MODESTUM
OPEN ACCESS

EURASIA Journal of Mathematics, Science and Technology Education, 2018, 14(12), em1610
ISSN:1305-8223 (online)
Research Paper https://doi.org/10.29333/ejmste/94227

Using Technology to Support Teaching Computer Science: A Study with Middle School Students

Yizhou Qian [1*], James Lehman [1]
[1] College of Education, Purdue University, West Lafayette, USA

Received 13 February 2018 ▪ Revised 30 June 2018 ▪ Accepted 16 July 2018

ABSTRACT
Expansion of computer science education in K-12 schools is driving the need for quality computer science teachers. Effective computer science teachers need both knowledge of computer science and pedagogical content knowledge (PCK), which includes an understanding of student misconceptions. In this study, by integrating an automated assessment system, we identified common misconceptions of Chinese middle school students in an introductory programming course. We found that students' limited English ability and existing math knowledge contributed to their misconceptions in learning to program. We also noted that Chinese students with better English ability made fewer programming mistakes. This finding differs from previous studies on English speakers that found that students' English ability had negative impacts on the learning of programming commands. Our results suggest that computer science teachers should integrate appropriate technology into instruction to support identifying and addressing specific student misconceptions. We recommend that teacher training programs in computer science pay attention to developing teachers' technological pedagogical content knowledge (TPACK), the knowledge for effective teaching with technology.

Keywords: computer science education, misconceptions, pedagogical content knowledge (PCK), technological pedagogical content knowledge (TPACK)

INTRODUCTION

With technological developments over the past few decades, computing technology has become ubiquitous in today's world. Computer science, as an academic discipline, is receiving increasing attention of education researchers, policymakers, and practitioners (Webb et al., 2017). Computer science education for K-12 students has been expanding in many countries, such as the U.S. (Guzdial, 2016), the U.K. (Brown, Sentance, Crick, & Humphreys, 2014), New Zealand (Bell, Andreae, & Robins, 2014), and others. While such expansion aims to make computer science available to more students, a shortage of effective computer science teachers exists in many countries (Gal-ezer & Stephenson, 2014; Webb et al., 2017). Teacher educators are challenged to effectively prepare teachers of computer science to meet the growing demand.

An effective computer science teacher needs both knowledge of content and knowledge of how to teach (Hubwieser, Magenheim, Mühling, & Ruf, 2013; Shulman, 1986; Yadav, Berges, Sands, & Good, 2016). Content knowledge of a teacher is the knowledge about the subject matter. An effective computer science teacher needs to know fundamental theories of computer science, programming concepts and languages, common algorithms and data structures, and so forth. Teachers also possess pedagogical knowledge, a kind of content knowledge related to the aspects of the content most relevant to its teaching (Shulman, 1986). However, simply knowing content and pedagogy does not make a qualified teacher (Carlsen, 1999; Shulman, 1986). An effective teacher also must have pedagogical content knowledge (PCK), which blends both content and pedagogy and aims to make instructional content comprehensible to students (Shulman, 1987). PCK enables teachers to transform subject matter into a form that students can understand and is considered a crucial part of teachers' professional competence (Hill, Ball, & Schilling, 2008; Hubwieser et al., 2013; Sadler, Sonnert, Coyle, Cook-Smith, & Miller, 2013).

✉ qian47@purdue.edu **(*Correspondence)** ✉ lehman@purdue.edu

Contribution of this paper to the literature

- We used an automated assessment system and its data to identify student misconceptions in introductory programming. Prior studies on such systems paid little attention to identifying student misconceptions, yet knowledge of misconceptions is important for teachers' pedagogical content knowledge.
- Our subjects were Chinese middle school students, while previous studies mainly focused on college English speakers.
- Our results indicated that for Chinese students, many programming misconceptions are related to their limited English ability, and better English ability may reduce misconceptions. In contrast, previous research on English speakers found that English is a factor that can interfere with the learning of programming.

Despite the importance of PCK, we have limited understanding of computer science teachers' PCK, i.e. CS PCK (Saeli, Perrenet, Jochems, & Zwaneveld, 2011; Yadav et al., 2016). Nevertheless, researchers agree that teachers' understanding of student misconceptions is a core component of PCK (Carlsen, 1999; Sadler et al., 2013; Saeli et al., 2011; Shulman, 1986). When Shulman (1986) proposed the notion of PCK, he pointed out that teachers should have "an understanding of what makes the learning of specific topics easy or difficult: the conceptions and preconceptions that students of different ages and backgrounds bring with them to the learning of those most frequently taught topics and lessons" (p. 9). In other words, an effective teacher needs to know what misconceptions students encounter and be able to tailor the instruction to students' misconceptions and existing knowledge.

Unfortunately, computer science teachers often have limited or incorrect understanding of their students' misconceptions (Brown & Altadmri, 2017; Guzdial, 2015). More importantly, their teaching experience may be ineffective in enhancing such understanding (Brown & Altadmri, 2017). Therefore, it is important to examine potential ways to develop computer science teachers' understanding of student misconceptions. This paper reports on an exploratory study that investigated common misconceptions that students encountered in a middle-school level programming class by integrating a technology tool into instruction and examined factors that might contribute to those misconceptions. This paper also discusses the potential of technology for enhancing teachers' understanding of student misconceptions in computer science.

STUDENT MISCONCEPTIONS IN INTRODUCTORY PROGRAMMING

In introductory-level computer science courses, computer programming is usually the core content (Guzdial, 2015). Learning to program is challenging, and students often exhibit a variety of misconceptions (Brown & Altadmri, 2017; Qian & Lehman, 2017; Sorva, 2013), which are students' inadequate or inaccurate understandings of academic concepts (Smith, diSessa, & Roschelle, 1994; Taber, 2013). In the literature regarding student misconceptions in introductory programming, researchers have used various terms to describe student misconceptions, such as bugs, errors, difficulties, and mistakes (Qian & Lehman, 2017). We use the term *misconception* in this paper as it is broadly inclusive and widely used by researchers (Qian & Lehman, 2017; Sorva, 2013).

Many studies have identified student misconceptions in introductory programming. For example, some researchers have focused on cataloguing common compilation errors in student programs (Becker, 2016; Brown & Altadmri, 2017). A compilation error is an error in code that makes a program fail to be compiled prior to program execution. By analyzing millions of errors in students' programs, Altadmri and Brown (2015) reported that the most frequent student error in Java programming was unbalanced parentheses, brackets, or quotation marks. In Java syntax, parentheses, brackets, and quotations marks are used in pairs, and students often omit one member of the pair or the other. Previous studies have also discussed student misconceptions not producing compilation errors (Qian & Lehman, 2017). Such misconceptions include misunderstandings of semantics and inappropriate use or lack of programming strategies (Clancy & Linn, 1999; Kolikant & Mussai, 2008). For instance, students often have difficulties in understanding complicated programming concepts such as classes and objects (Ragonis & Ben-Ari, 2005). Furthermore, novices often lack strategies for planning, composing, and debugging programs (Clancy & Linn, 1999; Lister, Simon, Thompson, Whalley, & Prasad, 2006) such as forgetting to consider boundary conditions and failing to locate errors in their programs (Fisler, Krishnamurthi, & Siegmund, 2016; Fitzgerald et al., 2008; McCauley et al., 2008). Some beginners even believe that a successfully compiled program means a correct solution even though it produces unexpected results (Kolikant & Mussai, 2008). While existing studies have captured common student misconceptions in introductory programming, most of these studies have focused on college students. Few studies have explored the programming misconceptions of middle school students.

According to the constructivist perspective, students' prior knowledge significantly influences the construction of new knowledge and is a dominant source of misconceptions (Jonassen, 1991; Qian & Lehman, 2017; Smith et al., 1994). Students' existing math knowledge is one factor that has been shown to contribute to misconceptions in introductory programming (Clancy, 2004; Qian & Lehman, 2017). For example, students may fail to distinguish variables and variable operations in algebra and those in computer programming which are quite different (Clancy, 2004). On the other hand, students' math ability is also considered as a key predictor of success in learning to program (Bennedsen & Caspersen, 2005). In other words, students' prior math knowledge may have both positive and negative impacts on their learning of programming. Another factor that can interfere with the learning of programming constructs is students' natural language (Bonar & Soloway, 1985; Bruckman & Edwards, 1999; Miller, 2014). Students often inappropriately link a word's meaning in English with the same word in the programming language (Bonar & Soloway, 1985; Miller, 2014). As most programming languages are based on English, students' English ability may have different impacts on misconceptions of English and non-English speakers. While prior research has investigated factors contributing to student misconceptions in learning to program, most of them have focused on English speakers. Limited research has examined the potential impact of non-English speakers' English ability on their misconceptions in computer programming.

Moreover, student misconceptions in computer science are usually latent and difficult to detect (Sorva, 2013; Yadav et al., 2016). Computer science teachers often fail to accurately gauge novices' difficulties because of their "expert blind spot" (Guzdial, 2015). Basically, experienced programmers often forget what it was like when they first began programming and so fail to see the mistakes novices make. Thus, exploring the potential of technology for improving teachers' understanding of student misconceptions can be important. In computer science education, one widely used instructional tool is the automated assessment system (De-La-Fuente-Valentín, Pardo, & Delgado Kloos, 2013; Douce, Livingstone, & Orwell, 2005). An automated assessment system is a tool that can automatically evaluate the correctness of students' programs by applying certain assessment techniques and algorithms. Such systems can reduce instructors' evaluation workload and identify erroneous student solutions. The data collected by an automated assessment system can be a good resource for analyzing student errors. However, few studies have discussed the use of automated assessment systems for identifying common student misconceptions.

PURPOSE OF THE STUDY

The goal of this exploratory study was to identify common misconceptions that Chinese middle school students encountered in introductory programming by using an automated assessment system. As prior studies indicated that students' math and English ability were two important factors that might affect their misconceptions in introductory programming, another focus of this study was to investigate the relationships between students' programming misconceptions and their math and English ability. Based on the results, the potential of using technology to enhance computer science teachers' understanding of student misconceptions is discussed. This study was guided by the following two research questions:

RQ1. What are common misconceptions held by Chinese middle school students in introductory programming?

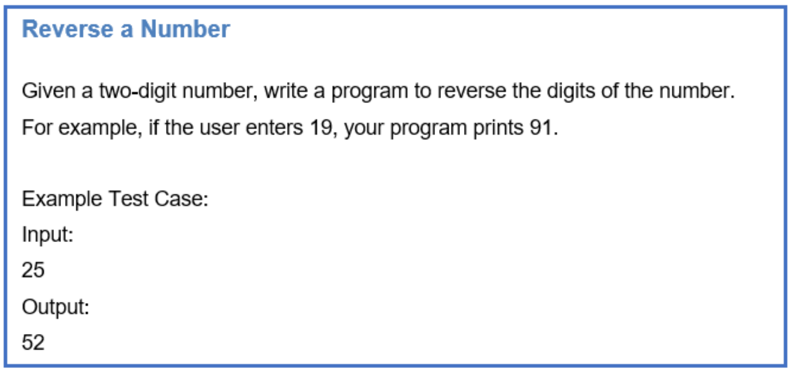
RQ2. Are there any relationships between students' programming misconceptions and their math and English ability?

METHODOLOGY

Subjects and Research Context

The research subjects in this study were a group of 33 middle school students (16 girls and 17 boys) from a city in East China. These students were in Grade 7, which is the first year of middle school in China. They took an introductory 14-week computer science course focused on Pascal programming. Students attended one 90-minute course block every week. Programming concepts covered in the course included Program Structure, Input and Output, Variables, Conditionals, Loops, and Functions. The programming environment used in the course was Dev-Pascal 1.9.2. Computer science was not a core course but was required for all the students in this middle school. None of the students in the study had taken any formal computing classes prior to this course. Students in this study had learned basics of algebra, which is often considered as a prerequisite of learning computer programming. For instance, they had learned how to solve linear equations and inequalities. English was taught as a required academic subject beginning in the 3rd grade in schools of the city. Thus, these students had had about four years' experience in English.

In this study, the SangTian Programming Learning Environment (SPLE) was employed in the instruction to support the identification of student misconceptions. SPLE is an automated assessment system for learning Pascal programming and was designed by the course instructor (the first author). SPLE has a pool of 56 programming problems, which were presented in Chinese. **Figure 1** presents a translated example problem. Each problem had several test cases, which were pairs of input data and expected output. To solve a problem in SPLE, a student had

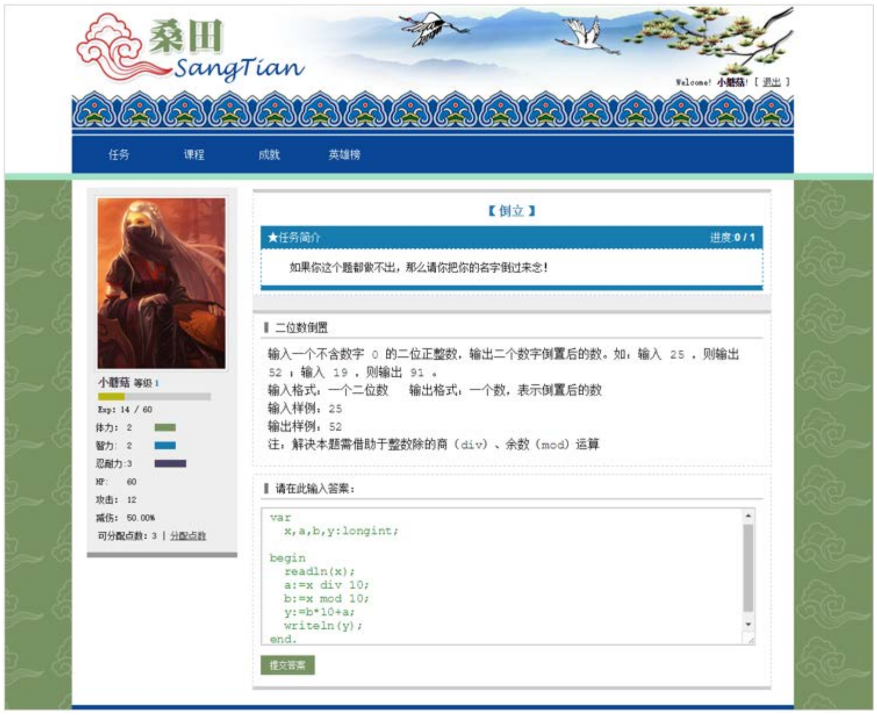


Reverse a Number

Given a two-digit number, write a program to reverse the digits of the number.
For example, if the user enters 19, your program prints 91.

Example Test Case:
Input:
25
Output:
52

Figure 1. A Translated Example Problem in SPLE



【 倒立 】

★任务简介 进度:0 / 1

如果你这个题都做不出，那么请你把你的名字倒过来念！

小蘑菇 等级 1
Exp: 14 / 60
体力: 2
智力: 2
忍耐力:3
HP: 60
攻击: 12
减伤: 50.00%
可分配点数: 3 | 分配点数

二位数倒置

输入一个不含数字 0 的二位正整数，输出二个数字倒置后的数。如，输入 25 ，则输出 52 ；输入 19 ，则输出 91 。
输入格式：一个二位数　　输出格式：一个数，表示倒置后的数
输入样例：25
输出样例：52
注：解决本题需借助于整数除的商 (div) 、余数 (mod) 运算

请在此输入答案：

```
var
  x,a,b,y:longint;

begin
  readln(x);
  a:=x div 10;
  b:=x mod 10;
  y:=b*10+a;
  writeln(y);
end.
```

Figure 2. A screen Shot of the User Interface of SPLE

to write a program to produce output that correctly matched the expected output. If the solution was incorrect, the student was informed that errors existed in the solution and could submit new solutions until his or her solution was correct. If the solution was correct, the problem was considered successfully solved and could not be solved again by that student.

In addition, SPLE was designed in a game-based way (see **Figure 2** for the user interface). Each student created his or her own avatar in SPLE to solve problems. When a student solved a problem in SPLE, his or her avatar gained experience points. Every problem in SPLE had a difficulty level, which was from 1 (lowest) to 10 (highest). Solving a higher difficulty-level problem gave more experience points. When the student's avatar accumulated enough experience points, higher difficulty-level problems would be unlocked, and the student could attempt the harder problems. As the course progressed, students could use the achievement system and the leaderboard to track their learning progress and compare their performance with others. Thus, the game-based design motivated students to solve more programming problems.

The use of SPLE in the instruction also benefited the instructor. In every class, the instructor presented several worked-out examples to demonstrate ways of applying the learned statements to solving problems. After the demonstration, the students would do individual practice and solve problems in SPLE. The instructor did not have to grade students' solutions because SPLE assessed the solutions automatically. The instructor only needed to offer necessary help upon request. Moreover, the instructor could use the backend to track students' learning progress. Hence, the instructor could focus on providing personalized support for students and enhancing the instructional design, rather than grading.

As an automated assessment system, SPLE collected all student solutions and output. These data were valuable for researchers to investigate students' learning difficulties (e.g., misconceptions) and understand the evolution of students' (mis)conceptions of certain programming concepts. In this study, students' solutions and output were the data source for misconception analysis. As we also examined the relationships between students' misconceptions and their math and English ability, we collected students' academic scores in their math and English courses from the academic assessment department of the school as a proxy measure of students' math and English ability. These scores were averages of each student's scores on the formal exams in each subject. The maximum score for each subject was 100 points.

Data Analysis

Students' incorrect solutions in SPLE were used to analyze their misconceptions. When the same error existed in different students' solutions, these students might share common misconceptions. Hence, to identify common student misconceptions, the first step was to find common errors shared by different students. Two types of errors in students' programs were analyzed: compilation errors and non-compilation errors. A compilation error is an error that makes a program fail to be compiled. When a student solution to a problem has no compilation errors but produces incorrect output, it means that this solution has non-compilation errors.

Compilation errors were not specific to the programming problems in SPLE but were mainly Pascal syntax errors and certain semantic errors associated with the incorrect use of programming constructs. In this study, a common compilation error was defined as an error that was identified in solutions of at least one-quarter of the students. In other words, when eight or more students had the same compilation error in their solutions, this compilation error was considered common. Students' incorrect solutions that produced common compilation errors were further analyzed for evidence of specific misconceptions.

Non-compilation errors were problem-specific because they were the result of failing to apply appropriate programming knowledge and strategies to solve a given problem. However, non-compilation errors of different problems might share the same underlying student misconception. In this study, when a problem was solved by more than half of the students (>=17), but a quarter or more students who solved this problem shared the same non-compilation error, this error was defined as a common non-compilation error. The incorrect student solutions that produced these common non-compilation errors were analyzed for student misconceptions.

To answer RQ2, quantitative analysis was conducted. In the literature of computer science, no widely accepted measure of student misconceptions has been established, but some researchers have used the number of errors made by students as an approximate measure (see Becker 2016). In this study, every student's *error rate* of solving problems in SPLE was calculated as an approximate measure of students' programming misconceptions. When a student solved a problem, his or her error rate of this problem was the percentage of the incorrect solutions he or she submitted to solve this problem. For example, if Mike submitted 4 solutions (3 incorrect solutions and 1 correct solution) to solve problem X, his error rate of problem X was 75%. An overall error rate for each student was calculated as the mean of the error rates of all the problems he or she solved. Hence, when a student had a lower error rate, it indicated that he or she made fewer mistakes and might have fewer misconceptions. Correlations were computed to analyze relationships between students' error rates and their academic performance in math and English. In addition, a stepwise regression analysis was used to identify significant factors that predicted the error rate. The significance levels for entering and removing a variable were set to .15 and .075 respectively, as recommended by Derksen and Keselman (1992).

RESULTS

RQ1: What are Common Misconceptions held by Chinese Middle School Students in Introductory Programming?

In total, the thirty-three students submitted 2380 solutions, of which 1293 solutions were incorrect. Among the incorrect solutions, 349 had compilation errors, and 944 had non-compilation errors. Ten common compilation errors and nine common non-compilation errors were identified (see **Table 1** and **2**). The two tables also present the programming concepts relevant to and the occurrence rates of these errors. For the occurrence rate of a compilation error, the numerator is the number of students who had the error, and the denominator is the sample size (33). For the occurrence rate of a non-compilation error, the numerator is the number of students who had the error, and the denominator is the number of students who solved the relevant problem. In addition, for non-compilation errors, the specific problems are included in the table. Details about these errors and the underlying student misconceptions are described and discussed in the following sections.

Table 1. Common Compilation Errors

#	Error	Occurrence Rate	Relevant Concept
1	missing semicolon	22/33 (67%)	Program Structure
2	missing *begin*	16/33 (48%)	Program Structure
3	full stop error	8/33 (24%)	Program Structure
4	Chinese character error	15/33 (45%)	Program Structure
5	identifier not found	15/33 (45%)	Variables, Functions
6	illegal expression	14/33 (42%)	Variables
7	incompatible types	12/33 (36%)	Variables
8	colon issue	10/33 (30%)	Variables
9	mismatched parenthesis	9/33 (27%)	Program Structure, Variables
10	missing *then*	8/33 (24%)	Conditionals

Table 2. Common Non-Compilation Errors

#	Error	Problem Title	Occurrence Rate	Relevant Concept
1	missing newline	Multiples of Three	11/24 (46%)	Output, Loops
2	missing newline	Natural Numbers II	14/26 (54%)	Output, Loops
3	missing newline	The Multiples of 7	6/19 (32%)	Output, Loops
4	missing newline	Stars	12/33 (36%)	Output, Loops
5	missing newline	Adding Numbers	8/27 (30%)	Output, Loops
6	infinite loop	Prime Factorization	5/20 (25%)	Loops
7	wrong output	Prime Factorization	5/20 (25%)	Output
8	missing output	Prime Factorization	6/20 (30%)	Output
9	wrong operation	Arithmetic Operations	7/28 (25%)	Variables

Common compilation errors and underlying misconceptions

Three common compilation errors were directly related to Pascal program structure, including **missing semicolon (#1)**, **missing *begin* (#2)**, and **full stop error (#3)**. In Pascal programming, the semicolon is a necessary punctuation mark to terminate a statement. However, students often forgot to add semicolons, and missing semicolon was the most common compilation error in this study. In a correct Pascal program (see **Figure 3**), the main program block, which includes all the executable statements, should start with the keyword *begin* and end with the keyword *end* with a full stop (.). Our analysis of student code showed that students often forgot to add the *begin* keyword when starting the main program block. Interestingly, while code blocks inside the main block should be enclosed within a pair of *begin* and *end* followed by a semicolon (;), students did not miss that *begin*. However, many of them confused "*end.*" with "*end;*" and got the **full stop error**. The *end* keyword at the end of the main program block should be followed by a full stop (.), but many students used a semicolon or forgot to add the full stop. While the three errors appear to be different, the underlying misconception was the same: students' deficient understanding of Pascal program structure. It is not surprising that these novices often forgot necessary punctuation or structure keywords as the Pascal syntax was new to them.

The **Chinese character error (#4)** was also related to program structure, but it was caused by the use of Chinese punctuation. Pascal is an English-based programming language and requires English punctuation. However, when the Chinese students in this study typed their code, they sometimes mistakenly used Chinese punctuation. Chinese punctuation symbols such as ：；。（） (respectively a colon, a semicolon, a full stop, a pair of parentheses) look almost identical to the English equivalents but are not recognized by the compiler. Therefore, it was difficult for these Chinese middle school students to enter proper punctuation in their programs. This was a special challenge for Chinese students.

In Pascal, an identifier is a sequence of characters that represents the name of a variable, a function, and so forth. The **identifier not found (#5)** error occurs when an identifier such a variable name or a function name cannot be found. After analyzing students' code, we found that the major source of this error was failing to declare variables before using them. In Pascal, variable declaration using the *var* keyword is required before a variable can be used (see line 2 in **Figure 3**); however, students in this study often failed to do this. Another source of the **identifier not found** error was misspelling the identifiers. For example, one student declared the variable *sum* but then spelled it *sun* in the code. Some students also failed to correctly spell built-in functions. For example, the function *writeln*, which means "write a line on the screen", was misspelled as *writein* by some students. The function *sqrt* calculates the square root of a number, but one student spelled it as *sprt*. Because these Chinese students might not know the English meaning of the identifiers, correct spelling was a special challenge for them.

```
 1  program example01;
 2  var a, b : integer;
 3  begin
 4    readln(a, b);
 5    if a>b then
 6       begin
 7          writeln(a, ' is greater than ', b);
 8       end;
 9    if a=b then
10       begin
11          writeln(a, ' is equal to ', b);
12       end;
13    if a<b then
14       begin
15          writeln(b, ' is greater than ', a);
16       end;
17  end.
```

Figure 3. An Example Pascal Program

The errors **illegal expression (#6)**, **incompatible types (#7)**, and **colon issue (#8)** were related to variables and variable operations. An illegal expression is an expression that Pascal cannot understand. In Pascal, the assignment operator is a colon and an equals sign (:=), and a single equals sign (=) is the equality operator that checks whether the values of two operands are equal. However, many students failed to distinguish the two operators. For example, one student tried to use this expression *p=(a+b+c)/2;* to assign the value of expression on the right side to the variable *p*, but he got the **illegal expression** error because he used the equality operator rather than the assignment operator. One student wrote this expression *2c:=(a+b)*(b-a+1);* in her code and thought the value of *c* would be calculated automatically by Pascal. However, this was an illegal expression because *2c,* while an acceptable expression in math, is not a valid expression in Pascal. The **incompatible types** error occurs when the programmer tries to assign a value to a variable with a different data type or perform operations that are not allowed by the variable type. For example, the result of division (/) in Pascal is a real type (real number), but students in this study often tried to assign the real value to an integer variable. One problem in SPLE asked students to output *n* asterisks (*) on the screen, and one student used this line *writeln(n * '*');* in her solution. However, an integer value cannot be multiplied by a character, so an **incompatible types** error occurred. The **colon issue** is related to variable declaration. When declaring variables in Pascal, a colon should be placed between variable names and variable types (see line 2 in Figure 3). The **colon issue** occurs when the colon is missing, or an unnecessary equals sign is added after the colon. The underlying misconception of these three errors was confusion between variables and variable operations in math and computer programming. In math, students do not need to define variable types, precision, and allowed operations, and the equals sign means equality. However, in Pascal programming, every variable has a type that defines the precision and allowed operations, and the equals sign is a relational operator that checks for equality.

The **mismatched parenthesis (#9)** error seems to be straightforward, but our analysis of students' code indicated that most of the time it was caused by illegal expressions in parenthesis rather than missing parentheses. Code pieces like *writeln(2+3a);* and *readln(a''b''c);* were invalid in Pascal, and because the Pascal compiler could not interpret the code after the open parenthesis it failed to "see" the closing parenthesis and produced the mismatched parenthesis error. The **missing *then* (#10)** error was caused by similar issues. Here are two examples of student code that produced this error: (1) *if (a>b),(b>c) then*; (2) *if b mod a:=0 then*. In the first example, the comma (,) was an illegal operator; a Boolean *and* operator should be used instead. In the second example, the assignment operator (:=) should be replaced by the equality operator (=). Because of the incorrect use of operators, the Pascal compiler reported the keyword *then* was missing. In summary, these two errors resulted from a variety of student misconceptions, including problematic knowledge of the program structure, confusion between math and programming expressions, and misunderstandings of operator meanings.

Common non-compilation errors and underlying misconceptions

Among the nine common non-compilation errors, five were the same **missing newline** error. When solving problems in SPLE, students were required to have a newline at the end of the output of their programs. This rule was easy to implement by using the built-in function *writeln*. We found that all five problems with the missing newline error were among the first several Loops problems. The loop construct is often considered challenging to beginners. When students in this study started to solve loops-related problems, they might have experienced high cognitive load (Sweller, 1988) and forgot the newline requirement. Thus, this error did not frequently occur until students started to use the loop constructs. The underlying problem here probably was not students' flawed

Table 3. Correlations between Error Rate and Academic Performance (N = 33)

	Error Rate	**Math**	**English**
Error Rate	-		
Math	-.39*	-	
English	-.49**	.47**	-

Note. * p < .05. ** p < .01

understanding of the output construct but rather their insufficient awareness or misunderstandings of loops in Pascal programming.

Three common non-compilation errors were related to the problem Prime Factorization. Two of them, **wrong output** and **missing output**, were related to the concept Output. This problem asked students to write a program to perform prime factorization of a number *n* and present the results like *180=2*2*3*3*5*. The number *n* is a variable, and the user will enter its value. A quarter of the students who solved this problem directly output the character "n" instead of the value of *n* and got the **wrong output** error. Six students forgot to output the value of n and the equals sign and got the **missing output** error. The other error was the **infinite loop** error. Because nested loops were used to solve this problem, these novices had difficulty distinguishing the loop variables of the inner and outer loop. As a result, it is not surprising that infinite loops existed in some of the programs. While the three errors seemed to be unrelated, they probably were all due to students' difficulties in understanding and applying nested loops. When the nested loops were incorrectly constructed, students made an infinite loop. When the nested loops were successfully handled, students failed to manage the output statement.

Another common non-compilation error was related to variable operations. The Arithmetic Operations problem asked students to write a program that reads two integers from the user and outputs the expressions of the four basic arithmetic operations of the two integers. For the division, this problem requires integer division, which means only presenting the integer quotient. For example, if the two numbers are 10 and 4, for the arithmetic operation division, a correct solution should output *10/4=2*, rather than *10/4=2.5*. However, seven students output the decimal places and made a **wrong operation** error. In Pascal, the operator *div* is designed for integer division. As this special operator does not exist in math, students had difficulty using it as necessary. This error again was related to the student misconception resulting from confusing variables and variable operations in math and programming.

RQ2: Are there any Relationships between Students' Programming Misconceptions and their Math and English Ability?

Correlations between students' error rate and their academic scores in math and English were computed (see **Table 3**). Students' error rate in the programming course significantly correlated with their academic performance in math (r = -.39, p < .05) and English (r = -.49, p < .01). In other words, students with better English or math ability made fewer errors in their programs.

To further understand the relationships, a stepwise regression analysis was conducted to examine important factors to predict students' error rate (see **Table 4** for summaries). The results of the stepwise regression indicated that students' English score significantly predicted the error rate, β = -.49, p < .01, and the best model had only the English score as the predictor, R^2= .24, F(1, 31) = 9.87, p < .01. The results suggest that when the English ability of Chinese students is considered, their math ability becomes unimportant for explaining their programming misconceptions. In other words, for Chinese students, better English ability, rather than math ability, may help to reduce their misconceptions in learning to program.

Table 4. Summaries of the Models and Stepwise Selection

a. Summary of Model with Math and English Scores

Variable	*Beta*	*Std. Error*	*β*	*t*
Intercept	255.63	74.23		3.44**
Math	-.46	.39	-.21	-1.19
English	-2.05	.92	-.39	-2.23*
R^2	.27			
F	5.71**			

Note. * *p* < .05. ** *p* < .01

b. Summary of Model with English Score

Variable	*Beta*	*Std. Error*	*β*	*t*
Intercept	262.06	74.53		3.52**
English	-2.56	.81	-.49	-3.14**
R^2	.24			
F	9.87**			

Note. ** *p* < .01

c. Stepwise Selection Summary

Step	Variable	Added/Removed	R^2	Adj. R^2	C(p)	AIC	RMSE
1	English	addition	0.24	0.22	1.93	255.72	10.98

Note. RMSE: Root Mean Square Error

DISCUSSION

Student Misconceptions in Computer Science

This study identified common student misconceptions in computer science using thousands of incorrect programs of a group of Chinese middle school students. Some misconceptions identified have also been reported in previous studies. For example, students' deficient understanding of the program structure (e.g., missing semicolon and mismatched parenthesis) has been found to be the most common student misconception in previous studies (Altadmri & Brown, 2015; Becker, 2016). Using an advanced programming environment that supports identifying syntax errors and/or providing novice-friendly error messages can help to address this misconception (Becker, 2016; Qian & Lehman, 2017). In addition, students' confusion between their math knowledge and programming knowledge was also noted as a common misconception in prior research (Clancy, 2004; Qian & Lehman, 2017). Explicitly teaching the differences between variables and variable operations in computer programming and math may help to ameliorate the problem. For example, using the cognitive conflict strategy, which explicitly presents details of the new knowledge that conflicts students' existing knowledge, can be an effective way to challenge students' misconceptions and promote conceptual change (Ma, Ferguson, Roper, & Wood, 2011).

On the other hand, our study identified special difficulties experienced by Chinese students. Our results suggest that Chinese students may have misconceptions related to their limited English ability. In this study, the programming language was English-based, but the students had limited English ability. While English was taught as an academic subject beginning in the 3rd grade in schools of that city, so the students had had four years' experience in English, these students may not have understood all the English words in the programming language and environment. They often misspelled identifiers and had difficulties understanding the English-based programming commands. Another interesting finding was students' inappropriate use of Chinese punctuation in their code. While English learners will not have such problems, the Chinese punctuation issue occurred repeatedly in these Chinese students' programs and was difficult to debug because the symbols look nearly identical to their English equivalents. The results of our study suggest that limited English ability may lead to certain difficulties in learning programming for non-English speakers using an English-based programming language. The results of the quantitative analysis also support this argument. Our results showed that students with better English ability made fewer mistakes in programming. While their math ability was also correlated with their programming error rate, only the English ability of these Chinese students was important for explaining the variance of the error rate in the regression model.

This finding conflicts with the results of previous studies on English speakers that found that students' natural language may have negative impacts on the learning of programming commands (Bruckman & Edwards, 1999; Miller, 2014). For instance, students who are English speakers often inappropriately link a word's meaning in natural language with the same word in the programming language (Bonar & Soloway, 1985; Miller, 2014). Students may believe that the value of the variable "smaller" is always smaller than the value of the variable "larger" even though variable names are independent of their values. According to Taber (2014), when a term's scientific meaning is different from how it is used in everyday life, students may form misconceptions about the term. However, when Chinese students learn to program, they may think in Chinese instead of English. Hence, the English meaning of those programming terms may not negatively affect their thinking. On the contrary, knowing the meaning of the English terms may significantly improve their learning of computer programming (Qian & Lehman, 2016). Therefore, when teaching computer programming to Chinese students, it is important to stress the meaning of the English-based programming commands to help them understand programming concepts.

Potential of Technology for Enhancing Computer Science Teachers' Understanding of Student Misconceptions

While knowledge of student misconceptions is a key part of a teacher's PCK, computer science teachers often show a weak understanding of student misconceptions, and their teaching experience may not help to enhance that understanding (Brown & Altadmri, 2017; Guzdial, 2015). On the one hand, the "expert blind spot" often prevents computer science teachers from precisely evaluating the difficulties faced by their students (Guzdial, 2015). On the other hand, most misconceptions in computer science are latent and may not be observed during traditional instruction (Sorva, 2013; Yadav et al., 2016). In this study, an automated assessment system was integrated into instruction to support identifying common student misconceptions. By using the data in the system and applying a simple misconception-identification algorithm, common misconceptions of this group of Chinese middle school students were captured. Currently, the automated assessment system used in this study, SPLE, can identify difficult problems (e.g., problems with high error rates), track students' learning progress (e.g., the number of solved problems, error rates, etc.), and also collect all student data (e.g., students' solutions to problems). The identification of common student misconceptions is not automatic, and this study relied on the researchers to gather and examine the data using the procedures described in the Methodology section. Future research of SPLE will focus on making the misconception identification automated and developing components that can track the evolution of students' (mis)conceptions of certain programming concepts by comparing successive versions of solutions to one problem developed by one particular student. When an automated assessment system has these functionalities built-in, it can not only help computer science teachers develop better understanding of student misconceptions but also provide support for improving instructional design.

In addition to automated assessment systems, researchers of computer science education have developed other types of technology tools to support teaching computer science. For example, code visualization tools such as Online Python Tutor (Guo, 2013) can illustrate program execution for students. As certain student misconceptions in computer science are caused by students' misunderstandings of code execution (Sorva, Karavirta, & Malmi, 2013), illustrating the steps of how a program executes may help to address misconceptions. In addition, an intelligent tutoring system, which can provide corrective feedback when students have mistakes in their solutions, is another useful tool (Gerdes, Heeren, Jeuring, & van Binsbergen, 2017; Rivers & Koedinger, 2017). Providing immediate feedback using an intelligent tutoring system may help to address students' misconceptions in a timely manner.

While technology tools have the potential to improve computer science teachers' PCK, effective teaching with technology requires technological knowledge (Koehler & Mishra, 2009). By adding the technology dimension to PCK, Mishra and Koehler (2006) proposed the notion of technological pedagogical content knowledge, which is now called TPACK (Koehler & Mishra, 2009). TPACK is the knowledge about how to effectively use technology in a specific teaching context to improve teaching. In other words, teachers' technological knowledge should not be independent of pedagogy or content but should be connected to content-specific pedagogy (Graham, 2011). Hence, it is important for computer science teacher training programs to consider developing teachers' TPACK. Nowadays, teachers' TPACK is considered as a core competency in disciplines such as mathematics and science and has been investigated by many researchers (Alrwaished, Alkandari, & Alhashem, 2017; Gonzalez & González-Ruiz, 2017; Jang & Chen, 2010; Schmidt et al., 2009). However, in computer science education, more research is needed to examine teachers' TPACK.

Limitations

While this study found useful results, it has several limitations. For one, the student sample used in this study may have threats to validity of the results. These middle school students were from one city in East China, and the sample size was relatively small. Hence, they may not be representative of the general population of middle school students learning introductory programming. Second, the use of students' academic scores in their math and English courses as a proxy measure of students' math and English ability might not be accurate. Some aspects of students' math and English ability might not be tested and measured in those academic exams. Third, certain common misconceptions were not addressed by this study. This study identified common student misconceptions using the data collected by SPLE. However, the problems in SPLE do not cover all the concepts of Pascal programming. For instance, students were not required to construct their own procedures or functions to solve problems in SPLE. Thus, their misconceptions related to Pascal procedures and functions may be ignored by this study. Finally, in this study, the course and problems were all based on Pascal, and thus findings may not generalize to other programming languages.

CONCLUSION

With the expansion of computer science education in K-12 schools, the need for quality computer science teachers is increasing. An effective computer science teacher needs to have both the knowledge of computer science and PCK. One core component of teachers' PCK is the understanding of student misconceptions. In this study, by integrating the SPLE system, we identified the common programming misconceptions of a group of Chinese middle school students. We found that students' limited English ability and existing math knowledge contributed to their misconceptions in learning to program. We also noted that students with better English ability made fewer mistakes in their programs. Computer science teachers need to be cognizant of these relationships and tailor their instruction to help students who may have difficulties caused by math knowledge and/or limited English ability.

Our results suggest that appropriate technology integration is able to enhance computer science teachers' PCK. Hence, training computer science teachers to use technology tools to support their teaching can be a potential way to produce more quality teachers. We recommend that teacher training programs in computer science pay attention to developing teachers' TPACK, which is the knowledge for effective teaching with technology. More research is needed to investigate computer science teachers' TPACK.

REFERENCES

Alrwaished, N., Alkandari, A., & Alhashem, F. (2017). Exploring in- and pre-service science and mathematics teachers' technology, pedagogy, and content knowledge (TPACK): What next? *Eurasia Journal of Mathematics, Science and Technology Education*, *13*(9), 6113–6131. https://doi.org/10.12973/eurasia.2017.01053a

Altadmri, A., & Brown, N. C. C. (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* (pp. 522–527). New York, New York, USA: ACM Press. https://doi.org/10.1145/2676723.2677258

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

Bell, T., Andreae, P., & Robins, A. (2014). A case study of the introduction of computer science in NZ schools. *ACM Transactions on Computing Education*, *14*(2), 1–31. https://doi.org/10.1145/2602485

Bennedsen, J., & Caspersen, M. E. (2005). An investigation of potential success factors for an introductory model-driven programming course. In *Proceedings of the 2005 international workshop on Computing education research - ICER '05* (pp. 155–163). New York, New York, USA: ACM Press. https://doi.org/10.1145/1089786.1089801

Bonar, J., & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction*, *1*(2), 133–161. https://doi.org/10.1207/s15327051hci0102_3

Brown, N. C. C., & Altadmri, A. (2017). Novice java programming mistakes: Large-scale data vs. educator beliefs. *ACM Transactions on Computing Education*, *17*(2), 7:1--7:21. https://doi.org/10.1145/2994154

Brown, N. C. C., Sentance, S., Crick, T., & Humphreys, S. (2014). Restart: The resurgence of computer science in uk schools. *ACM Transactions on Computing Education*, *14*(2), 1–22. https://doi.org/10.1145/2602484

Bruckman, A., & Edwards, E. (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* (pp. 207–214). New York, NY, USA. https://doi.org/10.1145/302979.303040

Carlsen, W. (1999). Domains of teacher knowledge. In J. Gess-Newsome & N. G. Lederman (Eds.), *Examining pedagogical content knowledge: The construct and its implications for science education* (pp. 133–144). Kluwer Academic Publishers.

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

Clancy, M. J., & Linn, M. C. (1999). Patterns and pedagogy. *ACM SIGCSE Bulletin*, *31*(1), 37–42. https://doi.org/10.1145/384266.299673

De-La-Fuente-Valentín, L., Pardo, A., & Delgado Kloos, C. (2013). Addressing drop-out and sustained effort issues with large practical groups using an automated delivery and assessment system. *Computers & Education*, *61*(1), 33–42. https://doi.org/10.1016/j.compedu.2012.09.004

Derksen, S., & Keselman, H. J. (1992). Backward, forward and stepwise automated subset selection algorithms: Frequency of obtaining authentic and noise variables. *British Journal of Mathematical and Statistical Psychology*, *45*(2), 265–282. https://doi.org/10.1111/j.2044-8317.1992.tb00992.x

Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic test-based assessment of programming. *Journal on Educational Resources in Computing*, *5*(3), 4:1-13. https://doi.org/10.1145/1163405.1163409

Fisler, K., Krishnamurthi, S., & Siegmund, J. (2016). Modernizing plan-composition studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education - SIGCSE '16* (pp. 211–216). New York, New York, USA: ACM Press. https://doi.org/10.1145/2839509.2844556

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, *18*(2), 93–116. https://doi.org/10.1080/08993400802114508

Gal-ezer, J., & Stephenson, C. (2014). A tale of two countries: Successes and challenges in K-12 computer science education in Israel and the United States. *ACM Transactions on Computing Education*, *14*(2), 8:1-8:18. https://doi.org/10.1145/2602483

Gerdes, A., Heeren, B., Jeuring, J., & van Binsbergen, L. T. (2017). Ask-Elle: An adaptable programming tutor for haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*, *27*(1), 65–100. https://doi.org/10.1007/s40593-015-0080-x

Gonzalez, M. J., & González-Ruiz, I. (2017). Behavioural intention and pre-service mathematics teachers' technological pedagogical content knowledge. *EURASIA Journal of Mathematics, Science and Technology Education*, *13*(3), 601–620. https://doi.org/10.12973/eurasia.2017.00635a

Graham, C. R. (2011). Theoretical considerations for understanding technological pedagogical content knowledge (TPACK). *Computers & Education*, *57*(3), 1953–1960. https://doi.org/10.1016/j.compedu.2011.04.010

Guo, P. J. (2013). Online python tutor: Embeddable web-based program visualization for CS education. *SIGCSE 2013 - Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, 579–584. https://doi.org/10.1145/2445196.2445368

Guzdial, M. (2015). Learner-centered design of computing education: Research on computing for everyone. *Synthesis Lectures on Human-Centered Informatics*, *8*(6), 1–165. https://doi.org/10.2200/S00684ED1V01Y201511HCI033

Guzdial, M. (2016). Bringing computer science to U.S. schools, state by state. *Communications of the ACM*, *59*(5), 24–25. https://doi.org/10.1145/2898963

Hill, H. C., Ball, D. L., & Schilling, S. G. (2008). Unpacking pedagogical content knowledge : Conceptualizing and measuring teachers' topic-specific knowledge of students. *Journal for Research in Mathematics Education*, *39*(4), 372–400.

Hubwieser, P., Magenheim, J., Mühling, A., & Ruf, A. (2013). Towards a conceptualization of pedagogical content knowledge for computer science. *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research - ICER '13*, 1–8. https://doi.org/10.1145/2493394.2493395

Jang, S. J., & Chen, K. C. (2010). From PCK to TPACK: Developing a transformative model for pre-service science teachers. *Journal of Science Education and Technology*, *19*(6), 553–564. https://doi.org/10.1007/s10956-010-9222-y

Jonassen, D. H. (1991). Objectivism versus constructivism: Do we need a new philosophical paradigm? *Educational Technology Research and Development*, *39*(3), 5–14. https://doi.org/10.1007/BF02296434

Koehler, M. J., & Mishra, P. (2009). What is Technological Pedagogical Content Knowledge (TPACK)? *Contemporary Issues in Technology and Teacher Education*, *9*(1), 60–70.

Kolikant, Y. B.-D., & Mussai, M. (2008). "So my program doesn't run!" Definition, origins, and practical expressions of students' (mis)conceptions of correctness. *Computer Science Education*, *18*(2), 135–151. https://doi.org/10.1080/08993400802156400

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education - ITICSE '06* (Vol. 38, pp. 118–122). New York, New York, USA: ACM Press. https://doi.org/10.1145/1140124.1140157

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2011). Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education*, *21*(1), 57–80. https://doi.org/10.1080/08993408.2011.554722

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: a review of the literature from an educational perspective. *Computer Science Education*, *18*(2), 67–92. https://doi.org/10.1080/08993400802114581

Miller, C. S. (2014). Metonymy and reference-point errors in novice programming. *Computer Science Education*, *24*(2–3), 123–152. https://doi.org/10.1080/08993408.2014.952500

Mishra, P., & Koehler, M. J. (2006). Technological pedagogical content knowledge: A framework for teacher knowledge. *Teachers College Record*.

Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education*, *18*(1), 1:1-1:24. https://doi.org/10.1145/3077618

Qian, Y., & Lehman, J. D. (2016). Correlates of success in introductory programming: A study with middle school students. *Journal of Education and Learning*, *5*(2), 73–83. https://doi.org/10.5539/jel.v5n2p73

Ragonis, N., & Ben-Ari, M. (2005). A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education*, *15*(3), 203–221. https://doi.org/10.1080/08993400500224310

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

Sadler, P. M., Sonnert, G., Coyle, H. P., Cook-Smith, N., & Miller, J. L. (2013). The influence of teachers' knowledge on student learning in middle school physical science classrooms. *American Educational Research Journal*, *50*(5), 1020–1049. https://doi.org/10.3102/0002831213477680

Saeli, M., Perrenet, J., Jochems, W. M. G., & Zwaneveld, B. (2011). Teaching programming in secondary school: A pedagogical content knowledge perspective. *Informatics in Education*, *10*(1), 73–88. Retrieved from https://search.proquest.com/docview/864687645?accountid=13360

Schmidt, D. A., Baran, E., Thompson, A. D., Mishra, P., Koehler, M. J., & Shin, T. S. (2009). Technological pedagogical content knowledge (TPACK): The development and validation of an assessment instrument for preservice teachers. *Journal of Research on Technology in Education*, *42*(2), 123–149. https://doi.org/10.1080/15391523.2009.10782544

Shulman, L. (1986). Those who understand: Knowledge growth in teaching. *Educational Researcher*, *15*(2), 4–14.

Shulman, L. (1987). Knowledge and teaching: Foundation of the new reform. *Harvard Educational Review*, *57*(1), 1–21.

Smith, J. P., diSessa, A. A., & Roschelle, J. (1994). Misconceptions reconceived: A constructivist analysis of knowledge in transition. *Journal of the Learning Sciences*, *3*(2), 115–163. https://doi.org/10.1207/s15327809jls0302_1

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, *13*(2), 1–31. https://doi.org/10.1145/2483710.2483713

Sorva, J., Karavirta, V., & Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education*, *13*(4), 1–64. https://doi.org/10.1145/2490822

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*, *12*(2), 257–285. https://doi.org/10.1016/0364-0213(88)90023-7

Taber, K. S. (2013). *Modeling learners and learning in science education*. New York: Springer.

Taber, K. S. (2014). Alternative Conceptions/Frameworks/Misconceptions. In R. Gunstone (Ed.), *Encyclopedia of Sci. Education* (pp. 1–5). Dordrecht: Springer Netherlands. https://doi.org/10.1007/978-94-007-6165-0_88-2

Webb, M., Davis, N., Bell, T., Katz, Y., Reynolds, N., Chambers, D. P., & Sysło, M. M. (2017). Computer science in K-12 school curricula of the 2lst century: Why, what and when? *Education and Information Technologies*, *22*(2), 445–468. https://doi.org/10.1007/s10639-016-9493-x

Yadav, A., Berges, M., Sands, P., & Good, J. (2016). Measuring computer science pedagogical content knowledge: An exploratory analysis of teaching vignettes to measure teacher knowledge. *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*, (October), 92–95. https://doi.org/10.1145/2978249.2978264