Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review

YIZHOU QIAN and JAMES LEHMAN, Purdue University

Efforts to improve computer science education are underway, and teachers of computer science are challenged in introductory programming courses to help learners develop their understanding of programming and computer science. Identifying and addressing students' misconceptions is a key part of a computer science teacher's competence. However, relevant research on this topic is not as fully developed in the computer science education field as it is in mathematics and science education. In this article, we first review relevant literature on general definitions of misconceptions and studies about students' misconceptions and other difficulties in introductory programming. Next, we investigate the factors that contribute to the difficulties. Finally, strategies and tools to address difficulties including misconceptions are discussed.

Based on the review of literature, we found that students exhibit various misconceptions and other difficulties in syntactic knowledge, conceptual knowledge, and strategic knowledge. These difficulties experienced by students are related to many factors including unfamiliarity of syntax, natural language, math knowledge, inaccurate mental models, lack of strategies, programming environments, and teachers' knowledge and instruction. However, many sources of students' difficulties have connections with students' prior knowledge. To better understand and address students' misconceptions and other difficulties, various instructional approaches and tools have been developed. Nevertheless, the dissemination of these approaches and tools has been limited. Thus, first, we suggest enhancing the dissemination of existing tools and approaches and investigating their long-term effects. Second, we recommend that computing education research move beyond documenting misconceptions to address the development of students' (mis)conceptions by integrating conceptual change theories. Third, we believe that developing and enhancing instructors' pedagogical content knowledge (PCK), including their knowledge of students' misconceptions and ability to apply effective instructional approaches and tools to address students' difficulties, is vital to the success of teaching introductory programming.

CCS Concepts: • **Social and professional topics → Computing education**; *CS1*;

Additional Key Words and Phrases: Misconceptions, difficulties, constructivism, introductory programming, conceptual change

ACM Reference format:
Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (October 2017), 24 pages.
https://doi.org/10.1145/3077618

This work is supported by the National Science Foundation under grant number 1502462. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.
Authors' addresses: Y. Qian and J. Lehman, Purdue University Department of Curriculum and Instruction, 100 N. University St. West Lafayette, IN 47907-2098; emails: {qian47, lehman}@purdue.edu.

https://doi.org/10.1145/3077618

1 INTRODUCTION

The United States is projected to need more STEM professionals than it will produce over the next decade, and the need for individuals in computer and information science is expected to grow faster than the average for all occupations (Bureau of Labor Statistics 2015). Because of the growing demand for computing professionals to drive innovation and economic progress, efforts are being made to improve the state of computer science education in the United States. These efforts include the development by the College Board and the computer science community of a new CS Principles course (http://www.apcsprinciples.org), intended to broaden participation in the discipline; the National Science Foundation's CS10K initiative, which aims to expand the number of high school computer science teachers (http://www.cs10kcommunity.org); and the White House's CS for All initiative, which aims to equip all students with computing knowledge and skills (https://www.whitehouse.gov/blog/2016/01/30/computer-science-all).

As computer science education expands, teachers of computer science are challenged to help students develop accurate understandings. Introductory programming courses are challenging to students (Guzdial 2015; McCracken et al. 2001), and novices often exhibit misconceptions and other difficulties in the context of learning to program that inhibit their ability to learn and make progress. Teachers' knowledge of student misconceptions and/or ability to assess students and reveal their misconceptions is thought to be important to effective teaching (Sadler et al. 2013). Thus, raising awareness of students' misconceptions and other difficulties, and strategies for addressing them, can help computer science teachers to better teach their students.

In this article, we first review relevant literature on general definitions of misconceptions. Second, we review studies about misconceptions and other difficulties that students have in introductory programming. Furthermore, we investigate the factors that contribute to the difficulties. Finally, strategies and tools to address difficulties including misconceptions are discussed.

2 DEFINITIONS OF MISCONCEPTIONS

2.1 Definitions in Science Education

In the learning of science, misconceptions are the flawed ideas held by students, often strongly, which conflict with commonly accepted scientific consensus (Clement 1993; Sanger and Greenbowe 1997; Smith et al. 1994). Interest in students' thinking that is at odds with the textbook view of science topics arose in the 1980s, and in the literature various terms have been applied to this kind of thinking including misconceptions, alternative conceptions, preconceptions, and others (Taber 2014). Different authors may ascribe particular meanings to particular terms. For example, some researchers use the term "alternative conceptions" because the term "misconceptions" may have a negative connotation (Clement 1993). Naïve conceptions, naïve knowledge, and intuitive knowledge are also used to describe the students' prior knowledge that interferes with learning and leads students to fail to understand the correct scientific concepts (diSessa 2014; Klopfer et al. 1983; Özdemir and Clark 2007). However, despite some differences in usage, these different labels may be considered broadly synonymous (Taber 2014).

2.2 Definitions in Computer Science Education

As in the natural sciences, early work on misconceptions in computer science and programming arose in the 1980s (Bayman and Mayer 1983; du Boulay 1986; Pea 1986; Sleeman et al.1986). Similar to the misconception studies in science, researchers in computer science education also use different terms to describe students' inaccurate or incomplete understandings of computer programming, such as "difficulties" (du Boulay 1986), "misconceptions" (Sorva 2013), "errors" (Sleeman et al. 1986), "bugs" (Pea 1986), "mistakes" (Altadmri and Brown 2015), and so forth. As

various terms are used to represent student difficulties, there is no commonly accepted definition of these problems in computer science education. Sorva (2013) defined misconceptions as "understandings that are deficient or inadequate for many practical programming contexts" (p. 8:5) including syntax errors, misunderstandings of concepts and control flow, challenges about using learned constructs, difficulties in planning and debugging programs, and so forth.

In this article, we examine a broad range of flawed or incomplete understandings on the part of learners of introductory computer programming. By introductory programming, we refer specifically to entry-level programming courses (e.g., college-level CS1 courses or similar precollege courses) that introduce students to programming and where instructors will grapple with students' misconceptions and other difficulties. While Sorva (2013) considers many kinds of difficulties as misconceptions, we acknowledge that there are qualitative differences among the various kinds of difficulties students may encounter. For learners of computer science studying loops, for example, these difficulties might range from syntax errors associated with loop statements, to misunderstandings about the execution of a loop that may relate to a student's (mis)conception of the loop construct, to a student's difficulties in using the loop construct to solve a problem. Misconceptions, per se, are probably best defined as errors in conceptual understanding, such as misunderstanding the loop construct in the example earlier. However, difficulties that students encounter in programming are not always neatly identifiable, and misconceptions may contribute to other kinds of difficulties or errors that students encounter. When the distinction is vague or unnecessary, we may use terms such as "errors," "misconceptions," "misunderstandings," "difficulties," "challenges," and so forth interchangeably.

3 MISCONCEPTIONS AND OTHER DIFFICULTIES IN INTRODUCTORY PROGRAMMING

3.1 Framework and Methodology

In previous studies, various frameworks for analyzing programming difficulties have been proposed, including surface and deep errors (Sleeman et al. 1986); misapplication of analogy, overgeneralizations, and interaction errors (du Boulay 1986); language-independent conceptual bugs (Pea 1986); categorizing misconceptions by causes (Clancy 2004); using compilation error frequencies of large-scale datasets (Brown et al. 2014); syntax errors, semantic errors, and logic errors (Hristova et al. 2003; McCall and Kölling 2014); and others. Some researchers only studied student difficulties in a specific programming language (e.g., Java) (Altadmri and Brown 2015) or a specific programming paradigm (e.g., object-oriented programming) (Ragonis and Ben-Ari 2005). Some studies focused on students' errors that were detectable at compile time (Brown et al. 2014). While Sorva (2012) made a comprehensive list of misconceptions of CS1 students, no specific method of categorization was used for the analysis.

In this article, in order to be inclusive, we analyze students' difficulties based on general types of programming knowledge. Learning to program can promote a chain of cognitive accomplishments (Linn 1985) or chain of cognitive changes (Mayer and Fay 1987), which include learning language features, learning the concepts within the domain of programming, and then learning to solve novel problems. Based on the cognitive changes in the learner, knowledge of programming can be categorized into three broad types: syntactic knowledge, conceptual knowledge, and strategic knowledge (Bayman and Mayer 1988; McGill and Volet 1997). Syntactic knowledge is the knowledge of the language features, basic facts, and rules. For example, in Java, double quotation marks are required to define strings, and a semicolon is required to end a statement. Conceptual knowledge refers to the knowledge of how programming constructs and principles work and what happens inside the computer. Examples of conceptual knowledge include how a loop works,

why semicolons are necessary in some languages, and what happens behind an assignment statement. Misconceptions are errors in conceptual understanding. Strategic knowledge refers to how to apply syntactic and conceptual knowledge of programming to solve novel problems. Planning, writing, and debugging programs including tracing or explaining code require strategic knowledge, in addition to knowing syntax and concepts of a programming language. We chose to use this framework because it provides a big-picture view of students' difficulties in learning to program that is grounded in both the educational computing literature and cognitive psychology and has been shown empirically to be useful (McGill and Volet 1997).

Based on this conceptual framework, we used the snowballing approach to find relevant literature. Snowballing, referring to identifying papers using reference lists and citations, is a reliable and efficient way of conducting systematic literature studies, since it makes good use of researcher expertise in an area and reduces noise in the literature search process (Wohlin 2014; Wohlin et al. 2013). The first step of the snowballing approach is to identify a starter set of papers regarding a topic (Wohlin 2014). In this study, to build the starter set, we searched for literature in the ACM Digital Library and ERIC database using keywords including "misconception," "difficulty," and "error" combined with "programming." Among the thousands of returned records sorted by relevance, we reviewed the titles and abstracts of the results in the first two pages of each search and selected 10 papers in total as the starter set (e.g., Ben-David Kolikant and Mussai (2008), Bonar and Soloway (1985), Kaczmarczyk et al. (2010), Pea (1986), Sirkia and Sorva (2012), and Sorva (2013), etc.). We then used the reference lists and citations to locate additional relevant papers. During the snowballing process, we also used the "Cited by" function of Google Scholar to find newly published papers regarding the topic. Because we focused on introductory programming, we excluded studies about non-introductory-level CS courses. For instance, studies such as Danielsiek et al. (2012), which discussed students' misconceptions in Algorithms and Data Structures, were not included in our review. Second, as strategic knowledge is complex and its relevant research covers a wide range of topics such as general problem-solving skills and expert-novice differences, we chose not to address every aspect of it in depth. For example, debugging can be considered as a specific instance of troubleshooting in a general problem-solving process that includes understanding the system, testing the system, locating the errors, and fixing the errors (Katz and Anderson 1987). There exist a considerable number of studies investigating novices' knowledge and difficulties in each stage of debugging (e.g., Ducassé and Emde (1988), Katz and Anderson (1987), and Ben-David Kolikant and Mussai (2008)). To limit the scope of our review of strategic knowledge, we focused on studies about general challenges and difficulties faced by novices in planning, writing, and debugging programs in order to present the big picture, rather than comprehensively addressing every detail.

3.2 Difficulties in Syntactic Knowledge

In introductory programming courses, it is common for students to exhibit errors in syntactic knowledge. By analyzing the large-scale student data from Blackbox, which collects Java code of BlueJ users, Altadmri and Brown (2015) reported that the most frequent student error is mismatched parentheses, brackets, or quotation marks. Other common Java programming errors students make in introductory programming include using irresolvable symbols, missing semicolons, and using illegal start of expressions (Jackson et al. 2005). The irresolvable symbol error usually results from failing to declare a variable before using it. The semicolon is a necessary terminating character in Java, but novices often forget it. The illegal start of expression error is usually caused by unfamiliarity with Java expressions, such as using malformed Boolean expressions. Another frequent novice error is mistakenly using the assignment operator (=) instead of the comparison operator (==) (e.g., if (a = b)) (Altadmri and Brown 2015; Hristova et al. 2003; Sirkia and Sorva

2012). Although many other syntactic-level errors are reported in previous research (see Altadmri and Brown (2015), Hristova et al. (2003), and Sorva (2012)), we do not discuss them in depth here, because problems in syntactic knowledge are often easy to detect and fix. Perhaps that is why they are often noted as the most frequent mistakes novices make (Altadmri and Brown 2015; Jackson et al. 2005). A compiler or a modern integrated development environment (IDE) may be able to find them and then provide error messages or hints for correction.

3.3 Difficulties in Conceptual Knowledge

Although students make the most errors in the basic mechanics of programming languages (Garner et al. 2005; Jackson et al. 2005), these errors are minor, superficial, and easily fixed. Problems in students' conceptual knowledge of programming can lead to deep and significant misconceptions that are related to students' mental models of code execution and the computer system (Bayman and Mayer 1983; Canas et al. 1994; Ma 2007; Sorva 2012). For example, variables are key components of computer programs used to store data values. Students may fail to understand that variables can only hold one value at a time (Doukakis et al. 2007; Sleeman et al. 1986) or that the order of statements assigning values to variables is important; that is, students may believe 5=A is equivalent to A=5 (du Boulay 1986; Ma 2007; Sirkia and Sorva 2012). Students may also mistakenly believe that the English meaning of the variable name affects the value of the variable (Kaczmarczyk et al. 2010; Sleeman et al. 1986). For instance, students might think the value of the variable "largest" is necessarily larger than the value of the variable "smallest," although variable names are arbitrary. The knowledge of variable scope, the context within which a variable is defined, can also confuse students; Fleury (1991) in a study of Pascal learners found that students often used local variables and global variables inappropriately.

Use of variables for input and output operations can also be difficult for students. Novices do not understand where the data come from and where they are stored (Bayman and Mayer 1983). Students may naively believe an output message printed on the screen such as "The value of X is 1" will change the value of the variable X to 1 (Haberman and Ben-David Kolikant 2001; Sleeman et al. 1986).

Conditionals are another difficult concept that leads to misconceptions (Green 1977; Sirkia 2012). Some students believe statements in both *if* and *else* blocks of a conditional expression will be executed (Sirkia 2012; Sleeman et al. 1986). Some students even mistakenly think that if the condition of an *if*-statement is *false,* the execution of the whole program stops (Sleeman et al. 1986).

The looping construct can also confuse novice students who often have problems understanding what the scope of a loop is, which lines will be repeated, how many times the code inside a loop will be executed, and so on (Sleeman et al. 1986). For example, if there are several lines inside a loop, and one of them is an output statement such as a print statement, students may think only the print statement is repeated by the loop because they only see the repeated output on the screen. In addition, as both for-loops and while-loops exist in many programming languages, novices often believe that one looping construct is better than the other one (e.g., while-loops are better than for-loops) based on their familiarity with one type of loop or the order they learned loop constructs, rather than understanding that each looping construct has benefits for solving different problems.

Understanding program execution may also be challenging to novices. Students may fail to understand that program instructions are executed sequentially (du Boulay 1986; Simon 2011). For instance, they may mistakenly believe that a conditional statement inside a loop will be executed whenever the condition is true, even if this occurs outside of the loop (Pea 1986). Even understanding a three-line swap is difficult for novices (Simon 2011). When using functions or methods, students may feel confused about where the parameter values come from and where the return value goes (Ragonis and Ben-Ari 2005).

As object-oriented programming (OOP) and Java have become popular since the 1990s, many studies have investigated students' misconceptions about OOP (Guzdial 1995; Holland et al. 1997; Ragonis and Ben-Ari 2005; Sorva 2013). Not surprisingly, novices often struggle with understanding OOP concepts and principles. For instance, one common challenge students often face is to understand classes, objects, instances, and their relationships (Holland et al. 1997; Kaczmarczyk et al. 2010; Ragonis and Ben-Ari 2005; Teif and Hazzan 2006). Novices may believe that a class is a collection of objects or an object is a subset of a class. Students may also exhibit difficulties in understanding the role of the main method and the relationship between objects and methods (Sajaniemi et al. 2008). For example, some beginners hold the belief that the local variables of the main method are accessible by referred objects. Misunderstandings about references and reference assignment are also common among students in introductory programming (Kaczmarczyk et al. 2010; Ma 2007). For example, some students think that the assignment a=b (where a and b are objects) will replace some of the attributes of a by the corresponding attributes of b (Ma 2007). In addition to the challenges of specific OOP concepts, novices may also experience difficulties in understanding OOP principles (Butler and Morgan 2007), building correct OOP notional machines (Sorva 2012), developing a decentralized mindset (Guzdial 1995), forming appropriate mental models (Ma 2007), and so forth.

3.4 Difficulties in Strategic Knowledge

Strategic knowledge of programming, which is also labeled as conditional knowledge in cognitive psychology, refers to expert-level knowledge about planning, writing, and debugging programs for solving novel problems using syntactic and conceptual knowledge (McGill and Volet 1997). Several different terms are used to describe strategic knowledge, such as strategies (de Raadt 2008), plans (Soloway 1986), schemas (Rist 1989), patterns (Clancy and Linn 1999; Muller 2005), and so on. In this article, we use strategic knowledge to represent all of these similar terms. Students' difficulties in strategic knowledge are highly correlated to their difficulties in syntactic knowledge and conceptual knowledge (Bayman and Mayer 1988; de Raadt 2008; Ebrahimi 1994; Lopez et al. 2008; Whalley et al. 2006; Venables et al. 2009). Students who exhibit more difficulties in basic syntax and inaccurate mental models of programming usually make more errors and show lower performance when they write programs to solve problems (Ebrahimi 1994). Based on a study of explicitly teaching strategic knowledge in introductory programming, de Raadt (2008) concluded that "programming knowledge is prerequisite to programming strategies" (p. 113).

Although without adequate syntactic and conceptual knowledge students will have difficulties in writing correct programs to solve problems, only knowing syntax and semantics does not necessarily produce a good programmer. Unlike experts, novices usually hold fragile programming knowledge and lack programming strategies and patterns (Clancy and Linn 1999; Davies 1993; Lister et al 2004; Lister et al 2006; Perkins and Martin 1986; Sajaniemi and Prieto 2005; Soloway 1986; Whalley et al. 2006). Their lack of well-established strategies and patterns often leads to various challenges in planning, writing, and debugging programs. In general, novices often show difficulties in understanding the task and decomposing the problem (Muller 2005; Robins et al. 2006). Students in introductory programming may know how to write a runnable program but fail to check the appropriate boundaries of conditions and unexpected cases in their program (Fisler et al. 2016; Sajaniemi and Kuittinen 2005; Spohrer and Soloway 1986). Even though the variable is a basic programming concept, novices often fail to correctly initialize and use specific variables such as sum and count variables (de Raadt 2008; Simon 2013). Another strategic challenge for students is to merge blocks of code that should be applied together (de Raadt 2008; Ginat et al. 2013; Muller et al. 2007; Spohrer and Soloway 1986). For example, when solving a problem requires interleaving two or more patterns, novices often only concatenate the patterns and "bypass" the

interleaving (Ginat et al. 2013). Other difficulties in strategic knowledge include choosing a correct loop construct in a specific context (de Raadt 2008; Fisler et al. 2016; Simon 2013), using a guard when necessary such as checking for division by zero (de Raadt 2008; Fisler 2014; Simon 2013), and so forth. Also, many novice students lack skills of effectively testing and debugging programs (Ahmadzadeh et al. 2005; Fitzgerald et al. 2008; McCauley et al. 2008). For instance, students often fail to evaluate the correctness of a program and believe that a program is partially correct if some of its code is written correctly (Ben-David Kolikant and Mussai 2008). When a novice programmer debugs a program, he or she typically reads and traces code in a local manner line by line without a holistic view about programming (Ben-David Kolikant and Mussai 2008; Lister et al. 2006). Thus, the major challenge of debugging faced by novices usually is not fixing the error but rather comprehending the program and locating the error (McCauley et al. 2008). Once errors are identified and located, even novice students can repair most of them (Fitzgerald et al. 2008).

4 FACTORS CONTRIBUTING TO MISCONCEPTIONS AND OTHER DIFFICULTIES

Previous research reveals a number of factors that may contribute to the misconceptions and other difficulties experienced by students. These factors include task complexity and cognitive load, natural language, existing math knowledge, flawed mental models, inadequate patterns and strategies, environmental factors, and teachers' instruction and knowledge.

4.1 Task Complexity and Cognitive Load

Task complexity may contribute to students' difficulties in programming by increasing students' cognitive load and leading to confusion (Anderson and Jeffries 1985; Muller et al. 2007; Spohrer and Soloway 1986). As novices who are learning to program are not familiar with all the new terms and syntax of the programming language, they may forget some basic syntactical parts of statements such as parentheses, braces, and semicolons (Altadmri and Brown 2015; Garner et al. 2005; Jackson et al. 2005), or they may confuse basic operators (Sirkia and Sorva 2012; Sleeman et al. 1986) when they write programs. In a study of CS1 students, Sanders and Thomas (2007) found that most of the students submitted flawless programs for the first assignment but when tasks became more challenging, students started to make more syntactic mistakes. By analyzing student errors in programming, Anderson and Jeffries (1985) found that the complexity of the task made students unable to retain information in their working memory and led them to omit necessary parts of the code. In a pattern-oriented introductory programming course, Muller et al. (2007) found that when simultaneously dealing with tasks relating to different levels of abstraction, novices may face difficulties in solving problems due to the high demand on cognitive load. By studying novice programmers' code tracing skills, Vainio and Sajaniemi (2007) reported that because of limited knowledge and understanding of programming, tracing programs becomes complex and requires high cognitive load on novices, and they often use wrong values for variables in their tracing.

4.2 Natural Language

Because programming language commands are based on natural language, students' understanding of natural language may interfere with their understandings of the meanings of programming commands (Bonar and Soloway 1985; Bruckman and Edwards 1999; du Boulay 1986; Miller 2014). For instance, some novices learning BASIC believed the code "INPUT A," which will input a number and store it in variable A, would input the letter A to the computer memory (Bayman and Mayer 1983). Taber (2014) noted that language can influence students' alternative conceptions when a term used in everyday life does not match the term's scientific meaning. Some students may think the if-statement is always waiting for the Boolean condition to be true, just as we use the word "if" in natural conversation (Pea 1986). Similarly, according to du Boulay (1986), English

words in programming languages have specific meanings, but the student may not understand them. For example, the word "and" is a Boolean operator in many programming languages, but it is a conjunction in everyday use. Natural language interference may differ for non-English speakers. In a study of Chinese students, Qian and Lehman (2016) reported that students' English ability was significantly correlated with their success in learning to program.

Novices often inappropriately use natural language in programming. After analyzing students' programs, Bonar and Soloway (1985) indicated that novices often made misleading links between SSK (step-by-step natural language programming knowledge) and PK (Pascal programming knowledge). For example, students may interpret the word "then" in the if-then-else construct in a programming language as it is used in natural language. As a result, they may add a "then" to the repeat-until construct, which is incorrect in Pascal syntax. Miller (2014) reported that novices often used metonymy, referring to something not by its own name but by something related, in programming, which contributed to reference-point errors. A reference-point error refers to mistakenly referencing another element that is related to the intended element. For example, an object *temperatureText* has an *id* attribute equal to "temperature" and a *value* attribute equal to 32, but novices may use this "givenTemperature = temperature" statement to get the *value* of *temperatureText*. They use metonymy in programming as they often would in natural language and communication and believe that computers are able to infer the intended referent.

4.3 Math Knowledge

Students' prior math knowledge can be another source of misconceptions. For instance, students may confuse variable assignments in a computer program with algebraic expressions, which look similar but mean something quite different (Clancy 2004; Doukakis et al. 2007). In high school algebra, students do not need to declare a variable before using it, which may be one reason that students frequently forget variable declarations in Java programming (Jackson et al. 2005). Students may also believe that values of variables in programming have no size limit and precision limit as they do in math (Doukakis et al. 2007). Another example of the differences between programming and math is that, in languages like C and Java, the result of the division of two integers, say, "1 / 2," is zero, while in math the result would be 0.5. Because students as young as middle school have had years of math experience, the notations used in programming, when they conflict with conventional math, become "hindrances to cognitive comprehension" (Ebrahimi 1994, p. 458).

4.4 Flawed Mental Models

Incomplete or inaccurate mental models of code execution and the computer system also can result in students' misconceptions (Bayman and Mayer 1983; Canas et al. 1994; Ma 2007). A mental model is the student's understanding of the hidden information process underlying the code and what happens inside the computer (Bayman and Mayer 1983). One essential but difficult mental model for novices is a notional machine, which refers to a mental model of an idealized computer that the programmer uses to execute code independent of programming languages (du Boulay 1986; Guzdial 2015; Sorva 2013). Unlike syntactical errors, such as missing parentheses and semicolons, which are obvious and easy to fix, students' misconceptions that result from an incomplete or inaccurate mental model are covert and resistant to change. Moreover, students' mental models are established at an early stage of programming learning (Sleeman et al. 1986). For example, beginners often assume the computer is intelligent like a human being. Pea (1986) found evidence of what he termed a superbug, in which beginning students attributed an intelligence to the computer such that it could interpret what they wished for the computer to do and carry it out. Sleeman et al. (1986) also indicated that a deep error results from students' belief that computers can think like humans.

Thus, an incorrect mental model leads to the misunderstanding about how the code is executed and difficulties in tracing code. Students may know how to use an assignment statement but may not know how variables and values are stored in computer memory (Ma 2007). For instance, some students learning BASIC programming mistakenly thought the whole statement "LET A = B + 1" was stored in memory (Bayman and Mayer 1983). With incomplete knowledge and understanding of program translation and compile-time type checking, it is not surprising that students do not understand the "need" for variable declarations and incorrectly use variables in a math-like way.

4.5 Inadequate Patterns and Strategies

Students who know the syntax and understand the concepts may still face difficulties in solving problems, because they often hold fragmentary knowledge instead of well-organized patterns, lack programming strategies, and fail to reason at an abstract level when comprehending, writing, and debugging code (Clancy and Linn 1999; Lister 2011; McCauley et al. 2008; Sajaniemi and Prieto 2005; Whalley et al. 2006). For instance, even with correct knowledge of syntax and semantics, a student is not necessarily able to explain or evaluate the correctness of a program properly (Ben-David Kolikant and Mussai 2008; Lister et al. 2006). The programming knowledge gained by novice programmers usually is not organized into meaningful patterns, and the connections between the pieces of knowledge are not well established. Perkins and Martin (1986) call it fragile knowledge that is incomplete, hard to retrieve, and often misused. Meanwhile, programming strategies (e.g., debugging strategies) usually are not explicitly taught in introductory programming (de Raadt 2008). Without explicit instruction, students often have difficulties in inferring reusable programming strategies naturally (Clancy and Linn 1999). For example, by explicitly teaching roles of variables, which describe common patterns of using variables in different contexts, Sajaniemi and Kuittinen (2005) reported that 35% of the students in the experimental groups started to use role names, but no students in a traditionally taught group showed evidence of gaining such understanding.

When people, regardless of age, develop expertise in a specific domain, they progress from less abstract to more abstract forms of reasoning (Lister 2011). Using neo-Piagetian theory, Lister (2011) proposed four main stages of cognitive development of novice programmers based on Piaget's stages: sensorimotor, preoperational, concrete operational, and formal operational. Novices at the sensorimotor stage usually have some programming knowledge but cannot reliably trace a short program. A preoperational novice may be able to trace and write programs correctly but often fails to see the connections between different parts of the program (e.g., understanding every line of a program without the comprehension of the purpose of the program). When reaching the concrete operational stage, students start forming programming patterns and can see the relationship between parts and the whole, but the abstract reasoning is "restricted to familiar situations" (Corney et al. 2012, p. 78). Expert programmers usually have reached the formal operational stage and can solve problems in novel situations and "reason logically, consistently, and systematically" (Lister 2011, p. 10). In introductory programming, many students can only reason at the first two stages and fail to reach the concrete operational stage (Teague et al. 2013). According to neo-Piagetian theory, simply forcing students at the sensorimotor and preoperational stages to write a lot of code may not help them reach the concrete operational stage and develop patterns and strategies naturally (Lister 2011). Teague and Lister (2014) indicated that it is essential to develop proper learning experiences (e.g., program comprehension tasks for sensorimotor novices) to help novices gain programming patterns and strategies and progress beyond the preoperational stage.

4.6 Environmental Factors

Though many misconceptions and difficulties result from students' existing knowledge, environmental factors, such as language features and programming environment, may also pose a barrier

to students' success in introductory programming (Ko and Myers 2005; Myers et al. 2004). For example, in some programming languages, such as Java, the addition operator (+) can be used to add integers, concatenate strings, or even concatenate integers with strings. The ambiguity brings flexibility to professionals but also confusion and challenges to novices (Myers et al. 2004). The cryptic and uninformative error messages from the compiler also make error correction difficult for beginners (Becker 2016; Denny et al. 2014). For example, a "cannot find symbol" error message in Java may not effectively help a novice understand and fix the error in his or her program. Using different programming languages in instruction may also have different effects on students' understanding. Students using functional programming may face different difficulties from those who use imperative programming (Fisler 2014; Fisler et al. 2016). In a study of students learning introductory programming with different languages, Kunkle and Allen (2016) reported that using complex object-oriented languages, such as C++ and Java, may result in better understanding of basic programming concepts than using a language like Visual Basic. In addition to programming languages, programming environments may also cause difficulties for students. For instance, beginners may face challenges when using debugging tools of programming environments that do not well support debugging activities (e.g., inspection of variable values at runtime) (Myers et al. 2004).

4.7 Teachers' Instruction and Knowledge

Sometimes teaching itself may contribute to students' inaccurate mental models; examples of problematic teaching strategies include using inappropriate analogies, models, and metaphors (Taber 2014). Describing a variable as a box, which is something many teachers of introductory programming courses do, is an example of misapplication of analogy in programming (Clancy 2004). Students may believe a variable can hold a number of things just as a box. While using analogies may help in explaining complex systems, inappropriate use of analogies may result in barriers preventing novices from establishing accurate mental models (Halasz and Moran 1982).

In addition, teachers' poor content knowledge may also lead to students' misconceptions (Sadler et al. 2013). Teachers who misunderstand the concepts themselves may pass on the inaccurate understanding to students. Even though they may not have misconceptions, teachers with weak content knowledge often teach "rules" rather than "reasons" (Even 1993). Thus, students may memorize syntactic knowledge but not really understand why and form an incorrect mental model.

4.8 Summary

We discussed several sources of students' misconceptions and other difficulties. However, the source of many problems, such as unfamiliarity with syntax, natural language, math knowledge, inaccurate mental models, and inadequate patterns and strategies, is related to students' existing knowledge or previous experience. According to the constructivist view of learning, the learner constructs understanding based on his or her experience (Jonassen 1991). Jonassen (1991) stated, "How one constructs knowledge is a function of the prior experiences, mental structures, and beliefs that one uses to interpret objects and events" (p. 10). Hence, ultimately, students' prior knowledge is a major source of their misconceptions and other difficulties in learning to program (Bonar and Soloway 1985; Smith et al. 1994). If we want to understand what difficulties students have and what causes the difficulties, we need to pay attention to students' prior knowledge.

Some problems are related to teachers' knowledge and instruction. Although teachers' poor content knowledge may lead to problems, even teachers with significant expertise in computer science may also exhibit troubles in instruction. Guzdial (2015) uses the term "expert blind spot" to describe the situation that experts may not be able to see with novices' eyes and therefore fail to comprehend novices' difficulties. Educators often exhibit a weak understanding about their

students' difficulties (Brown and Altadmri 2014). Hence, it is vital to develop teachers' pedagogical content knowledge (PCK), such as knowing students' misconceptions and other difficulties and applying effective teaching strategies and tools to address them.

5 STRATEGIES AND TOOLS TO ADDRESS MISCONCEPTIONS AND OTHER DIFFICULTIES

5.1 Instructional Approaches and Strategies

As misconceptions interfere with learning and are spread widely among students (Smith et al. 1994), many studies have focused on using different instructional approaches to help address students' misconceptions and other difficulties and so enhance students' performance in learning to program. One approach is to use good program examples. Well-chosen program examples are important, because novice programmers often construct their knowledge from them (Fleury 1991). Example programs should be clear and designed to help students build accurate understanding of programming and improve knowledge transfer. By embedding transfer-oriented tasks and worked-out examples in instructional materials, Ginat et al. (2011) reported reduction of students' cognitive load and improvement in solving novel problems. Furthermore, novices usually gain more benefits from worked-out examples than non-novices (Tuovinen and Sweller 1999). Requiring students to read, trace, and explain example programs is also a useful strategy (Mayer 1981; Teague and Lister 2014). Program comprehension is important for writing and debugging code (Lister et al. 2009; Lopez et al. 2008; McCauley et al. 2008). However, novice programmers usually lack program comprehension skills and simply interpret code line by line without seeing the "forest" (Lister et al. 2006; McCauley et al. 2008). Developing learning experiences such as reading and comprehending example programs may help to uncover students' misconceptions and other difficulties, and develop their skills of writing and debugging code (McCauley et al. 2008; Teague and Lister 2014; Vainio and Sajaniemi 2007).

Another approach is to teach programming strategies and patterns explicitly to students. A typical introductory programming course often focuses on covering features and rules of a programming language (Muller et al. 2007; Robins et al. 2003). However, explicitly teaching strategic knowledge of programming may be effective to address students' difficulties. After receiving explicit instruction in programming strategies, de Raadt (2008) indicated that students exhibited improved performance in programming and an increase in the ability to apply strategies while solving problems. By using pattern-oriented instruction (POI), Muller et al. (2007) reported that programming patterns and schemas can reduce students' cognitive load and enhance their competence in problem decomposition and solution construction. Teaching roles of variables also shows effectiveness. In introductory programming, 10 roles can cover 99% of all variables, and three of them, including fixed value, stepper, and most-recent holder, can represent 84% of all variables (Sajaniemi and Kuittinen 2005). After comparing traditional instruction and instruction integrating roles of variables, Sajaniemi and Kuittinen (2005) noted that while the traditional group made fewer superficial errors, such as misuse of control structures, the experimental groups exhibited fewer deeper errors, such as ignoring exceptional conditions of variables.

Other instructional approaches and strategies to tackle student misconceptions include developing a concept inventory (CI) (Goldman et al. 2010; Kaczmarczyk et al. 2010; Sanders and Thomas 2007; Taylor et al. 2014; Tew 2010) and Peer Instruction (PI) (Porter et al. 2013). A CI is an assessment designed to evaluate student understanding of a set of concepts (Goldman et al. 2010; Tew 2010). Developing a concept inventory and assessing students' understanding of the core concepts in introductory programming allows instructors to identify students' misconceptions and then to provide appropriate intervention in the future (Taylor et al. 2014). PI is an instructional approach

to engage students in building their own understanding of concepts using three steps: individually answering a question, discussing the question with peers, and responding to the question again (Simon et al. 2010). PI has been tested in different STEM fields and is considered a promising way to find challenges students face and improve their conceptual understanding (Crouch and Mazur 2001; Simon et al. 2010; Smith et al. 2009). In introductory programming, PI can effectively enhance students' learning and decrease failure rates (Porter et al. 2013; Simon et al. 2010).

5.2 Programming Environments and Learning Tools

In addition to the instructional approaches and strategies, many researchers and educators have designed and applied new tools or advanced computing techniques to address students' difficulties. As novice students often make mistakes in basic syntax, new types of programming environments have been developed that can help to find and prevent syntax errors (Kelleher and Pausch 2005). In many modern code editors, syntax errors such as mismatching quotation marks and missing semicolons are highlighted. With this feature, students can easily identify and fix these problems. Some other systems, such as the Scratch programming environment, prevent syntax errors by supporting graphical programming (Resnick et al. 2009). In this kind of programming environment, students can drag and drop the code to build programs without making syntax errors. Furthermore, graphical programming can help novices build a better understanding of some programming concepts (e.g., loops) that are difficult in text-based programming (Weintrop and Wilensky 2015). In addition, to reduce natural language interference in the learning of programming, natural-language-like programming languages such as Hypertalk and MOOSE (Bruckman and Edwards 1999) have been designed to help novices learn programming. After analyzing students' errors in MOOSE, Bruckman and Edwards (1999) indicated that fewer natural language errors were produced by students in a natural-language-like programming language such as MOOSE than in a formal language like Pascal.

Because inaccurate mental models of programming concepts and the computer system are an important source of students' misconceptions, visualization tools that illustrate the programming concepts and program execution have been helpful for students to develop their understandings of how computer programs function and the notional machine (Sirkia and Sorva 2012; Sorva et al. 2013). For example, Online Python Tutor is a well-known program visualization tool that is widely used by instructors who teach Python-based college-level CS1 courses (Guo 2013). Greenfoot is another tool that explicitly visualizes OOP concepts for improving students' conceptual understanding of OOP (Kölling 2010). UUhistle is a tool focusing on visualizing the notional machine for Python programming and provides automatic feedback for addressing students' misconceptions (Sorva 2012). While visualization tools are helpful to make the notional machine and code execution visible, they may also increase learners' cognitive load and so make things more complex (Sorva 2012; Sorva et al. 2013). Thus, educators should not assume visualization will necessarily address students' misconceptions (Guzdial 2015).

As novices also exhibit difficulties in debugging programs, tools for supporting debugging have been developed. For example, because raw compiler error messages are often uninformative and sometimes misleading, researchers have created tools that can provide enhanced error messages (Becker 2016; Denny et al. 2014). By applying Decaf, a tool that can provide user-friendly and detailed error messages based on raw Java error messages to CS1 students, Becker (2016) found that students using Decaf made significantly fewer errors than those who only saw raw Java error messages. During the debugging process, programmers usually ask two types of questions: "*why did* questions, which assume the occurrence of an *unexpected* runtime action, and *why didn't* questions, which assume the absence of an *expected* runtime action" (Ko and Myers 2005, p. 76). Whyline is a question-based debugging interface that allows the user to ask "why" questions when the program

fails. Ko and Myers (2005) reported that novice programmers who used Whyline spent less time debugging programs and completed 40% more tasks than those who did not.

Another widely used instructional tool in introductory programming is an automated assessment system (Douce et al. 2005). Such systems can automatically grade students' solutions, provide immediate feedback, and enhance students' learning experience (De-La-Fuente-Valentín et al. 2013; Gerdes et al. 2010). Based on historical student data, more advanced automated assessment systems can diagnose students' programs and personalize the feedback for students (Barnes and Stamper 2008; Rivers and Koedinger 2017; Xu and Chee 2003). These intelligent tutoring systems not only assess the correctness of students' programs but also provide customized hints based on specific student errors. When adequate student data are collected, learners' common misconceptions can be effectively discovered by such systems (Altadmri and Brown 2015; Britos et al. 2008), and pre-emptive hints can even be provided before students make mistakes (Dominguez et al. 2010).

5.3 Conceptual Change Theories

In the sciences and mathematics, conceptual change theories have been developed and applied to address students' misconceptions (Vosniadou and Skopeliti 2014). After decades of research, two prominent theoretical perspectives, knowledge-as-theory and knowledge-as-elements, have emerged (Özdemir and Clark 2007). The knowledge-as-theory perspective posits that learners often hold theory-like naïve knowledge structures, which lead to misconceptions. According to the classical knowledge-as-theory approach (Posner et al. 1982), presenting a cognitive conflict to make the learner abandon the misconception and take the new conception is essential in conceptual change. Posner et al. (1982) proposed that new conceptions should be intelligible, plausible, and fruitful. Intelligible means the new conception is easy to understand. Plausible means the new conception is convincing and appears to be able to solve the problems generated by the old conception. Fruitful means the new conception should show "the potential to be extended, to open up new areas of inquiry" (Posner et al. 1982, p. 214). Believing students' misconceptions are rooted in existing coherent naïve knowledge, researchers of the knowledge-as-theory perspective have indicated that conceptual change is a revolutionary replacement of learners' initial theory-like knowledge structures.

In contrast, the knowledge-as-elements perspective posits that learners' naïve knowledge consists of unstructured collections of quasi-independent elements (diSessa 1993). From this point of view, conceptual change is an evolutionary process of revising and refining the elements and the organization of the elements. DiSessa (1993, 2014) called students' naïve knowledge elements "p-prims" (phenomenological primitives), which refer to intuitive schema students use to explain physical phenomena. The p-prims are generated from learners' superficial interpretations of everyday experience and also used by learners to interpret their experience. The primitiveness of p-prims means that learners' naïve knowledge elements are often self-explanatory with no need for further justification. Unlike the knowledge-as-theory perspective, researchers of knowledge-as-elements believe that learners' existing naïve knowledge elements can be productive in learning, rather than always resulting in misconceptions (diSessa 2013, 2014). While learners can develop near-normative understanding by composing their naïve knowledge elements, studies about the productivity of intuitive knowledge are sparse (diSessa 2014). In addition to the productive aspects of learners' naïve knowledge, the knowledge-as-elements perspective also has proposed that students' misconceptions are contextually sensitive (diSessa 2013, 2014; Özdemir and Clark 2007). For example, in different situations, students may utilize different elements of knowledge to construct meaning because of the contextual sensitivity of fragmented knowledge. Even within similar contexts, students may develop various intermediate mental states during

conceptual change. In summary, rather than coherent theory-like knowledge structures, the knowledge-as-elements perspective posits that learners' naïve knowledge is in small pieces that can be used in building scientific understanding, and enhancing the organization of these knowledge pieces is vital to the success in addressing students' misunderstandings.

While these two perspectives seem conflicting to each other, recent studies have shown some convergence. For example, the framework theory approach (Vosniadou and Skopeliti 2014) agrees that conceptual change is gradual, knowledge elements such as p-prims exist, and misconceptions are not unitary and context independent, but it argues that learners' existing knowledge generates coherent framework theories for interpreting phenomena. According to the framework theory approach, before correctly developing scientific understanding, learners often form synthetic models within their framework theories after instruction (Vosniadou 1994; Vosniadou and Skopeliti 2014). A synthetic model refers to an intermediate state of knowledge with some explanatory power. These days, researchers of the different perspectives share some agreements, including that (1) learners' daily experiences influence knowledge acquisition, (2) learning is affected by learners' naïve knowledge, and (3) conceptual change is difficult and time consuming (Özdemir and Clark 2007).

Although debates among researchers still exist, conceptual change theories provide new avenues to address students' misconceptions. According to the classical theory (Posner et al. 1982), making students dissatisfied with their prior conceptions is important to help them build correct understanding. The framework theory (Vosniadou and Skopeliti 2014) suggests that carefully analyzing students' prior knowledge and planning the sequence of intended concepts are crucial in curricula design. The p-prims theory (diSessa 2014) advocates for considering learners' intuitive knowledge as a productive resource for knowledge construction. Instead of documenting and attacking misconceptions, one growing trend in conceptual change research focuses on describing the knowledge acquisition process and tracking the evolution of learners' understandings (Vosniadou 2013; diSessa 2013, 2014).

While conceptual change theories derive from research on learning of science and mathematics, they may inform the work of computer science educators by assisting in understanding novices' knowledge structures and providing specific recommendations for instruction to address students' misconceptions in learning to program. For instance, mismatching parentheses, brackets, or quotation marks are a very common mistake of novices (Altadmri and Brown 2015) and seem to be an obvious syntactic error. However, according to the knowledge-as-theory perspective, this error may be related to students' inaccurate understanding of program compilation. They may hold some naïve theories or mental models about compilers and may not understand why semicolons are necessary in some programming languages. Variables and assignments are basic programming concepts, but novices may have incorrect mental models of them. Ma (2007) reported that a cognitive conflict strategy was effective in helping students construct viable mental models of variables and assignments. As previous studies have noted that novices often show insufficient strategic knowledge because of their fragile syntactic and conceptual knowledge of programming (Clancy and Linn 1999; Lister et al 2004; Lister et al 2006; Soloway 1986), the knowledge-as-elements perspective may even help to guide the handling of students' difficulties in strategic knowledge. According to this view, novices may possess relevant knowledge elements for solving a certain problem but do not organize them in the way experts do. Thus, solving the challenges of using strategic knowledge may require developing and refining the organization of novices' syntactic and conceptual knowledge. In addition, considering the contextual sensitivity of misconceptions, instructors need to be aware that different instruction and programming languages might lead to different difficulties and misunderstandings for novices. Fisler et al. (2016) noted that when solving

the same problems, students using different programming paradigms (functional vs. imperative) showed distinct difficulties.

5.4 Summary

We discussed various strategies and tools for addressing students' misconceptions and other difficulties. Some strategies or tools are designed for handling specific difficulties. For example, the major purpose of natural-language-like programming languages is to reduce natural language interference in the learning of programming. However, other strategies or tools are aimed at addressing more general student difficulties in learning to program. For instance, reading and tracing program examples can help students develop not only accurate conceptual understanding of programming but also their skills of writing and debugging code. While conceptual change theories may be mainly beneficial for tackling students' misconceptions, they may also help students refine the organization of their syntactic knowledge and develop programming strategies. Table 1 provides a summary of exemplars of difficulties in different types of programming knowledge, their major related factors, and potential strategies and tools for addressing them. Strategies and tools that can be applied to difficulties in various types of programming knowledge are listed with the most relevant area of difficulties.

6 DISCUSSION AND IMPLICATIONS

Students inevitably exhibit various kinds of difficulties in introductory programming. In computing education research, myriad studies have focused on cataloging and tackling misconceptions and other difficulties by using various learning tools and instructional approaches. While effectiveness has been reported, the dissemination of the instructional approaches and tools may need more attention. By reviewing the literature on teaching introductory programming, Pears et al. (2007) reported that while many programming tools were developed, most of them were not used outside their home institutions for various reasons, including difficulties of adaptation, lack of continuous funds, and so forth. Sorva et al. (2013) indicated that learner engagement with new tools was not adequately investigated, and many students did not see the benefits of using such tools. However, successful examples exist. For instance, the BlueJ IDE is widely used by different institutions and its Blackbox data helps researchers and instructors understand and address students' misconceptions in introductory Java programming (Altadmri and Brown 2015; Pears et al. 2007). Therefore, in addition to creating new instructional tools and approaches, developers and researchers may also need to place emphasis on disseminating existing tools and approaches for addressing misconceptions of specific content (e.g., OOP concepts) by specific groups (e.g., college CS1 students or high school AP students) and their long-term effects (Pears et al. 2007; Sorva et al. 2013).

Moreover, computing education research should move beyond simply identifying individual errors and misconceptions and advance toward understanding the process of knowledge acquisition and the evolution of misconceptions by integrating conceptual change theories. In this article, we summarize factors contributing to misconceptions and other difficulties in learning to program and find that a major source of many problems is students' existing knowledge or previous experience. According to conceptual change theories, learners' prior knowledge plays an important role in forming misconceptions. Thus, analyzing students' prior knowledge is crucial to successful instruction. In addition, contexts should be taken into account when studying students' difficulties. For instance, different programming paradigms may lead to different challenges to the same group of students, and English speakers and non-English speakers may show different difficulties. Hence, investigating programming difficulties in a context-dependent fashion is important. In addition, it is vital to value the productivity of students' prior knowledge and

Table 1. Exemplars of Difficulties, Related Factors, and Potential Strategies/Tools for Addressing

Knowledge	Exemplars of Difficulties	Major Related Factors	Potential Strategies and Tools
Syntactic	Syntax errors such as • Mismatched parentheses *while ((a > b) && (a>c) { … }* • Incorrect use of the comparison operator *if (a = true) { … }*	• Task complexity can increase cognitive load and cause novices to forget basic syntax. • Programming environments may not support highlighting errors.	• Advanced editors or graphical programming environments can highlight or prevent syntax errors, reduce cognitive load, and help students with syntactical difficulties.
Conceptual	Misunderstanding of programming constructs or machine operation such as • Variables and assignment statements • Conditional expressions • Loops • Classes and objects • Sequential code execution	• Knowledge of natural language may interfere with learning of English-like statements. • Prior math knowledge may interfere with understanding of computer expressions. • Students' mental models of computer operation may be flawed. • Teachers' instruction and limited knowledge may contribute to students' misconceptions. • Some computing environments create conceptual difficulty by using the same symbol (+) for multiple functions (addition, concatenation).	• Well-chosen program examples can help students build accurate understanding of programming and improve knowledge transfer. • Visualization tools such as Python Tutor can help students to visualize code execution step by step and build correct mental models. • Natural-language-like programming languages such as MOOSE may reduce natural-language errors. • According to conceptual change theories, analyzing students' existing knowledge elements may help instructors better understand students' misconceptions to help them refine and reorganize their knowledge to be more complete and utilitarian. • A concept inventory can help instructors evaluate students' understanding of programming concepts. • Peer instruction can help students improve their understanding of concepts in programming.
Strategic	Difficulties applying programming to solve problems such as • Failing to choose an appropriate loop construct in a specific context • Failing to test and debug programs	• Novices often have fragmentary knowledge with an inadequate set of patterns and strategies for problem solving.	• Asking students to read and trace example programs can help students develop strategic knowledge. • Explicitly teaching debugging strategies and using enhanced debugging tools (e.g., providing detailed error messages) may improve students' debugging skills.

explore the evolution of their conceptions and misconceptions. Between learners' initial knowledge and the to-be-acquired knowledge, there exist many intermediate states, such as synthetic models (Vosniadou and Skopeliti 2014) and the compositions of p-prims (diSessa 2014). The intermediate models, which are developed by the learner using his or her prior knowledge, may not be completely incorrect and sometimes are near normative. Therefore, researchers and educators need to investigate how students form these understandings and track how they change their intermediate models to the scientific ones with instruction or spontaneously. Understanding the knowledge acquisition process can provide more sophisticated theoretical frameworks for designing novel learning tools and instructional approaches. Finally, current conceptual change theories have been mainly developed by researchers in science and mathematics education, and computing education researchers have not contributed much to expanding the theories. Although a few studies have discussed (Miller 2014; Sorva 2012) or integrated (Ginat and Shmallo 2013; Ma 2007) conceptual change theories, overall computing education research has not highlighted the value of conceptual change theories. As one growing trend in conceptual change research focuses on tracking the development of learners' (mis)conceptions using real-time data and microgenetic analysis (diSessa 2014), we computing education researchers have an opportunity. For example, with millions of records of the Blackbox dataset (Altadmri and Brown 2015), we may be able to go beyond identifying the most common errors to discover details about students' conceptual change paths. Even though the current tools may not have been designed for this function, when creating new tools, we definitely should take this into account.

Last but not least, it is important to develop and enhance instructors' PCK for handling students' misconceptions. Instructors often show a weak understanding about their students' misconceptions (Brown and Altadmri 2014) or fail to see novices' challenges (Guzdial 2015). Further, educators often view students' misconceptions as obstacles to learning, rather than resources for teaching and learning (Larkin 2012; Maskiewicz and Lineback 2013; Smith et al. 1994). From a knowledge-as-elements perspective, a successful teacher needs to learn to help learners refine and reorganize their existing knowledge to develop more complete and utilitarian conceptions, rather than trying to replace students' ideas with new ones (diSessa 2013; Smith et al. 1994). Hence, awareness of students' misconceptions and the ability to apply effective teaching strategies and tools to address misconceptions should be considered as an important part of a teacher's PCK and another measure of a teacher's competence (Sadler et al. 2013). All instructors should improve their own PCK for teaching introductory programming.

7 CONCLUSIONS

Although different names are used to represent students' difficulties in learning to program, there is general agreement that there is value in helping students to develop their conceptual understandings and overcome challenges. In introductory programming courses, students exhibit various misconceptions and other difficulties in syntactic knowledge, conceptual knowledge, and strategic knowledge. Previous studies reveal that the difficulties experienced by students are related to many factors, such as unfamiliarity of syntax, natural language, math knowledge, inaccurate mental models, lack of strategies, programming environments, inappropriate instruction, and teachers' knowledge. However, many sources of students' difficulties have connections with students' prior knowledge. To better understand and address students' misconceptions and other difficulties, various instructional approaches and tools have been developed. Nevertheless, the dissemination of these approaches and tools has been limited. Thus, we suggest enhancing the dissemination of existing tools and approaches and investigating their long-term effects. Second, we recommend that computing education research move beyond documenting misconceptions to address the development of students' (mis)conceptions by integrating conceptual change theories. Third, we believe

that developing and enhancing instructors' PCK, including their knowledge of students' misconceptions and ability to apply effective instructional approaches and tools to address difficulties, is vital to the success of teaching introductory programming.

Currently, the United States has a growing demand for computing professionals to drive innovation and economic progress. Efforts to improve computer science education are underway, and teachers of computer science are challenged in introductory programming courses to help learners develop their understanding of programming and computer science. Identifying and addressing students' misconceptions is a key part of a computer science teacher's competence. However, relevant research on this topic is not as fully developed in the computer science education field as it is in mathematics and science education. If we are to expand and improve computer science education, more research on students' misconceptions, especially on the evolution of (mis)conceptions, is needed and appropriate instructional strategies and tools to address them must be developed and disseminated.

ACKNOWLEDGMENTS

The authors also would like to thank the reviewers of TOCE who contributed feedback on this journal article.

REFERENCES

Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice computer science students. *SIGCSE Bulletin* 37, 3 (June 2005), 84–88. DOI: http://dx.doi.org/10.1145/1151954.1067472

Amjad Altadmri and Neil C. C. Brown. 2015. 37 Million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE'15)*. ACM, New York, 522–527. DOI: http://dx.doi.org/10.1145/2676723.2677258

John R. Anderson and Robin Jeffries. 1985. Novice LISP errors: Undetected losses of information from working memory. *Human–Computer Interaction* 1, 2 (1985), 107–131. DOI: http://dx.doi.org/10.1207/s15327051hci0102_2

Tiffany Barnes and John Stamper. 2008. Toward automatic hint generation for logic proof tutoring using historical student data. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems*, 373–382.

Piraye Bayman and Richard E. Mayer. 1983. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM* 26, 9 (September 1983), 677–679. DOI: http://dx.doi.org/10.1145/358172.358408

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

Brett A. Becker. 2016. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE'16)*. ACM, New York, 126–131. DOI: http://dx.doi.org/10.1145/2839509.2844584

Yifat Ben-David Kolikant and M. Mussai. 2008. "So my program doesn't run!" Definition, origins, and practical expressions of students' (mis) conceptions of correctness. *Computer Science Education* 18, 2 (2008), 135–151. DOI: http://dx.doi.org/10.1080/08993400802156400

Jeffrey Bonar and Elliot Soloway. 1985. Preprogramming knowledge: a major source of misconceptions in novice programmers. *Human-Computer Interaction* 1, 2 (1985), 133–161. DOI: http://dx.doi.org/ 10.1207/s15327051hci0102_3

Paola Britos, Elizabeth J. Rey, Dario Rodriguez, and Ramon Garcia-Martinez. 2008. Work in progress-programming misunderstandings discovering process based on intelligent data mining tools. In *Proceedings of the 38th Annual Frontiers in Education Conference*. IEEE, F4H-1–F4H-2. DOI: http://dx.doi.org/10.1109/FIE.2008.4720499

Neil C. C. Brown and Amjad Altadmri. 2014. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the 10th Annual Conference on International Computing Education Research (ICER'14)*. ACM, New York, 43–50. DOI: http://dx.doi.org/10.1145/2632320.2632343

Neil C. C. Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE'14)*. ACM, New York, 223–228. DOI: http://dx.doi.org/10.1145/2538862.2538924

Amy Bruckman and Elizabeth Edwards. 1999. Should we leverage natural-language knowledge? An analysis of user errors in a natural-language-style programming language. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'99)*. ACM, New York, 207–214. DOI: http://dx.doi.org/10.1145/302979.303040

Bureau of Labor Statistics, US Department of Labor. 2015. Occupational Outlook Handbook, 2014-15 edition, Computer and Information Research Scientists. Retrieved February 10, 2016, from http://www.bls.gov/ooh/computer-and-information-technology/computer-and-information-research-scientists.htm.

Matthew Butler and Michael Morgan. 2007. Learning challenges faced by novice programming students studying high level and low feedback concepts. In *Proceedings of Ascilite Singapore 2007*. ascilite, Tugun, QLD, Australia, 99–107.

José J. Canas, Maria T. Bajo, and Pilar Gonzalvo. 1994. Mental models and computer programming. *International Journal of Human-Computer Studies* 40 (1994), 795–811. DOI: http://dx.doi.org/10.1006/ijhc.1994.1038

Michael Clancy. 2004. Misconceptions and attitudes that interfere with learning to program. In *Computer Science Education Research*. London: Taylor & Francis Group, 85–100.

Michael J. Clancy and Marcia C. Linn. 1999. Patterns and pedagogy. In *The Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'99)*. ACM, New York, 37–42. DOI: http://dx.doi.org/10.1145/299649.299673

John Clement. 1993. Using bridging analogies and anchoring intuitions to deal with students' preconceptions in physics. *Journal of Research in Science Teaching* 30, 10 (1993), 1241–1257. DOI: http://dx.doi.org/10.1002/tea.3660301007

Malcolm Corney, Donna Teague, Alireza Ahadi, and Raymond Lister. 2012. Some empirical results for neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. In *Proceedings of the 14th Australasian Computing Education Conference - Volume* 123 *(ACE'12)*, Michael de Raadt and Angela Carbone (Eds.), Vol. 123. Australian Computer Society, Darlinghurst, Australia, 77–86.

Catherine H. Crouch and Eric Mazur. 2001. Peer instruction: Ten years of experience and results. *American Journal of Physics* 69, 9 (2001), 970–977.

Holger Danielsiek, Wolfgang Paul, and Jan Vahrenhold. 2012. Detecting and understanding students' misconceptions related to algorithms and data structures. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE'12)*. ACM, New York, 21–26. DOI: http://dx.doi.org/10.1145/2157136.2157148

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

Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. 2014. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE'14)*. ACM, New York, 273–278. DOI: http://dx.doi.org/10.1145/2591708.2591748

Michael de Raadt. 2008. *Teaching Programming Strategies Explicitly to Novice Programmers*. PhD Dissertation. University of Southern Queensland, Australia.

Luis De-La-Fuente-Valentín, Abelardo Pardo, and Carlos D. Kloos. 2013. Addressing drop-out and sustained effort issues with large practical groups using an automated delivery and assessment system. *Computers & Education* 61 (2013), 33–42. DOI: http://dx.doi.org/10.1016/j.compedu.2012.09.004

Andrea A. diSessa. 1993. Toward an epistemology of physics. *Cognition and Instruction* 10, 2–3, (1993), 105–225.

Andrea A. diSessa. 2013. A bird's-eye view of the "pieces" vs "coherence" controversy (from the "pieces" side of the fence). In *International Handbook of Research on Conceptual Change*. Taylor and Francis, New York, 31–48.

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

Anna K. Dominguez, Kalina Yacef, and James R. Curran (June 2010). Data mining for individualised hints in eLearning. In *Proceedings of the 3rd International Conference on Educational Data Mining*, 91–100.

Christopher Douce, David Livingstone, and James Orwell. 2005. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing* 5, 3, Article 4 (September 2005). DOI: http://dx.doi.org/10.1145/1163405.1163409

Dimitrios Doukakis, Maria Grigoriadou, and Grammatiki Tsaganou. 2007. Understanding the programming variable concept with animated interactive analogies. In *Proceedings of the 8th Hellenic European Research on Computer Mathematics & Its Applications Conference (HERCMA'07)*.

Benedict du Boulay. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73. DOI: http://dx.doi.org/10.2190/3LFX-9RRF-67T8-UVK9

M. Ducassé and A.-M. Emde. 1988. A review of automated debugging systems: knowledge, strategies and techniques. In *Proceedings of the 10th International Conference on Software Engineering (ICSE'88)*. IEEE Computer Society Press, Los Alamitos, CA, 162–171.

Alireza Ebrahimi. 1994. Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies* 41, 4 (1994), 457–480. DOI: http://dx.doi.org/10.1006/ijhc.1994.1069

Ruhama Even. 1993. Subject-matter knowledge and pedagogical content knowledge: Prospective secondary teachers and the function concept. *Journal for Research in Mathematics Education* 24, 2 (1993), 94–116. DOI: http://dx.doi.org/10.2307/749215

Kathi Fisler. 2014. The recurring rainfall problem. In *Proceedings of the 10th Annual Conference on International Computing Education Research (ICER'14)*. ACM, New York, 35–42. DOI: http://dx.doi.org/10.1145/2632320.2632346

Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing plan-composition studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE'16)*. ACM, New York, 211–216. DOI: http://dx.doi.org/10.1145/2839509.2844556

Sue Fitzgerald, Gary Lewandowski, Renee McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education* 18, 2 (2008), 93–116. DOI: http://dx.doi.org/10.1080/08993400802114508

Ann E. Fleury. 1991. Parameter passing: The rules the students construct. In *Proceedings of the 22nd SIGCSE Technical Symposium on Computer Science Education (SIGCSE'91)*. ACM, New York, 283–286. DOI: http://dx.doi.org/10.1145/107004.107066

David Ginat, Eti Menashe, and Amal Taya. 2013. Novice difficulties with interleaved pattern composition. In *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*. Springer, Berlin, 57–67.

David Ginat and Ronit Shmallo. 2013. Constructive use of errors in teaching CS1. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13)*. ACM, New York, 353–358. DOI: http://dx.doi.org/10.1145/2445196.2445300

David Ginat, Eyal Shifroni, and Eti Menashe. 2011. Transfer, cognitive load, and program design difficulties. In *Proceedings of 5th International Conference on Informatics in Schools: Situation, Evolution and Perspectives (ISSEP'11)*, 165–176.

Sandy Garner, Patricia Haden, and Anthony Robins. 2005. My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems. In *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42 (ACE'05)*, Alison Young and Denise Tolhurst (Eds.). Australian Computer Society, Darlinghurst, Australia, 173–180.

Alex Gerdes, Johan T. Jeuring, and Bastiaan J. Heeren. 2010. Using strategies for assessment of programming exercises. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE'10)*. ACM, New York, 441–445. DOI: http://dx.doi.org/10.1145/1734263.1734412

Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey L. Herman, Lisa Kaczmarczyk, and Craig Zilles. 2010. Setting the scope of concept inventories for introductory computing subjects. *Transactions on Computing Education* 10, 2, Article 5 (June 2010), 29 pages. DOI: http://dx.doi.org/10.1145/1789934.1789935

T. R. G. Green. 1977. Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology* 50, 2 (1977), 93–109. DOI: http://dx.doi.org/10.1111/j.2044-8325.1977.tb00363.x

Philip J. Guo. 2013. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13)*. ACM, New York, 579–584. DOI: http://dx.doi.org/10.1145/2445196.2445368

Mark Guzdial. 1995. Centralized mindset: A student problem with object-oriented programming. In *Proceedings of the 26th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'95)*, Curt M. White, James E. Miller, and Judy Gersting (Eds.). ACM, New York, 182–185. DOI: http://dx.doi.org/10.1145/199688.199772

Mark Guzdial. 2015. Learner-centered design of computing education: Research on computing for everyone. In *Synthesis Lectures on Human-Centered Informatics*, 1–165.

Bruria Haberman and Yifat Ben-David Kolikant. 2001. Activating "black boxes" instead of opening "zipper" - A method of teaching novices basic CS concepts. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'01)*. ACM, New York, 41–44. DOI: http://dx.doi.org/10.1145/377435.377464

Frank Halasz and Thomas P. Moran. 1982. Analogy considered harmful. In *Proceedings of the 1982 Conference on Human Factors in Computing Systems (CHI'82)*. ACM, New York, 383–386. DOI: http://dx.doi.org/10.1145/800049.801816

Simon Holland, Robert Griffiths, and Mark Woodman. 1997. Avoiding object misconceptions. In *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'97)*, James E. Miller (Ed.). ACM, New York, 131—134. DOI: http://dx.doi.org/10.1145/268084.268132

Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'03)*. ACM, New York, 153–156. DOI: http://dx.doi.org/10.1145/611892.611956

James Jackson, Michael Cobb, and Curtis Carver. 2005. Identifying top Java errors for novice programmers. In *Proceedings of the 35th Annual Frontiers in Education Conference (FIE'05)*, T4C-T4C.

David H. Jonassen. 1991. Objectivism versus constructivism: Do we need a new philosophical paradigm? *Educational Technology Research & Development* 39, 3 (1991), 5–14.

Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. 2010. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE'10)*. ACM, New York, 107–111. DOI: http://dx.doi.org/10.1145/1734263.1734299

Irvin R. Katz and John R. Anderson. 1987. Debugging: An analysis of bug-location strategies. *Human.-Computer Interactions* 3, 4 (December 1987), 351–399. DOI: http://dx.doi.org/10.1207/s15327051hci0304_2

Caitlin Kelleher and Randy Pausch. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys* 37, 2 (June 2005), 83–137. DOI: http://dx.doi.org/10.1145/1089733.1089734

Leopold E. Klopfer, Audrey B. Champagne, and Richard F. Gunstone. 1983. Naive knowledge and science learning. *Research in Science & Technological Education* 1, 2 (1983), 173–183. DOI: http://dx.doi.org/10.1080/0263514830010205

Andrew J. Ko and Brad A. Myers. 2005. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing* 16, 1 (2005), 41–84. DOI: http://dx.doi.org/10.1016/j.jvlc.2004.08.003

Michael Kölling. 2010. The Greenfoot programming environment. *Transactions on Computing Education* 10, 4, Article 14 (November 2010), 21 pages. DOI: http://dx.doi.org/10.1145/1868358.1868361

Wanda M. Kunkle and Robert B. Allen. 2016. The impact of different teaching approaches and languages on student learning of introductory programming concepts. *Transactions on Computing Education* 16, 1, Article 3 (January 2016), 26 pages. DOI: http://dx.doi.org/10.1145/2785807

Douglas Larkin. 2012. Misconceptions about "misconceptions": Preservice secondary science teachers' views on the value and role of student ideas. *Science Education* 96, 5 (2012), 927–959. DOI: http://dx.doi.org/10.1002/sce.21022

Marcia Linn. 1985. The cognitive consequences of programming instruction in classrooms. *Educational Researcher* 14, 5 (1985), 14–29.

Raymond Lister. 2011. Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In *Proceedings of the 13th Australasian Computing Education Conference - Volume 114 (ACE'11)*, John Hamer and Michael de Raadt (Eds.). Australian Computer Society, Darlinghurst, Australia, 9–18.

Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A multi-national study of reading and tracing skills in novice programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR'04)*. ACM, New York, 119–150. DOI: http://dx.doi.org/10.1145/1044550.1041673

Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'09)*. ACM, New York, 161–165. DOI: http://dx.doi.org/10.1145/1562877.1562930

Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 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 (ITICSE'06)*. ACM, New York, 118–122. DOI: http://dx.doi.org/10.1145/1140124.1140157

Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the 4th International Workshop on Computing Education Research (ICER'08)*. ACM, New York, 101–112. DOI: http://dx.doi.org/10.1145/1404520.1404531

Linxiao Ma. 2007. *Investigating and Improving Novice Programmers' Mental Models of Programming Concepts*. PhD Dissertation. University of Strathclyde, UK.

April C. Maskiewicz and Jennifer E. Lineback. 2013. Misconceptions are "so yesterday!". *CBE-Life Sciences Education* 12, 3 (2013), 352–356. DOI: http://dx.doi.org/10.1187/cbe.13-01-0014

Richard E. Mayer. 1981. The psychology of how novices learn computer programming. *ACM Computing Surveys* 13, 1 (March 1981), 121–141. DOI: http://dx.doi.org/10.1145/356835.356841

Richard E. Mayer and Anne L. Fay. 1987. A chain of cognitive changes with learning to program in Logo. *Journal of Educational Psychology* 79, 3 (1987), 269–279.

Davin McCall and Michael Kölling. 2014. Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference Proceedings (FIE'14)*. IEEE, 1–8.

Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: A review of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92. DOI: http://dx.doi.org/10.1080/08993400802114581

Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin* 33, 4 (December 2001), 125–180. DOI: http://dx.doi.org/10.1145/572139.572181

Tanya J. McGill and Simone E. Volet. 1997. A conceptual framework for analyzing students' knowledge of programming. *Journal of Research on Computing in Education* 29, 3 (1997), 276–297. DOI: http://dx.doi.org/10.1080/08886504.1997.10782199

Craig S. Miller. 2014. Metonymy and reference-point errors in novice programming. *Computer Science Education* 24, 2–3 (2014), 123–152.

Orna Muller. 2005. Pattern oriented instruction and the enhancement of analogical reasoning. In *Proceedings of the 1st International Workshop on Computing Education Research (ICER'05)*. ACM, New York, 57–67. DOI: http://dx.doi.org/10.1145/1089786.1089792

Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-oriented instruction and its influence on problem decomposition and solution construction. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'07)*. ACM, New York, 151–155. DOI: http://dx.doi.org/10.1145/1268784.1268830

Brad A. Myers, John F. Pane, and Andy Ko. 2004. Natural programming languages and environments. *Communications of the ACM* 47, 9 (September 2004), 47–52. DOI: http://dx.doi.org/10.1145/1015864.1015888

Gökhan Özdemir and Douglas Clark. 2007. An overview of conceptual change theories. *Eurasia Journal of Mathematics, Science & Technology Education* 3, 4 (2007), 351–361.

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

Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A survey of literature on the teaching of introductory programming. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR'07)*, Janet Carter and June Amillo (Eds.). ACM, New York, 204–223. DOI: http://dx.doi.org/10.1145/1345443.1345441

D. N. Perkins and Fay Martin. 1986. Fragile knowledge and neglected strategies in novice programmers. In *Empirical Studies of Programmers: First Workshop*, 213–229.

Leo Porter, Cynthia Bailey Lee, and Beth Simon. 2013. Halving fail rates using peer instruction: A study of four computer science courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13)*. ACM, New York, 177–182. DOI: http://dx.doi.org/10.1145/2445196.2445250

George J. Posner, Kenneth A. Strike, Peter W. Hewson, and William A. Gertzog. 1982. Accommodation of a scientific conception: Toward a theory of conceptual change. *Science Education* 66, 2 (1982), 211–227. DOI: http://dx.doi.org/10.1002/sce.3730660207

Yizhou Qian and James D. Lehman. 2016. Correlates of success in introductory programming: A study with middle school students. *Journal of Education and Learning* 5, 2 (2016), 73–83.

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

Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for all. *Communications of the ACM* 52, 11 (November 2009), 60–67. DOI: http://dx.doi.org/10.1145/1592761.1592779

Robert S. Rist. 1989. Schema creation in programming. *Cognitive Science* 13, 3 (1989), 389–414.

Anthony Robins, Patricia Haden, and Sandy Garner. 2006. Problem distributions in a CS1 course. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52 (ACE'06)*, Denise Tolhurst and Samuel Mann (Eds.). Australian Computer Society, Darlinghurst, Australia, 165–173.

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

Kelly Rivers and Kenneth R. Koedinger. 2017. Data-driven hint generation in vast solution spaces: A self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 37–64. DOI: https://doi.org/10.1007/s40593-015-0070-z

Philip M. Sadler, Gerhard Sonnert, Harold P. Coyle, Nancy Cook-Smith, and Jaimie L. Miller. 2013. The influence of teachers' knowledge on student learning in middle school physical science classrooms. *American Educational Research Journal* 50, 5 (2013), 1020–1049. DOI: http://dx.doi.org/10.3102/0002831213477680

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

Jorma Sajaniemi and Raquel N. Prieto. 2005. Roles of variables in experts' programming knowledge. In *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG'05)*, 145–159.

Jorma Sajaniemi, Marja Kuittinen, and Taina Tikansalo. 2008. A study of the development of students' visualizations of program state during an elementary object-oriented programming course. *Journal on Educational Resources in Computing* 7, 4, Article 3 (January 2008), 31 pages. DOI: http://dx.doi.org/10.1145/1316450.1316453

Michael J. Sanger and Thomas J. Greenbowe. 1997. Common student misconceptions in electrochemistry: Galvanic, electrolytic, and concentration cells. *Journal of Research in Science Teaching* 34, 4 (1997), 377–398.

Kate Sanders and Lynda Thomas. 2007. Checklists for grading object-oriented CS1 programs: Concepts and misconceptions. *SIGCSE Bulletin* 39, 3 (June 2007), 166–170. DOI: http://dx.doi.org/10.1145/1269900.1268834

Simon. 2011. Assignment and sequence: Why some students can't recognise a simple swap. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research (Koli Calling'11)*. ACM, New York, 10–15. DOI: http://dx.doi.org/10.1145/2094131.2094134

Simon. 2013. Soloway's rainfall problem has become harder. In *Proceedings of the 2013 Learning and Teaching in Computing and Engineering (LATICE'13)*. IEEE Computer Society, 130–135. DOI: http://dx.doi.org/10.1109/LaTiCE.2013.44

Beth Simon, Michael Kohanfars, Jeff Lee, Karen Tamayo, and Quintin Cutts. 2010. Experience report: Peer instruction in introductory computing. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE'10)*. ACM, New York, 341–345. DOI: http://dx.doi.org/10.1145/1734263.1734381

Teemu Sirkia. 2012. *Recognizing Programming Misconceptions: An Analysis of the Data Collected from the UUhistle Program Simulation Tool*. Master's thesis, Aalto University, Espoo, Finland.

Teemu Sirkia and Juha Sorva. 2012. Exploring programming misconceptions: An analysis of student mistakes in visual program simulation exercises. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research (Koli Calling'12)*. ACM, New York, 19–28. DOI: http://dx.doi.org/10.1145/2401796.2401799

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

John P. Smith, Andrea A. diSessa, and Jeremy Roschelle. 1994. Misconceptions reconceived: A constructivist analysis of knowledge in transition. *Journal of the Learning Sciences* 3, 2 (1994), 115–163. DOI: http://dx.doi.org/10.1207/s15327809jls0302_1

Michelle K. Smith, William B. Wood, Wendy K. Adams, Carl Wieman, Jennifer K. Knight, Nancy Guild, and Tin Tin Su. 2009. Why peer discussion improves student performance on in-class concept questions. *Science* 323, 5910 (2009), 122–124.

Elliot Soloway. 1986. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM* 29, 9 (September 1986), 850–858. DOI: http://dx.doi.org/10.1145/6592.6594

Juha Sorva. 2012. *Visual Program Simulation in Introductory Programming Education*. PhD Dissertation. Aalto University, Espoo, Finland.

Juha Sorva. 2013. Notional machines and introductory programming education. *Transactions on Computer Education* 13, 2, Article 8 (July 2013), 31 pages. DOI: http://dx.doi.org/10.1145/2483710.2483713

Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A review of generic program visualization systems for introductory programming education. *Transactions on Computer Education* 13, 4, Article 15 (November 2013), 64 pages. DOI: http://dx.doi.org/10.1145/2490822

James C. Spohrer and Elliot Soloway. 1986. Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM* 29, 7 (July 1986), 624–632. DOI: http://dx.doi.org/10.1145/6138.6145

Keith S. Taber. 2014. Alternative conceptions/frameworks/misconceptions. In *Encyclopedia of Science Education Vol. A*. Springer, New York, 37–41. DOI: http://dx.doi.org/10.1007/978-94-007-6165-0_88-2

Cynthia Taylor, Daniel Zingaro, Leo Porter, Kevin C. Webb, Cynthia Bailey Lee, and M. Clancy. 2014. Computer science concept inventories: Past and future. *Computer Science Education* 24, 4 (2014), 253–276.

Donna Teague, Malcolm Corney, Alireza Ahadi, and Raymond Lister. 2013. A qualitative think aloud study of the early neo-piagetian stages of reasoning in novice programmers. In *Proceedings of the 15th Australasian Computing Education Conference - Volume 136 (ACE'13)*, Angela Carbone and Jacqueline Whalley (Eds.), Vol. 136. Australian Computer Society, Darlinghurst, Australia, 87–95.

Donna Teague and Raymond Lister. 2014. Programming: Reading, writing and reversing. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE'14)*. ACM, New York, 285–290. DOI: http://dx.doi.org/10.1145/2591708.2591712

Mariana Teif and Orit Hazzan. 2006. Partonomy and taxonomy in object-oriented thinking: Junior high school students' perceptions of object-oriented basic concepts. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR'06)*. ACM, New York, 55–60. DOI: http://dx.doi.org/10.1145/1189215.1189170

Allison E. Tew. 2010. *Assessing Fundamental Introductory Computing Concept Knowledge in a Language Independent Manner*. PhD Dissertation. Georgia Institute of Technology, Atlanta, GA.

Miki K. Tomita. 2008. *Examining the Influence of Formative Assessment on Conceptual Accumulation and Conceptual Change*. PhD Dissertation. Stanford University, Palo Alto, CA. UMI Number: 3343949.

Juhani E. Tuovinen and John Sweller. 1999. A comparison of cognitive load associated with discovery learning and worked examples. *Journal of Educational Psychology* 91, 2 (1999), 334–341.

Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. 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 - Volume 52 (ACE'06)*, Denise Tolhurst and Samuel Mann (Eds.). Australian Computer Society, Darlinghurst, Australia, 243–252.

Vesa Vainio and Jorma Sajaniemi. 2007. Factors in novice programmers' poor tracing skills. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'07)*. ACM, New York, 236–240. DOI: http://dx.doi.org/10.1145/1268784.1268853

Anne Venables, Grace Tan, and Raymond Lister. 2009. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the 5th International Workshop on Computing Education Research Workshop (ICER'09)*. ACM, New York, 117–128. DOI: http://dx.doi.org/10.1145/1584322.1584336

Stella Vosniadou, 1994. Capturing and modeling the process of conceptual change. *Learning and Instruction* 4, 1, (1994), 45–69.

Stella Vosniadou, 2013. Conceptual change in learning and instruction: The framework theory approach. In *International Handbook of Research on Conceptual Change*. Taylor and Francis, New York, 11–30.

Stella Vosniadou and Irini Skopeliti. 2014. Conceptual change from the framework theory side of the fence. *Science & Education* 23, 7, (2014), 1427–1445.

David Weintrop and Uri Wilensky. 2015. Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In *Proceedings of the 11th Annual International Conference on International Computing Education Research*, 101–110.

Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE'14)*. ACM, New York, Article 38, 10. DOI: http://dx.doi.org/10.1145/2601248.2601268

Claes Wohlin, Per Runeson, Paulo Anselmo da Mota Silveira Neto, Emelie Engström, Ivan do Carmo Machado, and Eduardo Santana De Almeida. 2013. On the reliability of mapping studies in software engineering. *Journal of Systems and Software* 86, 10, (2013), 2594–2610. DOI: http://dx.doi.org/10.1016/j.jss.2013.04.076

Songwen Xu and Yam San Chee. 2003. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering* 29, 4 (2003), 360–384. DOI: http://dx.doi.org/10.1109/TSE.2003.1191799

Received May 2016; revised March 2017; accepted March 2017