27 Student Knowledge and Misconceptions

Colleen M. Lewis, Michael J. Clancy, and Jan Vahrenhold

1 Introduction

Students' knowledge is at the center of computing education (CEd). Research relating to this knowledge has frequently focused on the difficulties experienced by students when learning, which are often referred to as "misconceptions."

Awareness of the misconceptions commonly encountered by students has several benefits for teachers. For example, as teachers gain experience, they are believed to gain knowledge of typical student misconceptions (Sadler et al., 2013). This can partially explain improved learning outcomes for students with more experienced instructors (Ladd & Sorensen, 2017). Moreover, understanding these potential difficulties can help counteract our "expert blindspot" (Guzdial, 2015; Nathan & Petrosino, 2003) where, as relative experts in the domain, we have difficulty anticipating the difficulties that will be experienced by novices.

There is a considerable body of literature on misconceptions in computing. A book chapter (Clancy, 2004) enumerated patterns of misconceptions in computer science (CS). A current summary and review of misconceptions research within introductory CS was published in 2017 by Qian and Lehman. Section 2 reviews the literature in order to demonstrate that documentation of misconceptions in computing extends beyond introductory topics.

There has been, however, some controversy in this area. For example, research on misconceptions has been critiqued as being preoccupied with what students cannot do rather than what students can do or can learn to do (Smith et al., 1993). A research paradigm referred to as "expert-novice" research catalogs the ways in which novices demonstrate different knowledge and skills from experts (Bransford et al., 1999). For example, a popular finding from expert-novice research in physics education is that experts classify physics mechanics problems based upon the physics principle that could be used to solve them; perhaps unsurprisingly, novices do not do this (Chi et al., 1981). Instead, they attempt to classify problems using features of the problem, such as grouping all problems that involve an inclined plane. While the characterization of the skills of experts may be helpful in establishing learning goals, simply cataloging the fact that novices do not yet have this expertise can be done in ways that are not illuminating and are dismissive of students. Instead, students' prior knowledge can be seen as a resource for learning, which might enable more effective pedagogy.

We are sympathetic to the argument that misconceptions research may point out trivial differences between experts and novices and may distract from the goal of helping students learn. Despite that, our experience as educators speaks to the value of accumulating an understanding of where students have difficulty. Rather than pointing out deficiencies in students, we can see this research as helping educators recognize their likely weaknesses caused by the expert blindspot (Guzdial, 2015; Nathan & Petrosino, 2003) in not anticipating and understanding students' learning difficulties.

Beyond identifying *what* students have difficulty with, in this chapter, we hope to be able to provide perspectives from outside of science, technology, engineering, and mathematics (STEM) about *why* students experience particular types of difficulties. These perspectives on the difficulties of learning computing are essentially hypotheses based upon education research outside of CEd. In many cases, we also draw on CEd research (CEdR) to affirm these connections. Other chapters provide a more comprehensive review of the cognitive science literature (see Chapter 9). Here, we have selected literature that we believe provides a particular perspective that is helpful for considering students' knowledge and misconceptions. These perspectives are not exhaustive and are overlapping in many cases.

For researchers, we hope that this chapter charts potential directions for future research that connects CEdR with education research outside of computing. For educators, we hope that this chapter provides resources for understanding students' difficulties and that this may ultimately help them improve their teaching practice.

2 (Some) Misconceptions in CS

Qian and Lehmann (2017) provided an extensive literature review on programming-related misconceptions encompassing more than three decades of research. However, their scope did not extend beyond introductory programming classes. In this section, we highlight selected research results to illustrate the breadth of topics covered in computing misconceptions research. In our choice of papers discussed, we also aimed at presenting results that are representative of issues instructors may face in class. Given that some of the results presented are very recent at the time of this writing, not all observations have led to documented interventions yet; instead, they should be considered as the basis for additional research projects.

2.1 Theory of Computation: Reduction

Gal-Ezer and Trakhtenbrot (2016) report on a five-year exploratory study involving the written work of roughly 650 students from offerings of a theory of computation class. The focus of this study was on misconceptions related to reductions, an abstraction technique that transforms ("reduces") any input for some type of problem into an input for another, sometimes seemingly unrelated type of problem. The goal of such a reduction is usually to show that complexity bounds carry over from one problem to the other. The best-known use of reductions is to establish that any input to one problem that is known to be undecidable or NP-complete can be transformed in time polynomial in the input size to an input to another problem. In such a case, the transformation establishes the undecidability or, if the second problem is in NP as well, NP-completeness of the second problem.

Through their analysis, the authors identified misconceptions related to a "bigger is harder" mental model. Students were found to assume that any problem from a class included in a larger class of problems was reducible to any problem from that larger class. In particular, a corollary of this general misconception was that every problem A in the complexity class P should be polynomially reducible to every problem B in the complexity class NP (which contains P). Variations of this general misconception were that the mapping reduction from a decidable (or recursively enumerable) set A to a set B would be possible if and only if A was a subset of B ("B is bigger than A") or, since the class of enumerable problems contains the class of decidable problems, that any decidable problem C could be reduced to any enumerable problem D.

The authors conjecture that the effect of the "bigger is harder" mental model is strengthened by the typographically similar notations for inclusion and reduction. They suggest to actively address these issues by first presenting a "proof" based upon one of the misconceptions and then discussing its shortcomings using both intuitive and formal lines of reasoning followed by a correct proof or a proof that the sought reduction does not hold.

2.2 Algorithms and Data Structures: Heapsort

Danielsiek et al. (2012) investigated misconceptions related to algorithms and data structures by analyzing 400 written exams from offerings of a first-year CS course. Here, we focus on a heapsort test item (given a set of elements, sort it using the heapsort algorithm and show the steps), an item for which both the attempt rate and the average score were lowest across the exams for an introductory course taught by a third party. This was despite the fact that heapsort was discussed in class and was covered in programming assignments. Forty percent of the students did not attempt to solve the heapsort test item. In addition, the average score awarded for completed items was 43 percent.

Several written exams showed margin notes in which students had transformed the input given for the test item (an array of numbers) into a tree structure. From these notes, the authors could observe the following two points: (1) several students had transformed the input into a binary search tree and then (unsuccessfully) tried to run the heapsort algorithm on this structure; and (2) several students appeared to use a (correct) tree-based representation and then inferred the result for the array-based representation. Issues like the ones discussed in the work of Seppälä et al. (2006) (e.g., not executing recursive calls or including spurious swaps) could not be observed.

The authors established and tested hypotheses during weekly recital sessions and conducted think-aloud interviews to follow up on the above observations. The analysis suggested that students might have problems with associating the array-based implementation of a heap (usually discussed in the context of the heapsort algorithm) with the pointer-based tree model (usually discussed when introducing heaps). A closer look at the syllabus showed that the heapsort algorithm was taught in the first part of the course, which focused on algorithms. The heap data structure was taught in the second part of the course, which was immediately after binary search trees were discussed in class. Interestingly, not every student was comfortable with the tree-based heap representation. In a study with 155 students who were given both representations, 20 percent of the students could work with a tree-based representation but not with an array-based representation, whereas 14 percent could work with an array-based representation but not with a tree-based representation; 34 percent of the students could work well with either representation and the remaining 30 percent had problems with both representations.

In a follow-up study (Vahrenhold & Paul, 2014), the authors taught these topics in a different order: binary search trees were taught before heaps or heapsort. Afterwards the focus was on graph algorithms, which raised the need for an efficient implementation of the abstract data type PriorityQueue. For this, heaps were introduced and heapsort was presented as an example of a data structure-based sorting algorithm. On the exam, students were asked to show the intermediate steps on the array- and tree-based representations for the heap construction phase and two iterations of heapsort, working in parallel with both representations. The number of non-attempts dropped to 27 percent and the average score increased to 61 percent. While this change may have been due to other instructional effects as well (demographics were comparable), a follow-up study by Karpierz and Wolfman (2014) in which the topic order also avoided the problematic sequence was also unable to reproduce the misconceptions seen in Danielsiek et al. (2012).

2.3 Programming Languages: Scope, Mutation, and Aliasing

With a focus on upper-division programming language courses, Fisler et al. (2017) targeted misconceptions in the areas of scope, mutation, and aliasing in Java and Scheme; these languages have identical semantics with respect to the focus of the study, despite very different syntax and surface-level concepts. Working with two languages was one key feature of the research. Their intent was not only to determine whether students understood those topics, but also to see if advanced students transferred knowledge between languages.

The authors used three instruments. Two were multiple-choice: a pre-test at the start of the course and a post-test after scope, mutation, and parameter passing had been covered. Each contained 12 Scheme-based items and 6 Java-based items. The third instrument, which makes this study stand out from other work, consisted of several activities intended to enrich students' understanding of the topics; these activities included implementing a problematic language feature, writing a test suite, peer reviewing each other's test suite, and answering clicker questions during the lectures.

Preliminary results indicate that the students (n = 66) did poorly on the pre-test and much better on the post-test. Most of the improvement was in Scheme rather than Java, and improvements were not uniform across topics. Though the results were not rich enough to determine the effects of the instructional activities on learning, this work, and in particular the instruments used, is likely to provide a solid basis for future research.

2.4 Operating Systems: Indirections

Webb and Taylor (2014) report on their first steps toward developing a *concept inventory* to test operating systems concepts. A concept inventory is an instrument that belongs to the class of so-called formative assessments of instruction (Adams & Wieman, 2011). It consists of multiple-choice questions in which the incorrect answers ("distractors") are designed to indicate a particular form of students' misunderstanding. For more detail on the design process of such questions, see Adams and Wieman (2011) and Almstrum et al. (2006).

The paper by Webb and Taylor explores the following three operating systems concepts: indirection, input/output, and synchronization. Here, we focus on their discussion of indirection in a file system block map. During class, the authors asked a peer instruction question focused on the space requirements given either direct or indirect pointers. Forty-four percent of the students' initial, individual answers were correct. However, after students discussed the question in their groups, only 39 percent of the students' answers were correct. This drop, which is in contrast with general results regarding the effects of peer group discussions on performance, suggests that students have an incorrect but intuitive idea that is competing with the correct reasoning. The authors argue that indirection is difficult for students because "systems often use indirection as an optimization as opposed to it being necessary for correctness. … We believe that the challenge of indirection may stem from the fact that a solution using only direct pointers seems to be attractive to students because it would be relatively straightforward" (Webb & Taylor, 2014, p. 106).

They also examined two indirection-related questions on the final exam. One of the questions presented a scenario in which students were asked to determine the numerical properties of a hypothetical page table and then explain (as free-response text) whether or not they would recommend using multiple levels of page tables (indirection) in this scenario. The second question presented a similar hypothetical scenario of a file system block map and eventually asked students to justify the complexity of using indirect pointers. Unsurprisingly, students performed better on the second question, where they were implicitly told that the complexity was useful, as opposed to the former, in which students had to make that decision themselves. This further supports their hypothesis that students' difficulty comes from students' preference for the more straightforward implementation and that "students are more than capable of memorizing the details" (Webb & Taylor, 2014, p. 103).

To summarize, indirection is a difficult concept, and this difficulty may come from a lack of motivation for indirection rather than just a lack of understanding of the details. The authors suggest that students who are learning these concepts may benefit from instructors initially introducing an abstract form of indirection, perhaps by relating it to C memory pointers, which students are likely to find familiar. Discussing indirection abstractly by mapping it to a more comfortable topic may enable students to construct a mental model of indirection prior to adding the confounding details of memory management or file systems.

3 Perspectives from Outside CEd and How They Relate to CEd

There is a long history of education literature outside of CEdR, but often few connections are made across disciplines (Confrey, 1990). In the following sections, we seek to highlight particular strands of education research outside of CEdR and how they might apply within computing. When feasible, we summarize CEdR that we believe exemplifies this connection. While this is not an exhaustive list of potential connections outside of CEdR, we seek to encourage the development of additional connections and expect that some of the connections we describe could fuel promising future work in CEdR.

3.1 The Role of Intuition in Learning

Conceptual change research outside of CEd often focuses on the role of students' intuitions, particularly the ways in which these intuitions lead to misconceptions (see diSessa, 2014a, for a review). Similarly, CEd has focused on how students' interactions with programming languages are shaped by their experience conversing with a human (Pea, 1986). In both cases, understanding the ways in which students' intuitions lead them astray is seen as essential to effective teaching.

3.1.1 Evidence from Outside of Computing

Conceptual change involves understanding learning and has been particularly focused on exploring students' understanding of school-taught scientific content. Generally, conceptual change research rejects the idea that humans begin as a blank slate, and researchers wrestle with the complexity of learning as it is shaped by instruction and students' existing knowledge (Sawyer, 2014). Of particular focus is students' intuitions about particular scientific phenomena. For example, in their study of conceptual change in childhood, Vosniadou and Brewer (1992) found that in learning about the shape of the earth, some students came to believe that the earth was the shape of a pancake, both round and flat. Their study revealed that changing from an initial model of the earth based upon everyday experiences, such as walking on a seemingly flat surface, to the scientifically accepted model of a spherical earth "is slow and gradual" (Vosniadou & Brewer, 1992, p. 582).

As we discussed in the introduction, misconceptions research has been critiqued for focusing only on how prior knowledge fuels intuitions that inhibit learning processes (Smith, diSessa, & Roschelle, 1993). While anticipating students' misconceptions can be pedagogically valuable, an exclusive focus on the negative role played by prior knowledge misses an opportunity to identify the productive role that prior knowledge can play. This is an important critique to keep in mind when producing and reading research about students' understanding or lack thereof.

3.1.2 Connections to CS: Observations and Hypotheses

Previous research in CEd has focused on the role of prior knowledge in shaping students' intuitions. For example, CEdR has identified ways in which students' prior knowledge from math and English interfere with their understanding of particular programming constructs (Clancy, 2004). A common example of a misconception in computing that is tied to intuition is the intuition that the computer "knows" more than it does (Pea, 1986). This was originally referred to as a "superbug": "The superbug may be described as an idea that there is a *hidden mind* somewhere in the programming language that has intelligent, interpretive powers" (Pea, 1986, pp. 32–33, emphasis in original). Understanding students' intuitions is practically important for educators who could benefit from anticipating the difficulties students may experience. Pea's early claim is still true: "mapping conventions for natural language instructions onto programming results in error-ridden performances" (Pea, 1986, p. 33).

However, given that computer systems are designed by humans, students' intuitions can be seen as wrong only insofar as the designers of the system did not select the particular behavior that a student expects. For example, students sometimes incorrectly expect that a while loop will exit once the condition stops being met (Bonar & Soloway, 1989; Spohrer & Soloway, 1986). This is incorrect because the condition is only checked before beginning the loop body. However, it would be feasible for a loop to be designed such that the condition is checked after every expression in the loop body, which then would align with students' intuitions. That is, the relevance of students' intuitions is dependent on the programming language.

Understanding students' intuitions is incredibly important, but caution is needed because these intuitions may be incorrect for one programming language and correct for another. Because humans have designed these systems, there may be fewer truly counterintuitive things for students to learn in CS. Because of this, students' incorrect intuitions in computing may respond to instruction differently than intuitions about phenomena not within the complete control of humans.

3.2 Chunking

In CEd, we frequently want students to abstract from individual keywords or lines of code to understand the steps in an algorithm. This type of abstraction relates to the idea of chunking (Miller, 1956). The ability to chunk information is a well-established difference between experts and novices in a domain (Bransford et al., 1999). While Chapter 9 provides additional elaboration on chunking (Miller, 1956) and other information regarding working memory (Baddeley & Hitch, 1974), here we focus on the high-level overview so as to connect it to research in education and CEd.

3.2.1 Evidence from Outside of Computing

The idea of working memory (Baddeley & Hitch, 1974) gives rise to the common – and likely familiar – idea that people can typically remember seven things, or seven things plus or minus two. The idea of chunking is that, while working memory is finite (Baddeley & Hitch, 1974), the things that are stored in working memory can be "chunks" with a high density of information. This idea of chunking explains the otherwise perplexing result that people can remember approximately seven things, regardless of whether those things were numbers, letters, or monosyllabic words (Miller, 1956). As further elaboration, Miller (1956, p. 15) describes a study where researcher Sydney Smith was able to recall a sequence of 40 binary digits, far greater than the previously reported average of 9 (Pollack, 1953). Smith did this by converting sequences of binary digits into decimal digits, requiring extensive practice to automate this process. While this example related to binary digits might be the most relevant for CEd researchers, the canonical example of chunking comes from studies of chess players (Chase & Simon, 1973). Expert chess players were able to recall the locations of chess pieces with fewer errors than novice chess players. However, for chess board configurations that were not typical of chess game play, expert chess players were no better at recalling the locations of pieces. This is described as *domain-specific* chunking; the expert chess players had chunks to recall particular components of game board configurations. Contrary to conventional wisdom, the expert chess players did not have a greater ability to recall information, only a greater ability to recall particular domain-specific chunks (Bransford et al., 1999).

Chunking has been applied to understanding learning across domains (Baxter & Glaser, 1998; Gobet et al., 2001). For example, how humans process collections of letters into words can be explained as chunking (Gobet et al., 2001). Lane et al. (2000) also used chunking to explain how physics students construct diagrams of electric circuits. The idea of chunking appears to be broadly applicable to understanding learning and problem-solving (Gobet et al., 2001).

3.2.2 Connections to CS: Observations and Hypotheses

Ideas about chunking have also appeared within CEdR. Winslow summarizes that there is a "large number of studies concluding that novice programmers know the syntax and semantics of individual statements but they do not know how to combine these features into valid programs" (Winslow, 1996, p. 17). Robins et al. connect this idea to chunking and summarize Winslow as making the claim that novices "approach programming 'line by line' rather than using meaningful program 'chunks' or structures" (Robins et al., 2003, p. 140).

There are multiple examples from computing where educators attempt to make these chunks a learning goal in and of themselves. For example, Sajaniemi and Kuittinen (2005) argue for teaching students about the different roles that variables play in introductory programming tasks. They argue that just ten different roles account for 99 percent of all uses of variables in introductory programming instruction. Educators also use "Parsons' problems" (Denny et al., 2008; Parsons & Haden, 2006) that require students to order and indent complete lines of code to accomplish a particular goal. In providing these complete lines, they attempt to direct student attention to semantics rather than the details of the syntax. At a larger grain size, Gamma et al. (1995) published a set of 23 design patterns for software engineering, which has been used in helping students recognize and apply these larger patterns to solving problems.

A possible reason for teaching Big-O in introductory courses is that it forces the 10,000-foot view that is characteristic of chunking. Rather than seeing a for-loop as a sequence of steps, it can be seen as a more general pattern of repeating something a certain number of times. Similarly, a line of research has found correlations between students' ability to write code and to explain code "in plain English" (Murphy et al., 2012). This may be another example of domain-specific chunking that enables both improved coding performance and improved explanations of code.

3.3 Examining Cases

Introductory programming frequently begins with introducing basic operations such as conditionals (Rich et al., 2017). However, there are some long-documented weaknesses that humans demonstrate when reasoning about conditionals. This non-computing research appears to be consistent with CEdR of students' writing or evaluating Boolean expressions (Herman et al., 2012). From these findings we argue that providing a meaningful context for reasoning about conditionals promotes success. Additionally, people are prone to not enumerating all of the relevant cases, and they are much more successful at tasks when they are encouraged to enumerate and consider all of the relevant cases.

3.3.1 Evidence from Outside of Computing

Wason (1968) developed a test of human reasoning, referred to as the Wason selection task, which spurred an areas of research into people's reasoning about

Table 27.1 *Truth table for modus ponens, P → Q.*

Configuration	P (has an A on one side)	Q (has a 3 on one side)	P → Q
1	T	T	T
2	T	F	F
3	F	T	T
4	F	F	T

conditional logic. In an original version of the selection task, a person is told that each of four cards in front of them has a letter on one side and a number on the other side. Then the person is asked to determine which of the cards should be flipped to check if an additional rule is violated. For example, consider four cards showing, A, D, 3, and 7 and the rule: "If there is an A on one side of the card, then there is a 3 on the other side of the card." (Wason, 1968). We can think of this rule as using the general modus ponens form P → Q. That is, P (has an A on one side) implies Q (has a 3 on one side). Based upon this abstraction, the cards can be labeled as P (the A), ¬P (the D), Q (the 3), and ¬Q (the 7), as shown in the truth table in Table 27.1.

To check if the rule is violated by any of the cards, it is necessary to flip over the A and the 7. While this is likely not necessary for most readers, the following bullets explain the logic for each possibility (P, ¬P, Q, ¬Q).

- P (the A): Most people recognize that you need to flip over the A to ensure that there is a 3 on the other side. That is, it is necessary to distinguish between configurations 1 and 2 of the truth table in Table 27.1.
- ¬P (the D): It is not necessary to flip over the D because there are no rules related to letters other than A, and we know that the other side does not have an A, because we were guaranteed that every card has a letter on one side and a number on the other. That is, it is not necessary to distinguish between configurations 3 and 4 of the truth table in Table 27.1.
- Q (the 3): It is not necessary to flip over the 3 because regardless of the other side, it cannot violate the rule. That is, it is not necessary to distinguish between configurations 1 and 3 of the truth table in Table 27.1.
- ¬Q (the 7): Many people do not recognize that it is necessary to flip the 7 because having an A on the other side would violate the rule. That is, it is necessary to distinguish between configurations 2 and 4 of the truth table in Table 27.1.

Many variations of this task exist, and performance can vary greatly on the task when it is worded in different ways. The resulting body of research generally seeks to develop hypotheses about the cognitive mechanisms that explain this variation in task performance. This is an ongoing debate (von Sydow, 2006), but at this time we can abstract away three central points of relevance to computing teaching and learning.

First, this poor performance on the Wason selection task does not appear to be caused by a lack of understanding of the logical operator. In the original work (Wason, 1968), when people were asked to consider each card independently, they were generally able to do so. Additionally, O'Brien et al. (1998) showed that three- and four-year-olds were able to evaluate if a rule such as "If a boy is playing basketball, he is wearing red sneakers" was violated in pictures representing each of the configurations of the modus ponens truth table.

Second, changes in the context of the rule and instructions for the person can change average task performance significantly (Wason & Johnson-Laird, 1972). For example, people are more successful when asked which cards they should flip to determine if people are violating drinking age restrictions. Imagine cards that show a person's age on one side and whether they are drinking soda or alcohol on the other side. Consider the rule, "If someone is under 18, they cannot drink alcohol," and the following cards: P [age = 10], ¬P [age = 30], Q [drinking water], and ¬Q [drinking alcohol]. In this case, people tend to find it more intuitive to flip the cards with the person who is not old enough to drink alcohol and the card with the person who is drinking alcohol. Placing the problem in a more familiar context appears to improve task performance, as does, when appropriate, asking the person to assume the role of a checking authority (O'Brien et al., 2004, p. 101).

From the research described above, we should expect that students will make mistakes when reasoning about logical operators. As stated above, this is true even when they can demonstrate evidence of understanding the logical operator in other contexts.

3.3.2 Connections to Computing: Observations and Hypotheses

Consistent with the findings above, Herman et al. (2012) found that students often failed to enumerate or consider all of the cases when writing or evaluating Boolean expressions. However, when students were asked to enumerate all of the cases, they were able to correctly solve the problems. Herman et al. (2012) observed that there are many Boolean operators that are equivalent for a subset of the cases (see Table 27.2, gray cells). They found that students tended to substitute for the "easier" operator in the cases where the cases that they enumerated matched the "easier" operator. Table 27.2 shows some of the overlap between the operators AND, if-then, and if-and-only-if. It appears that when students work with less familiar Boolean operators, they fall back onto using familiar ones such as AND.

Herman et al. (2012) also found that students perform worst when they are answering the question in an unfamiliar context and best if it is a familiar context.

We hypothesize that this may help us understand why human-readable variable names are seen as important. Familiar context may make people more effective at reasoning, as was demonstrated in people's improved performance when enforcing drinking restrictions (O'Brien et al., 2004). Ultimately, human-readable variable names may help connect the code with the domain of the problem that they are trying to solve.

Table 27.2 *Overlap between logical operators AND, if-then, and if-and-only-if.*

A	B	AND	if-then	if-and-only-if
0	0	0	1	1
0	1	0	1	0
1	0	0	0	0
1	1	1	1	1

Additionally, we hypothesize that this may help us understand the importance of *test-driven development* (Beck, 2003). In test-driven development, programmers write test cases *before* writing the code. This allows them to focus on the behavior that is expected from the code and may support individuals in considering all of the relevant cases, which, as we see from the research above, frequently requires scaffolding.

3.4 Learning to Reason with Abstraction Symbols

Abstraction is frequently mentioned as a core skill developed when learning programming (Ginat & Blau, 2017). This is similar to the ways in which the learning of algebra is described (Sfard, 1995). Research about practices for helping students adapt to the abstraction involved in algebra may be helpful for identifying pedagogical strategies that could be applicable to CS. In particular, here we will focus on a sequence of instruction called concrete-to-representational-to-abstract, or CRA (Witzel et al., 2008), which we argue might be applicable to computing instruction.

3.4.1 Evidence from Outside of Computing

To introduce the basics of CRA we will use the example of a classroom where young students are learning addition. CRA begins by introducing a physical (i.e., *concrete*) object. For example, this could be physical blocks that could be counted to add them together. Once students are comfortable adding together sets of physical blocks, the class could advance to solving the same problems given only a picture (i.e., *representation*) of the blocks, but not the physical blocks. Once students are comfortable using only the pictures, the class could advance to solving the same problems using only numbers (i.e., *abstraction*). If a student has trouble adding together only numbers (i.e., working at the abstract level), they could be encouraged to draw pictures (i.e., returning to the representation level). If a student has trouble adding together numbers using a drawing, they could be encouraged to work with the physical blocks (i.e., returning to the concrete level).

As shown in this example, the concrete, representational, and abstract forms of the problem can be used to "promote overall conceptual understanding and procedural accuracy and fluency" (Witzel et al., 2008, p. 271). Witzel et al. (2008, pp. 271–272) argue that CRA instructional sequences provide the following benefits:

- Providing a connection between abstract content and students' understanding of the steps and definitions used in the concrete version;
- Making the content more memorable, which aids memory;
- Increasing engagement, which can increase students' enjoyment;
- Making the content relevant and personal;
- Enabling students to draw a picture of a concrete object rather than requiring them to manipulate that object directly.

While this particular sequence for teaching addition is likely pervasive across elementary school classrooms, the more general CRA sequence of instruction has been formalized and researched within special education. In their article, "A Meta-Analysis of Algebra Interventions for Learners with Disabilities and Struggling Learners," Hughes et al. (2014) found that the "concrete–representational–abstract sequence had the largest effect sizes" (Hughes et al., 2014, p. 36). Similar results exist within the broad mathematics education literature. Within mathematics education, the use of concrete objects is referred to as "manipulatives" because the student can interact directly with the concrete object. In a meta-analysis of 55 studies of mathematics instruction, there were small to moderate effect sizes "in favor of the use of manipulatives when compared with instruction that only used abstract math symbols" (Carbonneau et al., 2013, p. 380). Also from mathematics education, an earlier meta-analysis had found:

> The use of manipulative devices and drawings also emerged as an effective practice for teaching both computation and word problems. Manipulative devices with and without subsequent drawings were effective. Findings related to the use of drawings without manipulative devices, however, were mixed.
> (Miller et al., 1998, p. 21)

Despite the positive results referenced above, applying CRA and related pedagogies can be complex. While the meta-analyses show results that tend to be positive, studies comparing math instruction with and without manipulatives have shown null results, positive results, and negative results (Carbonneau et al., 2013, pp. 380–381). Witzel et al. (2008) provide practical advice for teachers who are designing CRA sequences within their math classrooms, which may be of practical value in CEd. They explain that "[w]ithout explicit awareness of how each stage connects with the next interconnected stage, the students may feel as though they are memorizing separate and arbitrary procedures to solve the same mathematical skill" (Witzel et al., 2008, p. 271).

3.4.2 Connections to CS: Observations and Hypotheses

Practices of demonstrating concrete instantiations of programming concepts may be much less common than concrete instantiations of mathematical concepts. Unlike the example of adding blocks, in CS, there are infrequently obvious real-world mappings for the abstractions. However, a few examples of common pedagogy in computing bear hallmarks of aligning with the CRA instructional sequence (Pollard & Duvall, 2006). For example, many instructors introduce a variable by showing it as a cup (i.e., the variable) that can hold a particular value or introduce an array of integers by showing a line of objects that each has a number written on it.

An experience from the first author further aligns with the findings described above. She had been struggling to teach students common summations such as $1 + 2 + 4 + \cdots + N / 2 + N$, and $1 + 2 + 3 + \cdots + N - 1 + N$. She had developed visual representations to try to help students build on their understanding of measurement and area, respectively. However, students were frequently unable to recalculate the sums. In the past year, she bought blocks like those that might be used in an elementary school classroom. These blocks could connect together, and she created a set of blocks for each of the terms in the sums. Then, under a document camera, she showed how they could be combined to add up to $2N - 1$ and $N(N + 1) / 2$, respectively. Only then did she show a representation of the same information in a slide. Through the remaining classes, the students appeared to be more capable of recreating the sums for the two sequences described above.

The use of concrete manipulatives as a bridge toward more abstract reasoning seems to be a promising direction for CEd. However, the mix of results as described above (Carbonneau et al., 2013) show some of the complexity of the pedagogical sequence. In particular, the focus on concrete manipulatives should not be mistaken for advocacy of learning styles (Willingham et al., 2015). Some might incorrectly argue that kinesthetic students need opportunities to physically move in order to be able to learn the content. Learning styles research that would make this incorrect claim has been debunked and rejected by the education community (Willingham et al., 2015). Instead, learning should be kinesthetic if the learning goal is kinesthetic. For example, you would teach a child to tie their shoelaces by having them practice the kinesthetic act of tying their shoelaces. If a concept can be practiced using multiple modalities, that additional exposure can be helpful, but the central principle is connecting the content to appropriate modalities and not matching modalities to students' learning styles (Willingham et al., 2015).

3.5 Learning Rule-Based Systems

Across domains, education often seeks to help people understand and apply rule systems. For example, when learning algorithms for arithmetic, students learn a set of rules and apply those rules to solve problems. There is a body of work looking at students "bugs" in learning and applying these algorithms (Young & O'Shea, 1981). We can see connections between students' learning of these algorithms and their understanding of the notional machine (see Chapter 1). Also, there are common goals across CEdR and mathematics education research of documenting and exploring mistakes or "bugs" in students' application of algorithms.

3.5.1 Evidence from Outside of Computing

There is a long history of research into the learning of arithmetic. By 1930, Buswell published a series of articles that reviewed 584 articles and books that “report investigations of the methods and results of teaching arithmetic” (p. 766). A common refrain across studies is that students’ understanding of algorithms for arithmetic should be rooted in their conceptual understanding of numbers and place value (Chan et al., 2014). Carpenter et al. (1988) argue that “there is general consensus about how children solve different problems” (p. 387). For example, they describe progressions of students’ strategies from counting to using a set of number facts to derive other answers (e.g., using 4 + 4 = 8 to solve 4 + 5 = 9).

Students’ learning of long division might best connect with students’ understanding of the notional machine. The algorithm for long division specifies a set of steps, but the actual execution of the algorithm depends upon the numbers in the problem. The algorithm for tracing code for a particular notional machine specifies the behavior of particular lines of code, but the tracing of the code depends upon the lines of code to be traced. These similarities may mean that teaching practices for long division could be helpful for teaching students to trace code. One seemingly common strategy for teaching long division is to use mnemonics to help students remember the steps (Rivera & Smith, 1988). The effectiveness of the use of mnemonics has been demonstrated outside of mathematics (Levin et al., 1992). However, clear recommendations do not appear to have developed from this body of work on teaching long division. Rivera and Smith, in describing a variety of pedagogical approaches for teaching long division, summarize that “research findings to support the efficacy of these strategies are limited” (Rivera & Smith, 1988, p. 77). Similarly, Chan et al. (2014) critique a common developmental model developed by Fuson and colleagues (see Fuson, 1998) for being “built upon informal observations of how children solved additional and subtraction problems” (p. 79) and that the “conceptual structures and developmental sequence remain untested empirically” (p. 79).

3.5.2 Connections to CS: Observations and Hypotheses

Understanding recursive code can be seen as an example of a rule-based system, not unlike arithmetic. In CEdR, students’ understanding of recursion is a popular area in which to identify “bugs” in students’ mental models. McCauley et al. (2015) surveyed over 35 research publications involving comprehension, evaluation, and construction of recursive programs across different programming languages and paradigms. The publications collectively used both quantitative and qualitative methods. Learner populations ranged from elementary school children through to university-level students. In one correct model, recursive procedures are seen to generate new instantiations of themselves, passing control and possibly data forward to successive instantiations and back from terminated ones. Kahney (1989) referred to this as a *copies* model.

An example of an incorrect model is a *looping* model (Kahney, 1989), which repeats the intended recursive code and stops the procedure when a base case is encountered, thus using it as a termination condition rather than a return condition. This may be partly due to a feature of the Logo language used in Kahney's work in which "STOP" means "return." However, the same difficulty was documented in college students more recently (Lewis, 2012).

Similarly, understanding object-oriented programming languages can be seen as an example of a rule-based system, and research has sought to document "bugs" in students' understanding and application of their knowledge. For example, Ragonis and Ben-Ari (2005) identified a total of 58 types of difficulties. They argued that these difficulties could be separated into the following four categories: "objects vs. class," "instantiation and constructors," "simple vs. composed classes," and "program flow." Roughly two-thirds of the difficulties were categorized as misconceptions, since they could be traced back to a specific misunderstanding (e.g., "an object cannot be the value of an attribute" or "attribute values are updated automatically according to a logical context").

In the light of the current interest in functional (see Section 4.5) and multiparadigmatic languages (e.g., Scala), we revisit early research on misconceptions related to functional programming and discuss the work of Davis et al. (1993). Consistent with our framing of the notional machine as based on a set of rules, the authors investigated *student-formed* rules in Common Lisp. They investigated the process of learning how Lisp evaluates expressions. The project extended over two semesters: the first to gather data and the second to test an intervention. In the first semester, a predicted list of incorrect rules was constructed, along with an assessment whose questions attempted to detect applications of the rules. Students' performance on the assessment was used to revise the list of incorrect rules. Each student's incorrect answers were then examined to determine what rule(s) could explain each result. In the second semester, several pre-lab activities were added to two of the lab assignments:

- Critique code and fix buggy rules;
- Invent function definitions and calls;
- Predict and explain the result or error message produced by given code.

The results revealed that the number of completely correct calls and the number of errors (incorrect rules used) on the assessment significantly improved in the intervention semester (Davis, 1995).

A more direct example of our framing of rules is a *rule-based* programming environment that has the following two components: a *knowledge* base and a set of *rules.* Each rule contains a *condition* that is checked against the knowledge base and one or more *actions*, which update the knowledge base. *Execution* of a program involves repeatedly identifying rules whose condition is true, then using the actions to update the knowledge base. Rule-based programming is popular in artificial intelligence applications, specifically *expert systems* that imitate the decision-making ability of a human expert. The Prolog language, created in 1972, is most commonly associated with the rule-based approach to programming. Prolog has been taught in introductory programming courses, mostly in Europe, since the early 1980s. It is perceived to be a difficult language to learn and use. As a result, a significant body of CEd work is based on Prolog. (See, for example, two special issues of *Instructional Science*: volume 16, issue 4–5, July 1990, pp. 247–416; and volume 20, issue 2–3, March 1991, pp. 81–266.)

3.6 Reading Literature

Programming is frequently compared to the learning of languages. For example, CEd researchers suggest that programming instruction should mimic reading instruction by having students read code as important preparation for learning to write code (Murphy et al., 2012). This and other connections to reading pedagogy seem to be a fruitful direction for consideration as there is a long-standing interest in understanding how humans learn to read. While there are many potential connections between the reading literature and computing, we focus on the pedagogical practice of rereading and close by highlighting a few additional potential connections to CEd.

3.6.1 Evidence from Outside of Computing

Reading education has identified that rereading is an effective practice for learners (Dowhower, 1994; Kuhn & Stahl, 2003; Samuels, 1979; Smagorinsky & Mayer, 2014). The pedagogical innovation is to have children reread the same text until they are able to read it relatively quickly with a low error rate. However, reading interventions have been difficult to study because of the ethical implications of having a control group (Kuhn & Stahl, 2003). For example, Stahl and Heubach (2005) report that "the results of the first year were so unexpectedly strong that we felt that denying treatment to a control set of classes was unethical" (p. 38). While repeated reading is praised as an effective practice (Smagorinsky & Mayer, 2014), Kuhn and Stahl (2003) warn that some of the positive results of the technique may be primarily attributed to additional time reading (p. 28).

3.6.2 Connections to CS: Observations and Hypotheses

The rereading literature described above presents the idea that repeating a particular task may be advantageous. Resources exist for students to practice specific skills in a programming language (e.g., https://codingbat.com, https://practiceit.cs.washington.edu, www.codestepbystep.com). These websites offer many isomorphic problems so that students can practice particular skills. However, based upon the rereading research, perhaps it is best for a student to solve a single problem multiple times before moving on. At a minimum, the reading literature suggests that achieving a level of fluency may require extended, deliberate practice (Kuhn & Stahl, 2003). Additionally, strategies for helping students make inferences when reading (Hansen & Pearson, 1983) may also be applicable. These connections between reading and programming may be a productive line of inquiry.

Another possible connection relates to two pedagogical phases of reading instruction. The reading literature talks about students' experiences as they transition from "learning to read" to the later stage of "reading to learn" in which students use their skills of reading to acquire new content knowledge (Smagorinsky & Mayer, 2014). In computing instruction, we often have a similar transition in which introductory courses focus on *learning to program*, and then later courses use programming as a way of illustrating particular algorithms, which we could similarly call *programming to learn*.

Lastly, we expect that most educators encourage students to plan their programs before starting to write code. Computing educators may also wish to mention to students that creating an outline improves the quality of a resulting essay (Kellogg, 1994, 2008), and the same may be true for programs. Again, this may be a fruitful direction for research.

3.7 Importance of Identity to Learning

There exists a large body of research about the importance of belonging and motivation for student learning. In particular, this work focuses on processes of identity development. Similarly, CEdR focused on equity and diversity frequently addresses students' feelings of belonging in computing (see Chapter 16). Identity development, and the underlying need for motivation and belonging, can further our understanding of the processes of learning in CS.

3.7.1 Evidence from Outside of Computing

To understand students' thinking and learning, many have proposed that it is insufficient to consider only the cognitive processes involved in learning (Lave & Wenger, 1991; Nasir & Hand, 2008; Nasir & Shah, 2011; Shah, 2017; Wortham, 2006). Instead, from a sociocultural perspective, it is important to consider students' motivations and other contextual aspects of their experience. An original emphasis on this comes from Lave and Wenger in their discussion of *situated learning*: "Rather than asking what kinds of cognitive processes and conceptual structures are involved, they ask what kinds of social engagements provide the proper context for learning to take place" (Lave & Wenger, 1991, p. 14).

Applying ideas from Lave and Wenger (1991), Stevens et al. (2008) focus on the experiences of individuals in engineering programs, and they propose a framework for engineering education to include disciplinary knowledge, navigation, and identification. Disciplinary knowledge focuses on students' achieving traditional learning outcomes, which may vary drastically as they progress from introductory to advanced courses. Navigation focuses on students' paths through engineering programs. Identification focuses on "[h]ow a person identifies with engineering and is identified by others as an engineer" (Stevens, et al., 2008, p. 356). This framework may serve as a model for CEd, particularly in how it challenges the more common narrative about an engineering “pipeline.” Stevens et al. explain: “[t]he problem with the pipeline metaphor is that the metaphor’s component parts seem to commit its users to a homogeneous view of people/fluids passing through the pipeline” (Stevens et al., 2008, p. 365).

3.7.2 Connections to CS: Observations and Hypotheses

Like in other disciplines, identity is related to learning. At a minimum, identity development within computing appears relevant to learning in computing if only because identity development relates to students’ decisions to pursue or leave computing. We can see ample connections to the processes of identity development, much of which is reviewed in Chapter 16. For example, Barker et al. (2002) described ways in which computing classrooms can reinforce the ideas that some students belong and others do not. Sapna Cheryan and colleagues have focused on how students’ sense of belonging is shaped by environmental cues (Cheryan et al., 2009), interacting with computer scientists who confirm or challenge typical stereotypes (Cheryan et al., 2011), and reading reports that discredit the validity of typical stereotypes of computer scientists (Cheryan et al., 2013).

4 Open Questions

4.1 How Do Misconceptions Related to Intuition Relate to Misconceptions in Other Domains?

Above, we argued that intuition in introductory programming is different from that in other domains because programming languages could (and are) designed to align with human intuition. At a minimum, this can help us to see students’ reasoning as relevant and a productive resource for educators. However, this difference in the nature of the domain leaves open the question of whether the processes of learning and the nature of the resulting knowledge are similar to or different from those in other disciplines. These questions are a central emphasis within the learning sciences and studies of conceptual change, which focus on beliefs that are deeply held and difficult to change (diSessa, 2014a). Are the misconceptions that we observe in computing similar in this respect?

4.2 How Should Teachers and Instruction Respond to Misconceptions from Intuition?

There is not a consensus among conceptual change researchers about the best response to misconceptions (diSessa, 2014a). Additionally, responses to misconceptions should likely take into account the nature of the processes of learning and the resulting knowledge. However, as we described in the previous section, these remain open questions. diSessa argues that this "[c]onfront and replace is an implausible instructional strategy" (diSessa, 2014a, p. 102) for conceptual change because of the sheer number of ideas that constitute conceptual understanding. Instead, diSessa argues for building upon students' existing ideas (diSessa, 2014b). However, this returns us to the focus on the nature of the knowledge to be learned. For example, a "confront and replace" strategy might be appropriate for helping students learn to recognize letters in an alphabet. For this type of declarative knowledge, we might find different instructional strategies effective.

4.3 How Should Information about Misconceptions be Disseminated among Teachers?

The question of how teachers should respond to misconceptions is interrelated with how information about misconceptions should be disseminated among teachers. Page Keeley and colleagues have published a set of books (e.g., Keeley et al., 2005) to guide teachers' formative assessment to be aligned with possible misconceptions. This format likely provides clearer paths to implementation than literature reviews about misconceptions (Qian & Lehman, 2017) or lists of teacher-identified misconceptions as can be found on CSTeachingTips.org.

4.4 What Evidence Should Be Required to Document a Misconception?

To date, multiple methods have been used to identify student misconceptions. For example, Qian and Lehman (2017) summarize misconceptions research that described misconceptions observed in a single student's response. In contrast, Stephens-Martinez et al. (2017) found that a small set of wrong answers (approximately 5 percent of all distinct wrong answers) accounted for approximately 60 percent of the wrong answers that students submitted. However, these wrong answers could include not distinguishing between whether the string X is printed as "x", 'x', or x. This type of common mistake might be valuable for educators, but might not match the typical definition of a misconception. Evidence from single cases might be valuable when it formalizes or explains patterns of student responses that are recognizable to educators. For example, Chung et al. (2017) draw on data from two students to argue that students may develop the misconception that data structures can only store strings or integers. This may simply be the result of a narrow set of examples presented to students, but this reliance on "simple" examples such as storing strings or integers is likely common. Documenting misconceptions might have practical relevance and/or help address the broader questions about the nature of students' knowledge. In the case of practical relevance, the question of what documentation is required may be inseparable from how teachers should integrate this information and how this information will be disseminated to teachers.

4.5 Which Curricular Factors Influence the Manifestation of Misconceptions and How?

As discussed in Section 2.2, even minor changes in how topics are presented can prevent the formation of misconceptions (Karpierz & Wolfman, 2014; Vahrenhold & Paul, 2014). A prominent example of how much more significant curricular decisions influence which misconceptions may or may not manifest themselves is recursion. As demonstrated by recent surveys (McCauley et al., 2015; Rinderknecht, 2014), a large number of research papers have dealt with issues related to teaching and understanding recursion.

Hamouda et al. (2017) present the development and validation of a basic recursion concept inventory. Based upon a literature review and advice from an expert panel, the authors developed test items focused on the following: (1) passive control flow after reaching the base case; (2) active control flow until reaching the base case; (3) how to formulate the recursive call; (4) how to formulate the stopping condition and when it will be triggered; (5) infinite recursion; (6) confusion with loop structures; and (7) unawareness of how variables are updated on every recursion (Hamouda et al., 2017, pp. 126–127).

As can be seen from the above list, all topics except for the last two are independent of whether the programming language used during instruction was functional, imperative, or object-oriented. In fact, the literature review about recursion by McCauley et al. reports on courses using "LISP, LOGO, BASIC, C, Python, SIMPLE, Scheme, Miranda, SOLO, Pascal, and pseudo-code" (McCauley et al., 2015, p. 38).

McCauley et al. (2015) cite one of the reasons why students may have issues with recursion as that it is difficult to find real-world examples and that students thus do not develop intuition about recursion. However, a curriculum focused on recursive data structures naturally lends itself to algorithms on these using (structural) recursion. One such curriculum is the How to Design Programs curriculum (Felleisen et al., 2000). A study by Fisler (2014) showed that students taught according to this curriculum performed better on a standard programming benchmark, the so-called Rainfall Problem (Soloway, 1986). However, Fisler concedes that one could argue that "Rainfall is biased in favor of functional programming, because [the components used for the solution] are standard, heavily exercised problems in functional CS1 courses" (Fisler, 2014, p. 42). As a consequence, while a functional, decomposition-based approach is promising with respect to understanding recursion, more research is needed to understand at which expenses, if any, this advantage comes.

In a curriculum that starts with object orientation (see Chapter 13 for more details), several types of misconceptions have been found to emerge. In particular, the high interconnectedness of concepts needed to understand object orientation (Pedroni & Meyer, 2010) provides challenges that have been observed and investigated in a number of studies.

Ragonis and Ben-Ari (2005) present the results of two yearlong studies with a total of 47 students in Grade 10. The students were novices with respect to concepts in object-oriented programming and were taught using BlueJ and Java. While the primary focus of the study was on which concepts in object orientation could be taught to students at that age, a second research focus was on which concepts students developed during the course. For the purpose of this, observations were recorded by taking field notes while attending the lectures. In addition, homework assignments, lab exercises, tests, and projects were analyzed as well.

Ragonis and Ben-Ari provide an encyclopedic description that shows the breadth of misconceptions and difficulties related to both object orientation in general and object orientation in Java in particular that can be encountered in class. They note, however, that most of these misconceptions and difficulties "appeared with low frequency and characterized a particular period of learning" (Ragonis & Ben-Ari, 2005, p. 218). Ragonis and Ben-Ari note that, at least in the course monitored, several difficulties disappeared as the course progressed, and they present a set of best-practice recommendations for teaching object orientation; see also similar discussions by Kölling and Rosenberg (2001) and Sanders and Thomas (2007).

In a revalidation study of multiple previous studies, Sanders and Thomas (2007) investigated whether previously reported misconceptions could be detected among a small group of students that were taught in an objects-first course designed by the first author and a colleague. They analyzed 71 artifacts (programs) submitted by the students at various points during the course. The authors were able to confirm students' difficulties with linking classes of a design such that objects of these classes could interact properly; in addition, hierarchies of classes and abstractions were found to be problematic. Furthermore, students were found to occasionally conflate objects and classes as well as classes and collections. Some students seemed to treat objects as nothing more than pieces of code, others created objects that would function within the program but did not correspond to the modeled domain. All of these behaviors confirm results from a previous study by Eckerdal and Thune (2005). On the other hand, Sanders and Thomas were unable to reconfirm a small set of misconceptions, including the conflation of attributes and identity that had been discussed by other authors (e.g., Holland et al., 1997). Even given a small sample size, these observations, or rather the lack thereof, seem to imply that at least some of the misconceptions reported in earlier work can be avoided by carefully selecting examples used when teaching; see Holland et al. (1997) or Kölling and Rosenberg (2001). To aid practitioners, Sanders and Thomas present two checklists containing indicators for students' understanding of concepts in object orientation and indicators for the presence of misconceptions.

In summary, this section demonstrates that a curricular decision may have a significant impact on the formation or avoidance of misconceptions. Educators and researchers thus need to take into account the curricular context when addressing a particular misconception.

5 Conclusion

We see the continued exploration of students' learning of computing as fruitful for both improving CEd and expanding our understanding of more general processes of learning. One important lesson from the education research we have surveyed is that learning takes time. In CEd, we often lament the difficulty students have in learning foundational content. The literature we have surveyed leads us to argue that this should be seen as part of the natural learning processes or possibly as evidence of deficiencies in current teaching methods. That mind-set may best drive us to better understand how to help students build a robust knowledge base in computing.

References

Adams, W. K., & Wieman, C. E. (2011). Development and validation of instruments to measure learning of expert-like thinking. *International Journal of Science Education*, 33(9), 1289–1312.

Almstrum, V. L., Henderson, P. B., Harvey, V. J., Heeren, C., Marion, W. A., Riedesel, C., Soh, K.-L., & Tew, A. E. (2006). Concept inventories in computer science for the topic discrete mathematics. *SIGCSE Bulletin*, 38(4), 132–145.

Barker, L. J., Garvin-Doxas, K., & Jackson, M. (2002). Defensive climate in the computer science classroom. In *Proceedings of the 33rd ACM Technical Symposium on Computer Science Education (SIGCSE 2002)* (pp. 43–47). New York: ACM Press.

Baddeley, A. D., & Hitch, G. (1974). Working memory. *Psychology of Learning and Motivation*, 8, 47–89.

Baxter, G. P., & Glaser, R. (1998). Investigating the cognitive complexity of science assessments. *Educational Measurement: Issues and Practice*, 17(3), 37–45.

Beck, K. (2003). *Test-Driven Development by Example*. Boston, MA: Addison-Wesley.

Buswell, G. T. (1930). Summary of arithmetic investigations (1929). *The Elementary School Journal*, 30(10), 766–775.

Bonar, J., & Soloway, E. (1989). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human–Computer Interaction*, 1(2), 133–161.

Bransford, J. D., Brown, A., & Cocking, R. (1999). *How People Learn: Mind, Brain, Experience, and School*. Washington, DC: National Research Council.

Carbonneau, K. J., Marley, S. C., & Selig, J. P. (2013). A meta-analysis of the efficacy of teaching mathematics with concrete manipulatives. *Journal of Educational Psychology*, 105(2), 380–400.

Carpenter, T. P., Fennema, E., Peterson, P. L., & Carey, D. A. (1988). Teachers' pedagogical content knowledge of students' problem solving in elementary arithmetic. *Journal for Research in Mathematics Education*, 19(5), 385–401.

Chan, W. W. L., Au, T. K., & Tang, J. (2014). Strategic counting: A novel assessment of place-value understanding. *Learning and Instruction*, 29, 78–94.

Cheryan, S., Plaut, V. C., Davies, P. G., & Steele, C. M. (2009). Ambient belonging: How stereotypical cues impact gender participation in computer science. *Journal of Personality and Social Psychology*, 97(6), 1045–1060.

Chase, W. G., & Simon, H. A. (1973). Perception in chess. *Cognitive Psychology*, 4(1), 55–81.

Cheryan, S., Plaut, V. C., Handron, C., & Hudson, L. (2013). The stereotypical computer scientist: Gendered media representations as a barrier to inclusion for women. *Sex Roles*, 69(1–2), 58–71.

Cheryan, S., Siy, J. O., Vichayapai, M., Drury, B. J., & Kim, S. (2011). Do female and male role models who embody STEM stereotypes hinder women's anticipated success in STEM? *Social Psychological and Personality Science*, 2(6), 656–664.

Chi, M. T., Feltovich, P. J., & Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. *Cognitive Science*, 5(2), 121–52.

Chung, A., Shao, P., & Vasquez, A. (2017). Students' misconceptions about the types of values data structures can store. *Journal of Computing Sciences in Colleges*, 32(4), 72–78.

Clancy, M. (2004). Misconceptions and attitudes that interfere with learning to program. In S. Fincher & M. Petre (Eds.), *Computer Science Education Research* (pp. 85–100). Abingdon, UK: Taylor and Francis.

Confrey, J. (1990). A review of the research on student conceptions in mathematics, science, and programming. *Review of Research in Education*, 16, 3–56.

Danielsiek, H., Paul, W., & Vahrenhold, J. (2012). Detecting and understanding students' misconceptions related to algorithms and data structures. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE 2012)* (pp. 21–26). New York: ACM Press.

Davis, E. A., Linn, M. C., Mann, L. M., & Clancy, M. J. (1993). Mind your P's and Q's: Using parentheses and quotes in LISP. In C. R. Cook, J. C. Scholtz, & J. C. Spohrer (Eds.), *Empirical Studies of Programmers: Fifth Workshop* (pp. 63–85). Norwood, NJ: Ablex.

Davis, E. A., Linn, M. C., & Clancy, M. J. (1995). Learning to use parentheses and quotes in LISP. *Computer Science Education*, 6(1), 15–31.

Denny, P., Luxton-Reilly, A., & Simon, B. (2008). Evaluating a new exam question: Parsons' problems. *Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08)* (pp. 113–124). New York: ACM Press.

diSessa, A. A. (2014a). A history of conceptual change research: Threads and fault lines. In R. K. Sawyer (Ed.), *The Cambridge Handbook of the Learning Sciences*, 2nd edn. (pp. 88–108). New York: Cambridge University Press.

diSessa, A. A. (2014b). The construction of causal schemes: Learning mechanisms at the knowledge level. *Cognitive Science*, 38(5), 795–850.

Dowhower, S. L. (1994). Repeated reading revisited: Research into practice. *Reading & Writing Quarterly: Overcoming Learning Difficulties*, 10(4), 343–358.

Eckerdal, A., & Thuné, M. (2005). Novice Java programmers' conceptions of "object" and "class", and variation theory. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '05)* (pp. 89–93). New York: ACM Press.

Felleisen, M., Findler, R. B., Flatt, M., & Krishnamurthi, S. (2000). *How to Design Programs: An Introduction to Programming and Computing*. Cambridge, MA: MIT Press.

Fisler, K. (2014). The Recurring Rainfall Problem. In *Proceedings of the International Computing Education Research Conference (ICER 2014)* (pp. 35–42). New York: ACM Press.

Fisler, K., Krishnamurthi, S., & Tunnell Wilson, P. (2017). Assessing and teaching scope, mutation, and aliasing in upper-level undergraduates. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2017)* (pp. 21–26). New York: ACM Press.

Fuson, K. C. (1998). Pedagogical, mathematical, and real-world conceptual-support nets: A model for building children's multidigit domain knowledge. *Mathematical Cognition*, 4(2), 147–186.

Gal-Ezer, J., & Trakhtenbrot, M. (2016). Identification and addressing reduction-related misconceptions. *Computer Science Education*, 26(2–3), 89–103.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley.

Ginat, D., & Blau, Y. (2017). Multiple levels of abstraction in algorithmic problem solving. In *Proceedings of the 48th ACM Technical Symposium on Computer Science Education (SIGCSE 2017)* (pp. 237–242). New York: ACM Press.

Gobet, F., Lane, P. C. R., Croker, S., Cheng, P. C.-H., Jones, G., Oliver, I., & Pine, J. M. (2001). Chunking mechanisms in human learning. *Trends in Cognitive Sciences*, 5(6), 236–243.

Guzdial, M. (2015). Learner-centered design of computing education: Research on computing for everyone. *Synthesis Lectures on Human-Centered Informatics*, 8(6), 1–165.

Hamouda, S., Edwards, S. H., Elmongui, H. G., Ernst, J. V., & Shaffer, C. A. (2017). A basic recursion concept inventory. *Computer Science Education*, 27(2), 121–148.

Hansen, J., & Pearson, P. D. (1983). An instructional study: Improving the inferential comprehension of good and poor fourth-grade readers. *Journal of Educational Psychology*, 75(6), 821–829.

Herman, G. L., Loui, M. C., Kaczmarczyk, L. C., & Zilles, C. B. (2012). Describing the what and why of students' difficulties in Boolean logic. *ACM Transactions on Computing Education*, 12(1), 3.

Holland, S., Griffiths, R., & Woodman, M. (1997). Avoiding object misconceptions. In *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education (SIGSCE 1997)* (pp. 131–134). New York: ACM Press.

Hughes, E. M., Witzel, B. S., Riccomini, P. J., Fries, K. M., & Kanyongo, G. Y. (2014). A meta-analysis of algebra interventions for learners with disabilities and struggling learners. *Journal of the International Association of Special Education*, 15(1), 36–47.

Kahney, H. (1989). What do novice programmers know about recursion? In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 209–228). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Karpierz, K., & Wolfman, S. A. (2014). Misconceptions and concept inventory questions for binary search trees and hash tables. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '13)* (pp. 109–114). New York: ACM Press.

Keeley, P., Eberle, F., & Farrin, L. (2005). *Uncovering Student Ideas in Science, Vol. 1: 25 Formative Assessment Probes*. Arlington, VA: National Science Teacher Association Press.

Kellogg, R. T. (2008). Training writing skills: A cognitive developmental perspective. *Journal of Writing Research*, 1(1), 1–26.

Kellogg, R. T. (1994). *The Psychology of Writing*. New York: Oxford University Press.

Kölling, M., & Rosenberg, J. (2001). Guidelines for teaching object orientation with Java. In *Proceedings of the 6th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '01)* (pp. 33–36). New York: ACM Press.

Kuhn, M. R., & Stahl, S. A. (2003). Fluency: A review of developmental and remedial practices. *Journal of Educational Psychology*, 95(1), 3–21.

Ladd, H. F., & Sorensen, L. C. (2017). Returns to teacher experience: Student achievement and motivation in middle school. *Education Finance and Policy*, 12(2), 241–279.

Lane, P. C., Cheng, P. C. H., & Gobet, F. (2000). CHREST+: A simulation of how humans learn to solve problems using diagrams. *AISB Quarterly*, 103, 24–30.

Lave, J., & Wenger, E. (1991). *Situated Learning: Legitimate Peripheral Participation*. Cambridge, UK: Cambridge University Press.

Levin, J. R., Levin, M. E., Glasman, L. D., & Nordwall, M. B. (1992). Mnemonic vocabulary instruction: Additional effectiveness evidence. *Contemporary Educational Psychology*, 17(2), 156–174.

Lewis, C. M. (2012). *Applications of Out-of-Domain Knowledge in Students' Reasoning about Computer Program State* (PhD dissertation). University of California, Berkeley.

McCauley, R., Grissom, S., Fitzgerald, S., & Murphy, L. (2015). Teaching and learning recursive programming: A review of the research literature. *Computer Science Education*, 25(1), 37–66.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63(2), 81–97.

Miller, S. P., Butler, F. M., & Lee, K.-H. (1998). Validated practices for teaching mathematics to students with learning disabilities: A review of literature. *Focus on Exceptional Children*, 31(1), 1–24.

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 International Computing Education Research Conference (ICER '12)* (pp. 111–118). New York: ACM Press.

Nasir, N. S., & Hand, V. (2008). From the court to the classroom: Opportunities for engagement, learning, and identity in basketball and classroom mathematics. *Journal of the Learning Sciences*, 17(2), 143–179.

Nasir, N. S., & Shah, N. (2011). On defense: African American males making sense of racialized narratives in mathematics education. *Journal of African American Males in Education*, 2(1), 24–45.

Nathan, M. J., & Petrosino, A. (2003). Expert blind spot among preservice teachers. *American Educational Research Journal*, 40(4), 905–928.

O'Brien, D., Dias, M. G., Roazzi, A., & Cantor, J. B. (1998). Pinocchio's nose knows: Preschool children recognize that a pragmatic rule can be violated, an indicative conditional can be falsified, and that a broken promise is a false promise. In M. D. S. Braine & D. P. O'Brien (Eds.), *Mental Logic* (pp. 447–457). Hillsdale, NJ: Lawrence Erlbaum Associates.

O'Brien, D. P., Roazzi, A., Dias, M. G., Cantor, J. B., & Brooks, P. J. (2004). Violations, lies, broken promises, and just plain mistakes: The pragmatics of counterexamples, logical semantics, and the evaluation of conditional assertions, regulations, and promises in variants of Wason's selection task. In K. Manktelow & M. C. Chung (Eds.), *Psychology of Reasoning: Theoretical and Historical Perspectives* (pp. 95–126). Hove, UK: Psychology Press.

Parsons, D., & Haden, P. (2006). Parson's programming puzzles: A fun and effective learning tool for first programming courses. In *Proceedings of the 8th*

Australasian Conference on Computing Education (pp. 157–163). Darlinghurst, Australia: Australian Computer Society.

Pea, R. D. (1986). Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, 2(1), 25–36.

Pedroni, M. & Meyer, B. (2010). Object-oriented modeling of object-oriented concepts: A case study in structuring an educational domain. In *Proceedings of the 4th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives (ISSEP 2010), Lecture Notes in Computer Science 5941* (pp. 155–169). Berlin, Germany: Springer.

Pollack, I. (1953). Assimilation of sequentially encoded information. *The American Journal of Psychology*, 66(3), 421–435.

Pollard, S., & Duvall, R. C. (2006). Everything I needed to know about teaching I learned in kindergarten: Bringing elementary education techniques to undergraduate computer science classes. In *Proceedings of the 37th ACM Technical Symposium on Computer Science Education (SIGCSE 2006)* (pp. 224–228). New York: ACM Press.

Qian, Y. & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education*, 18(1), 1.

Ragonis, N., & Ben-Ari, M. (2005). A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education*, 15(3), 203–221.

Rich, K. M., Strickland, C., Binkowski, T. A., Moran, C., & Franklin, D. (2017). K–8 learning trajectories derived from research literature: Sequence, repetition, conditionals. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)* (pp. 182–190). New York: ACM Press.

Rivera, D., & Smith, D. D. (1988). Using a demonstration strategy to teach midschool students with learning disabilities how to compute long division. *Journal of Learning Disabilities*, 21(2), 77–81.

Rinderknecht, C. (2014). A survey on teaching and learning recursive programming. *Informatics in Education*, 13(1), 87–119.

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

Sadler, P. M., Sonnert, G., Coyle, H. P., Cook-Smith, N., & Miller, J. L. (2013). The influence of teachers' knowledge on student learning in middle school physical science classrooms. *American Educational Research Journal*, 50(5), 1020–1049.

Sajaniemi, J., & Kuittinen, M. (2005). An experiment on using roles of variables in teaching introductory programming. *Computer Science Education,* 15(1), 59–82.

Samuels, S. J. (1979). The method of repeated readings. *The Reading Teacher*, 32(4), 403–408.

Sanders, K., & Thomas, L. (2007). Checklists for grading object-oriented CS1 programs: Concepts and misconceptions. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '07)* (pp. 166–170). New York: ACM Press.

Sawyer, R. K. (2014). Introduction: The new science of learning. In R. K. Sawyer (Ed.), *The Cambridge Handbook of the Learning Sciences*, 2nd edn. (pp. 1–18). New York: Cambridge University Press.

Seppälä, O., Malmi, L., & Korhonen, A. (2006). Observations on student misconceptions – A case study of the build-heap algorithm. *Computer Science Education*, 16(3), 241–255.

Sfard, A. (1995). The development of algebra: Confronting historical and psychological perspectives. *The Journal of Mathematical Behavior*, 14(1), 15–39.

Shah, N. (2017). Race, ideology, and academic ability: A relational analysis of racial narratives in mathematics. *Teachers College Record*, 119(7), 1–42.

Smagorinsky, P., & Mayer, R. E. (2014). Learning to be literate. In R. K. Sawyer (Ed.), *The Cambridge Handbook of the Learning Sciences*, 2nd edn. (pp. 605–625). New York: Cambridge University Press.

Smith, III, J. P., diSessa, A. A., & Roschelle, J. (1993). Misconceptions reconceived: A constructivist analysis of knowledge in transition. *Journal of the Learning Sciences*, 3(2), 115–163.

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

Spohrer, J. G., & Soloway, E. (1986). Analyzing the high frequency bugs in novice programs. In *Papers Presented at the First Workshop on Empirical Studies of Programmers* (pp. 230–251). Norwood, NJ: Ablex.

Stahl, S. A., & Heubach, K. M. (2005). Fluency-oriented reading instruction. *Journal of Literacy Research*, 37(1), 25–60.

Stephens-Martinez, K., Ju, A., Parashar, K., Ongowarsito, R., Jain, N., Venkat, S., & Fox, A. (2017). Taking advantage of scale by analyzing frequent constructed-response, code tracing wrong answers. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (pp. 56–64). New York: ACM Press.

Stevens, R., O'Connor, K., Garrison, L., Jocuns, A., & Amos, D. M. (2008). Becoming an engineer: Toward a three dimensional view of engineering learning. *Journal of Engineering Education*, 97(3), 355–368.

Vahrenhold, J., & Paul, W. (2014). Developing and validating test items for first-year computer science courses. *Computer Science Education*, 24(4), 304–333.

von Sydow, M. (2006). *Towards a flexible Bayesian and deontic logic of testing descriptive and prescriptive rules: Explaining content effects in the Wason selection task* (PhD dissertation). University of Göttingen.

Vosniadou, S., & Brewer, W. F. (1992). Mental models of the earth: A study of conceptual change in childhood. *Cognitive Psychology*, 24(4), 535–585.

Wason, P. C. (1968). Reasoning about a rule. *The Quarterly Journal of Experimental Psychology*, 20(3), 273–281.

Wason, P. C., & Johnson-Laird, P. N. (1972). *Psychology of Reasoning: Structure and Content*. Cambridge, UK: Harvard University Press.

Webb, K. C., & Taylor, C. (2014). Developing a pre- and post-course concept inventory to gauge operating systems learning. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE 2014)* (pp. 103–108). New York: ACM Press.

Willingham, D. T., Hughes, E. M., & Dobolyi, D. G. (2015). The scientific status of learning styles theories. *Teaching of Psychology*, 42(3), 266–271.

Winslow, L. E. (1996). Programming pedagogy – A psychological overview. *ACM SIGCSE Bulletin*, 28(3), 17–22.

Witzel, B. S., Riccomini, P. J., & Schneider, E. (2008). Implementing CRA with secondary students with learning disabilities in mathematics. *Intervention in School and Clinic*, 43(5), 270–276.

Wortham, S. (2006). *Learning Identity: The Joint Emergence of Social Identification and Academic Learning*. Cambridge, UK: Cambridge University Press.

Young, R. M., & O'Shea, T. (1981). Errors in children's subtraction. *Cognitive Science*, 5(2), 153–177.