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

Greg L. Nelson
University of Washington
Allen School, DUB Group
Seattle, Washington 98195
glnelson@uw.edu

Benjamin Xie
University of Washington
The Information School, DUB Group
Seattle, Washington 98195
bxie@uw.edu

Andrew J. Ko
University of Washington
The Information School, DUB Group
Seattle, Washington 98195
ajko@uw.edu

ABSTRACT

What knowledge does learning programming require? Prior work has focused on theorizing program writing and problem solving skills. We examine program comprehension and propose a formal theory of program tracing knowledge based on control flow paths through an interpreter program's source code. Because novices cannot understand the interpreter's programming language notation, we transform it into causal relationships from code tokens to instructions to machine state changes. To teach this knowledge, we propose a comprehension-first pedagogy based on causal inference, by showing, explaining, and assessing each path by stepping through concrete examples within many example programs. To assess this pedagogy, we built PLTutor, a tutorial system with a fixed curriculum of example programs. We evaluate learning gains among self-selected CS1 students using a block randomized lab study comparing PLTutor with Codecademy, a writing tutorial. In our small study, we find some evidence of improved learning gains on the SCS1, with average learning gains of PLTutor 60% higher than Codecademy (gain of 3.89 vs. 2.42 out of 27 questions). These gains strongly predicted midterms (R^2=.64) only for PLTutor participants, whose grades showed less variation and no failures.

KEYWORDS

knowledge representation, program tracing, notional machine

1 INTRODUCTION

Programming requires many complex skills, including planning, program design, and problem domain knowledge [47, 48, 74]. It also fundamentally requires, however, knowledge of how programs execute [13, 44, 49, 55, 86]. Unfortunately, many learners still struggle to master even basic program comprehension skills: two large multinational studies show more than 60% of students incorrectly answer questions about the execution of basic programs [45, 56].

Teachers and researchers have attempted to address learners' fragile knowledge of program execution in diverse ways, creating [37, 51, 62, 63, 65] or changing languages [15, 20, 41, 69, 73] and building writing [17, 39, 68] and visualization tools [29, 34, 34, 57, 81, 87, 91]. Pedagogy has also evolved, reordering [23, 61, 80, 84, 85] and changing what is taught [14, 50, 72], refining worked examples [58], explicitly teaching problem solving [48, 61] and program design [27], and exploring a discovery pedagogy [46].

Most of these diverse approaches have been evaluated in a *writing*-focused pedagogical context. People receive instruction on a programming construct's syntax and semantics, practice by writing code, then advance to the next construct (roughly a spiral syntax approach [76]). In contrast, little prior work has explored a *comprehension*-first pedagogy, teaching *program semantics*—how static code causes dynamic computer behavior—*before* teaching learners to write code. Prior work proposes some approaches and curricular ordering [6, 21, 38, 76, 79] but lacks implementations and evaluations on learning outcomes.

This leads us to our central research question: *What effects does a comprehension-first and theoretically-informed pedagogy have on learning program comprehension and writing skills?*

We argue that comprehension-first is not just another pedagogical strategy, but instead requires a new conceptualization of what it means to "know" a programming language. We therefore contribute a theory of program tracing knowledge, derived from abstract control flow paths through a programming language interpreter's execution rules. Based on this theory, we contribute the first comprehension-first pedagogy that teaches and assesses tracing skills for a Turing-complete portion of a programming language, without learners writing or editing code and without requiring them to infer program behavior from input and output. Based on this pedagogy, we built an interactive tutorial, PLTutor, to explore preliminary answers to our research question. We then conducted a formative experimental comparison of PLTutor and a writing-focused Codecademy tutorial, investigating the effects of a comprehension-first pedagogy on CS1 learning outcomes.

2 RELATED WORK

Prior work on *tools* has enabled comprehension-first pedagogy, but has lacked high-quality evaluations of its effects on comprehension and writing. For instance, program visualization can help learners comprehend detailed low-level operations in programs [57] or low-level visual program simulation [81], but these have not been applied in a comprehension-first pedagogy. UUhistle has some technical features that would support a learner independently following a comprehension-focused curriculum (e.g., by choosing from a list of programs with instructor annotations), but that has not been evaluated [81]. Bayman et al. compared the effects of combinations of syntax and semantic knowledge in diagrams on writing and comprehension practice [5]. Computer tutors tried low-level evaluation exercises, but sequenced with writing exercises; this had no benefits for writing skills vs. only having writing exercises [2].

Other work implements a comprehension-first pedagogy but has limitations in their evaluations or the breadth of what they teach. Dyck et al. only assessed writing ability after using a computer-based manual with rudimentary assessments of program tracing [25]. To our knowledge, the only self-contained tool that implements a comprehension-first pedagogy is Reduct [3], an educational game that teaches operational semantics. Its design focuses on engagement, does not include a textual notation, does not cover variables, scope, loops, and nested or ordered statements, and shows some inaccurate semantics for Javascript. Its evaluation also lacks a validated assessment, a pre-test, or comparison to other tools.

Prior *theoretical* work on program comprehension spans both cognitive studies of program comprehension and pedagogical approaches to teaching program tracing. Neither of these areas have a theory for what comprehension knowledge is.

Researchers since the 1970s have theorized about *how* people comprehend code [11, 32, 60, 75, 77]. This research has developed cognitive theories of comprehension processes, describing perception [32, 60], mental structures [11, 19, 55], and novices and experts differences [11, 19, 53, 77]. These theories facilitate questions about perceptual strategies that novices and experts use to comprehend code, what features of code experts use to comprehend code, and what kinds of knowledge experts use. These theories focus on comprehension process and behavior; we contribute a theory that 1) specifies what knowledge people must have to be able to execute these processes and 2) formally connects this knowledge to syntax.

Prior work makes key distinctions between writing, syntax, and semantic knowledge (for example, [5, 32, 52, 54, 77]) but lacks formal connections across levels of semantics knowledge and a principled way to derive it. Mayer divides semantics into micro (statement) and macro (program) levels, and describes *transactions* at a sub-statement level as action, object, and location [52, 54]. However, these natural language descriptions lack connections to the sub-expression parts of the code that causes them.

Berry generated animated program visualizations from operational semantics, a formalism used by PL researchers for proofs and reasoning [7]. We instead propose the knowledge needed to learn program tracing is not the *abstract formal* semantics for a language, but the semantics as actually implemented in a language's interpreter, mapped to a notional machine to facilitate comprehension.

Within CS education, early approaches utilized writing tasks that required program comprehension, focusing on teaching syntax and semantics one language construct at a time, while gaining writing knowledge about the construct [74, 76, 85]. In contrast, around 1980 Deimel et al. [21] and Kimura [38] briefly proposed, without evaluating, a comprehension-first curriculum starting with running programs and looking at I/O to infer semantics. Major literature reviews fail to mention their existence, even since the 1990s [67, 74, 90]. Both of these pedagogies lack a definition of the knowledge learned, making one unable to determine when language features have been fully covered. They also lacked assessment methods beyond I/O prediction, making it hard to give targeted practice, diagnose misconceptions, and correct them with feedback.

3 PROGRAM TRACING KNOWLEDGE

The critical gaps in prior work are in both tools and theory. No theories describe the knowledge necessary for program tracing skills, and no tutorials or visualization tools have been designed or evaluated with a comprehension-first pedagogy. Therefore, in this section, we present a theory of what program tracing knowledge is and build upon it in later sections to inform the design and evaluation of a tutor that teaches program tracing knowledge.

Our first observation about tracing is that inside the interpreter that executes programs for a particular programming language (PL) is the exact knowledge required to execute a program. It just happens to be represented in a notation designed for *computers* to understand rather than people. For example, in many PL courses, students write an interpreter for a calculator language; it reads text such as `2+3` and executes that code. The interpreter contains definitions of execution rules like `if(operator == "+") { result = left + right; }`. We argue that this logic is the knowledge required to accurately trace program execution.

Unfortunately, because this logic is represented as code, it is not easily learned by novices. First, few materials for learning a language actually show the interpreter's logic explicitly. Moreover, even if this logic was visible, novices would not likely understand it because they do not understand the notation that it is written in. This provides a key theoretical explanation for why learning to trace programs is hard—this notational barrier can only be overcome once you understand programming languages, creating a recursive learning dependency.

Execution rule logic, however, is not alone a suitable account of the knowledge required for tracing. Our second claim is that to know a programming language, learners also must be able to *map* these execution rules to the syntax and state that determines what rules are followed and in which situations. Therefore, knowledge of a PL is also the *mapping* between syntax, semantics, and state.

To illustrate this mapping, Table 1 shows an interpreter in pseudocode, showing the three conventional stages of transforming program source code into executable instructions, for a simple JavaScript-like expression `x == 0`. The first stage translates characters into tokens; the second stage translates tokens into an abstract syntax tree (AST); the final stage translates the AST into machine instructions that ultimately determine program behavior. We argue that learners do not need to understand these stages themselves, but rather that they need to understand *each path* through these stages that map syntax and state to behavior. We show one example of a path underlined in Table 1, which specifically concerns the `0` in our `x == 0` expression, showing its translation from character to token to a machine instruction that pushes `0` onto an accumulator stack for comparison to `x` by the `==` operator. This simple mapping rule—that a numerical literal like `0` is a token in an expression that results in a number being pushed onto a stack for later comparison—is just one of the execution rule paths; we argue learners must understand all possible paths to know the whole language.

Some rules have one path (for example, the `0` in `x==0` only has one in our example language), but some execution rules have multiple control flow paths, depending on the code or runtime state involved. For example, `if` statements in many imperative languages can optionally include an `else` statement. If an AST has an `else`

Stage	Transformation Rule (Pseudocode)	Example Output (input for next row)
1. Tokenize	Any number => Number	Name(x)
	Operator => Op	Op(==)
	Variable name => Name	Number(0)
2. Parse	Parse(*toks*) =>	*AST*(
	AST(Parse(*toks*$_L$), Op, Parse(*toks*$_R$))	*AST*(Name(*x*))
	ELSE *AST*(Number)	Op(==)
	ELSE *AST*(Name)	*AST*(Num(0)))
3. To Machine Code	Code(*AST*) =>	
	IF *AST*$_1$ Op(==)*AST*$_2$:	LOOKUP_AND_-
	Code(*AST*$_1$)	PUSH "x"
	Code(*AST*$_2$)	PUSH 0
	DO_EQUALS_OP	DO_EQUALS_OP
	ELSE IF Number(*n*): PUSH *n*	
	ELSE IF Name: LOOK_UP_AND_PUSH name	

Table 1: An interpreter with pseudocode notation. For input `x==0`, the example column shows instances of tokens in the 1st row, ASTs on the 2nd, and instructions on the last. An example semantic path is underlined for a PUSH instruction for the number token.

statement, the mapping is different, because the end of the `if` block must include a jump past the `else` block. Learners must understand these syntax-dependent branches in compilation and execution.

We therefore view the set of *all* possible paths through all of the compilation and execution rules in all of a programming language's constructs as the knowledge required to "know" a PL. This symbolic representation preserves the fidelity of the knowledge because we derive the knowledge directly from the interpreter. As we will show, we can then make these abstract paths concrete by presenting programs that cover these paths.

4 PEDAGOGY AND PLTUTOR

In this section we propose a comprehension-first pedagogy that embodies our theory of program tracing knowledge. Our design focuses on helping people learn 1) PL semantics and 2) program tracing skills; our prototype focuses specifically on JavaScript. The central pedagogical strategy in PLTutor is to build upon the experience of using a debugger, but 1) allow stepping forward and back in time, 2) allow stepping at an instruction-level rather than line-level granularity, and 3) interleave conceptual instruction about semantics throughout the program's execution.

Figure 1 shows the experience of using PLTutor to observe program execution over time. For example, Figure 1.a shows the path underlined in the first and second row of Table 1 for the token `0` in `x==0`, and Figure 1.b shows the third row in Table 1. The change to machine state displays on the next time step (see `0` on top of stack at Figure 1.c). This representation addresses the notational barrier to accessing the information in Table 1. As a side note, for brevity Figure 1 assumes `x` has value zero (e.g. at first time step).

The next two subsections discuss in detail how our pedagogy (partially implemented in PLTutor) teaches learners both the syntax, state, and semantics mapping and program tracing.

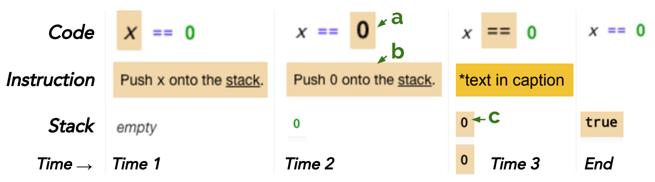


Figure 1: Stepping through three semantic paths covered by the example program `x==0`. *text is "Pop 0 and 0 off the stack, compute `0==0`, and push the result onto the stack".

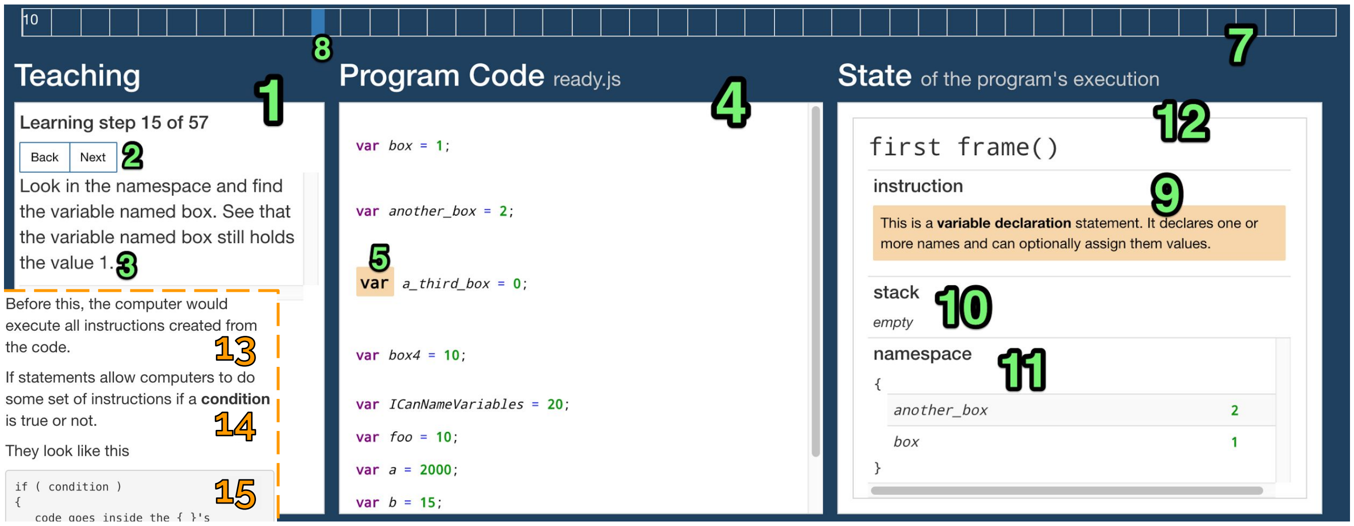
4.1 Programming Language Semantics

Now we theorize how learning may occur within PLTutor's pedagogy. We start with an example in another domain, then return to learning PL semantics. We start from a theory of learning for "a causes b" relationships (also called causal inference). For instance, a child may see a switch flipped then a light goes on, and they may infer the switch caused the light to go on, just from one example. How can this rapid learning occur? A theory of causal inference identifies three key enablers for this learning: *ontology* (differentiating/recognizing entities and their causal types), *constraints* on relationships (how plausible are relationships), and *functional forms* of relationships (from how effects combine and compare, to specific forms like $f = ma$) [28]. For the light example, *ontology* includes a light and a switch as mechanical entities, *constraints* includes closeness in time (switches tend to control things quickly), and *functional form* may include knowing the switch up position activates ("turns on") whatever it controls. With these enablers, causal learning then occurs by *observing examples of the causal relationships* [28].

We theorize learning PL semantics as grasping the *causal relationships* from code to machine behavior. To enable causal learning, our key insight is to convert the abstract path representations from the prior section into a concrete causal visualization, showing the causal relationship between *tokens*, *machine instructions*, and the machine state changes caused by the instructions.

Now we summarize how our pedagogy provides the three key enablers for causal learning, for program semantics; afterwards we will discuss them in a particular implementation (PLTutor). To provide *ontology*, we start learning with conceptual content describing entities (code, instruction, and machine model), show them in the visualization, and explain how to recognize them in the visualization. To provide *constraints* on relationships, we provide conceptual content emphasizing the mechanical relationship (to avoid the "computer as person" misconception [66]; causal learning theory gives us a new lens for why this misconception may be so damaging - having the wrong relationship misleads the causal learning process). We also give *functional forms* via syntax highlights for the token part, text descriptions of the instruction part, and the change in machine model state over each small time step.

To create this causal model, we extract patterns from interpreter implementations. We introduce two abstractions—the *instruction* and the *machine model*—to simplify this knowledge, while retaining fidelity. Typically, paths through the interpreter end with manipulating the state of the program being executed (either by generating machine code or with PL statements (for an implementation hosted in another language)). We encode these state changes as one or more instructions for a machine model (which serves as a model



Learning step 15 of 57

Look in the namespace and find the variable named box. See that the variable named box still holds the value 1.

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
```

```
var box = 1;

var another_box = 2;

var a_third_box = 0;

var box4 = 10;

var ICanNameVariables = 20;
var foo = 10;
var a = 2000;
var b = 15;
```

first frame()

instruction

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

stack

empty

namespace

```
{
  another_box   2
  box           1
}
```

Figure 2: PLTutor showing an early lesson on variables: left, 1) the learning content and assessment area with 2) stepping buttons and 3) conceptual instruction; 4) program code with 5) token-level highlighting to show what caused the instruction; machine model: 7) timeline of instructions executed for the program 8) current step, 9) current instruction's description, 10) stack, 11) namespace, 12) call frame. Lower left inset shows content for conceptual instruction for a later `if` lesson.

notional machine [24]). The interpreter produces a list of instructions, which are then sent to the machine model. This separates the state of the program's execution entirely into the machine model. Using the machine instruction as a bridge between syntax and state transformation, we can connect the path through the interpreter to machine state changes.

We can also connect instructions back to the code that causes them. These connections follow the semantic path, going back to the program code via the tokens that caused values in them. For example, for the path underlined in Table 1, the `PUSH 0` instruction connects to the `0` in the source code via the token Number(0); the `LOOKUP_AND_PUSH "x"` instruction connects to the x in Name(x).

Figure 2 shows PLTutor, our web-based prototype that visualizes these causal connections. In this figure, PLTutor is visualizing the beginning of a JavaScript variable declaration statement, which is mapped to the *var* keyword in 2.5. The machine state is on the right with the machine instruction shown at 2.9.

In PLTutor, our pedagogy has learners *observe examples of the causal relationships* by stepping forward in time in the program's execution, one instruction at a time. A single step changes the token being highlighted, the current instruction, and machine state. For example, Figure 1 shows three steps for the JavaScript program `x == 0`. Our prototype supports the full semantics of JavaScript, providing mappings between all constructs in the language and the machine instructions that govern JavaScript program execution.

Within this representation, our pedagogy identifies a unique causal role for the *instruction* between code and the machine model. The instruction *makes visible* the causal relationship between syntax and machine behavior. The instruction also provides *constraints* on relationships between code and machine state changes (a parser generates instructions from code, which is the only way code causes machine behavior). We also designed instructions to provide a set of *functional forms* that simplify and localize relationships; all instructions only 1) push onto an accumulator stack (either from the literals in the code or the namespace) 2) pop values from the stack, do a local operation only with those values, and push the result onto the stack, 3) set a value in the namespace, 4) pop a value and change the program counter (for conditionals, loops, and functions), or 5) clear the stack (which we map to the `;` token). These forms provide further *constraints* on relationships: for example, values from code tokens only change the stack, namespace values only come from the stack, and instructions only come from the code. These constraints and functional forms should help learners' causal inference [28].

Besides this causal role, the instruction also makes visible how a language executes syntax. Without it, stepping through the interpreted program line by line requires understanding how the computer navigates the program code notation. In contrast, stepping through a list of instructions only requires moving forward and backwards through an execution history and comprehending changes to machine state (see Figure 1).

PLTutor also conveys constraints, functional forms, and ontology by showing natural language explanations of the actions instructions are taking. PLT reinforces functional forms by showing an instruction's form as a description filled in with concrete values (see Figure 2.9, which explains how a variable declaration begins). PLT also shows learning content at 2.3 throughout the lesson.

To scaffold causal inference, the curriculum starts with *ontology*, with 5-10 minutes of conceptual instruction about the computer, code, state, the interpreter, instructions, and machine model. It provides ontology by describing the entities and how they are shown in the interface; for example, "The namespace is where variables are stored. This is like a table with two columns...". It also gives constraints on their relationships; for example, "In general, the list of instructions does not change as a computer executes a program." This information also serves as organizing concepts [10].

4.2 Program Tracing

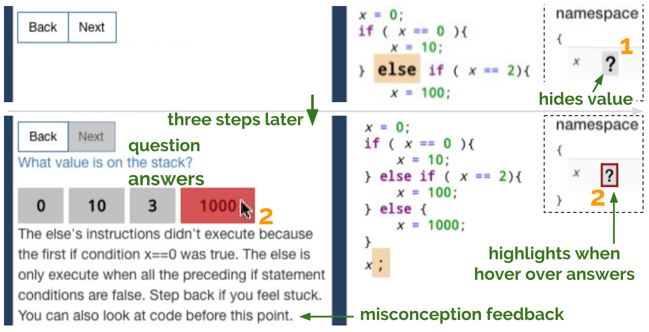
Our theory of program tracing knowledge suggests a general approach for a pedagogy: show a faithful representation of each interpreter path, assess the learners' knowledge of each path, correct misconceptions, and cover all the paths for completeness.

To cover most of the paths, PLTutor uses a fixed, sequenced curriculum of example programs. Instead of stepping through execution steps directly as in prior work, each program has a list of *learning* steps, specifying 1) the learning content or assessment to show and 2) which execution step to show. This decoupling allows the curriculum to navigate anywhere in the program's execution when the learner advances to the next learning step. There are three types of learning steps in the tutorial: *conceptual* steps (show conceptual instructional content), *execution* steps (show an instruction executing), and *assessment* steps (prompt a learner to fill in values, described shortly). Learners advance forward or backward through these steps by clicking the "Back" and "Next" buttons (Figure 2.2) or using keyboard arrow keys. Learners may also drag the bar (Figure 2.8) to scroll through execution steps. On the final learning step of a lesson, a "Next Program" button appears, which navigates to the next program in the curriculum. This contrasts with prior work, in which learners must find and choose a next program for themselves, constructing their own curriculum from a menu [81].

Our pedagogy interleaves these three types of steps (conceptual, execution, and assessment) through a program's execution. To illustrate this, we describe the learning steps for a first lesson for a construct (such as `if`). These follow a pattern: reference relevant prior knowledge, contrast what is new ("Before this..." see Figure 2.13), present the goal of the construct ("If statements allow...", Figure 2.14), present the syntactic pattern (see 2.15), then scaffold learning strategies ("step through to see how it works", and, later, prompts mental execution "What do you think this next if statement will do? Read through it and think, then step through it."). Where possible, steps introduce constructs with "equivalent" programs, based on similar instructions or state changes. For instance, in our arrays lesson, `is_day_free_0=false; is_day_free_1=true;` precedes `is_day_free = [false, true];` this may help transfer and provide *constraints* for causal inference. After such code, steps show an example path and a low-level assessment (described shortly), then address common misconceptions in turn by: 1) showing learning content against it, 2) executing a counter-example, 3) stepping through code with assessments for the misconception.

Besides the ordering of steps within each program, the stepping interface scaffolds perception of conceptual and execution information. On steps with learning content, learners experience a slight 1-second pause before they can advance, to encourage reading. At key points in the middle of a program's execution, learning steps stop advancing execution to show conceptual instruction. For example, PLTutor pauses in the middle of a condition evaluation to describe what is happening when first learning `if` statements. Execution steps show causality temporally, and having many steps shows many examples of the causal relationships, as recommended by [28]. Their granular, sub-expression level of detail may help structure inferences from these lower level steps into higher level inferences across multiple steps and lines of code [28].

To aid higher-level causal inferences, PLTutor assesses knowledge of each path at multiple levels. For the single execution step level, it navigates to a step and hides a value for the learner to fill in. Figure 3 shows an assessment across multiple steps. Using value hiding and the learning step's ability to control what execution step is shown, this enables scaffolded assessments, showing some instruction or code navigation, across multiple levels of granularity, including the simple effects of one execution step (done in prior work [82]) to showing the resulting state of many execution steps as if it were one large step. PLTutor also scaffolds links between assessment question phrases and machine state by showing which question to fill in; hovering over any answer shows a box around the corresponding value (see Figure 3.2). It also shows misconception feedback for inaccurate answers (see bottom Figure 3).



```
x = 0;
if ( x == 0 ){
    x = 10;
} else if ( x == 2){
    x = 100;
```

What value is on the stack?

The else's instructions didn't execute because the first if condition x==0 was true. The else is only execute when all the preceding if statement conditions are false. Step back if you feel stuck. You can also look at code before this point.

```
x = 0;
if ( x == 0 ){
    x = 10;
} else if ( x == 2){
    x = 100;
} else {
    x = 1000;
}
x ;
```

Figure 3: Assessments scaffold state, hiding values with a ? (see 1), so learners mentally execute semantics to answer. The assessment shows three steps later. This allows assessing from the step to multiple line or program level granularity, without requiring navigation restrictions to hide values.

In addition to introduction and practice lessons, our pedagogy includes review lessons, which apply constructs together and occur after the end of operators, conditionals, and the loops material. These lessons describe how constructs can be used with each other. They contrast larger "equivalent" code segments, justifying language features by appealing to "good" properties of code like readability, brevity, and ability to modify or reuse. We include these integration lessons to increase retention and motivate learning by showing how constructs are used together for actual problems [10], and to connect knowledge to the design goals of the language.

4.3 PLTutor Limitations

PLTutor only partially implements the principles we have discussed for a comprehension-first pedagogy. It covers all the paths in our JavaScript interpreter except for strings, I/O, objects, some unary and binary operators (like `--` and modulus), and error paths (like invalid variable names or syntax errors). For example, we do not show the failing examples required to fully specify variable naming patterns. We expect later writing pedagogy to cover them, and language runtimes make them visible with error messages.

While PLTutor's assessments directly or indirectly cover much of each semantic path, it leaves some out and does not fade scaffolding entirely. It directly assesses machine state changes by having users fill in values in the machine state via linked assessments. It indirectly assesses control flow via simple value changes (which depend on the variable assignment path in earlier lessons); Figure 3 shows one. PLTutor does not assess ontology directly or fully remove scaffolding in its curriculum; for example, it always shows machine state, instruction execution steps, and allows stepping.

Finally, at the time of our evaluation, PLTutor was very much a research prototype. When we evaluated it, it had usability issues and the environment made little effort to engage learners, introducing numerous barriers to sustained engagement and learning.

5 EVALUATION

What effects does a comprehension-first and theoretically-informed pedagogy have on learning code comprehension and writing skills? To investigate, we conducted a formative, block-randomized, between subjects study comparing a comprehension-first tutorial, PLTutor, with Codecademy [16], chosen for its traditional writing-focused spiral pedagogy [76] and quality from 4 years of curriculum refinement. We label the PLTutor condition *PLT* and the other *CC*.

5.1 Method

Our inclusion criteria were undergraduates that had not completed a CS1 course in college and had not used a Codecademy tutorial. We recruited students starting a CS1 class that followed a procedural-first writing pedagogy using Java [72]. Participants came to a Saturday 10:30am–6pm workshop, took a pre-survey and a pre-test, used a tutorial for 4.33 hours, and then took a post-test and post-survey. As a pre/post-test we used the SCS1 [26, 64], a validated measure of CS1 knowledge. Both surveys used validated measures for fixed vs. growth mindset [8] and programming self-efficacy [71]. The pre-survey also measured daytime/chronic sleepiness using the Epworth Sleepiness Scale [4], as prior work argues these affect learning [18, 22, 33, 59, 70].

At the first two lectures and via email, we advertised the study as a chance to excel in the class, potentially biasing towards motivated and at-risk students. Overall, 200 of 988 students responded to an emailed recruitment survey and 90 met our inclusion criteria. Using this survey, we block randomized [31] participants into a condition using hours of prior programming, self-reported likelihood of attendance, age, and gender, then invited subjects. From these blocks, we randomly invited participants from each block, ultimately having 41 attend the workshops. After confirming attendance by email, we sent subjects the room for their condition.

The two instructors of the one day workshops followed a written experimental protocol and coordinated to make any necessary day-of changes jointly. They then showed an introductory SCS1 test directions video, gave the pre-test, then showed video instructions for their condition's tutorial and stated students would have 4 hours and 20 minutes to learn the material. They served lunch during the tutorial period at 1PM. After a 10-minute break following the tutorial, students had 1 hour for the SCS1 post-test and could start the post-survey when done. The instructors then served dinner.

We operationalized learning outcomes by proxying program comprehension with SCS1 score and writing skills with midterm grade. Learning gain (posttest score−pretest score) is noisy because it combines pre and post test measurement error [9]; we also counted per-question and per-individual performance from incorrect on pre-test to correct on post-test (which we call *FT*, for false-to-true), as well as likely prior knowledge as correctly answering a question on both the pre and post test (*TT*, for true-to-true). We defined *learning capacity* for a person as (the # of questions not left blank)-TT and *learning capacity* for each question as (# of people that did not leave the question blank)-TT. We defined *LCL*, the % learned that could learn, as *FT* / learning capacity (like normalized gain in [83]).

The SCS1 system randomly lost some tests; we dropped those participants, reducing sample size to 18 in PLT and 19 in CC. We separated novices (operationalized as less than 10 hours of self-reported experience and no prior CS class) from experienced students (had prior CS class or >10 hours of self-reported experience).

5.2 Results

Despite random assignment, we found differences that may have confounded measures of learning gains. We analyzed these and other differences by default with the Wilcoxon rank-sum test for non-normal data. We add and note a t-test when Shapiro-Wilks's normality test had $p > 0.1$ for each group (still has low power for our sample size). Pre-test SCS1 differences between the two conditions were large and marginally significant (CC−PLT mean=1.65, W $p < .111$, t-test $p < 0.058$), but self-reported prior programming experience and mindset did not differ significantly.

While many individuals in both groups achieved higher SCS1 scores, comparing *within* each condition, only PLT's post-scores were significantly different from its pre-scores (p<.0044) (for CC p<.089). Figure 4 shows descriptive statistics; for comparison, students near the end of a CS1 course score from 2 to 20, $m = 9.68, SD = 3.5$ [64]. Comparing conditions, PLT had higher individual *FT*, with Cohen's d=.59 (p<.12, t-test p<.075); for *learning gain*: Cohen's d=.398 (p<.41, t-test p<.24); for the % learned that each person could learn (*LCL*): d=.34 (p<.39, t-test p<.31). To control our analysis for other variables, we tried to fit post-score with linear and binomial generalized linear models, but residuals strongly violated modeling assumptions.

When we consider the specific questions in the SCS1, PLT outperformed CC on 37% of questions and CC outperformed PLT on 22%, based on between group difference of $\geqslant .1$ in LCL (the % of people who got the question right that did not already get that question right on the pre-test). Figure 5 shows questions sorted from left to right by this LCL difference, with * at bottom for non-overlap of their 95% confidence intervals of the probability p of a binomial distribution estimated from x=FT and n=capacity (Wilson [12]).

CC did better for some code completion (Q18,25,21) and writing conceptual questions (e.g., Q1: When would you not write a for loop... Q6: Imagine writing code for each task - can you do it without conditional operators...). It also did better on topics missing from PLT's curriculum like strings (Q18) and a tracing question with modulus (Q19). In contrast, PLT did better for tracing (Q2,23,3,12,24,14,8), a tracing conceptual question relying on subexpression detail (Q10), and complex code completion questions (Q13,Q26). PLT also did better on topics missing from CC's curriculum like recursion (Q14 and even Q24 involving strings). PLT also did better on a tracing question with strings that only used array syntax and semantics (Q15).

PLT had less variation and a more normal distribution for later writing outcomes compared to CC (see Figure 4.f, 4.e). Shapiro-Wilks normality test rejected CC's midterm distribution (p<.013) but not PLT's (p=.4723). For PLT, midterm fit a linear model on each of: *learning gain*: (adj-R^2=.6469 (95% CI: .28, .84 by [35, 36])

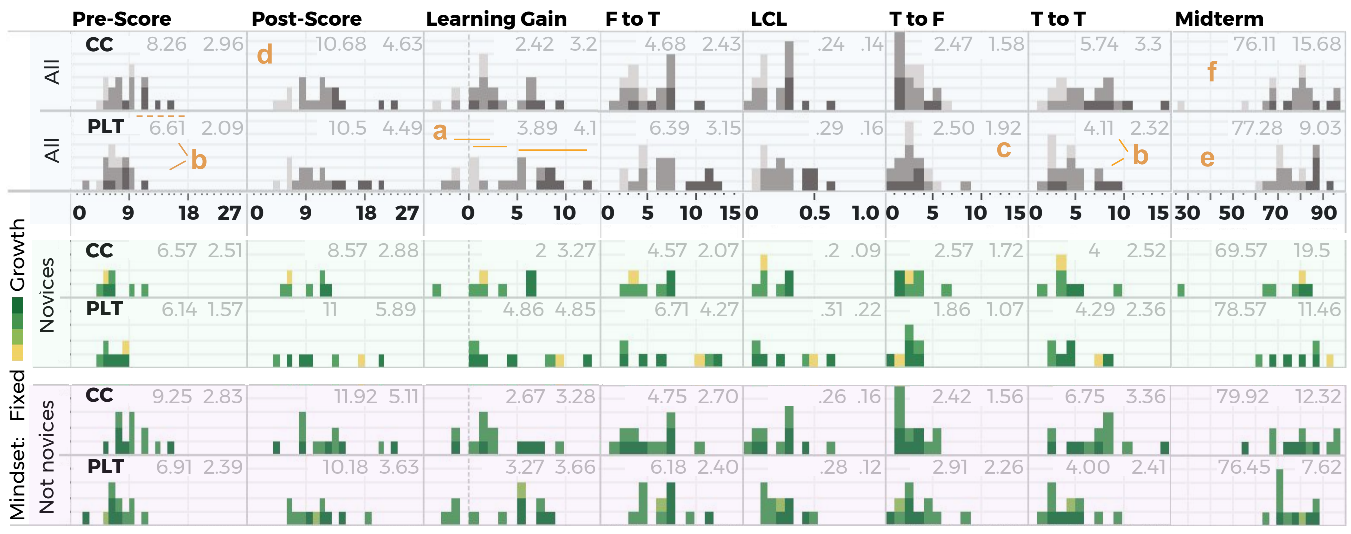


Figure 4: Histograms (with mean then *SD* inside) by condition (top) colored by *post-score* (dark grey: ⩾ 13 (about a *SD* above the mean of students finishing CS1 from [64], grey: within a *SD*, light grey for below), then experience (mindset color).

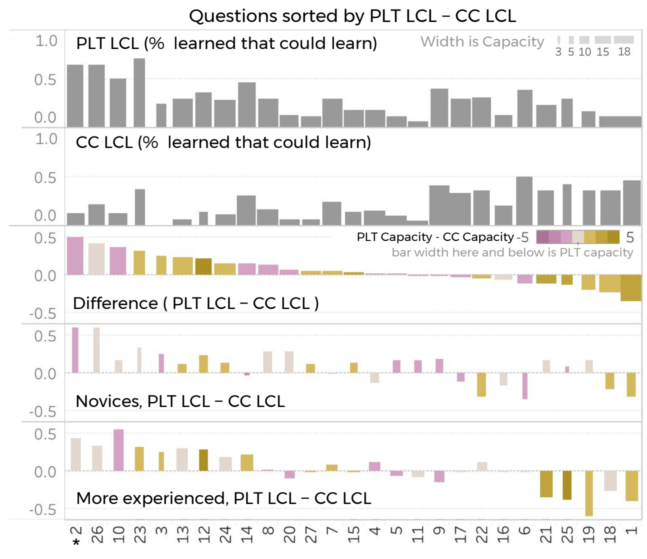


Figure 5: Per-question statistics, ordered by PLT's LCL minus CC's LCL. Bar width shows capacity.

p<.00004 residual SE=5.37), *post-score*: (adj-R^2=.5502 p<.00025), *LCL*: (adj-R^2=.53 p<.00037), and *FT*: (adj-R^2=.4983 p<.00064). For CC, only a post-score model had significance (adj-R^2=0.1784, p<.041, SE 14.21). PLT had no midterm failures (vs. 2 in CC). Midterm average did not differ significantly between PLT or CC (see 4.e), or those in the recruitment group that met the inclusion criteria and did not participate, midterm $m = 72.1, SD = 22.1$ (n=38).

To partly check the validity of midterm as a proxy for writing skill, we offered $6 for a photo of the midterm (with per question grades) to those that met the inclusion criteria (90) and got 17. A linear model from total score predicted the writing part (adj-R^2=.85) much better than the other parts (adj-R^2=.61), suggesting total midterm score varies fairly closely with the writing portion.

As a manipulation check, two months after the workshops we offered post-midterm tutoring. In each session, before tutoring, we conducted a think-aloud interview for tracing mental model granularity by prompting learners to "Underline the code as the computer sees it, then describe what the computer does" for Java versions of program B2 & G3 from [78]. Learning gains and midterms mostly increased with more sub-expression tracing (except P5). Two participants (P1 and P2) responded from CC. P1 had 2 TT, 2 FT, and a 54 midterm; her tracing model had some sub-expression (but not for control structures and had an early loop exit error); P2 (2 TT, 6 FT, 79 midterm) was mostly line-level but separated assignment (like y=1+1;) into 3 steps: the left side y, then =, then the rest 1+1;. Four participants (P3 to P6) responded from PLT. Compared to P2, two showed more but non-uniform sub-expression with self-caught misconceptions, P3 (4 TT, 7 FT, 72 midterm) and P4 (5 TT,7 FT, 77 midterm). P5 (8 TT, 12 FT, 86 midterm) had a line level model. P6 had a consistent sub-expression model (3 TT, 11 FT, 85 midterm).

We also asked which tutorial features they remembered, as a gross check on importance or causality. Better learning outcomes mostly increased with more correct and complete feature recall, suggesting they impacted learning. In CC P1 and P2 both recalled writing, exercise feedback, and help; only P2 had the print output. In PLT, P3 recalled learning content and (incorrectly) writing code only, with the stack shown briefly during execution with no stepping controls; the others recalled content, stepping and assessments; for the state display (see right side of Figure 2), P4 had namespace (2.11), P5 had steps bar (2.8), P6 had all except instruction (2.9).

6 THREATS TO VALIDITY

While we made efforts to ensure validity (minimizing confounds, block-randomizing group assignment, measuring confounding factors, avoiding early-riser effects[22], mitigating experimenter bias, etc.), there are still several threats to validity.

Differences between our study and the validation of SCS1 complicate the interpretation of our results. We post-tested within 4.3 hours of the pre-test; a carry-over effect may inflate post-scores (e.g.

remembering questions) [1]. Guessing, especially by novices, may have impacted scores. While the SCS1 is the best publicly available measure, it's validity arguments do not formally generalize i) to novice test takers, ii) in a pretest-posttest context, and iii) as a measure of learning gains. Our measure of writing skills (the midterm) had unknown measurement error and lacks validation.

Motivation differences, participant fatigue, measurement error, unmeasured participant variation, and differences in workshop setting also threaten validity. Internally, instructor variations may favor the Codecademy condition, which had a more experienced teacher. The PLTutor instructor also had to leave the room for 45 minutes to handle a lunch issue. Externally, the study's short duration may create a ceiling effect on learning gains. The study protocol, curricular quality, time-on-task, program domain, and pedagogical and lack of adaptive tutorial features affect results. In particular, in informal interviews, participants reported frustration with repetitive practice in the PLTutor curriculum, which may have reduced engagement and therefore learning.

7 DISCUSSION

We have presented a new theoretical account of program tracing knowledge, a new pedagogy for teaching this knowledge embodied by PLTutor, and empirical evidence of the effects of PLTutor on program tracing and writing skills. These effects included:

- Higher total and question-specific learning gain than Codecademy (overall 37% of questions and 70% for novices).
- Less midterm variation and no failure on the midterm. Learning gains from the tutorial also strongly predicted the midterm, suggesting a strong relationship or shared factors between learning rate in the tutorial and the class.
- More learners who started with low pre-scores had large learning gains (see dark grey in pre-score and *TT* (likely prior knowledge) in Figure 4.b).

Our study suggests greater learning gains for PLTutor compared to Codecademy. PLTutor matched Codecademy's post-scores even with a significant initial deficit (see Figure 4.b). This might just be mean-reversion for Codecademy (guessing on the pre-test with less luck on the post) but true to false shows little to no difference (see 4.c). The other interpretation is that PLTutor brought its less experienced group to parity with Codecademy (see 4.d); if the writing tutorial was better at teaching program tracing, it ought to have magnified initial differences. PLTutor also had more learning at the question level, doing better on 37% vs. 22% for Codecademy (1.68 times more). Question-level differences might come from sampling error, which is hard to model without item response theory parameters for the SCS1. However, these differences always aligned with curricular differences (no recursion in CC, no strings or modulus in PLTutor) and theoretical explanations—for example, writing did better on 3 out of 9 code completion problems, as did PLTutor for 8 out of 13 tracing problems. This supports the interpretation that our results do not just come from noise or guessing differences, though our small study still has threats to validity.

Our empirically strongest result is that PLTutor normalized midterm outcomes. PLTutor had no failures vs. 2 in CC (see 4.f vs. 4.e). With only one early measurement, in our small study only PLTutor learning gains predicted midterm (adj-R^2=.64), among the best of work predicting CS1 outcomes (adj-R^2 .44 to .46 [54, 88, 89]), better even than those using mid-course measures like homework or self-efficacy (adj-R^2 .35, .58, .61) [40, 43, 88]). In pre-score in Figure 4.b, PLTutor also has more dark high post-scorers coming from lower scores vs. CC. This improved equity in outcomes may help scale learning in diverse populations. Future work should confirm this pattern and see if normalizing has the downside of reducing outlier high outcomes (compare far right of 4.e & 4.f).

We also saw learning gains comparable to full quarter or semester long courses with both tutorials in $\sim$4 hours (see Figure 4.a), similar to prior work [42], yet unexplained by prior CS knowledge or traits we measured. Some gains varied from losing to gaining 2-3 points on the post-test, perhaps guessing noise. However, our participants' post-test distribution looked similar to those near the end of CS1 in [64]; this was either genuine learning, recruiting bias that skewed our sample towards more motivated or at-risk students, or test-retest carry-over score inflation. Even extreme learning was not uncommon; in $\sim$4 hours, in PLTutor 4 learners (22%) moved from a below average pre-score to nearly above a *SD* of [64]'s mean (3 (16%) above 13.18), and one from a score of 8 to 20, the maximum from 189 students in [64] (in CC, one 8 to 14 and an 11 to 20 also). In PLT these outcomes continue in the midterm (see dark at 4.e).

What skills or knowledge explains such fast learning without prior domain knowledge? Can we teach it and dramatically improve CS1 and even other CS education? Most learning theories frame learning as hard and time consuming, and transfer as fragile; they poorly explain these results. In contrast, causal inference theory says learning is facile and transfer instantaneous with the right conditions. We applied this theory in our pedagogy design and saw large gains, making it a promising direction. Future work may search for factors that lead to rapid learning by measuring learners' prior knowledge or traits then analyzing learning outcomes only, or jointly change the design of pedagogy or tools used, in an attempt to either increase or reduce extreme learning gains.

Decades of studies have attempted to improve outcomes for learning programming; we found something one can measure in only 8 hours (learning gain from PLTutor) which is highly predictive of long-term outcomes. We might be able to use this (or other good predictors) to improve the rate of experimentation and discovery, going from 3-4 studies per year using course outcomes to one per day using a proxy (if larger studies confirm their predictive ability).

Future work should investigate tools and curricula based on comprehension-oriented strategies, especially given the comparative lack of exploration and positive early results (ours and others like [30]). PLTutor had as good or better overall performance compared to a mature writing-oriented tutorial created with millions in funding. It seems unlikely that our team of three people, with almost no curriculum experimentation, has found the ceiling for comprehension tutorials or pedagogy.

ACKNOWLEDGMENTS

This material is based upon work supported by Microsoft, Google, Adobe, and the National Science Foundation (Grant No. 12566082, 1539179, 1314399, 1240786, and 1153625). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect funder views.

REFERENCES

[1] Mary J. Allen and Wendy M. Yen. 2001. *Introduction to Measurement Theory.* Waveland Press.

[2] John R. Anderson, Frederick G. Conrad, and Albert T. Corbett. 1989. Skill acquisition and the LISP tutor. *Cognitive Science* 13, 4 (1989), 467–505. DOI: http://dx.doi.org/10.1016/0364-0213(89)90021-9

[3] Ian Arawjo, Cheng-yao Wang, Andrew C Myers, Erik Andersen, and François Guimbretière. 2017. Teaching Programming with Gamified Semantics. In *Proceedings of the SIGCHI conference on Human factors in computing systems Reaching through technology - CHI '17.* ACM Press, New York, New York, USA.

[4] Sally Bailes, Eva Libman, Marc Baltzan, Rhonda Amsel, Ron Schondorf, and Catherine S. Fichten. 2006. Brief and distinct empirical sleepiness and fatigue scales. *Journal of Psychosomatic Research* 60, 6 (2006), 605–613. DOI: http://dx.doi.org/10.1016/j.jpsychores.2005.08.015

[5] Piraye Bayman and Richard E. Mayer. 1988. Using conceptual models to teach BASIC computer programming. *Journal of Educational Psychology* 80, 3 (1988), 291–298. DOI: http://dx.doi.org/10.1037/0022-0663.80.3.291

[6] Mordechai Ben-Ari. 2001. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching* 20, 1 (2001), 45–73.

[7] Dave Berry. 1991. *Generating Program Animators from Programming Language Semantics.* Ph.D. Dissertation. University of Edinburgh. DOI: http://dx.doi.org/10.1016/0377-0427(93)90083-N

[8] Lisa S Blackwell, Kali H Trzesniewski, and Carol Sorich Dweck. 2007. Implicit theories of intelligence predict achievement across an adolescent transition: a longitudinal study and an intervention. *Child development* 78, 1 (jan 2007), 246–63. DOI: http://dx.doi.org/10.1111/j.1467-8624.2007.00995.x

[9] Peter L. Bonate. 2000. *Analysis of Pretest-Posttest Designs.* CRC Press.

[10] J. D. Bransford, A. L. Brown, and R. R. Cocking. 2000. *How People Learn: Brain, Mind, Experience, and School: Expanded Edition.* National Academies Press. 1–27 pages. DOI: http://dx.doi.org/10.1016/0885-2014(91)90049-J

[11] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18, 6 (jun 1983), 543–554. DOI: http://dx.doi.org/10.1016/S0020-7373(83)80031-5

[12] L.D. Brown, T.T. Cai, and A. DasGupta. 2001. Interval Estimation for a Binomial Proportion. *Statist. Sci.* 16, 2 (2001), 101–133. https://www.scopus.com/inward/record.uri?eid=2-s2.0-0000460102&partnerID=40&md5=0997011d7da77720486e29c728a95d34 cited By 988.

[13] Teresa Busjahn and Carsten Schulte. 2013. The use of code reading in teaching programming. *Proceedings of the 13th Koli Calling International Conference on Computing Education Research* (2013), 3–11. DOI: http://dx.doi.org/10.1145/2526968.2526969

[14] William Campbell and Ethan Bolker. 2002. Teaching programming by immersion, reading and writing. In *32nd Annual Frontiers in Education*, Vol. 1. IEEE, T4G–23–T4G–28. DOI: http://dx.doi.org/10.1109/FIE.2002.1158015

[15] David Clark, Cara MacNish, and Gordon F Royle. 1998. Java as a teaching languagefi!?opportunities, pitfalls and solutions. *Proceedings of the 3rd Australasian conference on Computer science education* (1998), 173–179. DOI: http://dx.doi.org/10.1145/289393.289418

[16] Codecademy. 2016. https://www.codecademy.com. (2016). Accessed: 2016-12-12.

[17] CodingBat. 2016. https://www.codingbat.com. (2016). Accessed: 2016-12-12.

[18] Giuseppe Curcio, Michele Ferrara, and Luigi De Gennaro. 2006. Sleep loss, learning capacity and academic performance. *Sleep Medicine Reviews* 10, 5 (2006), 323–337. DOI: http://dx.doi.org/10.1016/j.smrv.2005.11.001

[19] J Dalbey and Marcia C Linn. 1985. The demands and requirements of computer programming: A review of the literature. *Journal of Educational Computing Research* 1, 3 (1985), 253–274. DOI: http://dx.doi.org/10.2190/BC76-8479-YM0X-7FUA

[20] M. De Raadt, M. Toleman, and R Watson. 2002. Language Trends in Introductory Programming Courses On the internet. *Informing Science* (2002), 329–337. http://proceedings.informingscience.org/IS2002Proceedings/papers/deRaa136Langu.pdf

[21] Lionel Deimel and David Moffat. 1982. A More Analytical Approach to Teaching the Introductory Programming Course. In *Proceedings of the National Educational Computing Conference.* 114–118.

[22] Julia F. Dewald, Anne M. Meijer, Frans J. Oort, Gerard A. Kerkhof, and Susan M. Bögels. 2010. The influence of sleep quality, sleep duration and sleepiness on school performance in children and adolescents: A meta-analytic review. *Sleep Medicine Reviews* 14, 3 (2010), 179–189. DOI: http://dx.doi.org/10.1016/j.smrv.2009.10.004

[23] Allen Downey and Lynn Stein. 2006. Designing a small-footprint curriculum in computer science. In *Proceedings. Frontiers in Education. 36th Annual Conference.* IEEE, 21–26. DOI: http://dx.doi.org/10.1109/FIE.2006.322660

[24] Benedict du Boulay, Tim O'Shea, and John Monk. 1981. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies* 14, 3 (apr 1981), 237–249. DOI: http://dx.doi.org/10.1016/S0020-7373(81)80056-9

[25] Jennifer L. Dyck and Richard E. Mayer. 1989. Teaching for Transfer of Computer Program Comprehension Skill. *Journal of Educational Psychology* 81, 1 (1989), 16–24. DOI: http://dx.doi.org/10.1037//0022-0663.81.1.16

[26] Allison Elliott Tew. 2010. Assessing fundamental introductory computing concept knowledge in a language independent manner. December 2010 (2010), 147. http://search.proquest.com/docview/873212789

[27] Matthias Felleisen, Robert Bruce Findler, Mathhew Flatt, and Shriram Krishnamurthi. 2001. How to Design Programs. *MIT Press* (2001), 720. DOI: http://dx.doi.org/10.1136/bjsm.27.1.58

[28] Thomas L. Griffiths and Joshua B. Tenenbaum. 2009. Theory-based causal induction. *Psychological Review* 116, 4 (2009), 661–716. DOI: http://dx.doi.org/10.1037/a0017201

[29] Philip J. Guo. 2013. Online python tutor. In *Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13.* ACM Press, New York, New York, USA, 579. DOI: http://dx.doi.org/10.1145/2445196.2445368

[30] Matthew Hertz and Maria Jump. 2013. Trace-Based Teaching in Early Programming Courses. *Proceedings of the 44th ACM Technical Symposium on Computer Science Education* (2013), 561–566. DOI: http://dx.doi.org/10.1145/2445196.2445364

[31] Klaus Hinkelmann and Oscar Kempthorne. 2008. *Design and Analysis of Experiments, Volume I: Introduction to Experimental Design.* Wiley.

[32] Jean-Michel Hoc and Anh Nguyen-Xuan. 1990. Chapter 2.3 - Language Semantics, Mental Models and Analogy. In *Psychology of Programming*, J.-M. Hoc, T.R.G. Green, R. Samuray, and D.J. Gilmore (Eds.). Academic Press, London, 139 – 156. DOI: http://dx.doi.org/10.1016/B978-0-12-350772-3.50014-8

[33] Aaron Hochanadel and D. Finamore. 2015. Fixed And Growth Mindset In Education And How Grit Helps Students Persist In The Face Of Adversity. *Journal of International Education Research fi First Quarter* 11, 1 (2015), 47–51. DOI: http://dx.doi.org/10.19030/jier.v11i1.9099

[34] Ville Karavirta, Riku Haavisto, Erkki Kaila, Mikko-Jussi Laakso, Teemu Rajala, and Tapio Salakoski. 2015. Interactive Learning Content for Introductory Computer Science Course Using the ViLLE Exercise Framework. *2015 International Conference on Learning and Teaching in Computing and Engineering* (2015), 9–16. DOI: http://dx.doi.org/10.1109/LaTiCE.2015.24

[35] Ken Kelley. 2007. Confidence Intervals for Standardized Effect Sizes :. *Journal of Statistical Software* 20, 8 (2007), 1–24. DOI: http://dx.doi.org/10.18637/jss.v020.i08 arXiv:arXiv:0908.3817v2

[36] Ken Kelley. 2007. Methods for the Behavioral, Educational, and Social Sciences: An R package. *Behavior Research Methods* 39, 4 (nov 2007), 979–984. DOI: http://dx.doi.org/10.3758/BF03192993

[37] John G Kemeny, Thomas E Kurtz, and David S Cochran. 1968. *Basic: a manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System.* Dartmouth Publications.

[38] Takayuki Kimura. 1979. Reading before composition. In *Proceedings of the tenth SIGCSE technical symposium on Computer science education - SIGCSE '79.* ACM Press, New York, New York, USA, 162–166. DOI: http://dx.doi.org/10.1145/800126.809575

[39] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. 2003. The BlueJ System and its Pedagogy. *Computer Science Education* 13, 4 (2003), 249–268. DOI: http://dx.doi.org/10.1076/csed.13.4.249.17496

[40] A. Krapp, S. Hidi, and K.A. Renninger. 1992. Factors Affecting Performance in First-year Computing. *The Role of interest in learning and development* 32, 2 (1992), 368.

[41] Olivier Lecarme. 1974. Structured programming, programming teaching and the language Pascal. *ACM SIGPLAN Notices* 9, 7 (jul 1974), 15–21. DOI: http://dx.doi.org/10.1145/953224.953226

[42] Michael J. Lee and Andrew J. Ko. 2015. Comparing the Effectiveness of Online Learning Approaches on CS1 Learning Outcomes. *Proceedings of the eleventh annual International Conference on International Computing Education Research - ICER '15* (2015), 237–246. DOI: http://dx.doi.org/10.1145/2787622.2787709

[43] Alex Lishinski, Aman Yadav, Jon Good, and Richard Enbody. 2016. Introductory Programming : Gender Differences and Interactive Effects of Students ' Motivation , Goals and Self-Efficacy on Performance. *Proceedings of the 12th International Computing Education Research Conference* (2016), 211–220. DOI: http://dx.doi.org/10.1145/2960310.2960329

[44] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *ACM SIGCSE Bulletin* 41, 3 (2009), 161. DOI: http://dx.doi.org/10.1145/1595496.1562930

[45] Raymond Lister, Otto Seppälä, Beth Simon, Lynda Thomas, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, and Kate Sanders. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin* 36, 4 (dec 2004), 119. DOI: http://dx.doi.org/10.1145/1041624.1041673

[46] J Littlefield, V Delclos, S Lever, K Clayton, J Bransford, and J Franks. 1988. Learning Logo: Methods of teaching, transfer of general skills, and attitudes toward school and computers. In *Learning computer programming: Multiple research perspectives*, Richard E. Mayer (Ed.). Erlbaum, Hillsdale, NJ, 111–135.

[47] Dastyni Loksa and Andrew J. Ko. 2016. The Role of Self-Regulation in Programming Problem Solving Process and Success. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16).* ACM, New

York, NY, USA, 83–91. DOI: http://dx.doi.org/10.1145/2960310.2960334

[48] Dastyni Loksa, Andrew J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 1449–1461. DOI: http://dx.doi.org/10.1145/2858036.2858252

[49] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. *Proceeding of the fourth international workshop on Computing education research - ICER '08* (2008), 101–112. DOI: http://dx.doi.org/10.1145/1404520.1404531

[50] Andrew Luxton-Reilly. 2016. Learning to program is easy. *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (2016), 284–289. DOI: http://dx.doi.org/10.1145/2899415.2899432

[51] John Maloney, Mitchel Resnick, and Natalie Rusk. 2010. The Scratch programming language and environment. *ACM Transactions on Computing Education* 10, 4 (2010), 1–15. DOI: http://dx.doi.org/10.1145/1868358.1868363.http arXiv:-

[52] Richard E. Mayer. 1979. *Analysis of a Simple Computer Programming Language: Transactions, Prestatements and Chunks.* Technical Report. Series in Learning and Cognition, Tech. Rep. No. 79-2, U. of California, Santa Barbara, California. 1–34 pages. https://eric.ed.gov/?id=ED207549

[53] Richard E. Mayer. 1981. The Psychology of How Novices Learn Computer Programming. *Comput. Surveys* 13, 1 (1981), 121–141. DOI: http://dx.doi.org/10.1145/356835.356841

[54] Richard E. Mayer. 1985. *Learning In Complex Domains: A Cognitive Analysis of Computer Programming.* Vol. 19. Academic Press. 89–130 pages. DOI: http://dx.doi.org/10.1016/S0079-7421(08)60525-3

[55] Anneliese Von Mayrhauser and a Marie Vans. 1995. Program Comprehension During Software Maintenence and Evolution. *Computer* 28, 8 (1995), 44–55. DOI: http://dx.doi.org/10.1109/2.402076

[56] Michael McCracken, Tadeusz Wilusz, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, and Ian Utting. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin* 33, 4 (dec 2001), 125. DOI: http://dx.doi.org/10.1145/572139.572181

[57] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. 2004. Visualizing programs with Jeliot 3. *Proceedings of the working conference on Advanced visual interfaces - AVI '04* (2004), 373. DOI: http://dx.doi.org/10.1145/989863.989928

[58] Briana Morrison, Lauren Margulieux, and Mark Guzdial. 2015. Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In *Proceedings of the eleventh annual International Conference on International Computing Education Research - ICER '15*. ACM Press, New York, New York, USA, 267–268. DOI: http://dx.doi.org/10.1145/2787622.2787744

[59] Laurie Murphy and Lynda Thomas. 2008. Dangers of a fixed mindset: implications of self-theories research for computer science education. *ACM SIGCSE Bulletin* 40, 3 (2008), 271–275. DOI: http://dx.doi.org/10.1145/1597849.1384344

[60] Michael O'Brien. 2003. *Software Comprehension - A review and research direction.* Technical Report UL-CSIS-03-3. University of Limerick. 1–29 pages.

[61] D. B. Palumbo. 1990. Programming Language/Problem-Solving Research: A Review of Relevant Issues. *Review of Educational Research* 60, 1 (jan 1990), 65–89. DOI: http://dx.doi.org/10.3102/00346543060001065

[62] John F Pane. 2002. *A programming system for children that is designed for usability.* Ph.D. Dissertation. Carnegie Mellon University.

[63] Seymour Papert. 1971. *A Computer laboratory for elementary schools.* Technical Report. Massachusetts Institute of Technology. Artificial Intelligence Laboratory. 19 pages.

[64] Miranda C Parker and Mark Guzdial. 2016. Replication, validation, and use of a language independent CS1 knowledge assessment. *Proceedings of the 12th International Computing Education Research Conference* (2016), 93–101. DOI: http://dx.doi.org/10.1145/2960310.2960316

[65] Randy Pausch, Wanda Dann, and Stephen Cooper. 2000. Alice : a 3-D Tool for Introductory Programming Concepts. *Journal of Computing Sciences in Colleges* 15, 5 (2000), 107–116.

[66] Roy D Pea. 1986. Language-independent conceptual" bugs" in novice programming. *Journal of Educational Computing Research* 2, 1 (1986), 25–36. DOI: http://dx.doi.org/10.2190/689T-1R2A-X4W4-29J2

[67] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A Survey of Literature on the Teaching of Introductory Programming. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '07)*. ACM, New York, NY, USA, 204–223. DOI: http://dx.doi.org/10.1145/1345443.1345441

[68] PracticeIt. 2016. http://practiceit.cs.washington.edu. (2016). Accessed: 2016-12-12.

[69] Anthony Ralston. 1971. Fortran and the First Course in Computer Science. *Acm Sigcse* 3, 4 (1971), 24–29. DOI: http://dx.doi.org/10.1145/382214.382499

[70] Vennila Ramalingam, Deborah LaBelle, and Susan Wiedenbeck. 2004. Self-Efficacy and Mental Models in Learning to Program. *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '04* 36, 3 (2004), 171–175. DOI: http://dx.doi.org/10.1145/1007996.1008042

[71] Vennila Ramalingam and Susan Wiedenbeck. 1999. Development and validation of scores on a computer programming self-efficacy scale and group analyses of novice programmer self-efficacy. *Journal of Educational Computing Research* 19, 4 (1999), 367–381. DOI: http://dx.doi.org/10.2190/C670-Y3C8-LTJ1-CT3P

[72] Stuart Reges. 2006. Back to basics in CS1 and CS2. *ACM SIGCSE Bulletin* 38, 1 (2006), 293. DOI: http://dx.doi.org/10.1145/1124706.1121432

[73] Eric Roberts. 2004. The dream of a common language. *Proceedings of the 35th SIGCSE technical symposium on Computer science education - SIGCSE '04* 36, 1 (2004), 115. DOI: http://dx.doi.org/10.1145/971300.971343

[74] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education* 13, 2 (jun 2003), 137–172. DOI: http://dx.doi.org/10.1076/csed.13.2.137.14200

[75] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H. Paterson. 2010. An introduction to program comprehension for computer science educators. *Proceedings of the 2010 ITiCSE working group reports on Working group reports - ITiCSE-WGR '10* (2010), 65. DOI: http://dx.doi.org/10.1145/1971681.1971687

[76] Ben Shneiderman. 1977. Teaching programming: A spiral approach to syntax and semantics. *Computers & Education* 1, 4 (jan 1977), 193–197. DOI: http://dx.doi.org/10.1016/0360-1315(77)90008-2

[77] Ben Shneiderman and Richard Mayer. 1979. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences* 8, 3 (jun 1979), 219–238. DOI: http://dx.doi.org/10.1007/BF00977789

[78] D. Sleeman, Ralph T. Putnam, Juliet Baxter, and Laiani Kuspa. 1986. Pascal and high school students: A study of errors. *Journal of Educational Computing Research* 2, 1 (1986), 5–23. DOI: http://dx.doi.org/10.2190/2XPP-LTYH-98NQ-BU77

[79] Juha Sorva. 2013. Notional machines and introductory programming education. *ACM Transactions on Computing Education* 13, 2 (2013), 1–31. DOI: http://dx.doi.org/10.1145/2483710.2483713

[80] Juha Sorva and Otto Seppälä. 2014. Research-based design of the first weeks of CS1. *Proceedings of the 14th Koli Calling International Conference on Computing Education Research (Koli Calling '14)* November 2014 (2014), 71–80. DOI: http://dx.doi.org/10.1145/2674683.2674690

[81] Juha Sorva and Teemu Sirkia. 2010. UUhistle: a software tool for visual program simulation. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research Koli Calling 10* (2010), 49–54. DOI: http://dx.doi.org/10.1145/1930464.1930471

[82] Juha Sorva and Teemu Sirkiä. 2011. Context-sensitive guidance in the UUhistle program visualization system. *Proceedings of the Sixth Program Visualization Workshop (PVW 2011)* (2011), 77–85.

[83] C. Taylor, D. Zingaro, L. Porter, K.C. Webb, C.B. Lee, and M. Clancy. 2014. Computer science concept inventories: past and future. *Computer Science Education* 24, 4 (2014), 253–276. DOI: http://dx.doi.org/10.1080/08993408.2014.970779

[84] Franklyn Turbak, Constance Royden, Jennifer Stephan, and Jean Herbst. 1999. Teaching recursion before loops in CS1. *Journal of Computing in Small Colleges* 14, May (1999), 86–101. http://cs.wellesley.edu/

[85] Jeroen J. G. Van Merrienboer and Hein P. M. Krammer. 1987. Instructional strategies and tactics for the design of introductory computer programming courses in high school. *Instructional Science* 16, 3 (sep 1987), 251–285. DOI: http://dx.doi.org/10.1007/BF00120253

[86] Anne Venables, Grace Tan, and Raymond Lister. 2009. A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. *Proceedings of the Fifth International Workshop on Computing Education Research Workshop - ICER '09* 2009 (2009), 117–128. DOI: http://dx.doi.org/10.1145/1584322.1584336

[87] Antti Virtanen, Essi Lahtinen, and Hannu-Matti Jarvinen. 2005. VIP, a Visual Interpreter for Learning Introductory Programming with C++. *Koli Calling 2005 Conference on Computer Science Education* November (2005), 125–130. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.9111

[88] Christopher Watson, Frederick W B Li, and Jamie L Godwin. 2014. No tests required: comparing traditional and dynamic predictors of programming success. *Proceedings of the 45th ACM technical symposium on Computer science education - SIGCSE '14* (2014), 469–474. DOI: http://dx.doi.org/10.1145/2538862.2538930

[89] Brenda Cantwell Wilson and Sharon Shrock. 2001. Contributing to success in an introductory computer science course: a study of twelve factors. *ACM SIGCSE Bulletin* 33, 1 (2001), 184–188. DOI: http://dx.doi.org/10.1145/366413.364581

[90] Leon E. Winslow. 1996. Programming Pedagogy - A Psychological Overview. *ACM SIGCSE Bulletin* 28, 3 (1996), 17–22. DOI: http://dx.doi.org/10.1145/234867.234872

[91] Cecile Yehezkel. 2003. Making program execution comprehensible one level above the machine language. *ITiCSE '03 Proceedings of the 8th annual conference on Innovation and technology in computer science education* (2003), 124. DOI: http://dx.doi.org/10.1145/961290.961547