A theory of instruction for introductory programming skills

Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li & Amy J. Ko

Published online: 25 Jan 2019.

Submit your article to this journal ✏

Article views: 6

View Crossmark data ✓

Routledge
Taylor & Francis Group

ARTICLE

Check for updates

A theory of instruction for introductory programming skills

Benjamin Xie[a], Dastyni Loksa[a], Greg L. Nelson[b], Matthew J. Davidson[c], Dongsheng Dong[c], Harrison Kwik[b], Alex Hui Tan[a], Leanne Hwa[a], Min Li[c] and Amy J. Ko[a]

[a]The Information School, University of Washington, Seattle, WA, USA; [b]Paul G. Allen School of Computer Science & Engineering, University of Washington, Seattle, WA, USA; [c]College of Education, University of Washington, Seattle, WA, USA

ABSTRACT

Background and Context: Current introductory instruction fails to identify, structure, and sequence the many skills involved in programming.

Objective: We proposed a theory which identifies four distinct skills that novices learn incrementally. These skills are tracing, writing syntax, comprehending templates (reusable abstractions of programming knowledge), and writing code with templates. We theorized that explicit instruction of these skills decreases cognitive demand.

Method: We conducted an exploratory mixed-methods study and compared students' exercise completion rates, error rates, ability to explain code, and engagement when learning to program. We compared material that reflects this theory to more traditional material that does not distinguish between skills.

Findings: Teaching skills incrementally resulted in improved completion rate on practice exercises, and decreased error rate and improved understanding of the post-test.

Implications: By structuring programming skills such that they can be taught explicitly and incrementally, we can inform instructional design and improve future research on understanding how novice programmers develop understanding.

1. Introduction: CS1 instruction could better teach programming skills

Programming requires many distinct skills. In addition to basic knowledge of programming constructs (Tew & Guzdial, 2010), programming also requires procedural skills to perform tasks with these constructs (e.g. tracing code and writing correct syntax) (Davies, 1993; Sanders et al., 2012; Winslow, 1996). For example, tracing is a critical skill (Lopez, Whalley, Robbins, & Lister, 2008; Venables, Tan, & Lister, 2009), which Nelson, Xie, and Ko (2017) formally defined as being able to look at code and predict state changes and outputs through the compilation and execution of a programming language's constructs. Explaining code is another

critical skill, defined as reading a piece of code and describing it in relation to the code's purpose (Whalley et al., 2006). And then, of course, there is the skill of writing code, typically characterized as composing syntactically correct code with a purpose in the context of a problem or specification (Robins, Rountree, & Rountree, 2003). Many works described and empirically investigated the relationships between these various skills, finding that while related, these skills are distinct (Corney, Lister, & Teague, 2011; Lopez et al., 2008; Robins et al., 2003; Venables et al., 2009; Winslow, 1996).

To teach programming skills effectively requires sequencing them. Educational psychologist Bruner (1966) argued that instruction requires specifying "the ways in which a body of knowledge should be structured so that it can be most readily grasped by the learner," as well as "effective sequences in which to present the materials to be learned." Studies of introductory computer science (*CS1*) instruction specifically have found gaps in the instruction of programming skills; for example, Kreitzberg and Swanson (1974) found that even when students had learned concepts, they could not readily apply them to programming skills; Sanders et al. (2012) identified that learning to program also required learning to apply *multiple skills*, and CS1 instruction lacked adequate instruction in these skills. In our own observations, we have found CS1 courses at our institution continue to have problems with overlapping instruction on programming skills in a way that makes instruction potentially inaccessible for novices. For example, on the very first day of our university's recent CS1 course, the instructor showed content on syntax and then proceeded with writing exercises for practice. Within an hour-long lecture, the instructor showed example programs and their output, then asked learners to trace code and determine correct outputs as well as write code to produce given outputs. Learners with no prior knowledge felt overwhelmed as they were asked to both trace and write code simultaneously. Theoretically, empirically, and anecdotally, the lack of sequencing of skills appears to be a longstanding problem in CS education.

One way to address the lack of structure in programming skills instruction is to provide explicit instruction on each skill, in sequence (Archer & Hughes, 2010; Doyle, 1983). For example, Soloway and Ehrlich (1984) proposed identifying and teaching programming skills by providing instruction on libraries of stereotypical solutions to problems, as well as strategies for coordinating and composing them. Others have proposed teaching skills separately and incrementally with the aim of automating more basic skills (e.g. writing correct syntax) to reduce confusion and improve learning by lowering cognitive demand (Anderson, Boyle, Farrell, & Reiser, 1984; Lister et al., 2004). The hope in these works was that this explicit instruction on skills would free learners to concentrate on more advanced skills relating to problem-solving (Buck & Stucki, 2000). Although there have been many theories with implications for CS1 instructional design for programming skills, few have been translated to concrete instruction, and when they have, they often lack evaluation.

In this paper, we build upon these prior theories of CS1 instruction, proposing a new theory of instruction for CS1 programming skills that is simple, has direct implications for instructional design, and that, when translated into concrete learning materials, may have measurable impacts on learning. Our theory structures and sequences four distinct skills: tracing, writing correct syntax, recognizing the parts and objectives of *templates* (reusable abstractions of programming knowledge), and using templates to solve problems. We hypothesize that explicit, incremental instruction on these four skills will result in the following effects:

(1) Learners will be more able to complete programming tasks
(2) Learners will make fewer errors
(3) Learners will have a greater understanding of the relationship between parts of the code and the overall purpose
(4) Learners will be more engaged in the learning process

To evaluate our hypotheses, we created learning materials for a subset of CS1 concepts that include instructional content, practice exercises with feedback, and a post-test which covers these four skills. We then conducted an exploratory, mixed-methods evaluation of this curriculum with nine novice programmers to explore the validity of our hypotheses and theory more broadly.

The structure of the paper is as follows: We further substantiate our claims about the current gaps between empirical findings on programming skills and theories of instruction in Section 2. We then propose our theory of programming skills instruction in Section 3. In Section 4, we provide a concrete example of learning materials (instruction, practice, post-test) that we designed based on our theory. In Section 5, we describe hypotheses based on our theory and detail a study we conducted to evaluate these hypotheses. We share the results of our evaluation in Section 6. Finally, we interpret the results, evaluate our hypotheses, and discuss the implications of our theory in Section 7.

2. Related work: CS1 skills and theories to inform instructional design

In this section, we explore three aspects of related work identifying different programming skills, gaps in prior theory, and the use of patterns to scaffold programming knowledge. This related work helps substantiate the need for a new theory of instruction as well as provide the foundation for it.

2.1. Programming skills (tracing, explaining, writing) are distinct and may develop sequentially

Much of the literature on the relationships between the skills of tracing, explaining, and writing code came from a common thread of research from the Leeds Working

Group and BRACElet workshops (Clear et al., 2011). This work investigated the teaching and learning of novice programmers, using the *Structure of Observed Learning Outcomes (SOLO) taxonomy* (Biggs & Collis, 2014) as a framework for evaluating novice programmers' responses to code tracing and explaining problems (Lister, Simon, Thompson, Whalley, & Prasad, 2006). Different studies used slight variations of the SOLO taxonomy (Murphy, Fitzgerald, Lister, & McCauley, 2012; Philpott, Robbins, & Whalley, 2007; Whalley et al., 2006), but they all typically referenced four hierarchical levels of student responses:

(1) *Prestructural*: Response demonstrates no relevant knowledge or is unrelated to the question.
(2) *Unistructural*: Response provides a description for a small portion of the code.
(3) *Multistructural*: Response is a line-by-line description of most of the code.
(4) *Relational*: Response provides a summary of what the code does in terms of the code's purpose.

Lister et al. (2006) found that responses from experts (educators) tended to manifest at the relational level of the SOLO taxonomy for tracing problems, as experts were "seeing the forest." In contrast, novices who could not produce relational level responses were "failing to see the forest" and were unable to extract the purpose or summary of what the code does at a more abstract level. This work suggested that novice programmers who could provide relational level responses demonstrated expertise. We used the SOLO taxonomy for evaluating depth of understanding among participants (see Section 6.3).

Using the SOLO taxonomy to analyze the quality of responses, the Leeds Working Group and BRACElet workshops made findings related to the programming skills novices learn (e.g. tracing, explaining, and writing code). Philpott et al. (2007) found that novices' mastery of code tracing indicated their readiness to reason about or explain the code, suggesting tracing was prerequisite knowledge for explaining code. Sheard et al. (2008) found a positive correlation between code explaining and writing tasks. Lopez et al. (2008) found a similar correlation and suggested a potential hierarchy of programming related tasks with the knowledge of program constructs at the bottom, code tracing and explaining ability as part of one or more intermediate levels, and writing ability at the top. Venables et al. (2009) found a causal relationship between code tracing and writing, and that the skills of tracing and explaining were strong predictors of performance on code writing.

This prior work shows that tracing, explaining, and writing code are distinct skills that are potentially dependent and develop sequentially. Furthermore, the SOLO taxonomy can be used to evaluate the quality of novices' responses to tracing and explaining questions.

2.2. Theory decomposing programming skills lacks connections to instruction and lacks a simplifying structure to make instructional design tractable

Being able to distinguish skills does not necessarily provide guidance on how they should be taught. To teach these distinct programming skills identified in the previous section, it would be helpful to have a theory to inform the design of CS1 instruction that considers these skills. In addition to needing a theory of instruction to structure and sequence programming skills, it would be ideal to have it also be translatable to concrete instruction. We found that prior work in theories of instruction was typically not easily translatable to instruction or did not adequately account for different programming skills.

2.2.1. Some theories provide abstract constraints without fully specifying instruction

Basing instruction off some theories has resulted in constraints for instructional design without full specification of learning activities. Taxonomies, such as the SOLO taxonomy discussed previously (Section 2.1), often serve as theories to provide constraints. Two examples illustrate uses of taxonomies: Recent revisions to Bloom's taxonomy described the cognitive development of novices across knowledge domains that are CS-specific (Fuller et al., 2007). Many have used Bloom's or the SOLO taxonomy to classify instructional content and assessment items (Thompson, Luxton-Reilly, Whalley, Hu, & Robbins, 2008). Similarly, Gluga et al. used neo-Piagetian theory to classify the difficulty of computer science instruction, finding instructors with varying backgrounds could reliably identify a problem's required level of development (after using a tutorial to learn those levels) (Gluga, Kay, Lister, Simon, & Kleitman, 2013; Gluga, Kay, Lister, & Teague, 2012). These taxonomies and related theories have provided general guidelines for instruction, but fall short of being able to be directly translated to instruction.

Rather than directly adapt more general theories from the learning sciences, other researchers developed new frameworks and structures specific to computer science to provide constraints for instructional design. Whalley and Kasto (2013) proposed a Block model for measuring the difficulty of code comprehension questions and compared it to the SOLO and Bloom taxonomies. Similarly, Fuller et al. (2007) developed a two-dimensional *matrix taxonomy* based on Bloom's taxonomy along dimensions of producing and interpreting and includes some concrete discussion of narrow slices of CS1, databases, and computing professionalism courses. Mead et al. (2006) combined many threads related to cognition and learning to describe an *anchor graph* to represent dependencies among concepts in a course. These prior works provide CS-specific frameworks or structures to support instructional design at a high-level, but still cannot be directly translated to instruction.

For each of these theories, we are unaware of work that uses them to make concrete instructional designs that cover all the skills/components in the theory. So while some theories provide a means for general classification of content, they are often too abstract to be directly translated to concrete instruction.

2.2.2. More specific theories do not translate to instruction that supports skill development

More specific theories of instructional design include instructional designs that have clear realizations, specifying how the knowledge is organized explicitly for learners, concrete examples of instruction, and concrete examples of practice and/or assessment. The users of these theories are researchers and teachers, and potentially also learners themselves. These more concrete theories align with the perspective of Bruner (1966), in which a theory of instruction must define how a body of knowledge should be structured and sequenced so it is interpretable for a learner.

Some concrete theories of CS instruction translate to instruction but do not mention programming skills. For example, Caspersen and Bennedsen (2007) incorporated cognitive load theory and cognitive skill acquisition into a model of human cognitive architecture and then presented an introductory object-oriented programming course. This course used a pattern-based approach to programming and schema acquisition, but did not specify how to incorporate different programming skills.

Other concrete theories of instruction provide mention of sequenced programming skills but did not detail instruction and practice to fully scaffold the acquisition of these skills. For example, Buck and Stucki (2000) proposed a hierarchical progression of skill sets based on Bloom's taxonomy. This hierarchy identified tracing as a "lower-level" skill, but their course description did not include tracing instruction or practice beyond predicting what line executes next. More specifically, the knowledge and comprehension levels of their modified Bloom's taxonomy preceded implementing skills at higher levels, yet the course design did not provide instruction or practice to support those levels of knowledge. While the instruction does describe executing code snippets, it does not assess whether learners can trace code. The only code comprehension exercise described involved "student[s] predict the next statement to which control will pass, throughout an entire execution of a procedure. If they predict incorrectly, they are shown the actual line to be executed next, and they continue from there…" In a follow-up paper, Buck and Stucki (2001) discussed the implementation of this comprehension practice, as well as practice that involves translating program code into a flowchart; much of the justification for that instructional design was to try to avoid teaching variable concepts, and it is unclear when/how variables are taught or assessed.

Similar to Buck and Stucki (2001) but drawing on neo-Piagetian theories of programming learning (Morra et al., 2012), Szabo, Falkner, and Falkner (2014) described a well-defined theory for course design, yet their course design was ambiguous and lacked practice and instruction for lower level skills. While Szabo applied their theory to designing a second programming course, the theory did not describe lower level instruction or assessments (e.g. tracing) at the sensorimotor or pre-operational levels for new concepts such as object-oriented language features or concepts. Instead, learners experienced "objects in real life and their interactions" and that seems to have counted for those skill levels. Yet in their theory they defined those levels as relating to tracing ability at the sensorimotor level as "low abstraction level, can barely trace code" and pre-operational level "can reliably trace code, but cannot understand functionality". This misalignment between the specifications in their theory of instructional design and the actual instructional design exists in other parts of the instruction as well.

Prior theoretical work presaged and reflected empirical findings that lower level tracing skills precede writing skills; however, prior theories of instructional designs do not fully specify focused practice, instruction, and assessment that covers all the knowledge required for pre-requisite skills, particularly for lower level pre-writing skills (e.g. program tracing).

2.3. Templates can help transition from learning a language to using it to problem solve

A potential way to incorporate skill development into instruction is by representing knowledge of what programs can do with the use of pattern-like chunks, which we will refer to as *templates* (Clancy & Linn, 1999). Templates are abstractions of programming knowledge that have generality and reusability (Clancy & Linn, 1992; 1999), similar to Rist (1989)'s notion of *schema*, Kreitzberg and Swanson (1974)'s notion of *meta-rules* of generalized problem-solving techniques, Anderson et al. (1984)'s notion of *weak schemata*, and Soloway and Ehrlich (1984)'s notion of *plans in the work*. Providing novices with a "repertoire of templates" (Clancy & Linn, 1992) can reduce the cognitive demands of writing programs by providing ways to decompose a problem, enabling novices to use these templates to support their planning and problem-solving process and write more complicated programs (Mead et al., 2006; Rist, 1989).

Templates can be incorporated into instruction by using it as a scaffolding technique. Linn and Dalbey (1985) proposed a *Chain of Cognitive Accomplishments* that should arise from ideal instruction of programming. This chain consisted of (1) features of language, (2) design skills relating to the procedural skills of planning, testing, and reformulating code using templates, and (3) problem-solving skills which are abstracted from specific languages and applied to learning new languages and situations. Clancy and Linn

(1999) suggested that exercises in code comprehension, identifying opportunities and "nonopportunities" for pattern reuse, considering multiple representations of patterns, and comparing related patterns could benefit students who were learning from instruction involving patterns.

In summary, prior work has not defined concrete instructional designs that consider different skills that are important to programming and how these skills develop. Prior theories tend to have either partially specified ambiguous designs or a lack of focused practice and instruction for specific skills (especially the lower level, pre-writing skills). This ambiguity or inconsistency may come from each theory's coverage of a very broad range of skills (e.g. the scope is often "programming rather than CS1" or other subsets). Prior work on theories of instructional design tended to focus on program writing instruction, often not adequately specifying their designs for instruction and practice for pre-requisite skills (such as tracing). A lack of instruction and practice for pre-requisite skills may result in gaps in learners' knowledge that exacerbate as they prematurely practice more advanced skills such as code writing. Common patterns such as templates may be able to support skill development by providing scaffolding in the transition from semantic understanding of code to problem-solving with code.

3. Theory: separating, structuring, and sequencing programming skills

In the previous section, we established that distinct programming skills exist, yet prior theories do not translate to concrete instruction that supports the development of these skills; we draw upon prior work to propose a theory that structures and sequences these skills and can be translated to instruction that scaffolds the development of these skills. In this section, we identify three claims we draw from prior work which serve as the foundation of our theory. We then describe our theory and how we structure knowledge across four programming skills (tracing, writing correct syntax, understanding templates as reusable abstractions of programming knowledge, applying templates to solve problems) which build upon each other. This theory is focused to novice programmers and by design does *not* explicitly account for skills including debugging, problem-solving, and solving problems that require inventing new or previously unlearned templates.

Table 1 shows our "quadrant" of introductory programming skills. We distinguish the skills of tracing (S1), writing correct syntax (S2), recognizing templates and their uses (S3), and using templates to solve problems (S4) across two dimensions. These dimensions are *skills* (read, write) and *knowledge* (semantics, templates). Skills refer to reading already written code and interpreting meaning from it, and writing code. Knowledge is either at a machine level (semantics) and at a task/objective level (templates).

Table 1. Decomposition of different skills across two dimensions.

	semantics related to code	templates related to goals/objectives
Read	S1. Predict effect of syntax on program behavior	S3. Recognize templates and their uses
Write	S2. Write correct syntax	S4. Use templates to complete objective

3.1. Connecting theory to prior work: differentiating and ordering skills

We based our theory of incrementally teaching decoupled programming skills on three claims that distinguished different skills novice programmers must learn:

(1) **C1**: *Tracing code is a different and precursory skill to writing syntactically correct code*
(2) **C2**: *Understanding the features of a programming language is different than solving a problem with code*
(3) **C3**: *Comprehending code templates is a different skill than using templates to write code to fulfill an objective*

Prior work described in Section 2 substantiates the three claims which are at the foundation of our theory.

The prior work in Section 2.1 on the proposed and empirically supported distinction between reading (tracing) and writing code helps substantiate our first claim (C1). While prior work tended to frame writing code as composing syntactically correct code that also has an objective, we distinguish between the skills of writing correct syntax and writing code that has an objective. For C1, we focus only on the skill of writing correct syntax. Although we have a more specific definition of writing, we still find that the prior work in Section 2.1 substantiates C1, the claim that tracing is a precursory skill to writing correct syntax.

To substantiate the next claim (C2), we look to the prior work in Section 2.3 on templates and the Chain of Cognitive Accomplishments. More specifically, we focus on how ideal instruction of programming teaches the features of the language (first chain) before design skills relating to using templates (second chain). Lastly, substantiating C3 requires drawing parallels between programming skills (from Section 2.1) and template use (from Section 2.3). We focus the definition of code explanation to say that it relates to recognizing templates and their uses (which we call *comprehending* templates) because both skills require looking at code and mapping it to an objective or purpose. As described in Section 2.1, multiple studies found that explaining code was a separate and precursor skill to writing code (Lopez et al., 2008; Sheard et al., 2008; Venables et al., 2009). Furthermore, Clancy and Linn (1999) suggested that code comprehension questions could support instruction on templates. We substantiate our third and final claim C3 with two points: code explanation (as defined by prior work) is similar to reading/comprehending

templates, and both skills precede writing code with a template. Therefore, prior work substantiates our three claims.

Our theory recommends *reading before writing* (as substantiated by C1, C3) and *semantics before templates* (as substantiated by C2).

3.2. A theory of instruction for four programming skills across two dimensions

To further explain our theory, we now describe each skill in incremental order from S1 (reading semantics) to S4 (writing templates). To provide an example of how a learner can demonstrate knowledge of each skill, Figure 1 decomposes a classic template of a variable swap operation (Sheard et al., 2008) into the four skills in our theory. All code is in Python syntax. We refer to this figure as we describe each skill below.

S1, Reading semantics (top left in Table 1) refers to the ability to accurately trace code and predict the effect of syntax on program behavior. We adopt the theory of program tracing defined by Nelson et al. (2017), which states that knowing programming tracing is understanding the set of all mappings between syntax, semantics, and state during compilation. After a learner develops the understanding of reading semantics, they are able to trace code and determine its intermediate and final states and outputs. They do not necessarily know how to write correct code or use code to perform a certain task. Reading semantics is a precursor skill to writing syntax (by C1) and using templates (by C2), so it is the foundation to all other skills. As a result, it is first in the sequence of skills in our instruction.

Learners demonstrate knowledge of reading semantics by being able to describe each line of the program and also being able to accurately trace the code and determine the final variable values (final state), as demonstrated in the top left of Figure 1. This knowledge is comparable to the *multistructural* level of the SOLO taxonomy (described in Section 2.1). Note that the purpose of the code as a whole (to swap two variables' values) is out of the scope of reading semantics. In the example, given a line of code y = temp, learners demonstrate knowledge of reading semantics by knowing that the value stored in variable y updates to the value stored in variable temp.

When they have a strong understanding of reading semantics for a given programming construct, learners should be able to understand how that construct affects the program statement and output for a piece of code. They do not necessarily understand the purpose of the construct in relation to the code or problem more broadly, as that comes later with template knowledge. They also do not necessarily know how to write correct syntax, but that is the next skill to learn.

S2, Writing semantics (bottom left in Table 1) refers to the translating of *unambiguous* natural language descriptions of language constructs into syntax

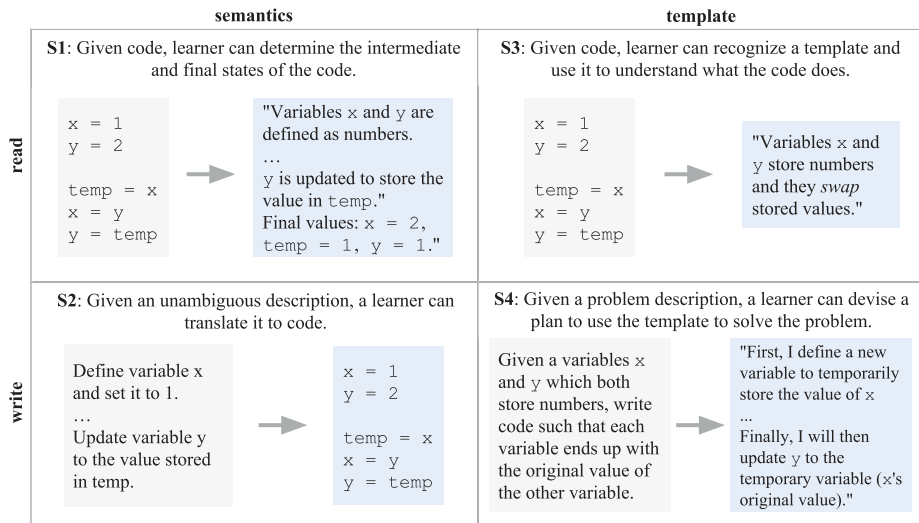|	semantics	template
read	**S1**: Given code, learner can determine the intermediate and final states of the code. ```\nx = 1\ny = 2\n\ntemp = x\nx = y\ny = temp\n``` → "Variables x and y are defined as numbers. … y is updated to store the value in temp." Final values: x = 2, temp = 1, y = 1."	**S3**: Given code, learner can recognize a template and use it to understand what the code does. ```\nx = 1\ny = 2\n\ntemp = x\nx = y\ny = temp\n``` → "Variables x and y store numbers and they *swap* stored values."
write	**S2**: Given an unambiguous description, a learner can translate it to code. Define variable x and set it to 1. … Update variable y to the value stored in temp. → ```\nx = 1\ny = 2\n\ntemp = x\nx = y\ny = temp\n```	**S4**: Given a problem description, a learner can devise a plan to use the template to solve the problem. Given a variables x and y which both store numbers, write code such that each variable ends up with the original value of the other variable. → "First, I define a new variable to temporarily store the value of x … Finally, I will then update y to the temporary variable (x's original value)."

Figure 1. Demonstrated knowledge relating to each of the four programming skills in our theory, using the example of a variable swap. Each skill is represented as a cell in the quadrant with a description of how a learner demonstrates the skill at the top of each quadrant. For each skill, a learner demonstrates that skill (for a variable swap) by translating a given prompt (gray box) into a response (blue box). For example, the first skill of reading semantics (S1, top left) relates to determining the intermediate and final program states and program output. A learner can demonstrate knowledge of this skill by taking a program and correctly explaining what each line of code does and determining the correct variable values after execution.

that will compile and execute as expected. Because this is a translation from an unambiguous specification, this skill does not require understanding the objective of the code at a relational level. To know how to write semantics is to be able to write syntactically correct code (construct a valid abstract syntax tree, AST), modify preexisting code without introducing syntax errors (fill in a missing part of a valid AST or modify a valid AST), and correct code with syntax errors (recognize an invalid AST). Visual blocks-based languages obviate this skill by enforcing correct syntax by providing drag-and-drop feedback and therefore not requiring a programmer to know how to write correct syntax. A novice must know how program constructs execute during compilation, so they must have an understanding of reading semantics (S1) before writing semantics.

The bottom left quadrant of Figure 1 is an example of writing semantics practice. A learner demonstrates knowledge of writing semantics by being able to translate a line-by-line description of code into lines of code with correct syntax. In the example, a learner can translate "define variable x and set it to 1" as x = 1 (in Python).

To write correct code that meets an unambiguous specification, a learner must have an understanding of reading semantics to know how code constructs affect execution and understanding of writing semantics to know how

to translate these code constructs to correct syntax. Weak knowledge of reading semantics but strong knowledge of writing semantics would result in a learner writing syntactically correct code which failed to meet the specification and does not execute as intended. Strong knowledge of reading semantics but weak knowledge of writing semantics would result in writing code which contains syntax errors and could look like a pseudocode that could meet the specifications.

S3, Reading templates (top right in Table 1) refers to the skill of identifying reusable abstractions of programming knowledge (which we will refer to as *templates*) and mapping them to an objective. Reading templates consists of being able to trace code and map parts of the code to parts of a template and identify what the objective or purpose of the code is. Novices acquire knowledge of templates by recognizing them in other programs, learning them through instruction, or devising them.

The top right quadrant of Figure 1 is an example of reading templates practice. A learner demonstrates knowledge of reading templates by being able to look at a program and recognize its purpose relative to a template they previously learned. In the example, a learner can look at the program in Figure 1 and recognize that parts of the code (e.g. the bottom three lines) implement a variable swap template. This description synthesizes knowledge across multiple lines of code, so it demonstrates *relational* knowledge, a higher level knowledge in the SOLO taxonomy.

To read templates from pre-defined code, a learner must have understanding of reading semantics knowledge to map syntax and semantics to changes in program state as well as reading templates knowledge to translate this machine-level understanding to a goal/purpose understanding. Weak knowledge of reading semantics would result in a novice having misunderstandings about how the code executes. This could result in them not recognizing a template in the code, recognizing a template but not recognizing it was incorrect, or recognizing the wrong template.

S4, Writing templates (bottom right in Table 1) requires a learner to start with a problem description that contains ambiguity, identify a template that they could use to solve the problem, and implement each component of the template in code. Extracting an objective from a natural language problem description is the first step to writing a template. From there, writing a template is a "reverse mapping" when compared to reading a template: whereas reading templates requires learners to map from code to parts of a template to a template objective, writing templates requires learners to map from an objective (extracted from a problem description) to parts of a template to code.

The bottom right quadrant of Figure 1 is an example of writing templates practice. A learner demonstrates knowledge of writing templates by being able to read a problem description, recognize the need for a previously learned template, and devise a plan which uses the template to solve the problem. In

the example, learners must recognize from the problem description that they can use a variable swap (perhaps cued by the fact that the problem description states "each variable ends up with the original value of the other variable"). They then devise a step-by-step plan which implements the variable swap template, a process utilizing knowledge of reading templates. They can then use this plan and write code to solve the problem, requiring knowledge of reading and writing semantics.

To write templates to solve a problem given a problem description, a learner must have an understanding of reading templates to know the objective and components of templates, and reading and writing semantics to write correct code. If they had weak knowledge of reading templates, they would not be able to recall the templates and would have to solve the problem by creating their own templates. If they had weak knowledge of reading or writing semantics, they would not be able to translate the template to correct syntax or semantics.

3.3. Summary of theory: read before write; semantics before templates

To summarize, we emphasize that the theory defines four distinct skills and sequences instruction, such that knowledge of each skill can be demonstrated and built upon knowledge of previous skills. The sequence emphasizes teaching reading before writing, semantics (features of a programming language) before templates (patterns of use). A key distinction we make is the separation of writing skills into writing correct syntax (S2) and writing meaningful code with the use of a template. By doing so, we can produce instruction which teaches semantics before templates. The structure of this theory has direct implications for instructional design.

4. Instruction: teaching skills incrementally

Having presented structure and sequence for four distinct skills, we now present several new genres of instruction that might be used to teach according to our theory. We do this by presenting a concrete example of learning materials that taught programming constructs (e.g. conditional statements) by teaching four programming skills in the order the theory proposed: Students learn the semantics by getting instruction, practice, and feedback on tracing (S1), then writing correcting syntax (S2), then reading a template (S3) related to that construct (e.g. using conditionals to find a maximum value), and finally on using a template to write code (S4). By providing instruction, practice, and feedback from each skill in the sequence proposed in our theory, this curriculum can support gradual skill development.

We include an outline of the learning materials to illustrate the sequence of constructs taught as well as the sequence of skills taught for each construct:

(1) Introduction
(a) Describe four skills
(b) Describe lesson structure
(c) Motivate metacognitive prompts
(d) Explain how code runs (teaching strategy from Xie, Nelson, and Ko (2018 memory tables))
(2) Data types**
(a) Reading semantics
(b) Writing semantics
(3) Variables
(a) Reading semantics (taught memory tables from Xie et al. (2018) to complete strategy on reading code)
(b) Writing semantics
(c) Reading template: Variable swap
(d) Writing template: Variable swap
(4) Arithmetic operators
(a) Reading semantics
(b) Writing semantics
(c) Reading template: digit processing
(d) Writing template: digit processing
(5) Print statements**
(a) Reading semantics
(b) Writing semantics
(6) Relational operators
(a) Reading semantics
(b) Writing semantics
(c) Reading template: float equality
(d) Writing template: float equality
(7) Conditional statements
(a) Reading semantics
(b) Writing semantics
(c) Reading template: find max/min value
(d) Writing template: find max/min value

*** Some constructs did not include learning a template because the ordering of constructs made it such that learning those constructs did not afford the learning of a new template. The participant needed to learn an additional construct before learning a new template.*

The learning materials began by explaining what it would cover ("basics of Python") and that it contained content to read and understand, practice exercises to attempt, and solutions to exercises with explanations. It also mentioned the importance of metacognition, thinking about one's own thinking (National Academies of Sciences, 2018; Zimmerman & Schunk, 2011), and explained the purpose of the metacognitive prompts contained within the lesson. These

retrospective prompts varied depending on the targeted skill: prompts to retrospectively reflect on code and explain the purpose of each line of code using code comments demonstrated reading semantics; prompts to explain how a given program might function demonstrated reading templates; prompts to preemptively write a plan in plain English for how they intend to solve a given programming problem demonstrated writing templates. The learning materials then provided an overview of how code runs ("typically…one line at a time from top to bottom, left to right") and explained how to read code line-by-line by following a simple "sketching" (Cunningham, Blanchard, Ericson, & Guzdial, 2017) strategy proposed and evaluated by Xie et al. (2018). It then provided an overview of the concepts covered before beginning instruction on the programming constructs.

In developing the learning materials, we assumed no prior programming knowledge, so taught the programming constructs of data types, variables, operators (arithmetic, comparison), print statements, and conditionals in Python 3. We ordered the constructs such that each construct only depended on knowledge of previously learned constructs. We chose Python because it is a common introductory language that appealed to a broad range of students (including non-majors) and did not require more advanced programming constructs such as methods or classes to execute (Ranum, Miller, Zelle, & Guzdial, 2006; Ranum & Miller, 2013). We drew our selected programming constructs from an adaptation of the first case study in *Designing Pascal Solutions* which used basic programming constructs to accomplish a concrete task (verifying a number is a valid passkey) (Clancy & Linn, 1992). The learning materials included both instructional content to read and exercises (and their solutions) for learners to practice applying each skill. We delivered the learning materials as a paper packet of 86 pages, printed single-sided so participants could easily reference previous pages.

We used instruction on conditional statements as an example to explain how the curriculum progressed across the four skills in our theory (S1-S4).

4.1. Instruction on semantics

The first skills the curriculum teaches are reading semantics (S1) and writing semantics (S2). These skills relate to understanding the features of a programming language.

4.1.1. Instruction on reading semantics (S1)

We began by connecting the new construct to previously learned ones and to relatable examples. Given that conditionals came after learning relational operators, we framed conditionals as a way to "do different things based on different relationships." We then provided a relatable example of a situation requiring a conditional: "If I don't have any homework tonight, then I will meet up with my friends." We then define the programming construct: "Conditional statements (also known as *if-statements*) enable different code to execute

based on a given relationship." After this, the instruction continues teaching how to trace code with conditional statements (S1).

To teach S1, the instruction provides examples framed around real-world contexts and incrementally adds complexity. For conditionals, it described a situation where a participant wanted to buy a beverage but only if it cost $1 or less. It then showed the code to reflect this basic conditional where a message instructing them to "buy the soda!" appeared if the `cost <= 1.00`. After explaining what the code in the example did, we added complexity by introducing the `else` statement. We motivated the `else` statement by framing it as a tool "to run different code if the condition executes to false" and expanded upon the previous example by having the `else` condition print a message warning against buying the soda. Because conditionals break the "top down, left to right" control flow they were previously familiar with, we then provided an annotated example showing which lines of code executed and which did not, as shown in Figure 2.

We repeated this process of motivating the need for the added complexity of an `else-if` (a way to add additional branching options) and then adding it to the previous example. After this, we provided practice exercises in the form of tracing questions and asked participants to determine the output of code that contained conditional statements.

Practicing reading semantics (S1) requires a learner to trace code and does not require knowledge of any other skills. This practice consisted of looking at fixed-code questions (McCartney, Moström, Sanders, & Seppälä, 2004) where a learner determines intermediate and final program states for a pre-defined piece of code. Figure 3 shows examples of practice exercises for S1.

If a learner has weak knowledge of S1, they would have misunderstandings about how tokens in the code affect program behavior. Thus, it is ideal for exercises to make misconceptions observable by revealing errors in the intermediate (variable values) and output (print statements) values when a novice traces code. For example, Figure 4 shows an error a learner makes relating to incorrectly tracing code. The learner sketched annotations to the program (writing the values of variables, crossing out lines that did not execute) that were encouraged in the instruction, but not required for that exercise. Tracing exercises that require learners to predict intermediate and final states help identify understanding and misconceptions relating to reading semantics.
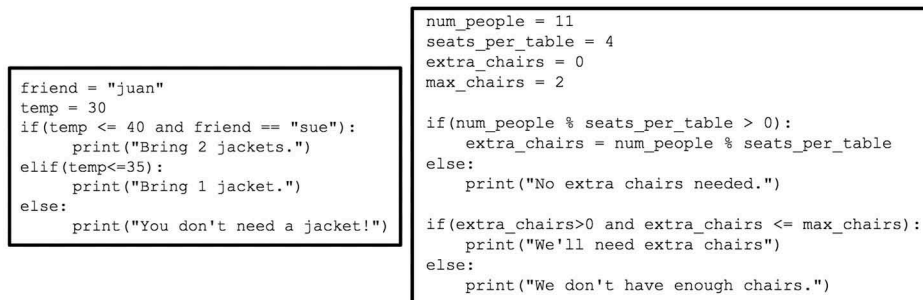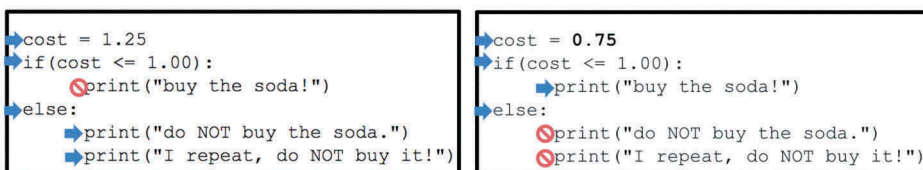


```
cost = 1.25
if(cost <= 1.00):
    print("buy the soda!")
else:
    print("do NOT buy the soda.")
    print("I repeat, do NOT buy it!")
```

```
cost = 0.75
if(cost <= 1.00):
    print("buy the soda!")
else:
    print("do NOT buy the soda.")
    print("I repeat, do NOT buy it!")
```

Figure 2. When teaching participants the skill of tracing (S1) for conditionals, we provided an explanation and this visualization to show how the control flow is different based on different inputs.

```
friend = "juan"
temp = 30
if(temp <= 40 and friend == "sue"):
    print("Bring 2 jackets.")
elif(temp<=35):
    print("Bring 1 jacket.")
else:
    print("You don't need a jacket!")
```
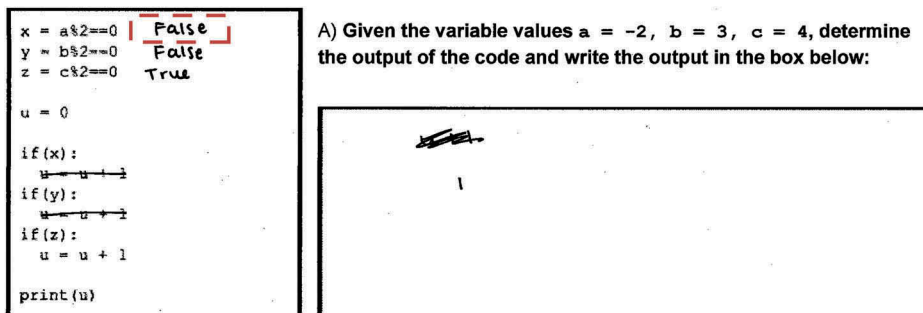
```
num_people = 11
seats_per_table = 4
extra_chairs = 0
max_chairs = 2

if(num_people % seats_per_table > 0):
    extra_chairs = num_people % seats_per_table
else:
    print("No extra chairs needed.")

if(extra_chairs>0 and extra_chairs <= max_chairs):
    print("We'll need extra chairs")
else:
    print("We don't have enough chairs.")
```

Figure 3. Practice exercise for tracing skill (S1) for conditionals. Participant reads the code, crosses out the lines of code that do not execute, and then determines what the code would output.

```
x = a%2==0    False
y = b%2==0    False
z = c%2==0    True

u = 0

if(x):
    u = u + 1
if(y):
    u = u + 1
if(z):
    u = u + 1

print(u)
```

A) **Given the variable values a = -2, b = 3, c = 4, determine the output of the code and write the output in the box below:**

Figure 4. Example of a reading semantics (S1) error. Here, the exercise provided the participant with a code snippet (left) and initial values for variables a, b, and c (above). The participant then traced through the code and determined the output (the dotted red box denotes the likely error the first line as suggested by the inline annotation, which was not required by the exercise).

4.1.2. Instruction on writing semantics (S2)

After practicing reading semantics and receiving feedback in the form of the correct solution and an explanation, learners then move on to learning how to write correct syntax (S2). To teach correct syntax, the instruction defined syntax rules for the construct (e.g. conditionals), adding nuance about the language grammar that was not necessarily visible when learning how to read semantics (S1). Figure 5 shows a table with these syntactic rules for conditionals as well as example code with these rules violated and then corrected.

When learners practice writing semantics, they would also need knowledge of S1 (reading semantics). Practicing writing semantics consists of translating lines of unambiguous natural language (e.g. "Declare a variable profit and set it to 87") to lines of code. Weak knowledge of S1 would result in a learner writing code constructs that do not align with the description of a given line. Weak knowledge of S2 would result in a learner knowing which constructs to

Rule	Broken Code	Fixed Code
The condition in an if statement has a colon at the line.	`if x < 2` `    print ("yes")`	`if (x < 2):` `    print ("yes")`
Indent the code that should run if a condition is true	`if(x<2):` `print ("yes")`	`if(x<2):` `    print ("yes")`

Other conditions (e.g. `elif`, `else`) for a given if statement should be at the same level of indentation.	`if(x<2):`
	`    print ("yes")`
	`    elif(x<5):`
	`        print("ok")`
	`    else:`
	`        print("too big")`
		`if(x<2):`
		`    print ("yes")`
		`elif(x<5):`
		`    print("ok")`
		`else:`
		`    print("too big")`

Else if statements begin with `elif`	`if(x<2):`
	`    print ("yes")`
	`else if(x<5):`
	`    print("ok")`
		`if(x<2):`
		`    print ("yes")`
		`elif(x<5):`
		`    print("ok")`

Figure 5. A table in writing semantics curriculum for conditionals which shows syntax rules relating to the construct and concrete examples of contrasting "bad code" which contains syntax errors with "fixed code" which corrects the errors.

use, but making syntax errors when writing those constructs. In both cases of weak knowledge of S1 or weak knowledge of S2, the learner may write code that contains errors such that the code would not run or the code would have unexpected behavior. To differentiate between the misunderstandings between S1 and S2, the instruction prompts learners to write comments to explain "in their own words" what each line of code is doing. Figure 6 provides an example of an exercise with an S2 error. By understanding what constructs a learner intended to write, we can differentiate between whether their misunderstanding is from a poor understanding of what semantic tokens do (S1) or weak understanding of how to write semantic tokens (S2).

4.2. Instruction on template knowledge (S3)

After learning to read and write semantics for a new construct, a learner then transitions to how to use templates of common code use patterns to apply knowledge of this construct. With each programming construct, we taught a template that reflected the application of that construct (often with other constructs) to accomplish a task.

4.2.1. Templates have an objective and multiple parts or steps

A template consisted of an objective as well as multiple parts or steps required to make the template perform its intended purpose. We included templates in instruction to motivate potential uses of programming and as a scaffolding technique to bridge between learning the features of a programming language and learning to use the language to problem solve. In total we taught four templates:

Figure 6. Example of errors in a writing semantics (S2) exercise which required a learner to translate the unambiguous natural language to Python syntax. The dotted red boxes denote errors in S2 relating to conditionals (forgetting colons after the conditional statements). The learner's comments reflect correct/intended behavior, but the written code is syntactically incorrect, suggesting an S2 error.

- *variable swap*: switching the values stored in 2 variables by using a temporary variable and variable updates.
- *digit processing*: accessing specific digits in an integer with multiple digits by repeatedly using modulus to access the rightmost digit, and then dividing and using integer truncation to drop the rightmost digit from the input integer.
- *float equality*: checking if two floats are approximately equal by comparing the absolute difference to a small threshold value using a relational operator.
- *max/min*: finding the maximum or minimum of three (or more) numbers using conditionals and the `and` operator.

Instruction on templates comes after instruction on reading and writing semantics and is also divided into reading and writing portions. The instruction first teaches learners how to read templates and recognize the purpose and components of a template that utilizes the construct they are learning (S3), then to apply these templates to write code to complete an objective (S4).

4.2.2. Instruction on S3 (reading templates)

Learning to read templates is fundamentally about learning patterns of computation that solve a class of problems. Our learning materials introduced learners to the "max/min" template, which they could use to find the maximum or minimum value from multiple variables that all store numbers (data structures were out of the scope of this instruction). Before introducing a template, we provided an example or visualization that attempted to make the objective and steps of the template more relatable and concrete. For the max/min template, we provided an explained visualization showing the step-by-step process of finding the maximum value from 3 variables. Figure 7 shows part of that visualized explanation.

The instruction then translated the same process to code and explained the relationship between the visualization and the code. Our intention was to

My next step is to compare B and C.

If B is larger than C, then I know B is the largest number. I do not have to compare A and B again because I already checked A previously! If we find that B is not larger than C, we can say that B is not the largest number and remove it.

Figure 7. Instruction for reading a template (S3) often began with an example or visualization to make the template objective and steps more concrete. For the template which uses if statements to find the max/min value among variables, we provided a visualization showing how learners could use pairwise comparisons to find the max/min values of more than 2 values.

promote comprehension and abstraction through mutual alignment of two (perhaps partially) understood situations (Kurtz, Miao, & Gentner, 2001). After doing so, the learning materials explicitly provided the steps involved in a max/min template in natural language:

To find the maximum (largest) or minimum (smallest) value, we do the following:

(1) *Use if statements to check one value against all other remaining values*
    (a) *We may need a compound conditional statement (using* and*)*
(2) *Ignore the value we just checked and repeat step one if there are at least two remaining values*
(3) *If there are no more values to compare against, then we reach our else condition.*

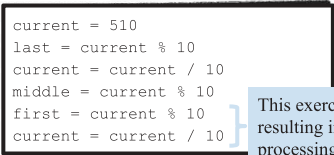The learning materials then transitioned to practice reading this template.

When learners practice reading templates, they would also need knowledge of S1. Practicing reading templates consists of looking at previously written code and determining if it correctly implemented a given template and if not, what part of the template was not properly implemented. Weak knowledge of S1 would result in a learner not being able to correctly trace the code. Weak knowledge of S3 would result in a learner not recognizing how parts of the code map to different components of a template. In both cases of weak knowledge of S1 or weak knowledge of S3, the learner would have trouble looking at code and identifying if it properly implemented a template. To differentiate between misunderstandings of S1 and S3, our instruction had learners read code and identify if it correctly implements each part of a template, while also explicitly tracking the state of the program. The instruction observed their ability to trace code (S1) by having them update memory tables (Xie et al., 2018) with variable declarations and updates. Errors relating

to changing program state suggest weak knowledge of S1. Failure to identify whether code implemented parts of a template correctly suggested weak knowledge of S3. Figure 8 illustrates an S3 error for the float equality template, which a learner could use to check if two floats are approximately equal.

4.2.3. Instruction on writing templates (S4)

After engaging with instruction on how to read a template, the lesson moved on to teaching how to apply a template to fulfill a computational objective. Obviously, problem-solving is important to this process, but out of the scope of this instruction. In our learning materials, we focused on the more narrow scope of providing rules to address errors that could arise when translating a template into code. Whereas writing semantics (S2) instruction specified rules to prevent syntactic errors, writing templates (S4) instruction specified rules to prevent logic errors which would be syntactically correct but result in code which did not perform to the specification of the template. Figure 9 provides an example of instruction to address errors relating to conditionals.

When learners practice writing templates, they also need knowledge of S1, S2, and S3. Practicing writing templates is consistent with typical code writing practice where learners read a problem description and must write code to solve the problem. Weak knowledge of S1 or S2 results in errors similar to what we could expect in practice for writing semantics. Weak knowledge of S3 results in a learner either not recognizing how they can use a template to solve the problem or not being able to write code to implement the template. To differentiate between misunderstandings, we asked learners to first write a plan to solve the problem in natural language, then write code to complete the task described to them, then annotate each line similar to what they would do when practicing writing semantics (S2). An incorrect plan suggests weak

3)
```
current = 510
last = current % 10
current = current / 10
middle = current % 10
first = current % 10
current = current / 10
```
This exercise reversed these 2 lines, resulting in incorrect digit processing. (correct answer is c)

Variable Name	Variable Value
current	510 → 51 → 5
last	0
middle	0̶ 1
first	1
~~current~~	

Select one:
- (a) The starting value is not an integer.
- (b) Digits are not properly extracted from starting value.
- (c) The starting value is not updated properly.
- (d) This code processes digits correctly.

Figure 8. Example of a reading templates (S3) error relating to the template to extract digits from a number. The learner erroneously thought that the extraction of the digit was not being done correctly (with the modulo operator) and selected option B. They correctly traced the code, as demonstrated by the properly completed memory table, so they did not make an S1 error. Therefore, they made an S3 error, likely failing to realize that the code was not properly updating the starting value `current`.

Rule	Bad Code	Explanation
Compare a value against all values that have NOT been eliminated.	```if (a > b and a > c): print(a) elif(b > c and b > a): print(b) else: print(c)```	The variable `a` has already been eliminated and does not need to be compared against again. This leads to unnecessary confusion.
If there are more than 2 values to compare, you will need combine if statements with `and`.	```if (a > b): if (a > c): print(a) elif(b > c): print(b) else: print(c)```	The bold code should be included as a compound conditional in the if statement above it: `if(a>b and a>c)`
Make sure the conditional statement matches the variable you are outputting.	```if (a > b and a > c): print(c) elif(b > c): print(b) else: print(c)```	The first if statement is checking to see if the variable `a` is the maximum but this code prints the variable `c`.

Figure 9. A table in writing templates curriculum for template to find maximum or minimum value from multiple number variables using conditionals. *Rules* to support the correct implementation of the template are shown in the left column, with the middle *bad code* column demonstrating code with a violation of the rule (in bold), and the rightmost *Explanation* column explaining the error.

knowledge of S4. We can differentiate between weak knowledge of S1 and S2 by looking at a learner's line-by-line annotations. Figure 10 provides an example of an S4 error where a learner wrote a plan which incorrectly defines a variable swap (this example is from a different unit than conditionals; we selected it because the error is more apparent here).

4.3. The post-test used a variety of exercises to evaluate each skill

We developed a summative assessment to measure how well learners were able to apply the four skills (S1-S4) in the context of the programming constructs and templates covered in the instruction. The assessment measured participants' ability to read and write semantics and templates. It consisted of seven questions which roughly increased in difficulty, based on face validity and performance of pilot tests with novice programmers.

We designed the questions to assess specific programming skills. Questions assessing reading semantics (S1) asked learners to trace stand-alone code segments that were not part of a larger code base, determine what initial program state would result in a given final state, and comment their own code; these questions were similar to tracing questions in prior studies (e.g. Lister et al. (2004)) and only the experimental group had practice on them. Questions

Example Use Case: Variable Swaps Practice

Joan has 2 battery-powered flashlights that also report the percentage of power left in the battery, stored in variables `power1` and `power2`. Her second flashlight has more energy in the battery, but it broke! She plans to swap the batteries in the flashlights so the working flashlight has more power. She wants you to write code to swap the values stored in these variables.
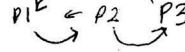
1) In plain English, describe a step-by-step plan for solving the problem:

Figure 10. Example of a writing templates (S4) error relating to the variable swap template. The participant was told to provide "a step-by-step plan for solving the problem" which involved swapping variables. They incorrectly planned the swap, suggesting an S4 error.

assessing writing semantics (S2) asked learners to translate an unambiguous description of program steps, and asked them to write correct syntax; while both groups had practice on these questions, the control group had more practice. Questions assessing reading templates (S3) asked learners to summarize in natural language what a program did; these questions were similar to code explanation questions in prior studies (e.g. Whalley et al. (2006)) and only the experimental group had practice on them. Questions assessing writing templates (S4) provided learners with a problem description (with ambiguity) and asked them to write a plan in natural language to solve the problem; while both groups had practice on these questions, the control group had more practice.

Some questions had multiple parts and assessed different skills at different parts. For example, we asked participants to write a plan, code, and comments for a program that determines whether a number is valid and prints appropriate messages. Given a four-digit number as input, the program should determine if the input has a correct "check digit." The check digit for a number is its rightmost digit and is correct if the check digit is equal to the remainder of the sum of the other three digits divided by seven. The program should print a given message if the input is valid and a different message if the input is not valid. We adopted this task from the first case study in *Designing Pascal Solutions* (Clancy & Linn, 1992). The learner solved this problem across three explicitly scaffolded steps: Firstly, they determined whether a specific template was applicable to solving the problem (S3 knowledge) and used a template create a step-by-step plan for solving the problem in plain English (S4 knowledge). In this case, the digit processing template (see Section 2.3) that they learned in the curriculum could

have been helpful. Next, they used their plan to write code to solve the problem (S2 knowledge). Finally, they commented the code, explaining the function of each line (S1 and S3 knowledge).

When learners write code to problem solve (such as in the example in the previous paragraph), they use skills relating to templates (S3, S4) before using skills related to semantics (S1, S2); this is backward relative to how we taught the skills. Teaching templates before semantics may better motivate problem-solving, but it can result in a problem-specific understanding of what constructs can do. To better support a more general understanding of what programming construct can do, we teach semantics before templates.

After each question, we asked them the same three questions to understand their perceptions about working through the question: (1) After reading the problem statement, what did you think of first? (If you were reminded of a construct in general or a general structure of solution, please note that.); (2) What was the most difficult part of this problem?; (3) On a scale of 1–7, rate your confidence in your solution: (Circle one). We adapted the first 2 questions from Fisler and Castro (2017). We asked what they thought of first as a manipulation check to determine if they thought about the templates we taught.

4.4. Comparing our curriculum to similar

To clarify nuances to our curriculum, we compare it to the similar curricula by Clancy and Linn (1992), de Raadt (2008), Hertz and Jump (2013), and Thota and Whitfield (2010).

Because we draw heavily from the work by Marcia Linn's use of templates for instruction (Linn & Clancy, 1992), our instruction on reading and writing templates (S3, S4) is very similar to how Clancy and Linn (1992) taught introductory programming in *Designing Pascal Solutions*. This introductory textbook taught the programming language Pascal through a case-based reasoning model (Kolodner & Guzdial, 2000), where each chapter was a case study with social and functional context. They relied on a model of cognitive apprenticeship (Collins, Brown, & Newman, 1988) to teach programming problem-solving, so situating the template in a relatable context was key to scaffolding. The textbook guides learners through the process of problem-solving with a template, beginning with conceptualizing the problem description, then breaking the problem into sub-problems, and then iterating on solutions. Throughout this process, the textbook asks self-test questions to allow students to assess their own understanding. The textbook lacked explicit instruction to teach programming constructs (S1 and S2 in our theory), instead relying on 1) prior learning of the basics of constructs, or 2) extensive repeated contextualized practice to inductively develop understanding of programming constructs. A selective evaluation of this curriculum found that learners that read expert solutions and commentary instead of writing their own solutions

outperformed learners that wrote their own solutions (Linn & Clancy, 1992), and another evaluation of an online template library had positive results (Schank, Linn, & Clancy, 1993). But as a whole, this curriculum primarily emphasized the skill of using templates to write code (S4 in our curriculum). Other instructional designs have also used templates to teach problem-solving skills (Muller, Haberman, & Ginat, 2007; Proulx, 2000), but they too do not emphasize differences between reading and writing skills. While *Designing Pascal Solutions* and similar instructional designs used templates to emphasize the problem-solving process, our learning materials emphasize instruction on programming skills that began with tracing instruction, as the empirical work we reviewed earlier (see Section 2.1) suggested.

Hertz and Jump (2013) contributed a curriculum design that provided more robust instruction on tracing, but lacked a theoretical foundation and did not focus on other skills. They sequenced instruction within constructs to first provide explicit instruction, examples, and practice on tracing ("trace-based teaching") involving intricate sketched program traces, and then provided practice writing programs. This instruction was all within a traditional spiral pedagogy from simpler to more advanced language constructs (Shneiderman, 1977). The work did not explicitly mention any theoretical grounding, so the mechanism of learning is not clear. Trace-based teaching had greater coverage of CS concepts than our instruction because it covered an entire Java-based data structures course. This method of teaching was similar in approach to our tracing (S1) instruction, as our instruction utilizes work by Xie et al. (2018), which partially builds off of trace-based teaching. Because Hertz and Jump (2013) focused on improving instruction on tracing, instruction on other skills (e.g. writing) were out of the scope of that work. In contrast to this work, our lesson had a theoretical grounding and focused on the development of four distinct skills that begin with tracing and ended with writing code with templates.

Prior work by de Raadt (2008) assessed four "aspects" similar to the four skills we proposed, but the instruction they created did not distinguish between these aspects. For assessment, they proposed a quadrant along dimensions that we viewed as parallel to ours: "comprehension-generation" (read-write in our skill decomposition) and "knowledge-strategy" (semantics-templates in our skill decomposition). They proposed these divisions for assessment and not for instructional design. Their instructional design integrated instruction on strategies (Soloway, 1986) which we viewed as similar to our framing of templates. While de Raadt (2008) incorporated strategies in their instructional design, they did not distinguish between the read-write dimension in their instruction (only in assessment).

Thota and Whitfield (2010) proposed and evaluated a holistic approach to designing an introductory object-oriented programming course that used the SOLO taxonomy to define assessment criteria. They grounded their pedagogy in phenomenography and constructivism, focusing on how a learner would

understand it. They explored these learning approaches and compared them to course performance. Whereas the emphasis of their work was in aligning curriculum with learners' approaches and preferences, our curriculum emphasizes the progression of programming skills, something not explicitly explored in Thota and Whitfield (2010).

Taken together, these curricula have some similarities in use of templates (Clancy & Linn, 1992; de Raadt, 2008), supporting lower-level skills Her (Hertz & Jump, 2013; Nelson et al., 2017), and skill progression (Thota & Whitfield, 2010), but we viewed our curriculum as unique in that it explicitly scaffolds a gradual transition from lower-level (e.g. tracing) skills to higher-level writing skills.

5. Evaluation of theory: exploratory experimental study

Evaluating any theory is an incremental process. Given that we have just proposed our theory in this article, evidence for its affordances to explain and predict learning and inform instructional design will necessarily have to develop across multiple publications. Evaluations to conduct include psychometric studies to determine the construct validity (Allen & Yen, 2001) of the practice items we proposed, feasibility studies to determine how instruction based on this theory could integrate with introductory computer science courses, and studies to measure learning outcomes for learners with diverse prior knowledge, learning contexts, and motivations. Because explicit instruction on incremental skills is a focal point of the theory we proposed, we sought to provide some initial evidence to evaluate the effects of such instruction on those skills. Given the broader work necessary to evaluate our theory, we focused on a detailed, formative investigation into learners' experience and outcomes with the specific structure and sequence of instruction we proposed, relative to more conventional learning materials that focus explicitly on program writing.

Our study was a between-subjects study to understand how learning materials reflecting our theory improve completion rates, reduce errors, improve the depth of understanding, and increases engagement. We provided the experimental condition a curriculum that reflects this theory (as described in Section 4) by labeling and providing practice for each skill. In contrast, we provided the control group with a curriculum with the same instructional material to read but no labeling of different skills, and practice on only writing skills (writing semantics, writing templates); we choose to have the control condition have writing-focused practice because this was consistent with much of the related work we found. To balance the amount of practice learners received, we provided the control group with additional writing practice when compared to the experimental group, which had reading and (less) writing practice. So while the type of practice varied between groups, we attempted to balance the *amount* of practice. Figure 11 compares the instruction and practice learners of each condition received.

control practices only writing skills (black squares)
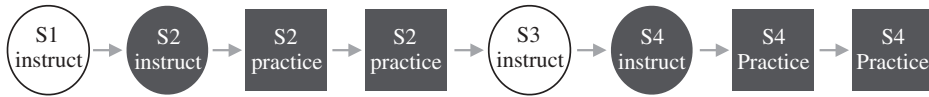
experimental practices both reading (white squares) and writing skills
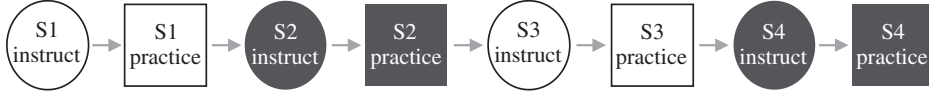

Figure 11. Diagram of differences in learning materials for study conditions where circles are instructional content and squares are practice and the numbers S1-S4 reflect the skills in the theory (see Section 3). Whereas the control condition got practice only in writing semantics (S2) and writing templates (S4) but more of it, the experimental condition got practice in reading semantics (S1) and reading templates (S4) as well.

5.1. Hypotheses: explicit practice improves learning & engagement

We make the following hypotheses to predict the effect of instruction that follows our theory of instruction:

H1: Novices who practice each skill will be able to *complete* more programming tasks. We predicted that novices who practiced each skill will have fewer repeat errors and will therefore be more able to complete more programming problems. In contrast, we predicted that novices who only receive practice consisting of multiple skills in combination will be less able to complete programming problems because weak knowledge of one or many skills hinders their ability to progress.

No difference in the completion rate would suggest that the different types of practice exercises that the experimental group received was unnecessary scaffolding as participants in the control group were still able to complete exercises that confounded multiple skills. The control group having a greater completion rate could suggest that practice for each skill was unnecessary, confusing, or inauthentic.

H2: Novices who practice each skill independently will make *fewer errors*. We predicted that errors in previous skills can appear in practice of later skills if a novice does not develop sufficient understanding of previous skills. If a novice did not develop sufficient understanding of earlier skills (S1, S2), our theory suggests that errors relating to more advanced skills (S3, S4) will compound because the skills build off of each other. That is, weak knowledge of S1 (tracing) can result in errors that reappear in later practice (e.g. S4, writing templates). We predicted that explicit instruction for each of the 4

skills can reduce the number of each type of error made, thus improving novice performance in reading and writing programs.

If novices who practice each skill incrementally were to make *more* errors (than novices who practiced using the skills in aggregate), then this could suggest that earlier skills (e.g. S1, S2) are easier to learn in combination.

H3: Novices who practice each skill will have a *greater depth* of understanding of the skills. Our theory claims that S1 S4 are incrementally built off of each other, so we predicted that providing explicit instruction and practice for these skills will result in a greater depth of understanding, as demonstrated by novices' ability to use these skills in unison and explain what they are doing for tasks such as writing code to complete an objective.

No change in the depth of understanding would mean that novices who learn the skills in conjunction (control group) had an equivalent depth of understanding, suggesting that it is unnecessary to differentiate between skills. If practicing each skill results in a *lesser* depth of understanding, then this would suggest that there may actually be benefits to practicing multiple skills in conjunction as current CS1 instruction typically does.

H4: Novices who practice each skill will demonstrate more *engagement* in the learning process. We predicted that a novice who learns each skill incrementally will engage more with the learning experience because teaching skills separately enables them to make perceptible progress. In contrast, novices could feel overwhelmed by instructional content that teaches multiple skills in unison without first developing understanding of separate skills.

If practicing each skill incrementally results in decreased engagement when compared to practicing an aggregate of multiple skills, this could suggest that novices found the incremental practice to be too easy or inauthentic to be helpful, or too mechanistic to be interesting.

5.2. Participants: undergraduates who were novice programmers

We recruited undergraduates who had an interest in programming but minimal prior experience (no formal coursework, <10 hrs ever spent programming) and were fluent in English. It was important for us to find "rank novices" who had as little prior programming as possible to minimize confounding situations where participants previously learned some programming concepts through a different form of instruction. We recruited from three introductory informatics and human-centered design & engineering courses from a large public university using in-person announcements, emails, and flyers. We received

approval from our Institutional Review Board prior to conducting this human subjects study and received permission from instructors prior to contacting their students. We framed the study as an opportunity to explore programming by spending a day learning it, offering no compensation. We stratified participants into (typically pair-wise) groups based on four factors in order of priority: prior programming experience, year in school, gender, and major. Participants within each group were randomly assigned to a condition.

After group assignment, we ended up with five participants in the experimental condition and four in the control condition. While the sample size is not large enough to make quantitative claims of statistical significance and effect size, we were able to make detailed analyses of each individual participant. All participants were undergraduates from the same institution except one (in experimental condition) who had completed her Bachelor's degree and was taking extra courses to prepare to apply for graduate school. All participants were pursuing or received a different humanities or social sciences major (e.g. communication, medical anthropology, economics), with no participants studying computer science, engineering, or informatics. Five participants identified as female and four as male; we balanced gender identities across conditions. Participants could identify as multi-ethnic, so five identified as Asian, four as White, one as Pacific Islander, and one as African-American. We relied on self-reported experiences to serve as proxies for prior knowledge. The control group had more prior knowledge, with two participants currently enrolled in at least one introductory course requiring programming and also having previously used an online learning tool (e.g. MIT Scratch, Codecademy). In contrast, none of the participants in the experimental group reported having taken or being enrolled in a course requiring programming at the time of recruitment, although one participant reported reading "the first couple chapters" of *Learn Python the Hard Way* (Shaw, 2017).

5.3. Procedure: learning from 1 of 2 instructional material variants

We conducted the study simultaneously in two classrooms with participants separated by condition. We began the study by explaining the objective, and had participants complete a pre-survey with questions on demographics, fatigue, mindset, and a computer programming self-efficacy scale from Ramalingam and Wiedenbeck (1998). We then explained the study schedule and moved on to the instructional portion of the study.

For the instructional time, we provided participants with a paper packet with instructional material, practice problems, and solutions to the practice problems. Participants in different conditions received different packets. We allotted them 3 h to work through the content at their own pace and learn everything they could to perform well on the post-test. We encouraged participants to work sequentially through the packet and attempt problems before looking at the

solution, but did not enforce this policy. To track their progress, we asked participants to initial each page they completed. To determine effort on practice exercises, we had participants self-report after each practice exercise whether they produced an answer without consulting the solutions and whether their answer matched the provided solution. Participants paced themselves and took breaks as necessary. We allowed participants to ask questions, which we answered by either referring to content in the paper packet or informing them that we were unable to answer the question until after the conclusion of the study. We recorded the questions asked and who asked them.

After the instructional period, we gave the students a 10- to 15-min break. Towards the end of the break, we gave them a brief (3 min) mental rotation test (Vandenberg & Kuse, 1978) as a distractor task. The objective of the distractor task was to mitigate short-term, temporary learning gains related to taking the post-test shortly after learning the material. A distractor task occupies participants' working memory with content unrelated to the post-test so participants would rely more so on long-term memory when working on the post-test (Liu & Fu, 2007; Stadler, 1995). We selected an assessment of spatial orientation (Ekstrom, Dermen, & Harman, 1976) as the distractor task because prior work has found a correlation between spatial reasoning and programming ability (Cooper, Wang, Israni, & Sorby, 2015), although we did not compare performances for this study because of confounding factors relating to fatigue and engagement. After the break concluded, participants spent 60 min taking the assessment and then completed a postsurvey.

6. Results: evaluating our hypotheses

In this section, we provide results to the evaluation we defined in the previous section. These results attempted to provide some initial evidence relating to the hypotheses we proposed relating to differences in the completion of practice, errors made in the post-test, depth of understanding on the post-test, and engagement throughout the study.

The objective of this evaluation was to provide evidence to suggest that practicing each of these skills actually improved learning outcomes. We wanted to try to observe longitudinal changes to understanding (H1) and engagement (H4) as participants learned from instruction that reflected our theory compared to more traditional instruction which did not provide practice on each skill. After participants engaged with these different learning materials, we wanted to understand differences in learning outcomes. Specifically, we sought to measure differences in understanding for each skill to provide evidence to support our sequencing of skills (H2). In addition, we sought to measure differences in the depth of understanding to provide evidence to suggest that explicit practice benefits learners (H3).

We choose to quantify our data on participant performance because additional qualitative prompting would have further fatigued participants and made data more difficult to interpret. Our learning materials took about 3 hours to work through because they spanned from basic concepts (to account for participants having minimal prior knowledge) to more advanced concepts (to enable coverage of multiple templates). As a result, we needed to balance collecting rich data on participants' thought processes with interrupting their learning process and excessively fatiguing them. We conducted pilot studies to evaluate the use of qualitative prompts such as think-aloud, follow-up questions, and annotations to indicate confusion. While these prompts could have provided rich data on learners' thought processes, we found that learners' rate and quality of responses varied individually and also tended to diminish as they progressed further and became more fatigued. We ended up using brief follow-up questions (multiple choice, short answer) and metacognitive prompts of planning and commenting code because they were lightweight and beneficial enough for participants to engage with them. We quantified self-reported feedback, scoring of the post-test, and qualitative coding of metacognitive prompts to identify potential longitudinal trends as learners progressed in the learning materials as well as potential trends relating to understanding different skills. We do not attempt any statistical analysis due to our small sample size and only report frequency counts. This initial evaluation provided initial evidence to support our theory and also potential trends to motivate future investigation.

6.1. H1: experimental condition completed more practice exercises

H1 predicted that the experimental condition would be able to complete more programming problems because they received practice in each of the four skills, and that results in better understanding of the skills. To evaluate completion of the instructional content, we looked at participants' self-reported measures of whether they solved a practice exercise and/or got the exercise correct without consulting the solution, which was provided.

We found a difference in completion rate between groups for the exercises they attempted, as shown in Figure 12. For the earlier units on data types and variables as well as the unit on *print* ($\approx 1/3$ of the lesson), participants in both conditions were able to both provide an answer and provide a correct answer without looking at the solutions. In later units (e.g. arithmetic operators, relational operators, conditionals), only about half of the control group produced an answer on their own and typically only 1 got the correct answer. In contrast, most if not all of the experimental group were typically able to provide an answer without looking at the solutions. But while participants in the experimental condition more consistently provided answers to exercises without looking at solutions, they often did not get exercises completely
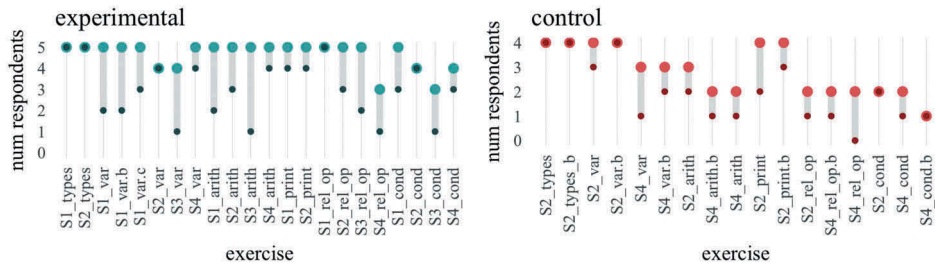
Figure 12. Completion and correctness rates for *all* exercises. Number of participants in each condition who reported completing each exercise (larger lighter dot) and getting an exercise correct (smaller, darker dot) without looking at the solution. While the experimental condition tended to make more errors (in part because of confusion with S3 exercises specific to their condition), they still tended to consistently complete exercises before consulting the solutions. In contrast, the control condition tended to complete fewer exercises as the instruction progressed.

correct. This could be because of issues with the practice exercises, as some participants expressed confusion about some exercises' instructions and response options (especially for practicing reading templates, S3). Comparing completion and correctness for all the exercises between conditions is confounded because many exercises are different between condition.

For more direct comparisons of performance, we compared rates of completion and correctness for the 10 exercises involving writing code that were present in both conditions' lessons. Only a subset of the writing exercises (S2, S4) are the same between conditions and they are shown separate from other exercises in Figure 13.

Every participant in the experimental group was able to produce an answer without consulting the solutions for most (6 out of 10) exercises. Furthermore, all but one participant from the experimental group got most (7 out of 10) of the exercises correct. In contrast, only half of the control group participants were able to produce an answer for half of the exercises. Furthermore, at most half of the control group were able to produce a correct answer for 7 out of 10 of the exercises. From this, we still observe that the experimental group was more consistently able to produce answers that were typically correct. We again see the trend that the completion and correctness rate tended to decrease after the first two units. These findings align with our hypothesis H1, where novices who get practice with each skill will be able to complete more programming tasks because they have a better understanding of earlier skills.

We can say that there is evidence to support H1, as participants in the experimental condition self-reported completing more exercises in the instruction and also reported getting them correct throughout the lesson, whereas the control group's rate of completion and correctness diminished later in the lesson.
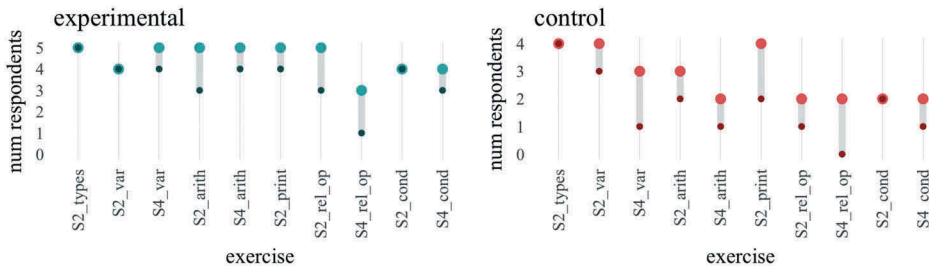
Figure 13. Completion and correction rates for *identical exercises* in both conditions. Number of participants in each condition who reported completing each exercise (larger lighter dot), getting an exercise correct (smaller, darker dot). When controlling for the exercises participants attempted, we see a more clear trend that the experimental group more consistently completes exercises and gets them correct even as they progress further in the lesson. In contrast, we see that after the first two units (data types, variables), there is a drop off in the rate of completion and correctness for the control group.

6.2. H2: experimental condition made fewer errors, especially on later skills

H2 predicted that participants in the experimental group would make fewer errors because practicing each skill would help better develop their programming skills by doing so incrementally. We predicted that if novices did not adequately understand programming skills with practice, errors would compound from earlier skills (S1, S2) to later skills (S3, S4), resulting in more errors in the later, more advanced skills. To evaluate error frequency and type, we broke down the post-test score by the four skills as shown in Figure 14. It shows that while there was large variation within both groups, participants in the experimental condition performed better. In part because of the small sample size and correcting for repeated statistical tests, we found no significant difference between the performance of the control and experimental group participants. Regardless, we found some evidence to support H2, as participants in the experimental condition made fewer errors on the post-test, especially in the later skills of reading and writing templates (S3 & S4).

6.3. H3: depth of understanding is greater for experimental group participants

H3 predicted that the experimental group would demonstrate a greater depth of understanding because they practiced each skill incrementally. To evaluate the depth of understanding, we analyzed participants' responses to the metacognitive prompts in the post-test. These included retrospective code comments, preemptive plans for the code they would write, and code explanation prompts. We developed our codebook based on the SOLO taxonomy (Castro & Fisler, 2017; Sheard et al., 2008) and followed a coding method from (Lister et al., 2010).
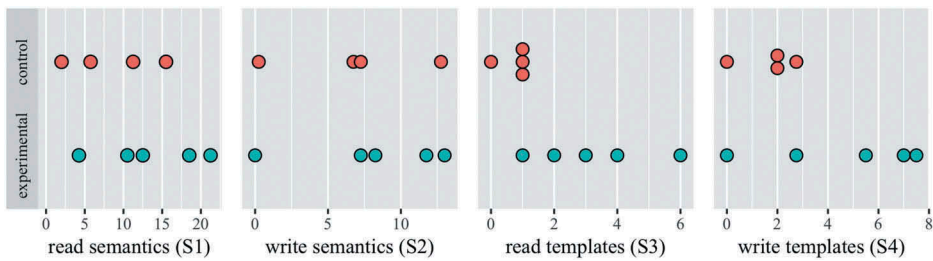
Figure 14. Post-test scores by skill and in total for participants in each condition. Overall, the experimental condition tended to score better than the control condition across all 4 skills. These differences were more notable for the later skills of reading and writing templates (S3, S4).

Following the process of Castro and Fisler (2017), we defined a multi-strand SOLO taxonomy for our codebook.

Our codebooks are as follows:

The adapted SOLO taxonomy reflects a hierarchy of depth of understanding.

- Relational (R): A summary of what the code does in terms of the use of a template (the forest). Learner provides reference to a template which has an objective and integrates it into a task. This reflects knowledge of S3 and/or S4 (and therefore S1 and/or S2) in our quadrant of skills.
- Multi-structural (M): Description relating to more than 1 line of code but not a template. This reflects knowledge of S1 and/or S2 in our quadrant of skills, but not S2 or S3.
- Unistructural (U): Description of one line of the code. Reflects S1 and/or S2 knowledge (but not S3 or S4). For qualitative coding of comments only.
- Other (O): Any other description of the part or all of the code, displaying no real evidence of understanding of the code as a whole; response has little to no alignment with code being described.
- No answer (NA): No answer provided or answer is unintelligible.

The adapted SOLO taxonomy above assumes completeness at each level, but this is not always the case. Another dimension to the depth of understanding is the completeness or lack thereof:

- Incomplete (I): Learner provides a response which is partially correct, but is missing parts or details that would make it complete.
- Error (E): Learner provides a response which has some correctness, but also some inaccuracies.

To assess the reliability of the coding scheme, three authors independently categorized all participant responses for all questions (186 responses in total).

Disagreements on the first pass were minimal and were due to ambiguities in the code definitions. After an additional pass, all three authors came to 100% agreement on every code across all responses.

We found that in all types of questions (explain in plain English, under what conditions, write a plan to solve this problem, comment your code), participants in the experimental group produced responses that reflected higher level *relational* understanding. Figures 15 and 16 show this difference in depth of understanding. Figure 15 shows that for questions that assessed reading template (S3) knowledge (explain in plain English, determine under what conditions something was true), all participants from the experimental group who answered provided multistructural responses which reflected knowledge of reading semantics (S1) and relational knowledge which reflected knowledge of reading templates (S3). In contrast, at least half of control participants provided responses that reflected incomplete knowledge on all question types, producing responses that were incomplete, lacking in evidence of understanding (other), or not responding at all. Figure 16 shows that participants in the experimental group wrote more comments which reflected relational (S3) knowledge than participants in the control group, although one participant in each condition did not write any comments. For one of the questions (7C), three out of five experimental group participants wrote comments where most lines demonstrated relational understanding. In contrast, only 1 of the 4 control group participants wrote any comments at all for the same question, and a lesser proportion of their comments reflected relational understanding. A confounding factor to this analysis is non-response, as some participants in both conditions noted feeling fatigued by the time they took the post-test. Regardless, we find that experimental group demonstrated higher level understanding at least twice as often as the control group did.

We found evidence to support H3, as participants in the experimental group produced question responses and code comments which reflected greater depth of understanding (as determined by the SOLO taxonomy) at least twice as often when compared to participants in the control group.
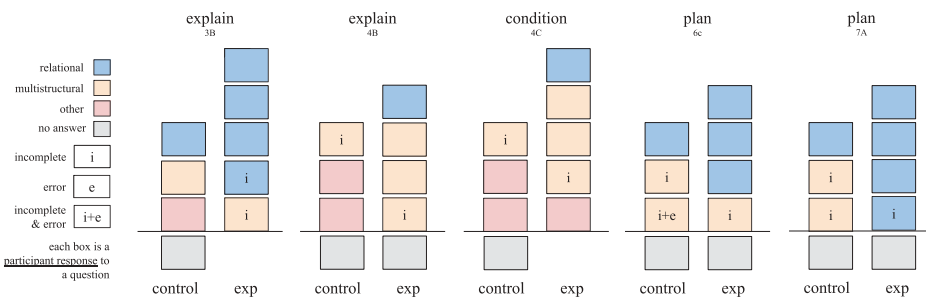

Figure 15. SOLO levels of understanding for post-test questions by participant. These questions asked participants to explain in plain English (3B, 4B), determine under what conditions something is true (4C), and plan code to solve a problem (6C, 7A). Each colored block represents a participant's response to an exercise. The experimental condition shows a higher level of understanding (more relational responses).
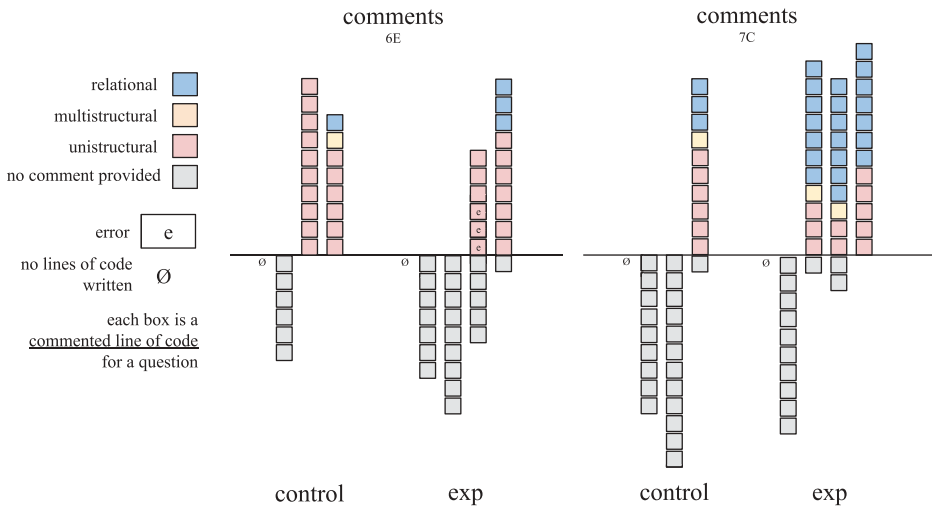
Figure 16. SOLO levels of understanding for comments to participants' code in writing templates (S4) post-test questions. We asked participants to write a comment to explain each line of code they wrote. Participants in the experimental condition tended to write comments which reflected the relational understanding of the templates they applied to solve the question.

6.4. H4: engagement varied by person

H4 predicted that because a novice learned skills incrementally, they would be more engaged with the learning experience.

To understand engagement, we reviewed researcher notes of participant interactions during the instruction and also looked to postsurvey questions. Researcher notes focused observations on participants (actions suggesting engagement or lack thereof) and responses to conversation that relate to fatigue, interest, and motivations relating to study participation. Postsurvey questions asked participants to rate on a numeric scale to what extent they found the content of the learning materials interesting, gave their best effort on the learning, and completed practice problems without consulting solutions. We also asked them what they did and did not enjoy about the study as free-response questions.

Overall, we found that the variation for participants within the same condition was too great to identify trends in engagement between conditions. When considering researcher notes, differences in participant behavior existed in both conditions, with some participants in both conditions able to engage for extended periods of time without interruption, and others being distracted (e.g. a member of the control group stepping out to buy coffee, a member of the experimental group stepping out for a phone call). When considering postsurvey response data, we found that Likert response data tended to vary wildly within conditions (as shown in Figure 17). We found responses to free-response questions to be unremarkable as they focused on features of the study that were
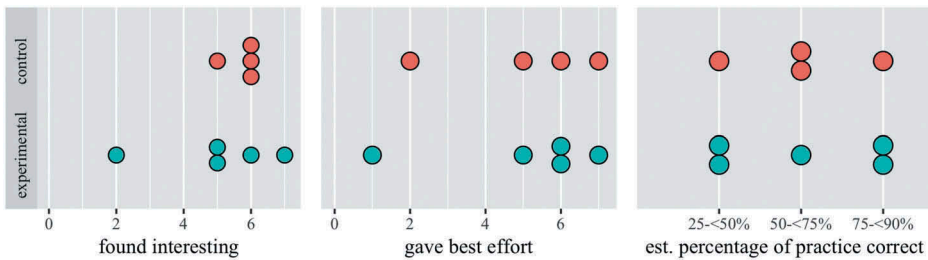
Figure 17. Participants' survey responses to questions relating to engagement with the learning materials, by condition. Engagement varied by person, so we did not identify any trend in engagement between conditions.

consistent between condition; for example, participants in both conditions noted that they felt the medium of the learning materials (80+ pages) and the length of the study (in total 5–6 hours) hindered engagement.

We can say we did not find much initial evidence to support or refute H4. We observed variation in individual participant behaviors and found postsurvey responses indistinguishable between conditions. Further investigation into engagement and perhaps refinement of the instructional design and measurement tools could yield stronger evidence.

7. Discussion: theory and "piece by piece" instruction may have helped

By proposing a theory of instruction for CS1 skill development, developing concrete learning materials from that theory, and evaluating potential learning outcomes with those materials, the contributions of this paper attempt to improve the design of CS1 instruction. In this section, we review each of the three contributions, offer interpretations of our results, note limitations, propose future work, and describe implications for this work to both research and practice.

7.1. Contributions: theory of instruction, learning materials, initial evaluation

The first contribution of the paper is the theory of CS1 instruction which structures and sequences programming skill. While prior work has identified that there are distinct skills which novices learn in sequence, theories of CS1 instruction have yet to consider this skill progression such that theories can translate to concrete learning materials. So, we proposed a theory which identifies distinct introductory programming skills that novices can learn sequentially. Skills are taught explicitly and incrementally so as not to overwhelm learners. The theory proposes that instruction first develop skills related to the semantics of a language by teaching tracing (S1: reading semantics) and

then writing correct syntax (S2: writing semantics). After explicit instruction on these skills, instruction should then use templates to teach comprehending common patterns in code (S3: reading templates) and then writing programs that implement these templates (S4: writing templates). We believe that the most salient part of this theory is not necessarily the four specific skills we defined, but rather the emphasis on being rigorous and explicit in how skills are defined and ordered in instruction; doing so will make a theory of CS1 instruction more useful for the design of learning materials.

The second contribution of this paper is the learning materials that we developed from the proposed theory. To provide a case study of translating this theory into concrete learning materials, we created an introductory Python curriculum which sequentially taught each of the four skills for introductory CS1 concepts. The learning materials focused instruction, practice, and feedback on one skill at a time. We developed new genres of practice items for writing semantics (translation of unambiguous description to code) and reading templates (determining if code implements a template correctly). We used these concrete learning materials as a case study to evaluate the efficacy of our theory.

The third and final contribution of the paper is the initial evaluation of the theory, where we sought to understand the impact of explicit instruction and practice of introductory programming skills on novices. We developed hypotheses from our theory that predicted that learners who received explicit practice on each of the four skills would be able to complete more practice exercises, make fewer errors, have a greater depth of understanding, and be more engaged. To evaluate these hypotheses, we conducted an exploratory study with novice programmers where the experimental group learned from the material that reflected the theory and distinguished between skills and provided explicit practice for all four skills. In contrast, the control group learned from the material that did not differentiate between skills in instruction and provided practice for only writing-related skills. Even though we attempted to control for amount of practice, we found evidence to suggest that those who received practice on all skills completed more exercises during practice, made fewer errors in the post-test (especially for later skills relating to templates), produced responses that reflected greater depth of understanding, but did not indicate differences in engagement. This evaluation provides some initial evidence relating to the efficacy of our theory, but there are multiple interpretations to these results.

7.2. Interpretation of results: explicit practice may have helped, but confounds exist

One interpretation of the results is that they provide some initial evidence to suggest that explicit practice for each of the four programming skills can improve CS1 instruction. We found that the participants who received practice

on all four skills more consistently completed practice exercises, made fewer errors on the post-test (especially for the more advanced skills of reading and writing templates), and produced responses that reflected deeper understanding on the SOLO taxonomy. We can interpret the results to suggest that this evidence provides support for our theory and that providing explicit, sequential instruction for programming skills (teaching them "piece by piece") helps novices learn to program when compared to more writing-focused instruction that does not distinguish between skills. This small-scale evaluation provides some evidence to support the theory and the broader claim that CS1 instruction can benefit from more structure and consideration of skill progression.

Another interpretation is that differences in learning outcomes were because of the unbalanced use of metacognitive prompts in the learning materials. Because participants in different conditions had different frequencies of question types and different questions had different prompts, there was an imbalance in metacognitive prompting between conditions. We relied on self-report from metacognitive prompts to identify sources of error. Metacognition is challenging for learners (National Academies of Sciences, 2018), so the quality of these self-reports varied. Furthermore, prior work has found that prompting learners to think metacognitively can improve their ability to read and write code (Loksa & Ko, 2016). This is a potentially confounding factor when evaluating differences in learning gains. But, the control condition had more practice with writing templates, the question with the most prompting. Therefore, the confound of the amount of metacognitive practice was biased *against* the experimental condition and in favor of the control condition.

Another interpretation of the results is that the difference in learning outcomes was a result of individual differences in participants. The small sample size for this initial study requires us to consider the variation between individuals. For example, the only two participants who reported having a fixed mindset (Dweck, 2008) ended up in the control group by chance. Fixed mindset can negatively impact learning to program (Murphy & Thomas, 2008), so differences in mindset could have been biased *in favor of* the experimental condition. In contrast, prior knowledge was biased *against* the experimental condition, as we assigned the two participants with the most prior knowledge, the only two who were currently enrolled in a programming class, to the control condition.

Other potential confounding factors were approximately balanced between conditions, such as programming self-efficacy and fatigue. We measured programming self-efficacy before and after the study using a survey that had undergone validation by Ramalingam and Wiedenbeck (1998). We found that for both conditions, participants reported low initial programming self-efficacy (average of 1–1.8 on 7-point Likert scale) and reported a final programming self-efficacy that was greater (1.83–5.83). Another confound we checked for was fatigue. Prior to the study, participants in both conditions all reported getting ample rest (6–8 hrs), although participants in both conditions reported feeling distracted or sleepy to

some extent during the instruction and post-test. We found similar patterns of increase in programming self-efficacy and levels of fatigue in both condition, so we do not have evidence to suggest that these potential confounds are impactful to the differences in learning outcomes.

7.3. Limitations and future work: improving rigor and breadth of theory

In this section, we note limitations and frame them as future work so we can improve upon the theory, the learning materials, and evaluating the effectiveness of these learning materials.

Future work could expand upon this theory by understanding how well skills transfer. Investigating how well different sub-skills transfer between programming languages can help improve skill development but is out of the scope of the learning materials we created, which taught introductory Python. This transfer between programming languages and environments is the third and final link in the Chain of Cognitive Accomplishments proposed by Linn and Dalbey (1985). Reading semantics (S1) and writing semantics (S2) likely do not transfer well because program syntax varies, but reading and writing templates (S3, S4) may transfer better as templates are supposed to reflect common patterns that exist across programming languages. Structuring learning materials according to this theory could decrease the teaching required to learn a second programming language.

Another potential thread of future work includes considering how to incorporate additional skills with the theory. As stated previously, many important programming skills are out of the scope of this theory. These skills include testing, debugging, problem-solving, and solving problems that require inventing new or previously unlearned templates ("creating" templates). For example, problem-solving is not explicitly addressed or taught in this theory of instruction. To determine if learning outcomes were relational in our adaptation of the SOLO taxonomy, we considered whether novices integrated templates to their solution. We did not, however, consider how novices conceptualized the problem description and solved the problem. Connecting this theory to theories of programming problem-solving such as the one by Loksa et al. (2016) could be promising. Another example of a skill we did not incorporate is how novices could develop their own templates, something that is more likely to happen as they learn more concepts and attempt more complex tasks. Finally, we taught semantic concepts (e.g. variables, conditional statements) in isolation and used templates to integrate these skills; Huang (2018) suggested that integrating skills across concepts may require instruction on additional skills (such as tracking program state and managing cognitive load) which are out of the scope of their theory. We believe that expanding and iterating upon our theory can result in theories of instruction that make CS1 instruction more robust.

Future work can also improve the learning materials with psychometric analysis of practice and assessment items and identifying the amount of practice required for each skill. Psychometric evaluation of the new item genres we proposed, which assess different skills, could help us better understand how distinct knowledge of each skill may be and how reliable interpretations of exercise performance are. This evaluation would be especially insightful in reading templates (S3) and writing templates (S4) practice (as defined in Section 4) as they are more novel in design. We found that for reading templates practice which asked learners to identify the mistake in the implementation of a template, the wording of the multiple choice options confused some participants in the experimental group. Those participants felt they correctly identified the location of the error in the code, but felt multiple answer responses could explain the error. Figure 12 may reflect this confusion as only 1 of the 5 experimental group participants reported getting the reading templates question correct even though 3–5 participants attempted each question. For writing templates practice, participants in both conditions often did not annotate the code with comments after writing the code. Confusion with the wording of reading templates questions and non-response relating to comments in writing practice makes the scores harder to interpret, so improving the design of exercises with psychometric evaluation is promising future work.

Another way to improve practice is to understand how much practice is required for each skill. Our theory suggests that some skills are precursory skills, so perhaps these more precursory skills (e.g. tracing) require less practice than more complex later skills (e.g. writing templates). We did not vary the amount of practice participants got (except in an attempt to counterbalance the amount of practice between conditions), so another open question is identifying how much practice participants require before understanding the skill well enough to progress.

Further evaluation of this theory could investigate how learners with different motivations engage with instruction derived from our theory. Learners all came from the same institution, self-selected to participate in this study, and received no compensation, so the study likely did not include learners who had more extrinsic motivations. Furthermore, the survey taken prior to participating in the study asked about mindset, programming self-efficacy, prior programming experience, attitudes towards programming, level of fatigue, ethnic identity, and parents' education, so stereotype threat could have affected their learning process and post-test performance (Schmader & Johns, 2003; Schmader, Johns, & Forbes, 2008). We found it necessary to collect some of this information prior to the study (e.g. mindset, self-efficacy, attitudes, level of fatigue), but we could have collected other information after the study. By designing additional learning materials based on this theory and conducting evaluations with participants in different contexts and with varied motivations, we can better understand how diverse novices conceptualize instruction based on this theory and how effectively they can learn from this instruction.

7.4. Implications and conclusion: specific theories can structure instruction for learners

Implications of this work extend to both research and practice and the interactions between them.

As a larger and more diverse body of learners begin to study CS1, there is a need to make instruction that is more effective, and, as a result, a greater need to connect research and practice on how people learn to program. Computing education research communities continue to grow around the world as do the communities of computing instructors. Yet, much of the current CS1 instruction still starts with "hello world" or an equivalent task, which has learners first and foremost writing code they do not understand to receive an output from a computer that is also ambiguous. CS1 instruction often does not structure knowledge relating to different programming skills, even though research suggests there are many skills that may develop sequentially. The gap between research and practice makes research findings difficult to translate to pedagogy and instruction.

Specific theories of instruction such as the one proposed in this paper can help inform and substantiate instructional design. Prior theories of instruction are not specific to CS1 or are too ambiguous to be easily translated into learning materials. To address this gap, we proposed a theory which sequences and structures knowledge relating to introductory programming skills in such a way that the theory can directly inform instructional design. To show this, we can develop an introductory Python lesson based on our theory and found some initial evidence to validate our theory. As detailed in the previous section, more work needs to be done to develop and evaluate this theory, but the promise is that instruction can be more approachable to future learners.

By developing and evaluating theories which structure skill progression and can easily translate to concrete learning materials, we can teach skills incrementally and avoid overwhelming learners.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

Notes on contributors

Benjamin Xie is a Ph.D. student at the University of Washington Information School, advised by Dr. Amy J. Ko in the Code & Cognition Lab. His research is in designing and developing interactions that have learners and intelligent agents collaborate to make learning computing more inclusive, work that spans human-computer interaction, artificial intelligence, and computing education. His vision is to computationally model how people learn programming to develop personalized online learning experiences where the learner is in control. He is a National Science Foundation (NSF) Graduate Research Fellow and was previously an MIT EECS-Google Research and Innovation Scholar. He received his Master's and Bachelor's degrees in computer science from MIT.

Dastyni Loksa is a Ph.D student at the University of Washington, advised by Prof. Andrew J. Ko in the Code & Cognition Lab. His research interests center on the mental processes of problem solving and design, specifically the metacognitive and self-regulation skills necessary for successful computer programming. He seeks to develop methods of learning and teaching cognitive skills for programming so that we can support learners from any background and cognitive style. He received his Bachelor's degree in informatics at University of California, Irvine and his Master's degree in information science from University of Washington.

Greg L. Nelson is a Ph.D student at the University of Washington, advised by Prof. Andrew J. Ko in the Code & Cognition Lab. His research interests center on rigorous theories of computer programming knowledge and using them to create better learning environments, but also include scientific process, statistical methods, HCI, and augmented reality. He seeks to foster a world where anyone can learn programming and sees programming as a medium that promises to revolutionize society, similar to widespread natural language literacy and the printing press. He has received a National Science Foundation (NSF) Graduate Research Fellowship, and received his BS in Computer Science and Physics from Georgetown; he hopes you judge him and others using the merits and an understanding of their work and, where he was taught to be a critical and reflective scientist and take awards and titles.

Matthew J. Davidson is a Ph.D. student in Measurement & Statistics at University of Washington, College of Education. His research centers on assessment of writing, especially of English language learners. He is particularly interested in investigating methods for analyzing data captured during the process of writing. He hopes to develop ways to use that data both as a tool for formative assessment and to investigate the validity of student response processes on writing tests. Ultimately, he is committed to making assessment data support student learning. He received his bachelor's degree in Philosophy, History, and English from the University of Texas, and his Master's of Education in Learning Sciences from the University of Washington.

Dongsheng Dong is a Ph.D. student in Measurement & Statistics at University of Washington, College of Education. Her research interests center on the development of K-12 STEM assessments, item development, and game-based learning. She is especially

interested in developing and optimizing test items and test accommodations for K-12 STEM assessments which could better reflect students' real potentials and assure the validity of assessments. She is also enthusiastic about using different methodologies to explore and describe students' thinking and behavior patterns through large-scale assessments. She received her Master's degree in TESOL from University of Pennsylvania, Graduate School of Education.

Harrison Kwik is a recent graduate of the University of Washington Computer Science department, but still continues to collaborate with members of the Code & Cognition Lab. During his undergraduate career, he assisted with various projects within the lab and also independently conducted research on transfer students within computer science. He is interested in continuing to pursue research and plans on applying to Ph.D. programs in the next coming months.

Alex Hui Tan is a recent graduate of the University of Washington Information School, and a current software developer at Hazel Analytics, Inc.. As an undergraduate, Alex taught Scratch, HTML and CSS to K-8 students through a partnership between Computing Kids and Seattle Public Schools. In pursuit of his interest in computing education, Alex assisted in the Code & Cognition lab, helping prototype tools for programming practice.

Leanne Hwa is a senior at the University of Washington Information School, supporting the Code & Cognition Lab on programming tutors while independently investigating the role of informal computing mentors amongst south Seattle teens. She is passionate about mentorship and representation in STEM and has served various leadership roles within UW Women in Informatics and also as a teaching assistant for the introductory Informatics course.

Min Li, an associate professor in Measurement & Statistics at College of Education, University of Washington, is an assessment expert with a deep interest in understanding how student learning can be accurately and adequately assessed both in large-scale testing and classroom settings. Her work reflects a combination of cognitive science and psychometric approaches in various projects on STEM assessments, including examining cognitive demands of science items, measurement issues in constructing instructionally sensitive test items, effects of context characteristics on item parameters, issues of testing linguistic minority students in mathematics and science, analyzing teachers' classroom assessment practices, development of instruments to evaluate teachers' assessment practices, and use of science notebooks as assessment tools. She received her bachelor's degree in psychology from Beijing Normal University and Ph.D. in education from Stanford University.

Amy J. Ko is an Associate Professor at the University of Washington Information School and an Adjunct Associate Professor in Computer Science & Engineering. She directs the Code & Cognition Lab, where she invents and evaluates interactions with people and code, spanning the areas of human-computer interaction, computing education, and software engineering. She is the author of 100 peer-reviewed publications, 9 receiving best paper awards and 3 receiving most influential paper awards. In '13, she co-founded AnswerDash, a SaaS company offering instant answers on websites using a selection-based search technology invented in her lab. In '10, she was awarded an NSF CAREER award for research on evidence-based bug triage. She received her Ph.D. at the Human-Computer Interaction Institute at Carnegie Mellon in 2008. She received degrees in Computer Science and Psychology with Honors from Oregon State University in 2002.

References

Allen, M. J., & Yen, W. M. (2001). *Introduction to measurement theory*. Long Grove, IL: Waveland Press.

Anderson, J. R., Boyle, C. F., Farrell, R., & Reiser, B. J. (1984). *Cognitive principles in the design of computer tutors* (No. TR-84-1-ONR). Retrieved from http://www.dtic.mil/docs/citations/ADA144825

Archer, A. L., & Hughes, C. A. (2010). *Explicit instruction: Effective and efficient teaching*. New York, NY: Guilford Press.

Biggs, J. B., & Collis, K. F. (2014). *Evaluating the quality of learning: The SOLO taxonomy (structure of the observed learning outcome)*. Cambridge, MA: Academic Press.

Bruner, J. S. (1966). *Toward a theory of instruction*. Cambridge, MA: Harvard University Press.

Buck, D., & Stucki, D. J. (2000). Design early considered harmful: Graduated exposure to complexity and structure based on levels of cognitive development. *SIGCSE Bulletin*, *32*(1), 75–79.

Buck, D., & Stucki, D. J. (2001). JKarelRobot : A case study in supporting levels of cognitive development in the computer science curriculum mathematical sciences department. *ACM SIGCSE* Bulletin, 33(1), 16–20.

Caspersen, M. E., & Bennedsen, J. (2007). Instructional Design of a Programming Course: A Learning Theoretic Approach. In *Proceedings of the Third International Workshop on Computing Education Research* (pp. 111–122). New York, NY: ACM.

Castro, F. E. V., & Fisler, K. (2017). *Designing a multi-faceted SOLO taxonomy to track program design skills through an entire course* (pp. 10–19). ACM Press. Retrieved from http://dl.acm.org/citation.cfm?doid=3141880.3141891

Clancy, M., & Linn, M. C. (1992). *Designing pascal solutions: A case study approach*. Rockville, MD: Computer Science Press.

Clancy, M. J., & Linn, M. C. (1999). Patterns and pedagogy. *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education* (pp. 37–42). New York, NY: ACM.

Clear, T., Whalley, J., Robbins, P., Philpott, A., Eckerdal, A., & Laakso, M. (2011, June). Report on the final BRACElet workshop: Auckland University of Technology, September 2010. *Journal of Applied Computing and Information Technology*, *15*. http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A431090&dswid=2905

Collins, A., Brown, J. S., & Newman, S. E. (1988). Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics. *Thinking: the Journal of Philosophy for Children*, *8*(1), 2–10.

Cooper, S., Wang, K., Israni, M., & Sorby, S. (2015). Spatial skills training in introductory computing. *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (pp. 13–20). New York, NY: ACM.

Corney, M., Lister, R., & Teague, D. (2011). Early relational reasoning and the novice programmer: Swapping as the hello world of relational reasoning. In *Proceedings of the Thirteenth Australasian Computing Education Conference Volume 114* (pp. 95–104). Australian Computer Society, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=2459936.2459948

Cunningham, K., Blanchard, S., Ericson, B., & Guzdial, M. (2017). Using tracing and sketching to solve programming problems: Replicating and extending an analysis of what students draw. *Proceedings of the 2017 ACM Conference on International Computing Education Research* (pp. 164–172). New York, NY: ACM.

Davies, S. P. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies*, *39*(2), 237–267.

de Raadt, M. (2008). *Teaching programming strategies explicitly to novice programmers* (Unpublished doctoral dissertation). University of Southern Queensland.

Doyle, W. (1983, June). Academic work. *Review of Educational Research*, *53*(2), 159–199.

Dweck, C. S. (2008). *Mindset: The new psychology of success*. New York, NY: Ballantine Books.

Ekstrom, R. B., Dermen, D., & Harman, H. H. (1976). *Manual for kit of factor-referenced cognitive tests* (Vol. 102). New Jersey, NJ: Educational Testing Service Princeton.

Fisler, K., & Castro, F. E. V. (2017). Sometimes, rainfall accumulates: Talk-alouds with novice functional programmers. In *Proceedings of the 2017 acm conference on international computing education research* (pp. 12–20). ACM. doi:10.1145/3105726.3106183

Fuller, U., Riedesel, C., Thompson, E., Johnson, C. G., Ahoniemi, T., Cukierman, D., … Thompson, D. M. (2007). Developing a computer sciencespecific learning taxonomy. *ACM SIGCSE Bulletin*, *39*(4), 152. Retrieved from http://portal.acm.org/citation.cfm?doid=1345375.1345438

Gluga, R., Kay, J., Lister, R., Simon, & Kleitman, S. (2013). Mastering cognitive development theory in computer science education. *Computer Science Education*, *23*(1), 24–57.

Gluga, R., Kay, J., Lister, R., & Teague, D. (2012). On the reliability of classifying programming tasks using a neo-piagetian theory of cognitive development. In *Proceedings of the ninth annual international conference on International computing education research ICER '12* (pp. 31). Retrieved from http://dl.acm.org/citation.cfm?doid=2361276.2361284

Hertz, M., & Jump, M. (2013). Trace-based teaching in early programming courses. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education* (pp. 561–566). Retrieved from http://dl.acm.org/citation.cfm?doid=2445196.2445364

Huang, Y. (2018). *Learner modeling for integration skills in programming* (Unpublished doctoral dissertation). University of Pittsburgh.

Kolodner, J. L., & Guzdial, M. (2000). Theory and practice of case-based learning aids. In Jonassen, D.H. & Land, S.M. (Eds.), *Theoretical Foundations of Learning Environments* (pp. 215–242). New York, NY: Routledge.

Kreitzberg, C. B., & Swanson, L. (1974). A cognitive model for structuring an introductory programming curriculum. In *Proceedings of the May 6–10, 1974, National Computer Conference and Exposition* (pp. 307–311). New York, NY: ACM.

Kurtz, K. J., Miao, C.-H., & Gentner, D. (2001, October). Learning by analogical bootstrapping. *Journal of the Learning Sciences*, *10*(4), 417446.

Linn, M., & Dalbey, J. (1985, September). Cognitive consequences of programming instruction: Instruction, access, and ability. *Educational Psychologist EDUC PSYCHOL*, *20*, 191206.

Linn, M. C., & Clancy, M. J. (1992). The case for case studies of programming problems. *Communications of the ACM*, *35*(3), 121–132. Retrieved from http://portal.acm.org/citation.cfm?doid=131295.131301

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M. (2004). A multi-national study of reading and tracing skills in novice programmers. In *ACM SIGCSE Bulletin* (Vol. 36, No. 4, pp. 119-150). New York, NY: ACM.

Lister, R., Clear, T., Bouvier, D. J., Carter, P., Eckerdal, A., Jacková, J., Lopez, M., McCartney, R., Robbins, P., Seppälä, O. and Thompson, E. (2010). Naturally occurring data as research instrument: Analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin*, *41*(4), 156–173.

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the solo taxonomy. In *Proceedings of the 11th annual sigcse conference on innovation and technology in computer science education* (pp. 118–122). ACM. doi:10.1145/1140124.1140157

Liu, Y., & Fu, X. (2007). How does distraction task influence the interaction of working memory and long-term memory? In D. Harris (Ed.), *Engineering psychology and cognitive ergonomics* (Vol. 4562, pp. 366–374). Berlin, Heidelberg: Springer Berlin Heidelberg.

Loksa, D., & Ko, A. J. (2016). The role of self-regulation in programming problem solving process and success. *Proceedings of the 2016 ACM Conference on International Computing Education Research* (pp. 83–91). New York, NY: ACM.

Loksa, D., Ko, A. J., Jernigan, W., Oleson, A., Mendez, C. J., & Burnett, M. M. (2016). *Programming, problem solving, and self-awareness: Effects of explicit guidance* (pp. 14491461). ACM Press.

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. Proceedings of the Fourth International Workshop on Computing Education Research (pp. 101–112). New York, NY: ACM.

McCartney, R., Moström, J. E., Sanders, K., & Seppälä, O. (2004). Questions, annotations, and institutions: Observations from a study of novice programmers. *Proceedings of the Fourth Finnish/Baltic Sea Conference on Computer Science Education* (pp. 11-19). Helsinki, Finland: Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory of Information Processing Science.

Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C. S., & Thomas, L. (2006). *A cognitive approach to identifying measurable milestones for programming skill acquisition* (*Working group reports on ITiCSE on Innovation and technology in computer science education ITiCSE-WGR '06* (December 2006), 182). Retrieved from http://portal.acm.org/citation.cfm?doid=1189215.1189185

Morra, S., Gobbo, C., Marini, Z., Sheese, R., Gobbo, C., Marini, Z., & Sheese, R. (2012). *Cognitive development : Neo-piagetian perspectives*. New York: Psychology Press.

Muller, O., Haberman, B., & Ginat, D. (2007). Pattern-oriented instruction and its influence on problem decomposition and solution construction. ACM SIGCSE Bulletin (Vol. 39, No. 3, pp. 151-155). New York, NY: ACM.

Murphy, L., Fitzgerald, S., Lister, R., & McCauley, R. (2012). Ability to explain in plain english linked to proficiency in computer-based programming. In *Proceedings of the ninth annual international conference on international computing education research* (pp. 111–118). ACM. doi:10.1145/2361276.2361299

Murphy, L., & Thomas, L. (2008). Dangers of a fixed mindset: Implications of self-theories research for computer science education. *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education* (pp. 271–275). New York, NY: ACM.

National Academies of Sciences. (2018). *how people learn II: Learners, contexts, and cultures*. Washington, DC: National Academies Press.

Nelson, G. L., Xie, B., & Ko, A. J. (2017). Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in Cs1. *Proceedings of the 2017 ACM Conference on International Computing Education Research* (pp. 2–11). New York, NY: ACM.

Philpott, A., Robbins, P., & Whalley, J. L. (2007). Assessing the Steps on the Road to Relational Thinking. *Proceedings of the 20th Annual Conference of the National Advisory Committee on Computing Qualifications* (pp. 286). Nelson, New Zealand: NACCQ.

Proulx, V. K. (2000). *Programming patterns and design patterns in the introductory computer science course* SIGCSE, 5.

Ramalingam, V., & Wiedenbeck, S. (1998, December). Development and validation of scores on a computer programming self-efficacy scale and group analyses of novice programmer selfefficacy. *Journal of Educational Computing Research*, *19*(4), 367–381.

Ranum, D., Miller, B., Zelle, J., & Guzdial, M. (2006). Successful approaches to teaching introductory computer science courses with python. In *ACM SIGCSE Bulletin* (Vol. 38, No. 1, pp. 396–397). New York, NY: ACM.

Ranum, D. L., & Miller, B. N. (2013). *Python programming in context* (2nd ed.). Burlington, MA: Jones & Bartlett Learning.

Rist, R. S. (1989, July). Schema creation in programming. *Cognitive Science*, *13*(3), 389414.

Robins, A., Rountree, J., & Rountree, N. (2003, June). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137–172.

Sanders, K., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Thomas, L., & Zander, C. (2012). Threshold concepts and threshold skills in computing. In *Proceedings of the ninth annual international conference on international computing education research* (pp. 23–30). ACM. doi:10.1145/2361276.2361283

Schank, P. K., Linn, M. C., & Clancy, M. J. (1993). Supporting Pascal programming with an on-line template library and case studies. *International Journal of Man-Machine Studies*, *38*(6), 1031–1048.

Schmader, T., & Johns, M. (2003). Converging evidence that stereotype threat reduces working memory capacity. *Journal of Personality and Social Psychology*, *85*(3), 440–452.

Schmader, T., Johns, M., & Forbes, C. (2008, April). An integrated process model of stereotype threat effects on performance. *Psychological Review*, *115*(2), 336–356.

Shaw, Z. A. (2017). *Learn python 3 the hard way: A very simple introduction to the terrifyingly beautiful world of computers and code*. Boston, MA: Addison-Wesley Professional.

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E., & Whalley, J. L. (2008). Going solo to assess novice programmers. In *Proceedings of the 13th annual conference on innovation and technology in computer science education* (pp. 209–213). ACM. doi:10.1145/1384271.1384328

Shneiderman, B. (1977, January). Teaching programming: A spiral approach to syntax and semantics. *Computers & Education*, *1*(4), 193–197.

Soloway, E. (1986, September). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, *29*(9), 850–858.

Soloway, E., & Ehrlich, K. (1984, September). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, *SE-10*(5), 595–609.

Stadler, M. A. (1995, May). Role of attention in implicit learning. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *21*(3), 674–685.

Szabo, C., Falkner, K., & Falkner, N. (2014). Experiences in course design using neo-piagetian theory. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research* (pp. 81–90). Retrieved from http://dl.acm.org/citation.cfm?id=2674691

Tew, A. E., & Guzdial, M. (2010). Developing a validated assessment of fundamental cs1 concepts. In *Proceedings of the 41st acm technical symposium on computer science education* (pp. 97–101). ACM. doi:10.1145/1734263.1734297

Thompson, E., Luxton-Reilly, A., Whalley, J. L., Hu, M., & Robbins, P. (2008). Bloom's taxonomy for CS assessment. In *Tenth australasian computing education conference ace*, Vol. 78.

Thota, N., & Whitfield, R. (2010). Holistic approach to learning and teaching introductory object-oriented programming. *Computer Science Education*, *20*(2), 103–127.

Vandenberg, S. G., & Kuse, A. R. (1978, December). Mental rotations, a group test of threedimensional spatial visualization. *Perceptual and Motor Skills*, *47*(2), 599–604.

Venables, A., Tan, G., & Lister, R. (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the fifth international workshop on computing education research workshop* (pp. 117–128). ACM. doi:10.1145/1584322.1584336

Whalley, J.L., & Kasto, N. (2013). Revisiting models of human conceptualisation in the context of a programming examination. In A. Carbone, & J. L. Whalley (Eds.), *Proceedings of the Fifteenth Australasian Computing Education Conference* (Vol. 136, pp.64–73). Adelaide, Australia: Australian Computer Society, Inc.

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A., & Prasad, C. (2006). An Australasian study of reading and comprehension skills in novice

programmers, using the bloom and SOLO taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education* (pp. 10). Australian Computer Society, Inc.

Winslow, L. E. (1996, September). Programming pedagogy-a psychological overview. *SIGCSE Bullettin*, *28*(3), 17–22.

Xie, B., Nelson, G. L., & Ko, A. J. (2018). An explicit strategy to scaffold novice program tracing. In *2018 acm sigcse technical symposium on computer science education*. ACM. doi:10.1145/3159450.3159527

Zimmerman, B., & Schunk, D. H. (2011). *Handbook of self-regulation of learning and performance*. New York, NY: Taylor & Francis.