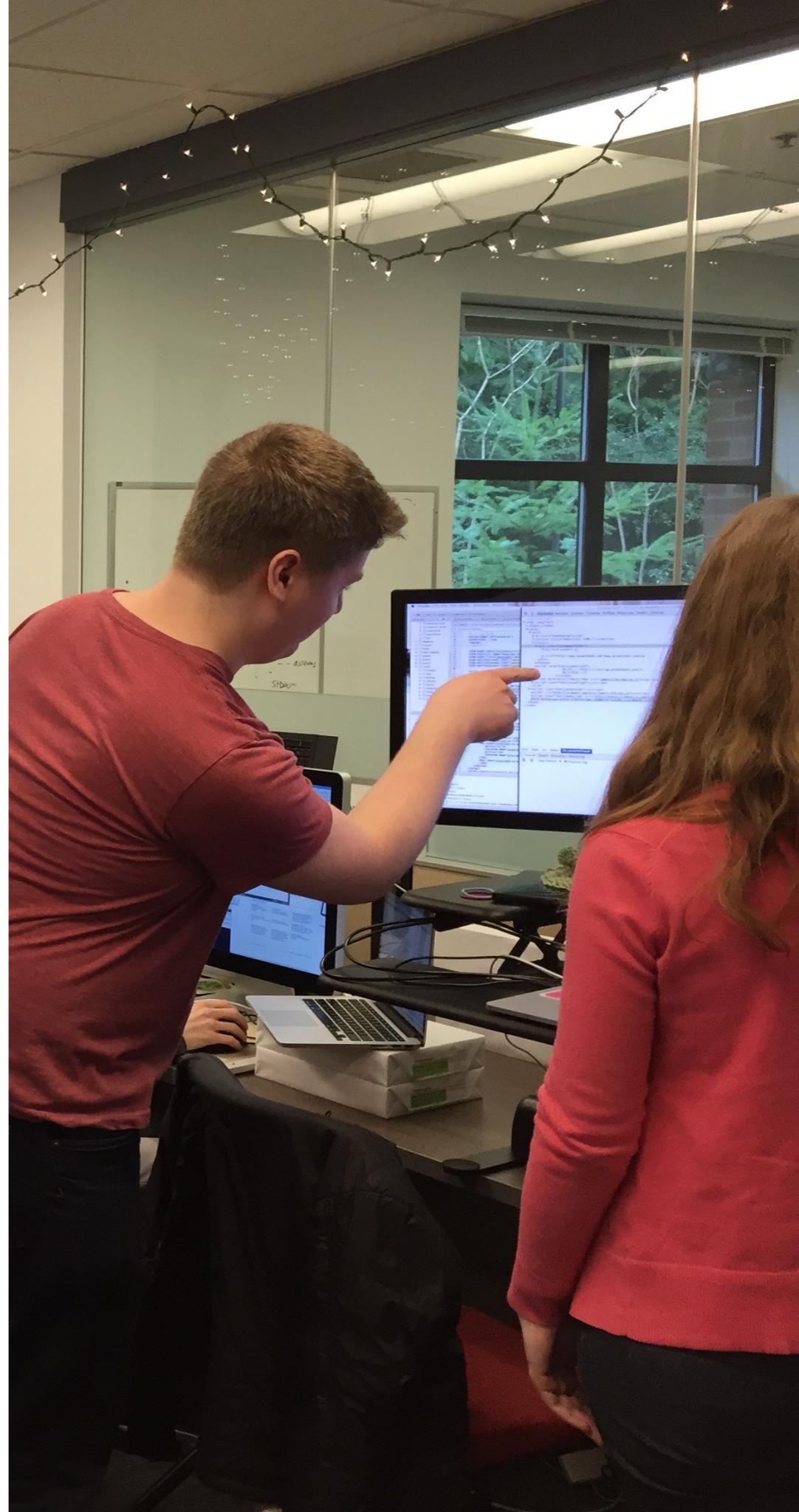
verify



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



Ada | DEVELOPERS ACADEMY

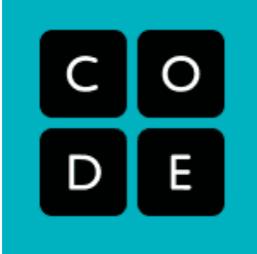
MEMBER
CsforALL
CONSORTIUM



csnyc

COMPUTING AT SCHOOL
EDUCATE • ENGAGE • ENCOURAGE

Part of BCS, The Chartered Institute for IT



SCRATCH

coursera

KHANACADEMY

CSEd

WEEK

amazon



Microsoft

facebook

Google

accenture

salesforce

\$1.3 billion

Are people learning?

we have (some) evidence

77% of Code.org's 500 million K-12 learners complete 0-2 puzzles

code.org

The screenshot shows a Minecraft-themed puzzle on the Code.org platform. At the top, there's a teal header with the Code.org logo, the title "Minecraft: Hero's Journey", a progress bar showing 1 of 10 steps completed, the message "I've finished my Hour of Code", a "Sign in" button, and a menu icon.

The main area features a 3D Minecraft scene where a character is inside a wooden house. A sheep is outside near a pressure plate. A chest is on the ground nearby. A lightbulb icon indicates a hint is available.

A large text box contains the following instructions:

The door is locked, but the Agent is here to help!
Snap a `move forward` block to the bottom of the `when run` block in the workspace to get the Agent to the pressure plate, then press "Run" and use the arrow keys to move out of the house to collect the chest.

Below the text box, the workspace shows two blocks: a blue `move forward` block and an orange `when run` block. There are also three other empty blocks in the workspace.

At the bottom, there are several control buttons: a yellow "Run" button, four small gray buttons for movement (left, right, up, down), and a "Need help?" link.

Record enrollment in AP CS, but most don't take exam, and 60% who do, fail it (especially underrepresented minorities) College Board



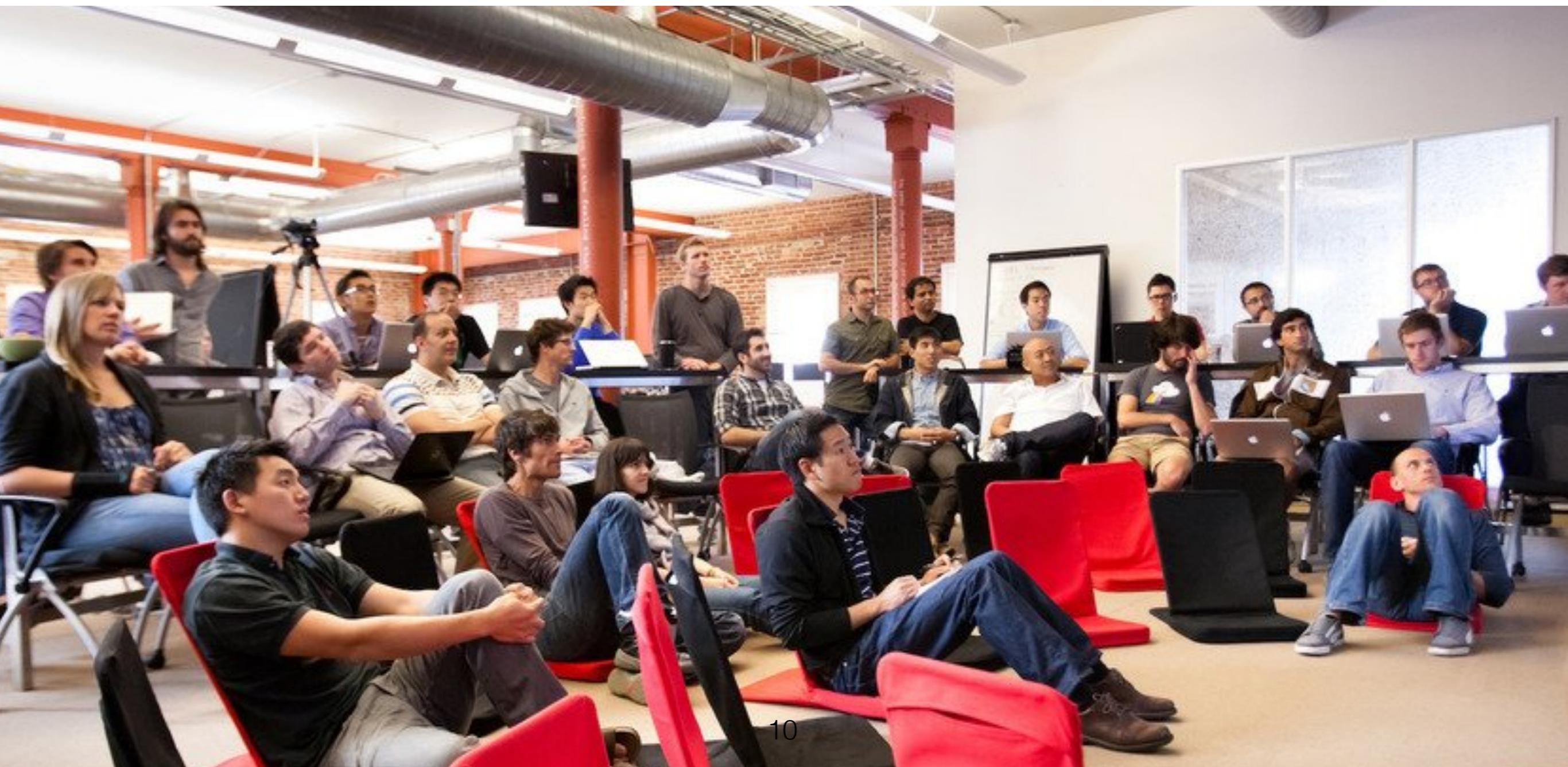
After a year of intro courses, most undergrads can't accurately predict the outcome of simple programs, or solve simple programming problems

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

Ko, A.J. and Davis, K. (2017). Computing Mentorship in a Software Boomtown: Relationships to Adolescent Interest and Beliefs. ACM ICER.

Ko, A.J. et al. (2018). Informal Mentoring of Adolescents about Computing: Relationships, Roles, Qualities, and Impact. ACM SIGCSE.

- 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

Ko, A.J. and Davis, K. (2017). Computing Mentorship in a Software Boomtown: Relationships to Adolescent Interest and Beliefs. ACM ICER.

Ko, A.J. et al. (2018). Informal Mentoring of Adolescents about Computing: Relationships, Roles, Qualities, and Impact. ACM SIGCSE.

- 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

Thayer, K. and Ko, A.J. (2017). Barriers Faced by Coding Bootcamp Students. ACM ICER.

- 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

Thayer, K. and Ko, A.J. (2017). Barriers Faced by Coding Bootcamp Students. ACM ICER.

- 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

codecademy

Catalog

Learn to code interactively, for free.



Sign Up Log In

Choose a username

Your email address

Choose a password

I'm not a robot

 reCAPTCHA
Powered by Google

Tutorials

Kim, A. and Ko, A.J. (2017). A Pedagogical Analysis of Online Coding Tutorials. ACM SIGCSE.

- 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

Kim, A. and Ko, A.J. (2017). A Pedagogical Analysis of Online Coding Tutorials. ACM SIGCSE.

- 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, editors matter, but only a little (e.g., Stefik & Weber 2013)
- Wrong IDE
 - Relative to text, drag and drop “blocks” editors reduce dropdown, but don’t improve learning (Cooper et al. 2001)
- Wrong biology
 - No evidence of “geek gene” or bimodal grade distributions (Patitsas et al. 2016)

Wrong
pedagogy

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.
-
- ```
if(dataIsValid && serviceIsOnline) {
 submit();
}
else {
 alert("Bad error message!");
}
```

# Knowing a entire PL

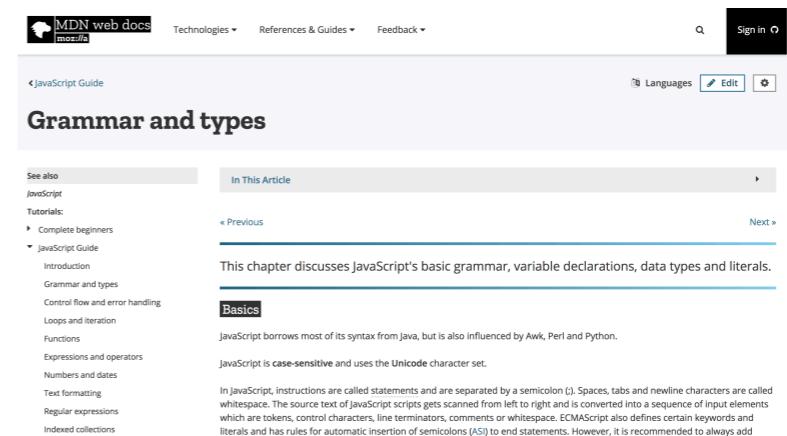
- Knowing *all* of JavaScript means knowing *all* of the semantics for JavaScript's entire grammar
- That's about **90 non-terminals** in the grammar, each with its own semantic nuances
- Most introductory programming courses never explain any of this:
  - In UW's CS1 course, the 1st homework is to write 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

$$\begin{array}{c} \frac{}{\mathbf{R} \vdash e_{empty} \Rightarrow \mathbf{R}} \text{ (EMPTY)} \quad \frac{e_2 \in \mathbf{R}}{\mathbf{R} \vdash e_1 = e_2 \Rightarrow \mathbf{R} \cup \{e_1\}} \text{ (ASSIGN)} \\ \frac{}{\mathbf{R} \vdash e_1.field* = e_2 \Rightarrow \mathbf{R} \cup \{e_1\}} \text{ (ASSIGN-FIELD)} \quad \frac{e_2 \in \mathbf{R}}{\mathbf{R} \vdash e_1[*] = e_2 \Rightarrow \mathbf{R} \cup \{e_1\}} \text{ (ASSIGN-ARRAY)} \\ \\ \frac{v \in \mathbf{R}; \text{pointer\_type.p}(v); \mathbf{R} \vdash e \Rightarrow \mathbf{R}'}{v \oplus e \Rightarrow \mathbf{R}' \cup \{v \oplus e\}} \text{ (BINOP1)} \\ \\ \frac{v \in \mathbf{R}'; \text{pointer\_type.p}(v); \mathbf{R} \vdash e \Rightarrow \mathbf{R}'}{v \oplus e \Rightarrow \mathbf{R}' \cup \{v \oplus e\}} \text{ (BINOP2)} \quad \frac{\mathbf{R} \vdash e_1 \Rightarrow \mathbf{R}'; \mathbf{R}' \vdash e_2 \Rightarrow \mathbf{R}''}{\mathbf{R} \vdash e_1; e_2 \Rightarrow \mathbf{R}''} \text{ (SEQ)} \\ \\ \frac{\mathbf{R} \vdash e_1 \Rightarrow \mathbf{R}_1; \mathbf{R}_1 \vdash e_2 \Rightarrow \mathbf{R}_2; \mathbf{R}_1 \vdash e_3 \Rightarrow \mathbf{R}_3; \mathbf{R}_2 \cup \mathbf{R}_3 \vdash e_4 \Rightarrow \mathbf{R}'}{\mathbf{R} \vdash \text{if } (e_1) \text{ then } \{e_2\} \text{ else } \{e_3\} e_4 \Rightarrow \mathbf{R}'} \text{ (IF)} \\ \\ \frac{\text{function\_type.p}(e) \quad \{e.args[i] \mid 0 < i < e.numargs \wedge \text{pointer\_type.p}(e.args[i])\} \cup \{v \mid \text{global\_p}(v)\} \vdash e.body \Rightarrow \mathbf{R}}{\{\} \vdash e \Rightarrow \mathbf{R}} \text{ (FUNCTION)} \end{array}$$

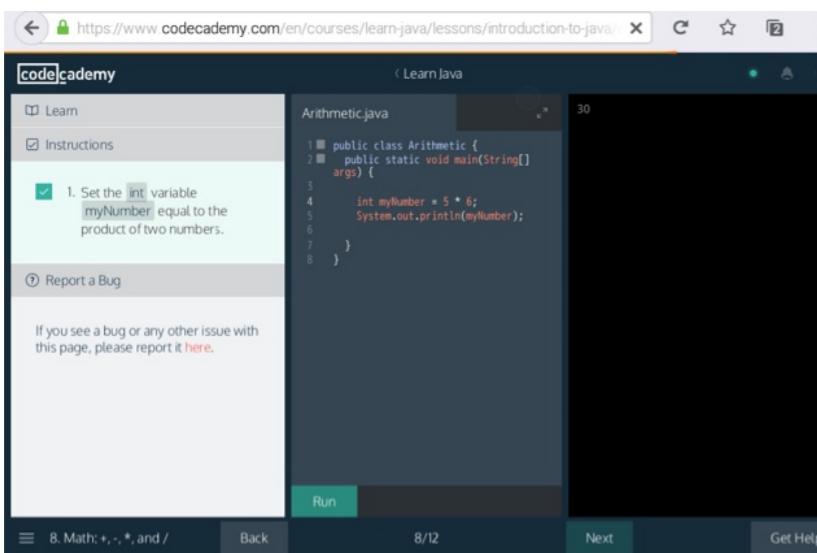
Explain via natural language



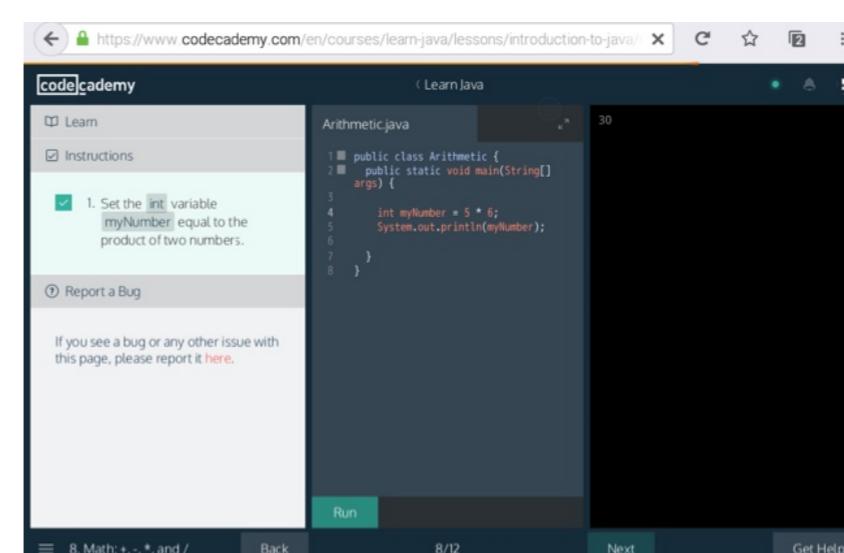
This chapter discusses JavaScript's basic grammar, variable declarations, data types and literals.

JavaScript borrows most of its syntax from Java, but is also influenced by Awk, Perl and Python. JavaScript is case-sensitive and uses the Unicode character set. In JavaScript, instructions are called statements and are separated by a semicolon (;). Spaces, tabs and newline characters are called whitespace. The source text of JavaScript scripts gets scanned from left to right and is converted into a sequence of input elements which are tokens, control characters, line terminators, comments or whitespace. ECMAScript also defines certain keywords and literals and has rules for automatic insertion of semicolons (ASI) to end statements. However, it is recommended to always add

Write code



Step through execution



# Four major pedagogies

Learn formal semantics

Explain via natural language

Formal semantics rules:

- $\frac{R \vdash e_{empty} \Rightarrow R}{e_2 \in R}$  (EMPTY)
- $\frac{R \vdash e_1 = e_2 \Rightarrow R \cup \{e_1\}}{e_2 \in R}$  (ASSIGN)
- $\frac{e_2 \in R}{e_2 \in R}$  (ASSIGN-FIELD)
- $\frac{e_2 \in R}{e_2 \in R}$  (ASSIGN-ARRAY)

No syntax mapping; Requires learning a notation

$$\frac{\{e.args[i] \mid 0 < i < e.numargs \wedge \text{pointer\_type\_p}(e.args[i])\} \cup \{v \mid \text{global\_p}(v)\} \vdash e.body \Rightarrow R}{\{\} \vdash e \Rightarrow R} \quad (\text{FUNCTION})$$

MDN web docs - JavaScript Guide - Grammar and types

This chapter discusses JavaScript's basic grammar, variable declarations, data types and literals.

See also: JavaScript Tutorials: Complete beginners, JavaScript Guide (Introduction, Grammar and types, Control flow and error handling, Loops and iteration, Functions, Expressions and operators, Numbers and dates, Text formatting, Regular expressions, Indexed collections)

In This Article: Previous, Next

Basics: JavaScript borrows most of its syntax from Java, but is also influenced by Awk, Perl and Python. JavaScript is case-sensitive and uses the Unicode character set. In JavaScript, instructions are called statements and are separated by a semicolon (;). Spaces, tabs and newline characters are called whitespace. The source text of JavaScript scripts gets scanned from left to right and is converted into a sequence of input elements which are tokens, control characters, line terminators, comments or whitespace. ECMAScript also defines certain keywords and literals and has rules for automatic insertion of semicolons (ASI) to end statements. However, it is recommended to always add

Write code

codecademy Learn Java

Arithmetic.java

```
1 public class Arithmetic {
2 public static void main(String[] args) {
3 int myNumber = 5 * 6;
4 System.out.println(myNumber);
5 }
6 }
```

Instructions: Set the `int` variable `myNumber` equal to the product of two numbers.

If you see a bug or any other issue with this page, please report it [here](#).

Run Back 8/12 Next Get Help

Step through execution

codecademy Learn Java

Arithmetic.java

```
1 public class Arithmetic {
2 public static void main(String[] args) {
3 int myNumber = 5 * 6;
4 System.out.println(myNumber);
5 }
6 }
```

Instructions: Set the `int` variable `myNumber` equal to the product of two numbers.

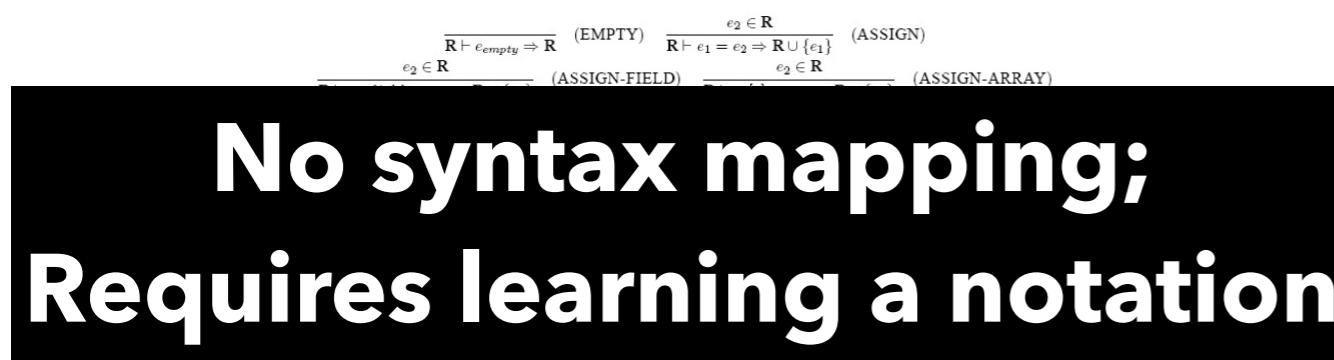
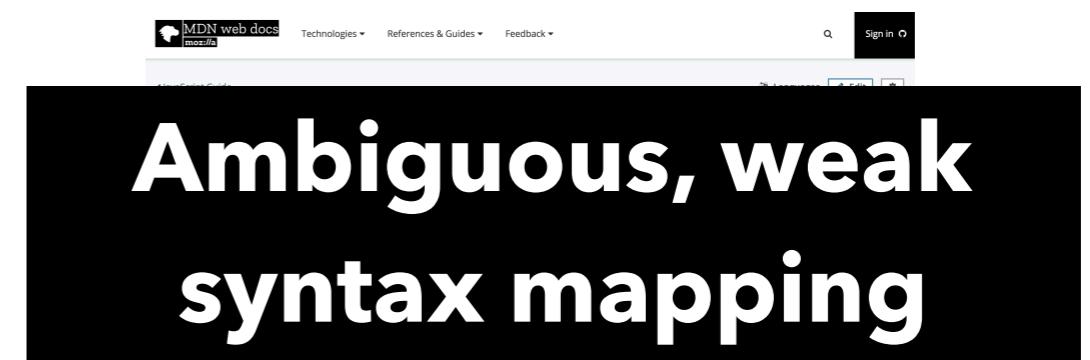
If you see a bug or any other issue with this page, please report it [here](#).

Run Back 8/12 Next Get Help

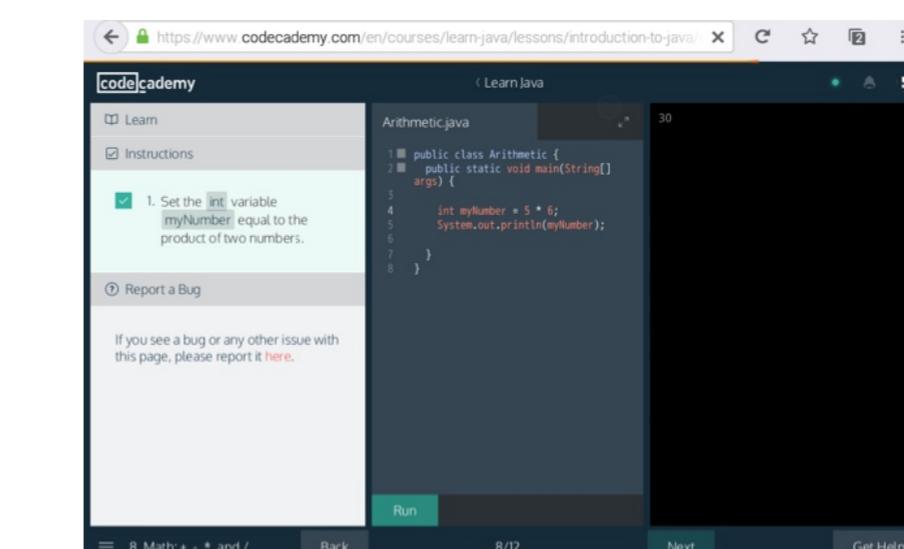
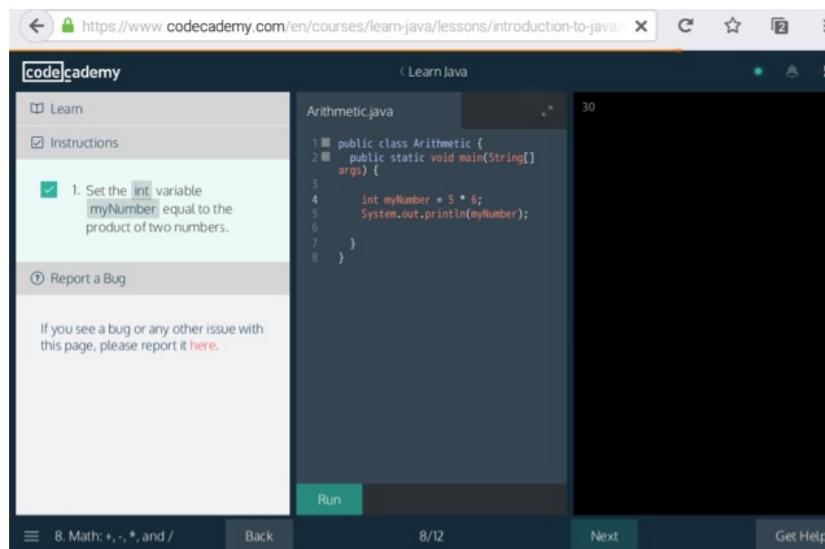
# Four major pedagogies

Learn formal semantics

Explain via natural language


$$\frac{\{e.args[i] \mid 0 < i < e.numargs \wedge \text{pointer\_type\_p}(e.args[i])\} \cup \{v \mid \text{global\_p}(v)\} \vdash e.body \Rightarrow R}{\{\} \vdash e \Rightarrow R} \text{ (FUNCTION)}$$


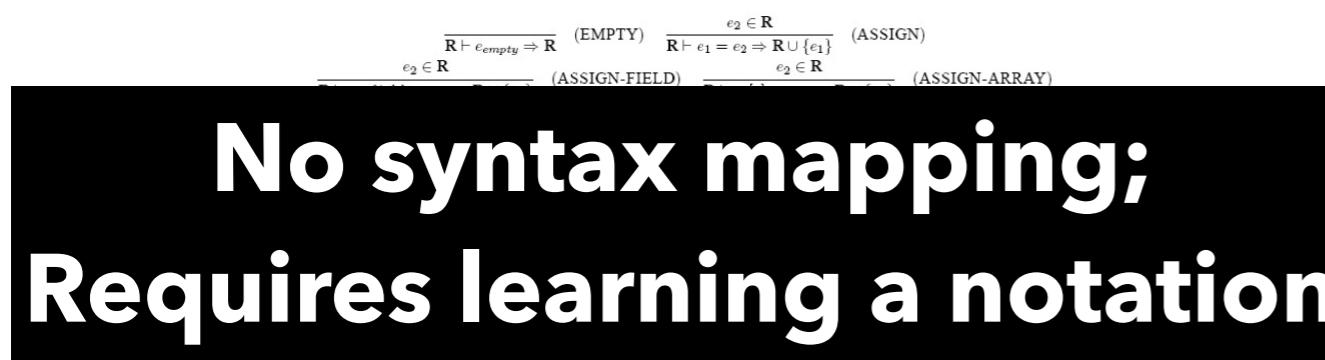
Write code



# Four major pedagogies

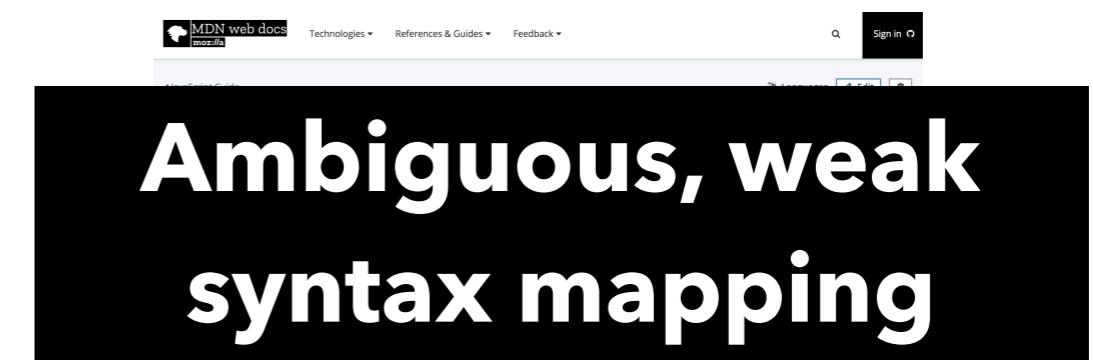
Learn formal semantics

Explain via natural language



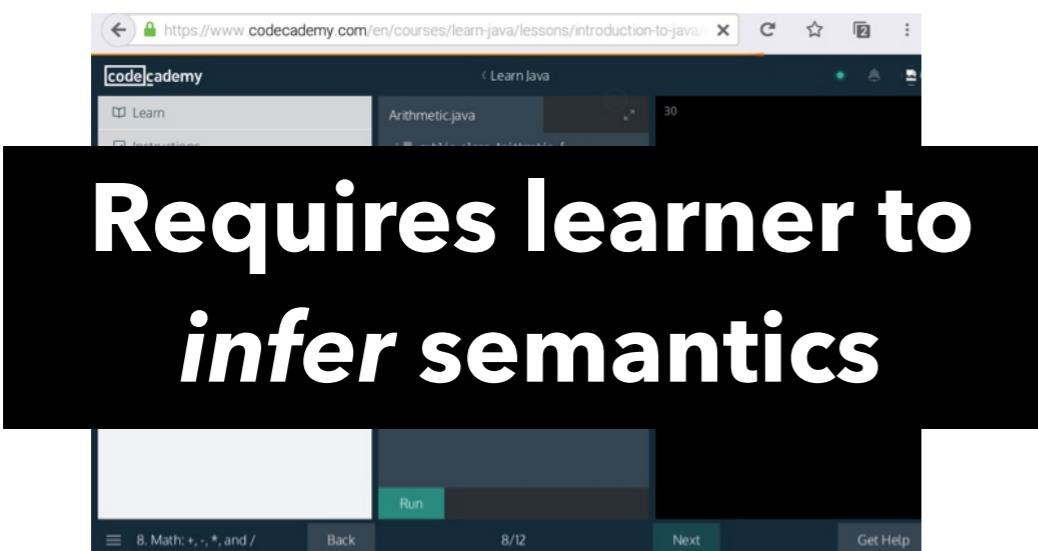
No syntax mapping;  
Requires learning a notation

$$\frac{\{e.args[i] \mid 0 < i < e.numargs \wedge \text{pointer\_type\_p}(e.args[i])\} \cup \{v \mid \text{global\_p}(v)\} \vdash e.body \Rightarrow R}{\{\} \vdash e \Rightarrow R} \quad (\text{FUNCTION})$$

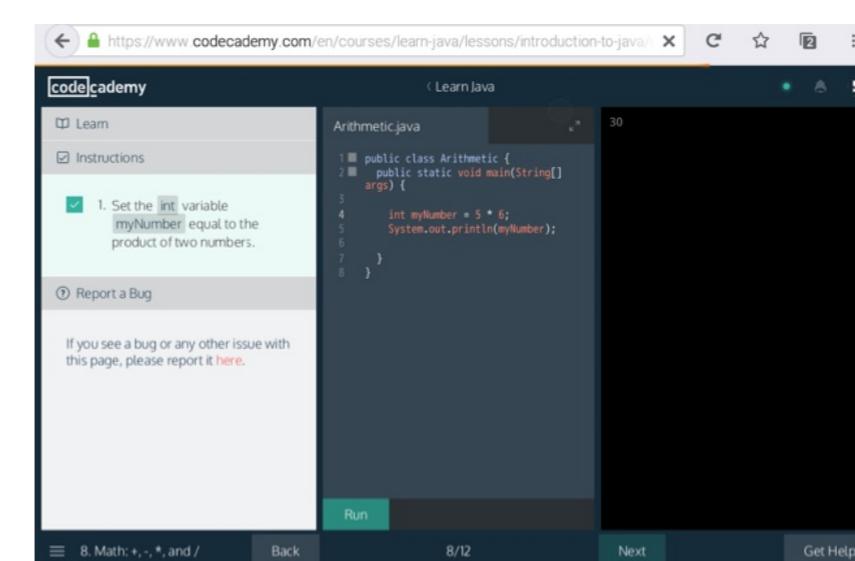


Numbers and dates  
Text formatting  
Regular expressions  
Indexed collections

Write code



Step through execution



# Four major pedagogies

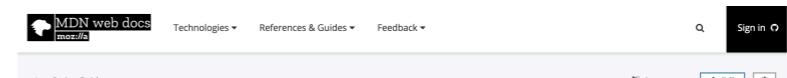
Learn formal semantics

Explain via natural language

$$\frac{\frac{\frac{R \vdash e_{empty} \Rightarrow R}{(EMPTY)} \quad \frac{e_2 \in R}{R \vdash e_1 = e_2 \Rightarrow R \cup \{e_1\}} \quad (ASSIGN)}{e_2 \in R} \quad (ASSIGN-FIELD) \quad \frac{}{e_2 \in R} \quad (ASSIGN-ARRAY)}{e_2 \in R}$$

No syntax mapping;  
Requires learning a notation

$$\frac{\{e.args[i] \mid 0 < i < e.numargs \wedge \text{pointer\_type\_p}(e.args[i])\} \cup \{v \mid \text{global\_p}(v)\} \vdash e.body \Rightarrow R}{\{\} \vdash e \Rightarrow R} \quad (\text{FUNCTION})$$

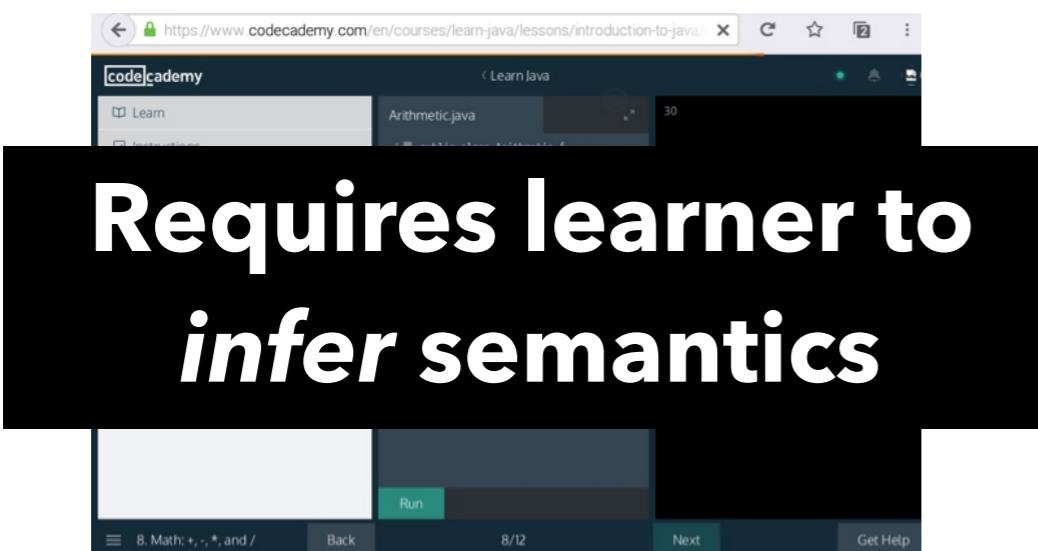


Ambiguous, weak  
syntax mapping

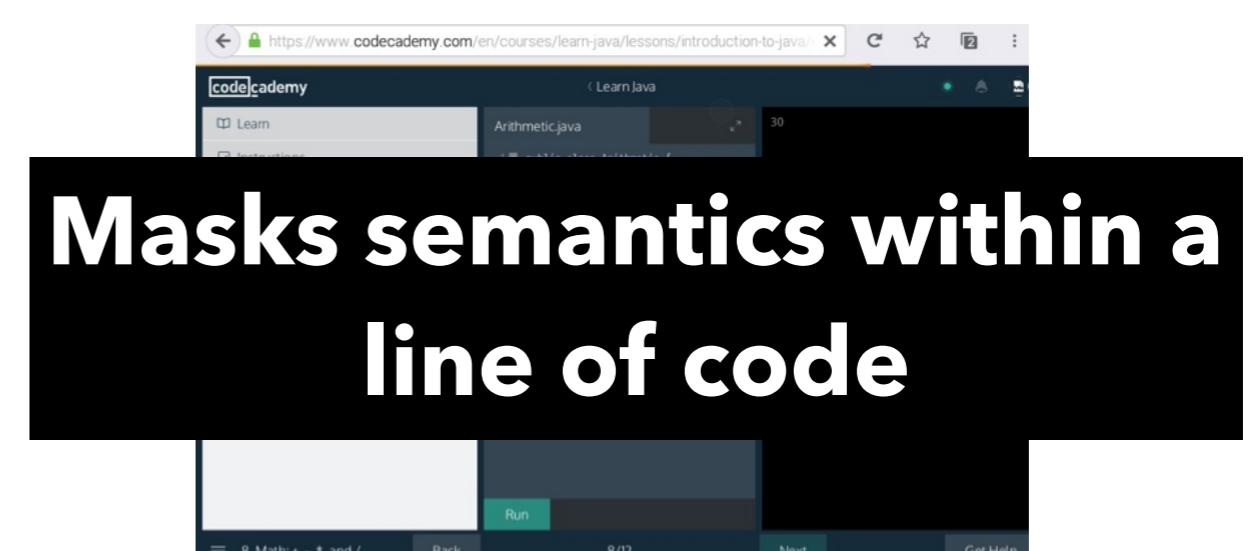
Numbers and dates  
Text formatting  
Regular expressions  
Indexed collections

In JavaScript, instructions are called statements and are separated by a semicolon (;). Spaces, tabs and newline characters are called whitespace. The source text of JavaScript scripts gets scanned from left to right and is converted into a sequence of input elements which are tokens, control characters, line terminators, comments or whitespace. ECMAScript also defines certain keywords and literals and has rules for automatic insertion of semicolons (ASI) to end statements. However, it is recommended to always add

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

## Widget



Mike Lee, Ph.D.

Learners *discover* semantics through debugging

## PLTutor



Greg Nelson

Tutor explicitly teaches semantics

## Tracing Strategies



Benji Xie

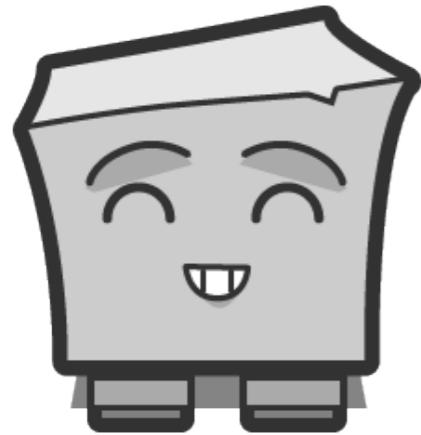
Learner reminded to follow semantics

# Widget

helpwidget.org



Mike Lee



- 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

# Gidget

helpgidget.org

Lee et al. 2014, VL/HCC



Mike Lee

The screenshot shows the Gidget website homepage. At the top, there are social media icons for Facebook and Twitter, followed by the word "Gidget" in a large, bold, black font against a blue sky with white clouds. To the right of the title are "Sign Up" and "Login" buttons. Below the title is a green grassy field featuring a central gray, blocky character with a smiling face, surrounded by various cartoon animals and objects: an orange cat, a pink pig, a white mouse, a yellow chick, a red car, and a yellow dog. To the right of this central image are three orange buttons with black text: "PLAY!", "Level Editor" (with a lock icon), and "About". At the bottom left is a "donate" button. Logos for NJIT, W OSU, and NSF are at the bottom center, along with a speaker icon indicating audio content. A "Q&A" button is visible in the bottom right corner. The overall theme is playful and educational.

# Gidget

helpgidget.org

Lee et al. 2014, VL/HCC



Mike Lee

The screenshot shows the Gidget website homepage. At the top, there are social media icons for Facebook and Twitter, followed by the word "Gidget" in a large, bold, black font against a blue sky with white clouds. To the right of the title are "Sign Up" and "Login" buttons. Below the title is a green grassy field featuring a central gray, blocky character with a smiling face, surrounded by various cartoon animals and objects: an orange cat, a pink pig, a white mouse, a yellow chick, a red car, and a yellow dog. To the right of this central image are three orange buttons with black text: "PLAY!", "Level Editor" (with a lock icon), and "About". At the bottom left is a "donate" button. Logos for NJIT, W OSU, and NSF are at the bottom center, along with a speaker icon indicating audio content. A "Q&A" button is visible in the bottom right corner. The overall theme is playful and educational.

# Widget

helpwidget.org

Lee et al. 2014, VL/HCC

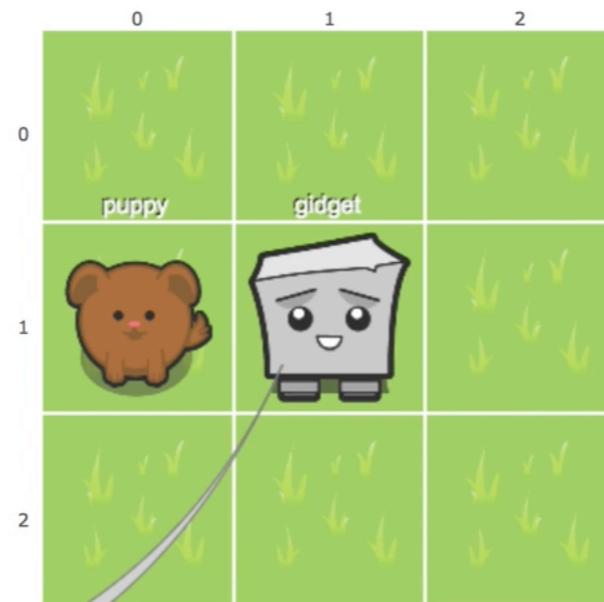


Mike Lee

Level 1. Let's get to the puppy!



world



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.

← Prev    Next →

# Widget

helpwidget.org

Lee et al. 2014, VL/HCC

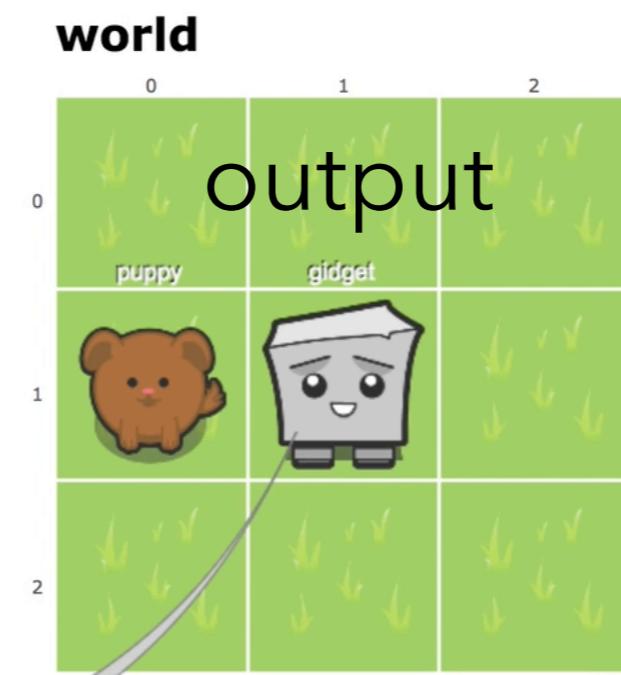


Mike Lee

Level 1. Let's get to the puppy!

The screenshot shows a 3x3 grid world. The top row is labeled '0', the middle row '1', and the bottom row '2'. The left column is labeled '0', the middle column '1', and the right column '2'. A brown puppy is at position (0, 0). A gray gidget is at position (1, 1). The word 'output' is written vertically across the center of the grid. The background is green with yellow grass tufts. There are three icons in the top right corner: a clipboard, a play button, and a menu icon.

code  
editor



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.

← Prev    Next →

test  
case

explanation

# Widget

helpwidget.org

Lee et al. 2014, VL/HCC



Mike Lee

37 levels teaching 12 semantics, including formative assessments to verify understanding

## Level 1. Let's get to the puppy!



**world**



# Widget

helpwidget.org

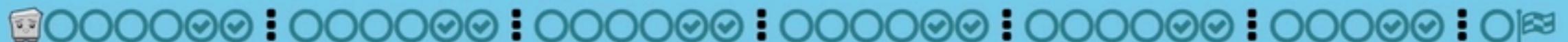
Lee et al. 2014, VL/HCC



Mike Lee

37 levels teaching 12 semantics, including formative assessments to verify understanding

## Level 1. Let's get to the puppy!



**world**



# Widget

helpwidget.org

Lee et al. 2014, VL/HCC



Mike Lee

level 20 teaches function calls

**Level 20. Press the button, open the gate!**



**world**

|   | 0                                                                                     | 1 | 2                                                                                     | 3 |
|---|---------------------------------------------------------------------------------------|---|---------------------------------------------------------------------------------------|---|
| 0 | basket                                                                                |   | piglet                                                                                |   |
| 1 |  |   |  |   |
| 2 |                                                                                       |   |                                                                                       |   |
| 3 |                                                                                       |   |                                                                                       |   |

# Widget

helpwidget.org

Lee et al. 2014, VL/HCC



Mike Lee

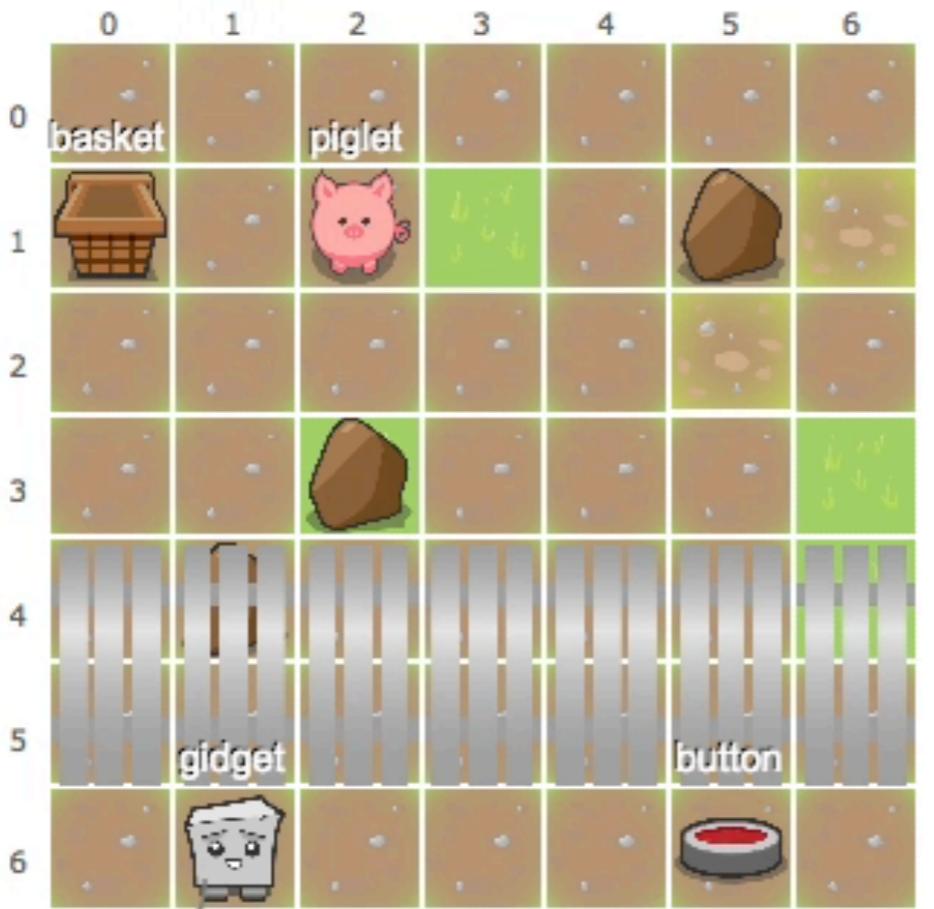
level 20 teaches function calls

**Level 20. Press the button, open the gate!**



**world**

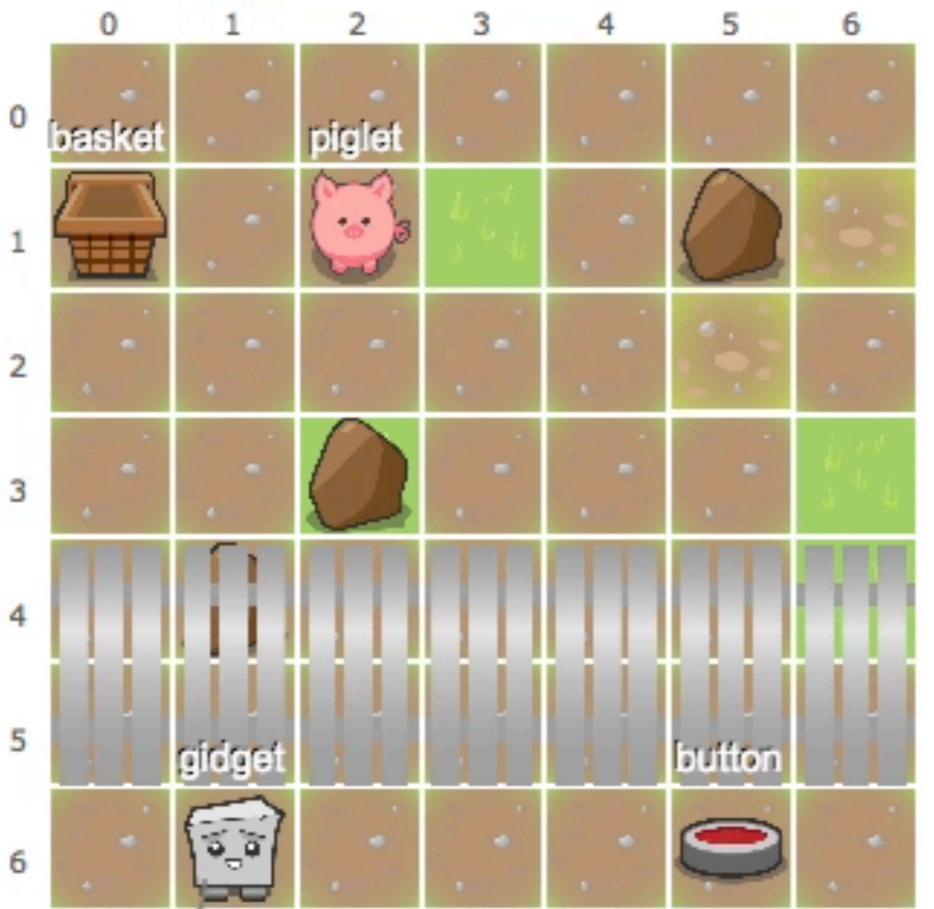
|   | 0                                                                                     | 1 | 2                                                                                     | 3 |
|---|---------------------------------------------------------------------------------------|---|---------------------------------------------------------------------------------------|---|
| 0 | basket                                                                                |   | piglet                                                                                |   |
| 1 |  |   |  |   |
| 2 |                                                                                       |   |                                                                                       |   |
| 3 |                                                                                       |   |                                                                                       |   |



All that grabbing & dropping made me remember a way to save time writing my programs though...

functions!

← Prev      Next →



All that grabbing & dropping made  
me remember a way to save time  
writing my programs though...

functions!

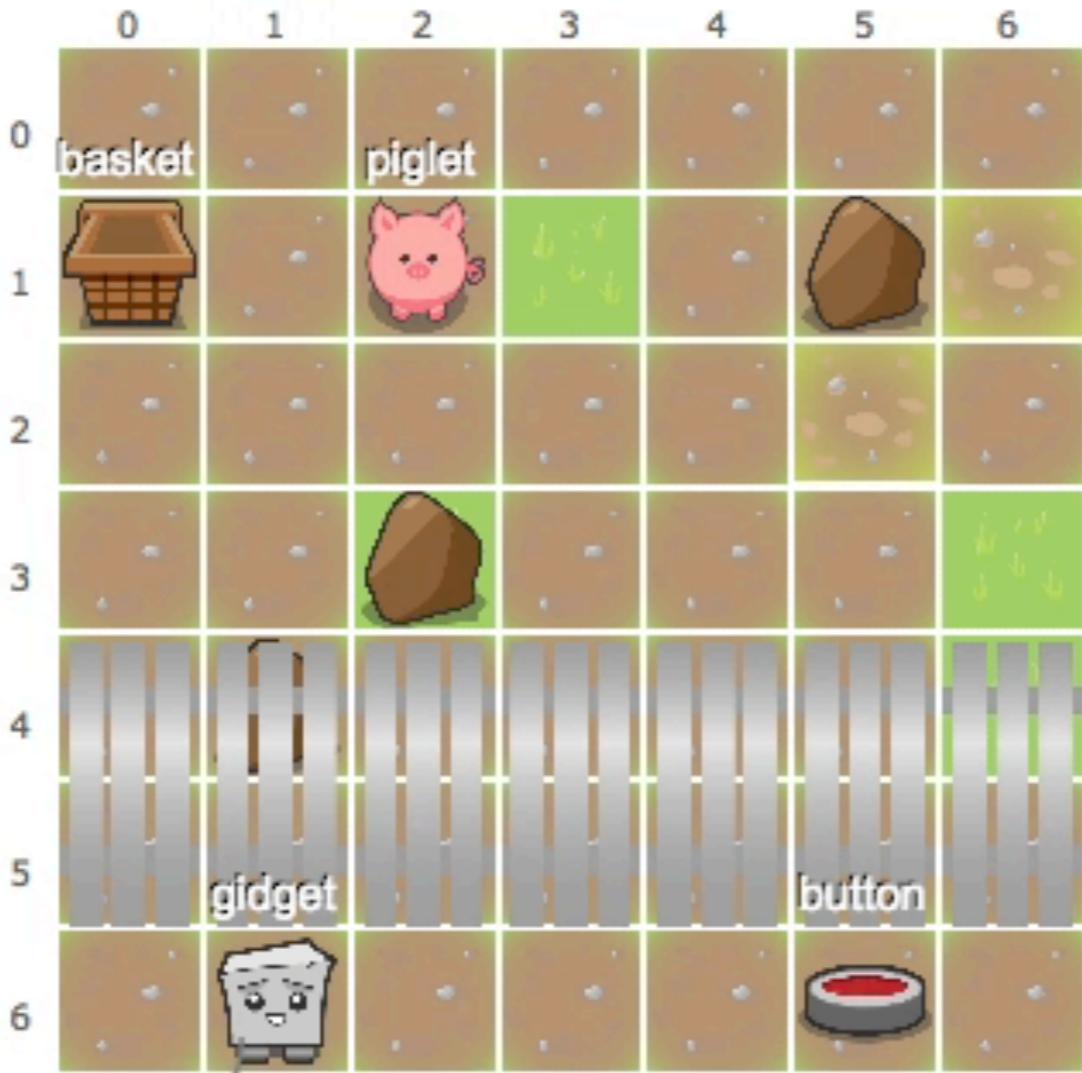
← Prev      Next →

# Function definition semantics

Q&A



# world

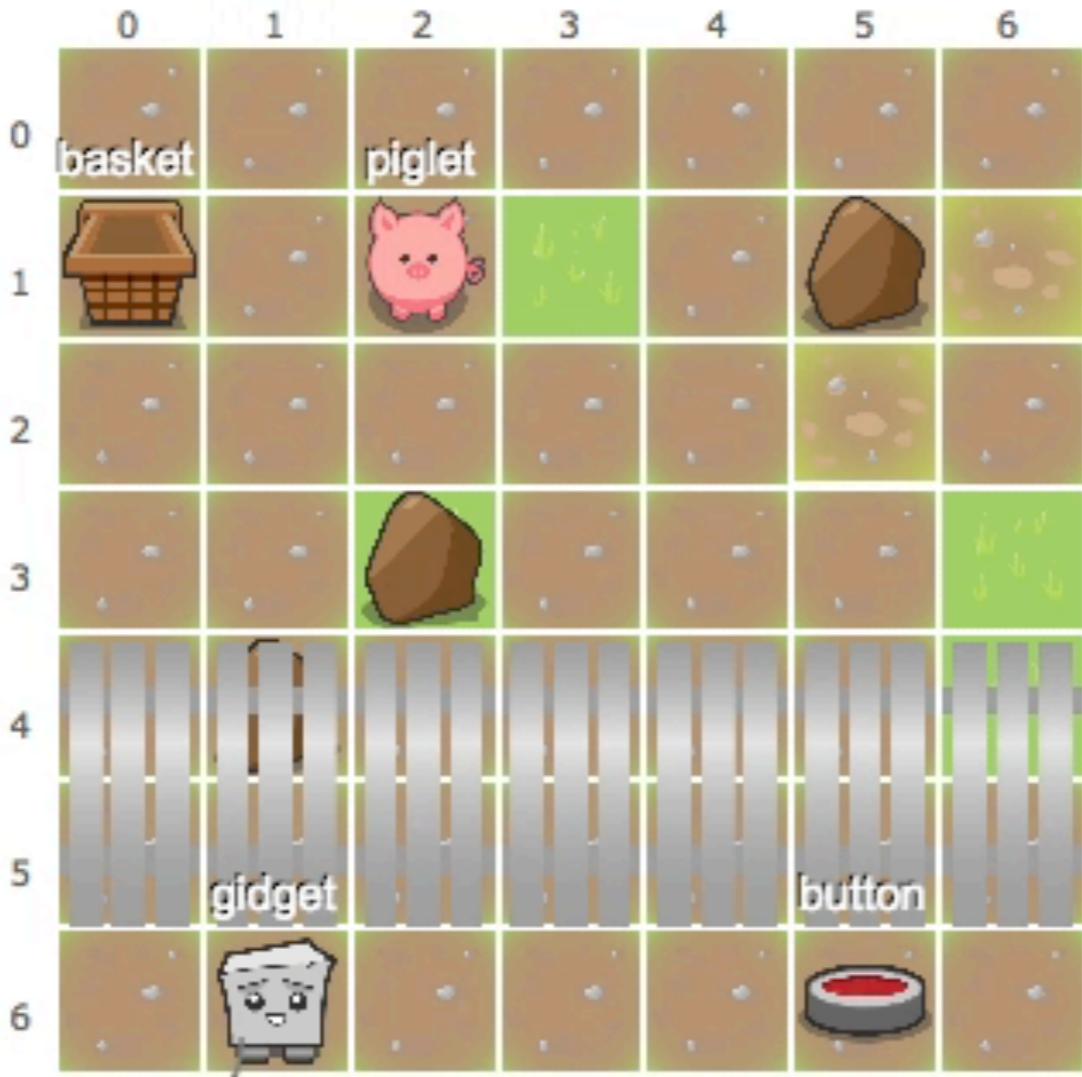


All that **grabbing** & **dropping** made  
me remember a way to save time  
writing my programs though...  
**functions!**

← Prev

Next →

# world



All that **grabbing** & **dropping** made me remember a way to save time writing my programs though... **functions!**

← Prev

Next →

Widget explains syntax and semantics



# Gidget

helpgidget.org

Lee et al. 2014, VL/HCC



Mike Lee

Level 20. Press the button, open the gate!



code

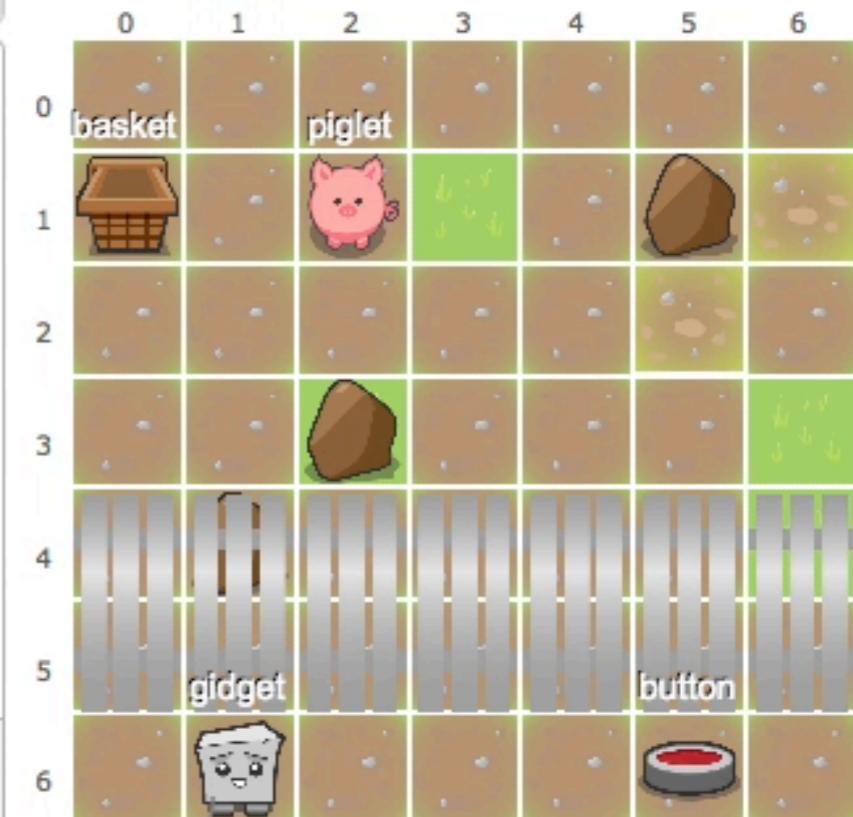
Original Code

Clear Code

```
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/
```

```
ensure /piglet/:nickname = "babe"
ensure /piglet/:age = 3
ensure # /piglet/ on /basket/ = 1
```

world



The **green line**  
and Gidget's  
speech bubble  
maps syntax to  
semantics

One step

One line

To end

Stop!

Don't forget you can click on objects to see their properties, and you should try running my code first to see what happens!

← Prev

Next →

# Gidget

helpgidget.org

Lee et al. 2014, VL/HCC



Mike Lee

Level 20. Press the button, open the gate!



code

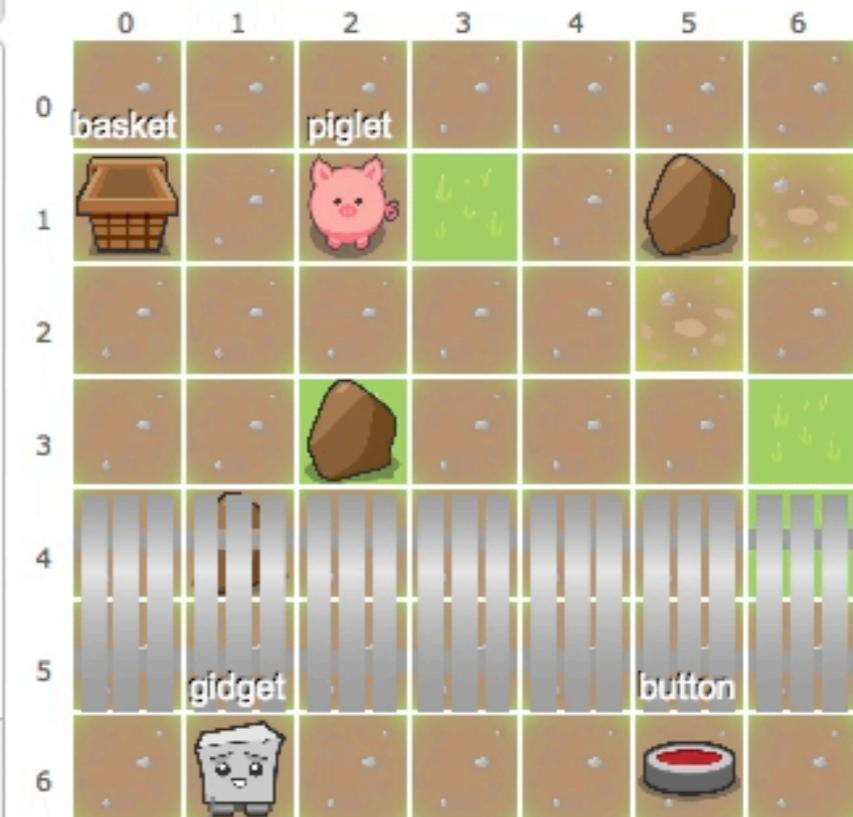
Original Code

Clear Code

```
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/
```

```
ensure /piglet/:nickname = "babe"
ensure /piglet/:age = 3
ensure # /piglet/ on /basket/ = 1
```

world



The **green line**  
and Gidget's  
speech bubble  
maps syntax to  
semantics

One step

One line

To end

Stop!

Don't forget you can click on objects to see their properties, and you should try running my code first to see what happens!

← Prev

Next →

# Widget

helpwidget.org

Lee et al. 2014, VL/HCC



Mike Lee

Level 20. Press the button, open the gate!



## code

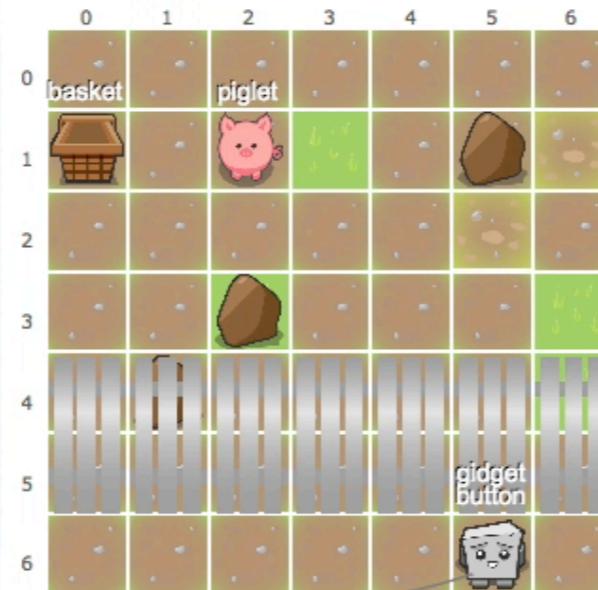
Original Code Clear Code

```
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/

ensure /piglet/:nickname = "babe"
ensure /piglet/:age = 3
ensure # /piglet/ on /basket/ = 1
```

One step One line To end Stop!

## world



## widget



|              |           |
|--------------|-----------|
| energy       | 97        |
| grabbed      | []        |
| image        | "default" |
| labeled      | true      |
| layer        | 1         |
| name         | "widget"  |
| position     | [6, 5]    |
| rotation     | 0         |
| scale        | 1         |
| transparency | 1         |
| code()       | [ ]       |

# Widget

helpwidget.org Lee et al. 2014, VL/HCC



Mike Lee

Level 20. Press the button, open the gate!



## code

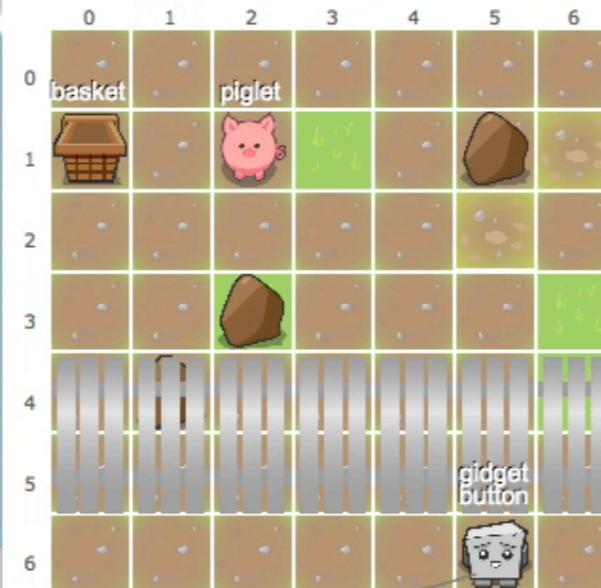
Original Code Clear Code

```
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/

ensure /piglet/:nickname = "babe"
ensure /piglet/:age = 3
ensure # /piglet/ on /basket/ = 1
```

One step One line To end Stop!

## world



## widget



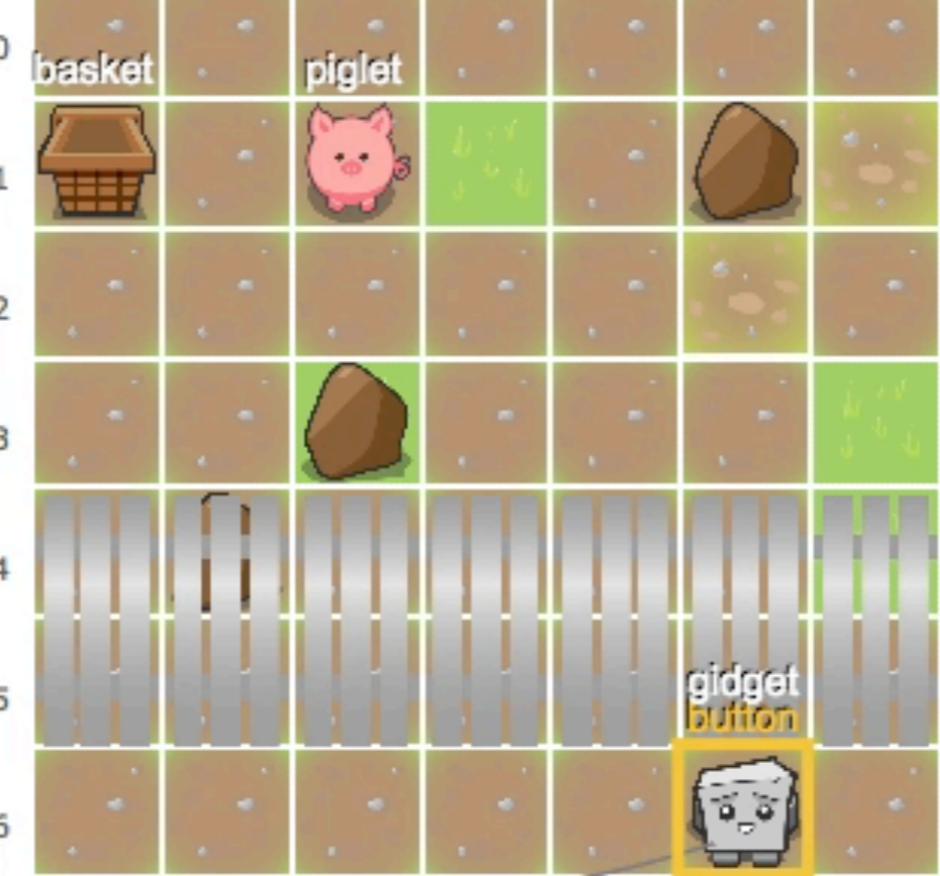
|              |           |
|--------------|-----------|
| energy       | 97        |
| grabbed      | []        |
| image        | "default" |
| labeled      | true      |
| layer        | 1         |
| name         | "widget"  |
| position     | [6, 5]    |
| rotation     | 0         |
| scale        | 1         |
| transparency | 1         |
| code()       | [ ]       |

Widget points  
out function  
definitions

to /button/  
y "Let's click the button to see its function  
utton/:openFence()  
unction getPiglet() -?  
goto /piglet/  
set /piglet/:nickname to "wilbur"  
set /piglet/:age to 3  
grab /piglet/  
tBird() -?  
tThePiggy() -?  
to /basket/

sure /piglet/:nickname = "babe"  
sure /piglet/:age = 3  
sure # /piglet/ on /basket/ = 1

One step      One line      To end      Stop!



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

Continue ➔

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

code()

[ ]

## button

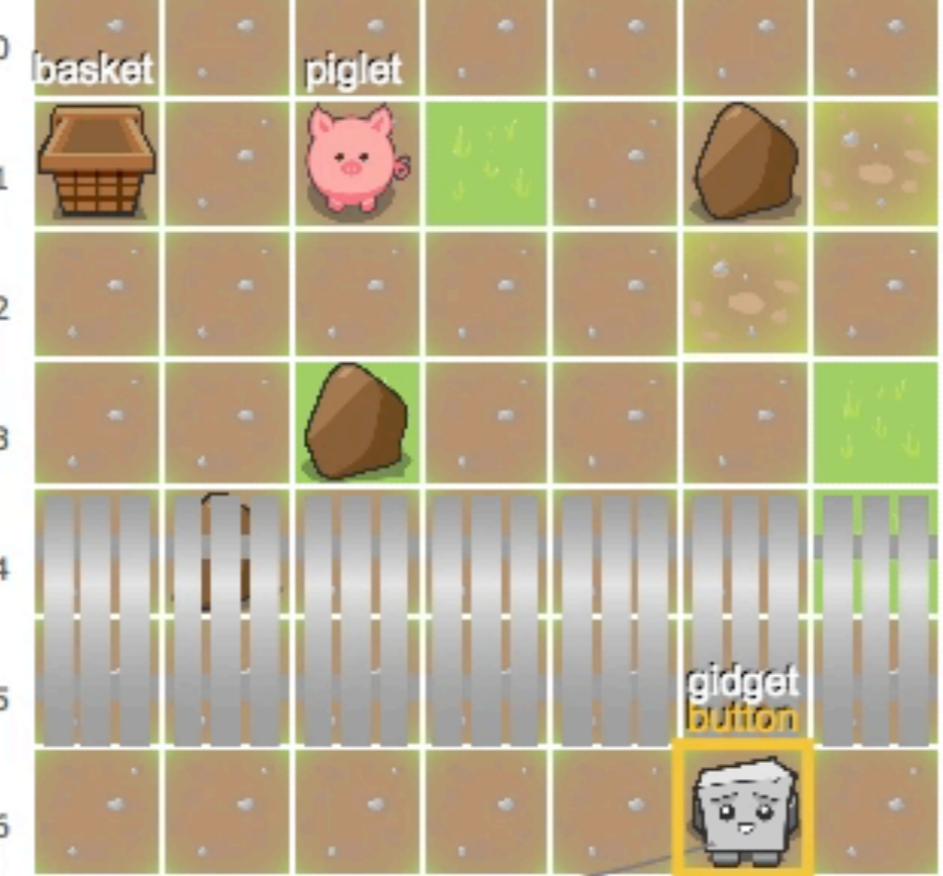
|              |            |
|--------------|------------|
| energy       | 100        |
| grabbed      | [ ]        |
| image        | "default"  |
| labeled      | true       |
| layer        | 1          |
| name         | "button"   |
| openGate     | openGate() |
| position     | [6, 5]     |
| rotation     | 0          |
| scale        | 1          |
| transparency | 1          |

code()

```
to /button/
y "Let's click the button to see its function
utton/:openFence()
unction getPiglet() -?
goto /piglet/
set /piglet/:nickname to "wilbur"
set /piglet/:age to 3
grab /piglet/
tBird() -?
tThePiggy() -?
to /basket/

sure /piglet/:nickname = "babe"
sure /piglet/:age = 3
sure # /piglet/ on /basket/ = 1
```

One step    One line    To end    Stop!



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

Continue ➔

Gidget explains  
name resolution  
semantics

|              |           |
|--------------|-----------|
| energy       | 97        |
| grabbed      | [ ]       |
| image        | "default" |
| labeled      | true      |
| layer        | 1         |
| name         | "gidget"  |
| position     | [6, 5]    |
| rotation     | 0         |
| scale        | 1         |
| transparency | 1         |
| code()       |           |
|              |           |

## button

|              |            |
|--------------|------------|
| energy       | 100        |
| grabbed      | [ ]        |
| image        | "default"  |
| labeled      | true       |
| layer        | 1          |
| name         | "button"   |
| openGate     | openGate() |
| position     | [6, 5]     |
| rotation     | 0          |
| scale        | 1          |
| transparency | 1          |
| code()       |            |
|              |            |



Q

# Widget

helpwidget.org Lee et al. 2014, VL/HCC



Mike Lee

Level 20. Press the button, open the gate!



## code

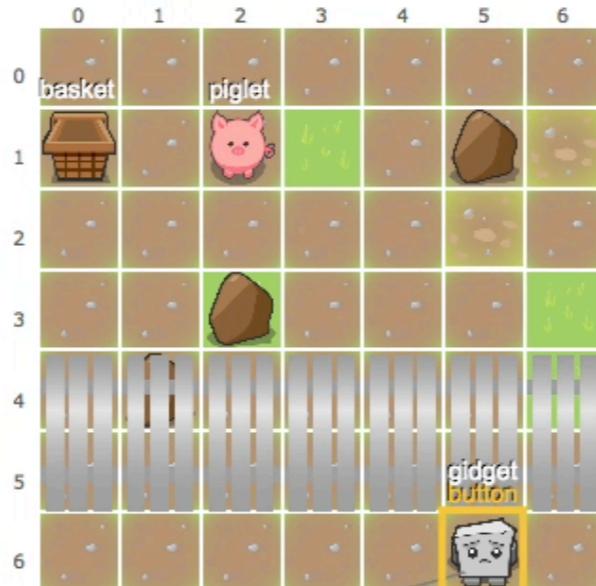
Original Code Clear Code

```
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/
```

```
ensure /piglet/:nickname = "babe"
ensure /piglet/:age = 3
ensure # /piglet/ on /basket/ = 1
```

One step One line To end Stop!

## world



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.

Stop

## widget



|              |           |
|--------------|-----------|
| energy       | 94        |
| grabbed      | []        |
| image        | "default" |
| labeled      | true      |
| layer        | 1         |
| name         | "widget"  |
| position     | [6, 5]    |
| rotation     | 0         |
| scale        | 1         |
| transparency | 1         |
| code()       | [ ]       |

## button



|              |            |
|--------------|------------|
| energy       | 100        |
| grabbed      | []         |
| image        | "default"  |
| labeled      | true       |
| layer        | 1          |
| name         | "button"   |
| openGate     | openGate() |
| position     | [6, 5]     |
| rotation     | 0          |
| scale        | 1          |
| transparency | 1          |
| code()       | [ ]        |

Q&A

# Widget

helpwidget.org

Lee et al. 2014, VL/HCC



Mike Lee

Level 20. Press the button, open the gate!



## code

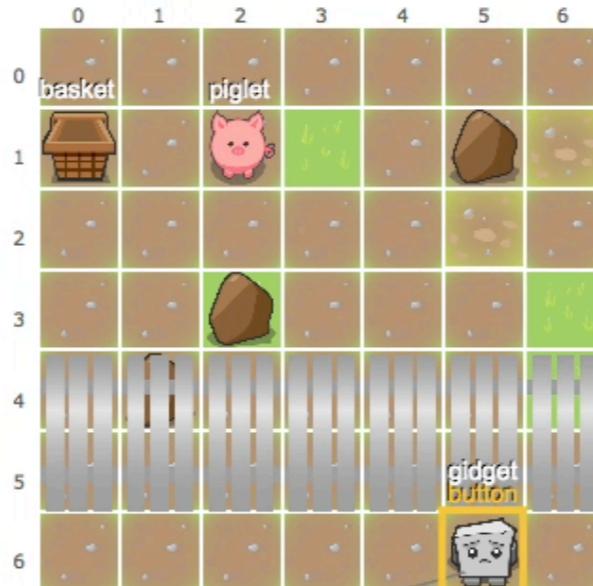
Original Code   Clear Code

```
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/
```

```
ensure /piglet/:nickname = "babe"
ensure /piglet/:age = 3
ensure # /piglet/ on /basket/ = 1
```

One step   One line   To end   Stop!

## world



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.

Stop

## widget



|              |           |
|--------------|-----------|
| energy       | 94        |
| grabbed      | []        |
| image        | "default" |
| labeled      | true      |
| layer        | 1         |
| name         | "widget"  |
| position     | [6, 5]    |
| rotation     | 0         |
| scale        | 1         |
| transparency | 1         |
| code()       | [ ]       |

## button



|              |            |
|--------------|------------|
| energy       | 100        |
| grabbed      | []         |
| image        | "default"  |
| labeled      | true       |
| layer        | 1          |
| name         | "button"   |
| openGate     | openGate() |
| position     | [6, 5]     |
| rotation     | 0          |
| scale        | 1          |
| transparency | 1          |
| code()       | [ ]        |

Q&A



# Gidget

helpgidget.org



Mike Lee



- 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

# Gidget

helpgidget.org



Mike Lee



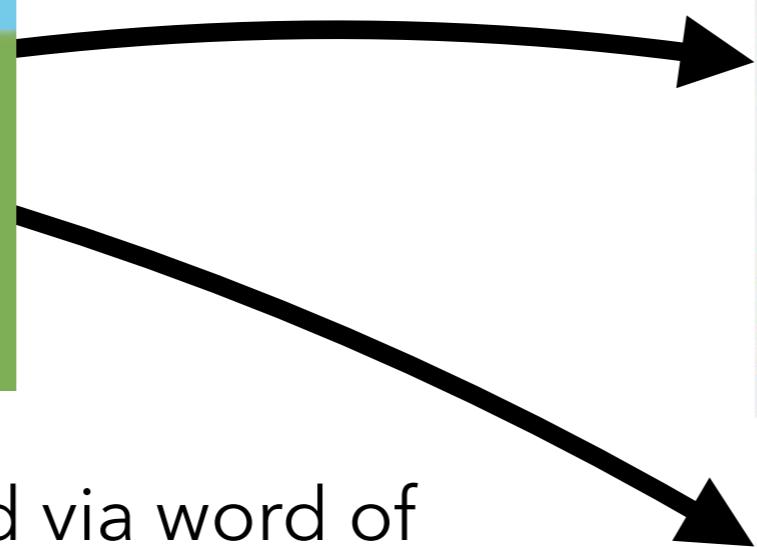
- Four online controlled experiments with over 1,000 adult learners:
  - Learning is **2x as fast** as Codecademy tutorial, **2x as much** as open-ended creative exploration Lee & Ko 2015
  - **Assessment levels** significantly increase learning efficiency Lee et al. 2013
  - Giving compiler a face and 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

# Gidget

helpgidget.org Lee et al. 2014, VL/HCC

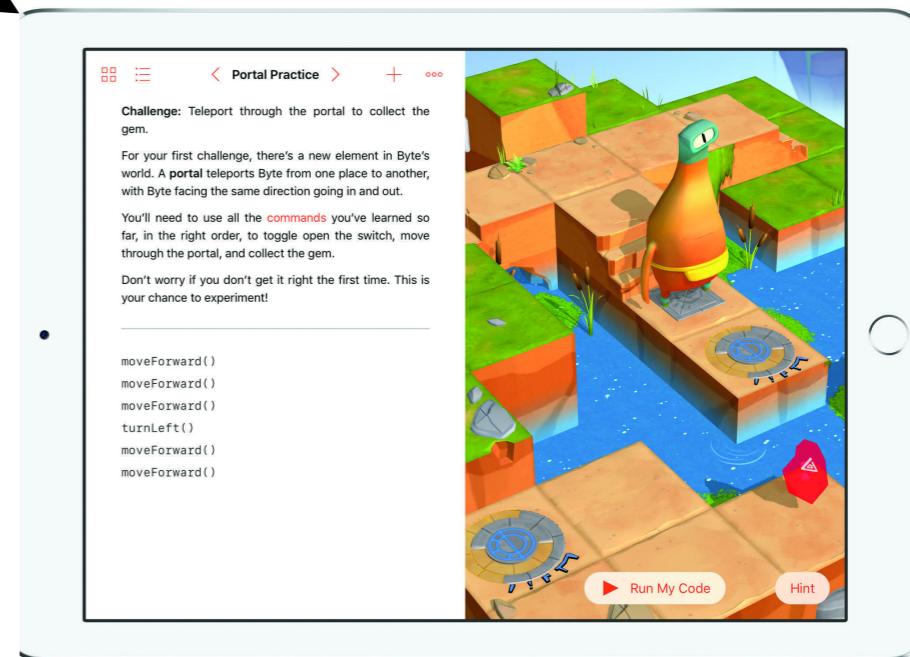


Mike Lee



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.

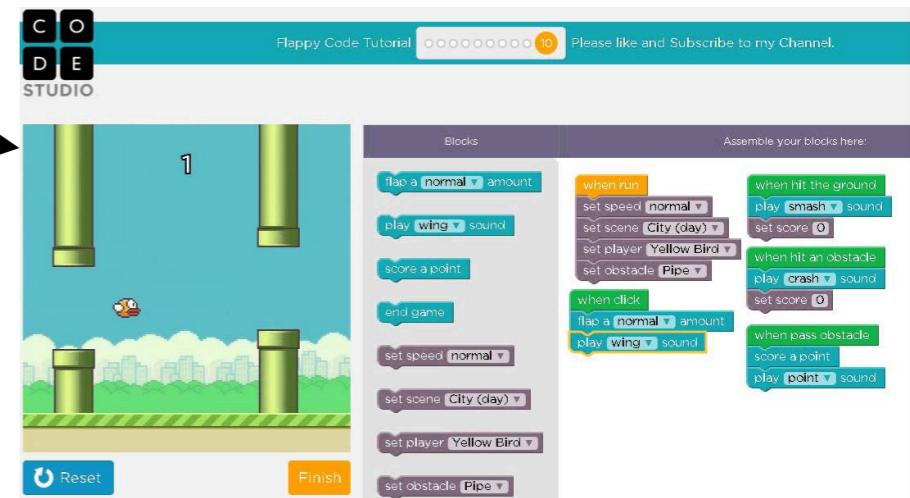
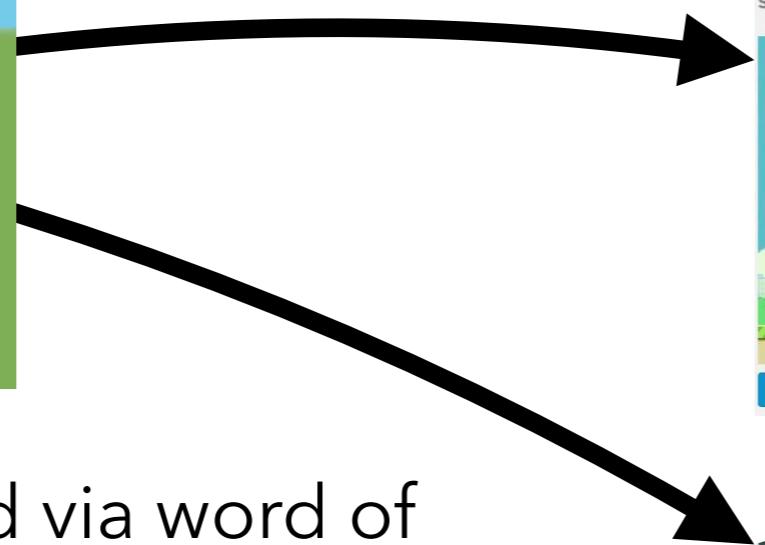


# Gidget

helpgidget.org Lee et al. 2014, VL/HCC

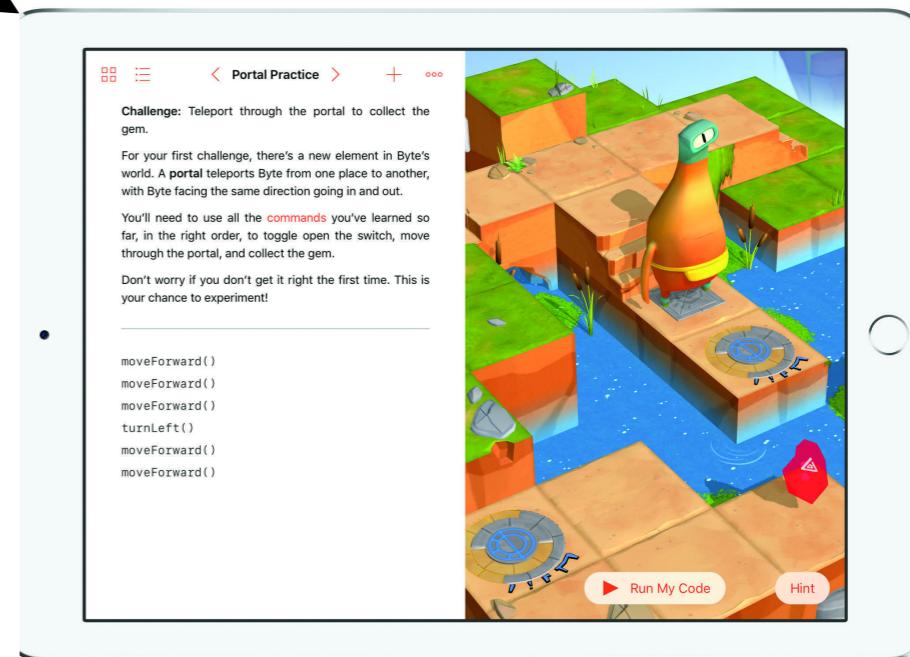


Mike Lee



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

## Widget



Mike Lee, Ph.D.

Learners discover  
semantics through  
debugging

## PLTutor



Greg Nelson

Tutor explicitly  
teaches  
semantics

## Tracing Strategies



Benji Xie

Learner  
reminded to  
follow semantics

# Three ideas

## Widget



Mike Lee, Ph.D.

Learners *discover* semantics through debugging

## PLTutor



Greg Nelson

Tutor explicitly teaches semantics

## Tracing Strategies



Benji Xie

Learner reminded to follow semantics

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

A screenshot of the PLTutor interface. On the left, there's a sidebar titled 'Programs' with a list of 'Program 1' through 'Program 5'. Below it is a 'Knowledge Units' section with topics like 'This Tutor, Learning, State, and the Stack', 'Variables', 'Boolean Values (True False)', 'Arrays', 'Operators + - \* / ()', 'Comparison Operators == != &gt; &gt;= &lt; &lt;=' , 'If statements', 'Operators Review', and 'Loops Enable Repeating'. The main area is divided into 'Program' and 'State'. The 'Program' tab shows a blank code editor. The 'State' tab shows a table with columns 'first frame()', 'instruction', 'stack', 'empty', 'namespace', and '{}'. A message in the state table says 'The program is done executing.'

A screenshot of a lesson step titled 'Learning step 1 of 180'. The title is 'If statements'. It contains text about how if statements work and what happens to the stack. Below this is a code editor with three snippets of JavaScript. The first snippet shows an if statement with a true condition. The second shows an if statement with a false condition. The third shows an if statement with a comparison. To the right is a 'State' panel showing the stack, which is empty, and the namespace, which contains 'x = 1000000'. The 'Program' tab shows the same three snippets of code.

- 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

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

A screenshot of the PLTutor interface. On the left, there's a sidebar titled 'Programs' with a list of 'Program 1' through 'Program 5'. Below it is a 'Knowledge Units' section with topics like 'This Tutor, Learning, State, and the Stack', 'Variables', 'Boolean Values (True False)', 'Arrays', 'Operators + - \* / ()', 'Comparison Operators == != &gt; &gt;= &lt; &lt;=' , 'If statements', 'Operators Review', and 'Loops Enable Repeating'. The main area is divided into 'Program' and 'State'. The 'Program' tab shows a blank code editor. The 'State' tab shows a table with columns 'first frame()', 'instruction', 'stack', 'empty', 'namespace', and '{}'. A message in the state table says 'The program is done executing.'

A screenshot of a lesson step titled 'Learning step 1 of 180'. The title is 'If statements'. It contains text about how if statements work and what happens to the stack. Below this is a code editor with three snippets of JavaScript. The first snippet shows an if statement with a true condition. The second shows an if statement with a false condition. The third shows an if statement with a comparison. To the right is a 'State' panel showing the stack, which is empty, and the namespace, which contains 'x = 1000000'. The 'Program' tab shows the same three snippets of code.

- 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

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).  
Comprehension First: Evaluating a Novel Pedagogy and Tutoring  
System for Program Tracing in CS1. ACM ICER.



Greg Nelson

Each chapter covers a set of semantics through a series of programs

The screenshot shows the PLTutor application interface. On the left, a sidebar titled "Programs" lists five programs: Program 1, Program 2, Program 3, Program 4, and Program 5. Below this, a section titled "Knowledge Units" lists various programming concepts: This Tutor, Learning, State, and the Stack; Variables; Variables - details and changing values; Boolean Values (True False); Arrays; Operators + - \* / (); Comparison Operators == != > >= < <=; and If statements. The main area is titled "Program" and contains a large, empty gray box. To the right, a section titled "State" displays the final state of the program execution:

```
first frame()
instruction
stack
empty
namespace
{}
```

A message in a red box states: "The program is done executing."

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

**Lesson** explains **Program** links  
purpose of syntax to semantics

**State** explains semantics

## ≡ Lesson

Learning step 1 of 180

Back

Next

### 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

```
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;
```

## State

first frame()

instruction

This is a **variable declaration** statement. It declares one or more names and can optionally assign them values.

stack

empty

namespace

{}

52

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

**Lesson** explains **Program** links  
purpose of syntax to semantics

**State** explains semantics

Learning step 1 of 180

Back Next

## 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

```
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;
```

## State

first frame()

instruction

This is a **variable declaration** statement. It declares one or more names and can optionally assign them values.

stack

empty

namespace

{}

# purpose of semantics

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

## Lesson

Learning step 1 of 180

Back Next

## 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

```
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

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

## Lesson

Learning step 1 of 180

Back Next

## 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

```
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){
```

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

**Next** moves through both program execution trace and instruction.

The screenshot shows the PLTutor interface. On the left, under 'Lesson', it says 'Learning step 1 of 180' with 'Back' and 'Next' buttons. The 'Next' button is highlighted with a hand cursor icon. Below it, the title 'If statements' is displayed in large, bold, dark font. A text block says: '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.' At the bottom, it says 'If statements allow computers to do some set'. On the right, under 'Program', there is a snippet of code with annotations:

```
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;
}
```

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

**Next** moves through both program execution trace and instruction.

The screenshot shows the PLTutor interface. On the left, under 'Lesson', it says 'Learning step 1 of 180' with 'Back' and 'Next' buttons. The 'Next' button is highlighted with a hand cursor icon. Below it, the title 'If statements' is displayed in large, bold, dark font. A text block says: '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.' At the bottom, it says 'If statements allow computers to do some set'. On the right, under 'Program', there is a block of code with annotations:

```
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;
}
```

A red diamond icon is in the bottom right corner.

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).  
Comprehension First: Evaluating a Novel Pedagogy and Tutoring  
System for Program Tracing in CS1. ACM ICER.



Greg Nelson

State teaches semantics in a runtime context,  
Lesson generalizes back to purpose

## ≡ Lesson

Learning step 7 of 180

Back

Next

Carefully step through the code and read all the explanations for the instructions, and look at the stack.



## Program

```
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 = 1000000; 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;
```

## State

first frame()

instruction

Before moving to the next statement, we remove the value this expression left on the stack, if any.

stack

empty

namespace

{

|   |   |
|---|---|
| x | 0 |
|---|---|

}

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).  
Comprehension First: Evaluating a Novel Pedagogy and Tutoring  
System for Program Tracing in CS1. ACM ICER.



Greg Nelson

State teaches semantics in a runtime context,  
Lesson generalizes back to purpose

## ≡ Lesson

Learning step 7 of 180

Back

Next

Carefully step through the code and read all the explanations for the instructions, and look at the stack.



## Program

```
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 = 1000000; 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;
```

## State

first frame()

instruction

Before moving to the next statement, we remove the value this expression left on the stack, if any.

stack

empty

namespace

{

|   |   |
|---|---|
| x | 0 |
|---|---|

}

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

Reverse execution allows learner to review instruction they didn't understand.

The screenshot shows the PLTutor interface. On the left, there's a large blue circular arrow icon with a white arrow pointing right, labeled "View Program". Above it, the "Lesson" section indicates "Learning step 15 of 180" with "Back" and "Next" buttons. The "Program" section contains the following JavaScript code:

```
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 10 is not equal to 0
 and that leaves true on the stack */
}
```

The "State" section visualizes the execution environment:

- first frame()**: A list of variables.
- instruction**: "Assign the value on top of the stack to x".
- stack**: Shows the value **1000000**.
- namespace**: A table with one row:

|   |         |   |
|---|---------|---|
| x | 1000000 | 0 |
|---|---------|---|

At the bottom of the slide, the number **56** is displayed.

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

Reverse execution allows learner to review instruction they didn't understand.

The screenshot shows the PLTutor interface. On the left, there's a large blue circular button with a white right-pointing arrow. Below it is the text "View Program". To the right of this is a vertical sidebar with the title "Lesson" and "Learning step 15 of 180". It has "Back" and "Next" buttons, with "Next" being highlighted. The main area is divided into three columns: "Program", "State", and "Stack".

**Program:**

```
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 10 is not equal to 0
 and that leaves true on the stack */
}
```

**State:**

- first frame()**
- instruction**: Assign the value on top of the stack to x
- stack**: 1000000
- namespace**: { x 0 }

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).  
Comprehension First: Evaluating a Novel Pedagogy and Tutoring  
System for Program Tracing in CS1. ACM ICER.



Greg Nelson

**Lesson** explains the side effect of the semantics before proceeding to further examples.

The screenshot shows a mobile application interface for 'PLTutor'. At the top, there's a blue header bar with a navigation menu icon and the word 'Lesson'. Below the header, the text 'Learning step 18 of 180' is displayed. There are two buttons: 'Back' (orange) and 'Next' (grey). A cursor arrow is positioned over the 'Next' button. The main content area contains the text: 'x has a new value now because the condition 10 > 0 was true. You can see it on the stack.' To the right of the main content, there are vertical blue bars with some code snippets visible: 'var x' and 'if (10 > 0)'.

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).  
Comprehension First: Evaluating a Novel Pedagogy and Tutoring  
System for Program Tracing in CS1. ACM ICER.



Greg Nelson

**Lesson** explains the side effect of the semantics before proceeding to further examples.

The screenshot shows a mobile application interface for 'PLTutor'. At the top, there's a blue header bar with a navigation menu icon and the word 'Lesson'. Below the header, the text 'Learning step 18 of 180' is displayed. There are two buttons: 'Back' (orange) and 'Next' (grey). A cursor arrow is hovering over the 'Next' button. The main content area contains the text: 'x has a new value now because the condition 10 > 0 was true. You can see it on the stack.' To the right of the main content, there are vertical blue bars with some code snippets visible: 'var x' and 'if ('. At the bottom right, there's a small red diamond icon.

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

**Assessments** embedded in execution trace require learners to predict side effects of semantics.

## Lesson

Learning step 29 of 180

Back

Next

What did  $0 > 10$  evaluate to (leave on the stack)?

true

10

-10

false

## Program

```
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 10 is not equal to 0
 and that leaves true on the stack */
}
```

## State

first frame()

instruction

If the if statement's condition is true, execute the true statements. Otherwise, skip it.

stack

?

namespace

{

x

0

}

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

**Assessments** embedded in execution trace require learners to predict side effects of semantics.

## Lesson

Learning step 29 of 180

Back

Next

What did  $0 > 10$  evaluate to (leave on the stack)?

true

10

-10

false

## Program

```
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 10 is not equal to 0
 and that leaves true on the stack */
}
```

## State

first frame()

instruction

If the if statement's condition is true, execute the true statements. Otherwise, skip it.

stack

?

namespace

```
{
 x 0
}
```

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

The screenshot shows the PLTutor interface. On the left, the 'Lesson' panel displays 'Learning step 1 of 180' and a section titled 'If statements'. It contains text explaining that if statements allow computers to do some set of instructions if a condition is true or not, and includes a code snippet: 

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

 Below it, a note says 'Let's step through one to see how it works.' In the center, the 'Program' panel shows three snippets of pseudocode. The first snippet shows an if block where the condition is true: 

```
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 = 100000;
}
```

 The second snippet shows an if block where the condition is false: 

```
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 = 100000;
}
```

 The third snippet shows an if block where the condition is equal: 

```
var x = 0;
if (0 == 10){
 /* the computer will NOT execute inside here
 because 0 is equal to 10
 leaves false on the stack */
 x = 100000;
}
```

 In the bottom right, the 'State' panel shows the 'first frame()' instruction. It includes a note: 'This is a variable declaration statement. It declares one or more names and can optionally assign them values.' The state is shown as: 

```
stack
empty
namespace
{}
```

- 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

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

The screenshot shows the PLTutor interface. On the left, the 'Lesson' panel displays 'Learning step 1 of 180' and a section titled 'If statements'. It contains text explaining that if statements allow computers to do some set of instructions if a condition is true or not, and includes a code snippet: 

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

 Below it, a note says 'Let's step through one to see how it works.' In the center, the 'Program' panel shows three snippets of pseudocode. The first snippet shows an if block where the condition is true: 

```
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 = 100000;
}
```

 The second snippet shows an if block where the condition is false: 

```
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 = 100000;
}
```

 The third snippet shows an if block where the condition is equal: 

```
var x = 0;
if (0 == 10){
 /* the computer will NOT execute inside here
 because 0 is equal to 10
 leaves false on the stack */
 x = 100000;
}
```

 In the bottom right, the 'State' panel shows the 'first frame()' instruction. It includes a note: 'This is a variable declaration statement. It declares one or more names and can optionally assign them values.' The state is shown as: 

```
stack
empty
namespace
{}
```

- 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

# PLTutor

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

The screenshot shows the PLTutor interface. On the left, a 'Lesson' panel titled 'If statements' provides an introduction to if statements. In the center, a 'Program' panel displays three snippets of JavaScript code illustrating different conditions: one where the condition is true, one where it is false, and one where it is equal. On the right, a 'State' panel shows a stack frame with an empty stack and a namespace.

- 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

Greg Nelson, Benjamin Xie, and Andrew J. Ko (2017).

Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. ACM ICER.



Greg Nelson

The screenshot shows the PLTutor interface. On the left, a 'Lesson' panel titled 'If statements' provides an introduction to if statements. In the center, a 'Program' panel displays three snippets of pseudocode illustrating different conditions: one where the condition is true, one where it is false, and one where it is equal. On the right, a 'State' panel shows a stack frame with an empty stack and a namespace.

- 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

## Widget



Mike Lee, Ph.D.

Learners *discover* semantics through debugging

## PLTutor



Greg Nelson

Tutor explicitly teaches semantics

## Tracing Strategies



Benji Xie

Learner reminded to follow semantics

# Three ideas

## Widget



Mike Lee, Ph.D.

Learners *discover* semantics through debugging

## PLTutor



Greg Nelson

Tutor explicitly teaches semantics

## Tracing Strategies



Benji Xie

Learner reminded to follow semantics



# Strategy

Benjamin Xie, Greg Nelson, and Andrew J. Ko (2017). An Explicit Strategy to Scaffold Novice Program Tracing. SIGCSE.

Benji Xie

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.

The handwritten notes include:

- A memory table titled "Memory : main" with columns for "name" and "value". It shows initial values for x=2, y=3, and f=16.0. A checkmark is written next to the table.
- A vertical column of numbers from 1 to 4.
- Annotations at the bottom left: "return 8" and "return 16.0".

- When learners have brittle knowledge of semantics, they often **guess** how programs will execute
- An explicit strategy for reading programs should outperform guessing



# Strategy

Benjamin Xie, Greg Nelson, and Andrew J. Ko (2017). An Explicit Strategy to Scaffold Novice Program Tracing.SIGCSE.

Benji Xie

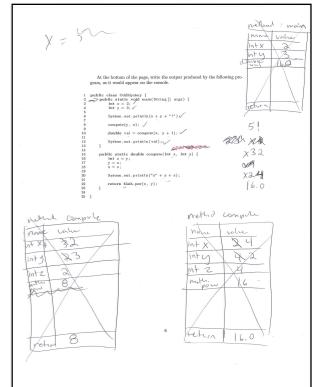
## STRATEGY: Understanding the Problem; Run the Code (like a computer).

### 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.





# Strategy

Benjamin Xie, Greg Nelson, and Andrew J. Ko (2017). An Explicit Strategy to Scaffold Novice Program Tracing.SIGCSE.

Benji Xie

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.

*(Handwritten notes: A wavy line starts from the first instruction and points to the second instruction. A checkmark is written next to the third instruction.)*

At the bottom of the page, write the output produced by the following program, as it would appear on the console.

```
1 public class OddMystery {
2 public static void main(String [] args) {
3 int x = 2; ✓
4 int y = 3; ✓
5
6 System.out.println(x + y + "!");✓
7
8 compute(y, x); ✓
9
10 double val = compute(x, y + 1); ✓
11
12 System.out.println(val);✓
13 }
14
15 public static double compute(int x, int y) {
16 int z = y;
17 y = x;
18 x = z;
19
20 System.out.println("x" + y + z);
21
22 return Math.pow(x, y);
23 }
24 }
```

~~method : main~~

| name       | value |
|------------|-------|
| int x      | 2     |
| int y      | 3     |
| double val | 16.0  |
| return     |       |

5!

~~BBSh~~ ~~X32~~

X32

~~0.0~~

X24

16.0

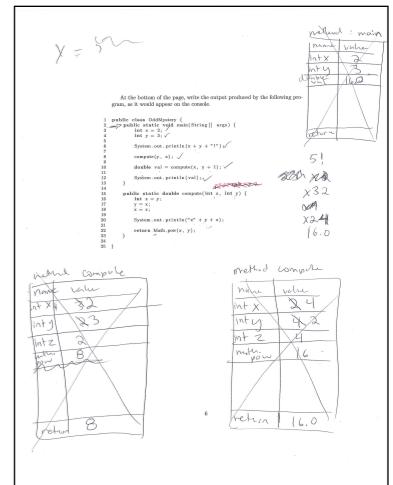


# Strategy

Benjamin Xie, Greg Nelson, and Andrew J. Ko (2017). An Explicit Strategy to Scaffold Novice Program Tracing.SIGCSE.

Benji Xie

- 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)



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.

# 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

Schraw, Crippen,, & Hartley (2006). Promoting self - regulation in science education. *Research in Science Education* 36, 1-2, 111 - 139.

- 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

Li, P., Ko, A.J., & Zhu, J. (2015)  
What Makes a Great Software Engineer? ICSE.



- 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



Dastyni Loksa

- 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



Loksa, D., Ko, A.J. (2016) . The Role of Self-Regulation in Programming Problem Solving Process and Success. ACM ICER.

Dastyni Loksa

- 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?

Loksa et al. (2016).  
Programming, Problem Solving,  
and Self-Awareness: Effects of  
Explicit Guidance. ACM CHI.



Dastyni Loksa

- 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

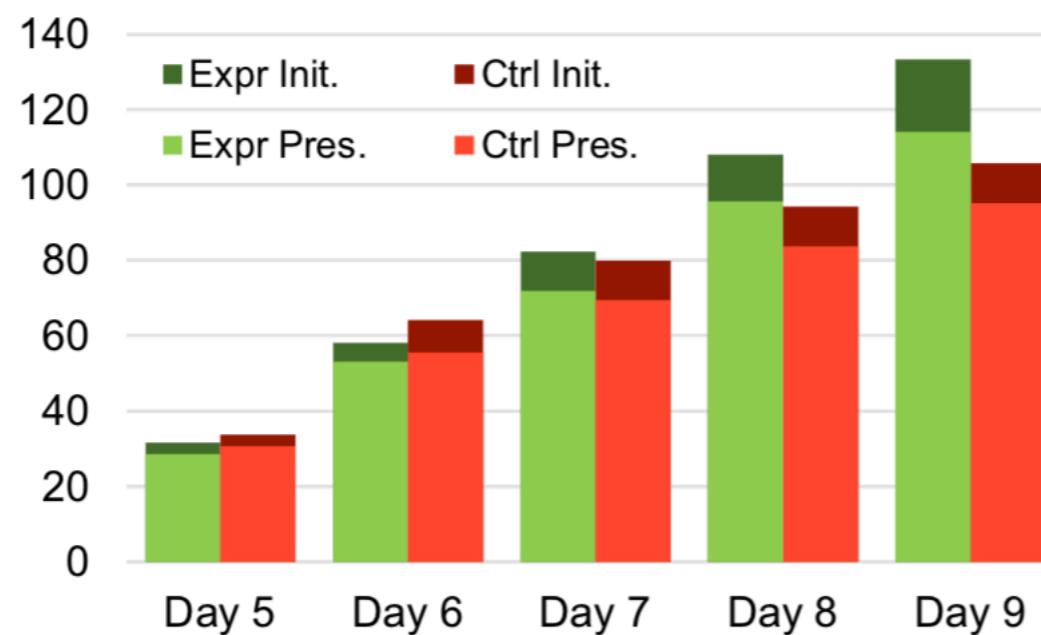
# Yes!

Loksa et al. (2016).  
Programming, Problem Solving,  
and Self-Awareness: Effects of  
Explicit Guidance. ACM CHI.

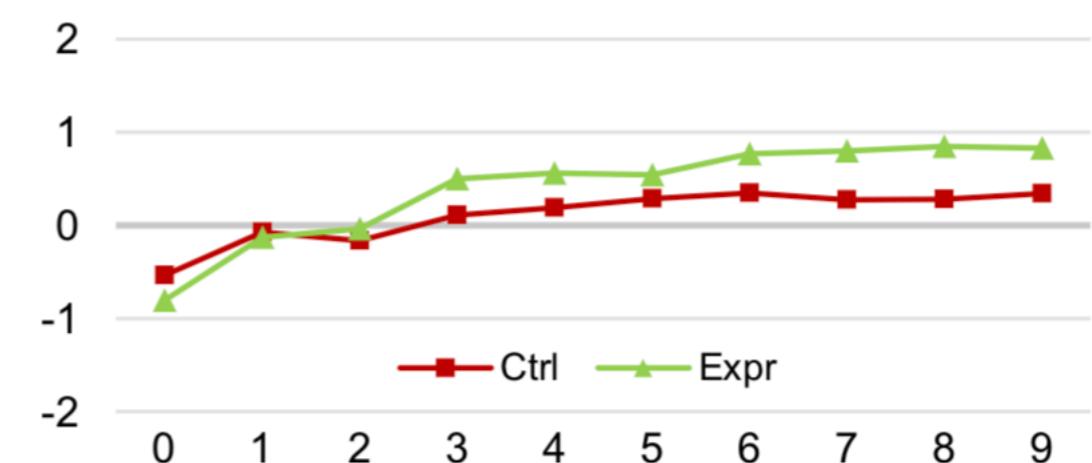


Dastyni Loksa

More productive



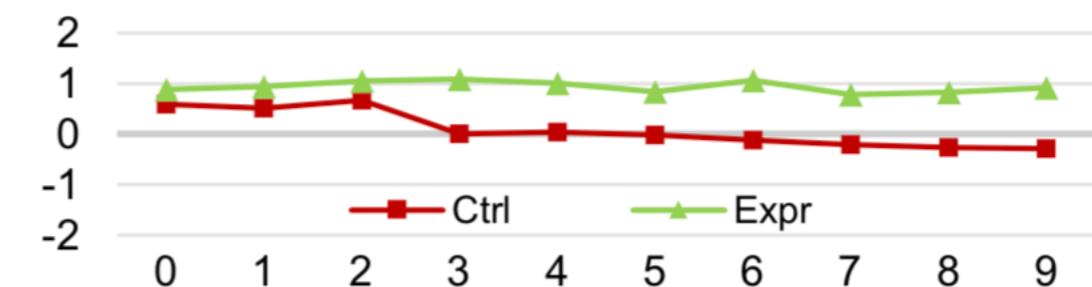
Higher programming self-efficacy



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



Min Li

Benji Xie

- 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

Dastyni Loksa Thomas LaToza

- 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



Mike Ernst   Kyle Thayer

- 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

Ada | DEVELOPERS ACADEMY

MEMBER  
**CsforALL**  
CONSORTIUM



coursera

CODE

COMPUTING AT SCHOOL  
EDUCATE • ENGAGE • ENCOURAGE  
Part of BCS, The Chartered Institute for IT

csnyc

KHANACADEMY

SCRATCH

CSEd  
WEEK



amazon

Microsoft

facebook

Google

accenture

salesforce

# Can't do it alone...

- Many great faculty contributing to computing education research from PL, Software Engineering, and HCI. **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



Microsoft



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!