3.1. Characteristics of computational thinking

In this section, we define CT and distinguish it from other types of thinking (e.g., systems thinking and mathematical thinking). We also discuss CT's relationship with computer science.

3.1.1. Components of CT

Wing (2006) argued that CT does not mean to think like a computer; but rather to engage in five cognitive processes with the goal of solving problems efficiently and creatively. These include:

1. *Problem reformulation* – Reframe a problem into a solvable and familiar one.
2. *Recursion* – Construct a system incrementally based on preceding information.
3. *Problem decomposition* – Break the problem down into manageable units.
4. *Abstraction* – Model the core aspects of complex problems or systems.
5. *Systematic testing* – Take purposeful actions to derive solutions.

Abstraction is the main element undergirding CT (Wing, 2008), where people glean relevant information (and discard irrelevant data) from complex systems to generate patterns and find commonalities among different representations (Wing, 2010). Abstraction has layers, so one must define each layer and clarify the relationships between layers. This involves: (a) abstraction in each layer, (b) abstraction as a whole, and (c) interconnection among layers. For instance, defining an algorithm is one kind of abstraction—the "abstraction of a step-by-step procedure for taking input and producing some desired output" (Wing, 2008, p. 3718).

In addition to abstraction and problem reformulation, Barr et al. (2011) argued that CT also consists of data organization and analysis, automation, efficiency, and generalization.

Automation is making a process or system operate automatically; efficiency means creating optimal solutions; and generalization involves applying CT strategies to solve new problems. Barr and colleagues also included certain dispositions important to CT, such as confidence, persistence in relation to solving complex tasks, and the ability to work well in teams. Similarly, CT described by Bers, Flannery, Kazakoff and Sullivan (2014) includes abstraction, generalization, and trial and error activities. They particularly emphasize the importance of debugging (i.e., identifying and fixing errors when solutions do not work as expected).

In a comprehensive report by the National Research Council, CT consists of five elements essential and universal across domains (NRC, 2010): (1) hypothesis testing, (2) data management, (3) parallelism, (4) abstraction, and (5) debugging. When solving a complex problem in any domain, one should generate and test hypotheses systematically to understand how the system works. It is impossible to test all possibilities, so selecting the right parameters to test is important. Data management involves gathering data from various sources, processing data patterns, and representing data in a meaningful way. Parallelism refers to simultaneously processing information from multiple sources or dimensions. Abstraction focuses on modeling the workings of a complex problem/system. Finally, debugging refers to finding and fixing errors after building up particular models. Recently, Anderson (2016) explicated five CT components: (1) problem decomposition, (2) pattern recognition, (3) abstraction (i.e., generalization of repeated patterns), (4) algorithm design for solutions, and (5) evaluation of solutions (i.e., debugging).

Based on the foregoing review, researchers have come up with similar CT constituent

skills. The components common among researchers are: decomposition, abstraction, algorithms, and debugging. In our competency model, CT similarly consists of decomposition, abstraction, algorithms, debugging, as well as iteration and generalization (see detailed definitions, subcategorizations, and justifications in Section 4).

3.1.2. Differences between CT and other types of thinking skills

Researchers are also studying the differences and similarities between CT and other types of thinking (e.g., Barr et al., 2011; Grover & Pea, 2013). In this section, we compare CT with mathematical, engineering, design, and systems thinking.

Mathematical thinking involves the application of math skills to solve math problems, such as equations and functions (Sneider et al., 2014). Harel and Sowder (2005) defined mathematical thinking as global across many problems and "…governs one's ways of understanding" (p. 31). Mathematical thinking consists of three parts: beliefs about math, problem solving processes, and justification for solutions. The main commonality between CT and mathematical thinking is problem solving processes (Wing, 2008). Figure 1 shows the full set of shared concepts of computational and mathematical thinking: problem solving, modeling, data analysis and interpretation, and statistics and probability.

FIGURE 1 ABOUT HERE

Engineering involves skills needed to build or transform things in the world in order to construct better lives (Bagiati & Evangelou, 2016) as well as "applied science and math, solving problems, and making things" (Pawley, 2009, p. 310). The overlap between CT and engineering includes problem solving, along with understanding how complex systems work in the real world

(Wing, 2008). However, unlike engineering, CT is intended to help humans understand complex phenomena through simulations and modeling, which can transcend physical constraints (Wing, 2010). To summarize, CT, mathematical thinking, and engineering stem from different disciplines. Their differences lie in specific applications in their own domain.

Design thinking requires one to solve problems by thinking as a designer (Razzouk & Shute, 2012). Computational thinking and design thinking both focus on problem solving. Design thinking, like engineering, focuses on product specification and the requirements imposed by both the human and the environment (i.e., practical problems). Again, CT is not limited by physical constraints, enabling people to solve theoretical as well as practical problems.

Systems thinking refers to the ability to understand various relationships among elements in a given environment (Shute, Masduki, & Donmez, 2010). According to the competency model developed by Shute et al. (2010), people with system thinking skills should be able to: (a) define the boundaries of a problem/system, (b) model/simulate how the system works conceptually, (c) represent and test the system model using computational tools, and (d) make decisions based on the model. Although CT and systems thinking both involve understanding and modeling systems, CT is broader than systems thinking, which focuses on identifying and understanding the workings of a system as a whole. CT aims to solve problems efficiently and effectively, going beyond modeling and understanding to include algorithmic design, automation, and generalization to other systems/problems. In conclusion, CT is an umbrella term containing design thinking and engineering (i.e., efficient solution design), systems thinking (i.e., system

understanding and modeling), and mathematical thinking as applied to solving various problems.

3.1.3. Relationship of CT with computer science and programming

Another area in need of clarification involves the relationships among CT, computer science, and programming (Czerkawski & Lyman, 2015). And although CT originates from computer science (Wing, 2006), it differs from computer science because it enables people to transfer CT skills to domains other than programming (Berland & Wilensky, 2015).

CT skills are not the same as programming skills (Ioannidou et al., 2009), but being able to program is one *benefit* of being able to think computationally (Israel et al., 2015). For instance, Shute (1991) examined the relationships among programming skills, prior knowledge, and problem-solving skills, within 260 college and technical school students. Participants who had no prior programming experience learned programming skills via an intelligent tutoring system. Several assessments were administered to measure learners' incoming knowledge (i.e., math and word knowledge), cognitive skills (i.e., working memory and information processing speed), and particular aspects of problem-solving skills (e.g., problem identification, sequencing, and decomposition). In addition, a criterion test was used to measure learners' programming skills and knowledge after the intervention. Results from a factor analysis and hierarchical regression showed that working memory, problem identification, and sequencing solutions are the best predictors of programming skill acquisition. Thus, CT and programming skills as well as problem solving are closely related.

In general, the field of computer science is broader than just learning about programming, and CT is broader than computer science (NRC, 2010; Wing, 2006) in that CT includes a way of

thinking about everyday activities and problems. In line with this perspective, Lu and Fletcher (2009) proposed that teaching CT should not even use programming languages; instead the language should be based on notions that are familiar to most students to engender the acquisition of concepts like abstraction and algorithms. Like with most of the research targeting CT, its particular relationship to computer programming is evolving.

3.2. Interventions to develop computational thinking

Researchers have attempted to leverage programming tools, robotics, games/simulations, and non-digital interventions to teach CT knowledge and skills in various educational contexts. The target population ranges from kindergarten to undergraduates. Table 1 summarizes all of the research studies we reviewed, arrayed by intervention tools.

TABLE 1 ABOUT HERE

3.2.1. Research on CT using programming tools

Due to its close relationship with computing and programming, CT skills appear to be improved via computational tools, such as Scratch (MIT, 2003). Regarding the equivalence of programming skills to CT skills, Cetin (2016) compared the effects of employing Scratch (experimental group) with C language (control group) to teach programming concepts to pre-service IT teachers. This experiment lasted for six weeks, and the participants (n = 56) completed pre- and posttests relative to their achievement on and attitudes toward programming. Additionally, nine participants per group were randomly selected to attend semi-structured interviews. Results showed the experimental group performed significantly better than the control group in terms of programming knowledge and skills, but there were no between group

attitudinal differences.

To promote algorithmic thinking via Scratch, Grover, Pea, and Cooper (2015) designed a seven-week Scratch-based CT course for 7th and 8th graders (*n* = 54). Similar to Cetin's (2016) study, CT gain was measured as pretest to posttest improvement on programming skills. The aim of this quasi-experiment was to see which approach (face-to-face instruction vs. face-to-face plus online) supported deep learning relative to computational concepts, such as algorithms, loops, conditionals, and decomposition. The two conditions were matched in terms of receiving comparable instruction for the same duration of time (i.e., four days per week, 55 minutes per day, across seven weeks). Findings revealed that both approaches lead to significantly higher CT gains, and students in the face-to-face plus online group performed significantly better than those in the face-to-face group. Moreover, both groups successfully transferred their programming knowledge and skills to text-based programming tasks.

The strength of Scratch is to help young people learn to think creatively, reason systematically, and work collaboratively, and thus is suitable to facilitate CT. It is easy to use with its drag-and-drop programming method, and provides a meaningful learning environment where learners engage in specific contexts. Alice (Carnegie Mellon University, 1999) functions similarly, equipped with ready-made code blocks. Compared with Scratch, it focuses on creating 3D programming projects. It also can be utilized to train CT. For example, Denner, Werner, Campe, and Ortiz (2014) randomly assigned 320 middle school students to either a dyadic work group or individual programming. Students' CT skills were measured by their ability to accomplish Alice tasks during one semester's course. Results demonstrated that students working

collaboratively achieved significantly higher CT scores than students working alone. And collaboration was especially beneficial to students with minimal programming experience. These findings are consistent with those reported in an earlier experiment conducted by Werner, Denner, Campe and Kawamoto (2012) testing 311 middle school students.

Basu, Biswas, and Kinnebrew (2017) similarly viewed CT constructs as programming-related concepts, such as sequencing, loops, and variables; and they considered iteration, problem decomposition, abstraction, and debugging as CT practices. They designed the CTSiM platform to integrate ecology and CT learning for 6th graders. Basu et al. employed a pretest/posttest design to test the effectiveness of scaffolding provided by a virtual agent embedded in CTSiM. Both the experimental (n = 52) and control (n = 46) groups learned CT and ecology via CTSiM, but only the experimental group received scaffolding. That is, when students failed a given task 3-5 times, scaffolding was triggered. In that case, the virtual agent provided conversation prompts, and the students answered by choosing one of the options, which in turn triggered a response from agent. Agent-student conversations were pre-programmed to help struggling students. In the control group, such functionality was disabled. The ecology tests required students to choose correct answers and provide rationales for their answers. The CT tests required students to predict outcomes after reading program segments, and build algorithms with CT constructs to model scenarios. Pre- and posttests showed significant gains in both groups on ecology and CT. And when pretest scores were used as a covariate, the experimental group (with scaffolding) significantly outperformed the control group in ecology learning gains and CT skills, with effect sizes of .36 and .31, respectively.

3.2.2. Research using robotics

Another fruitful area in which to develop CT skills is robotics, which is also closely related to programming. Lego robotics is particularly popular. For example, *Lego Mindstorms* (http://www.lego.com/en-us/mindstorms) was used to improve high school students' (n =164) CT skills for a year (Atmatzidou & Demetriadis, 2016). The 44 two-hour sessions focused on developing the following CT skills: decomposing problems, abstracting essential information, generalizing the solution to different problems, creating algorithms, and automating procedures.

Solving robot programming problems revealed students' CT skills, which were measured via rubrics related to the quality of problem-solving performance. Quantitative data showed that all participants, regardless of their age or gender, improved similarly on their CT skills following the intervention. Qualitative data generated from interviews and think-aloud protocols confirmed the effectiveness of robotics to develop CT concepts and solve problems effectively.

Lego robotics is a physical activity. Berland and Wilensky (2015) compared the effects of Lego robotics (n = 34) versus virtual robotics (n = 44) among 8th graders. Pre- and posttests on pseudo-code programming measured CT gains. Both groups of students significantly improved their CT skills, and there was no significant posttest difference between the two groups. It is interesting to note that the two groups perceived the problems differently. That is, the virtual robotics group tended to perceive the problem as a whole and then attempt to decompose the problem down to its details; while the physical robotics group initially focused on the constituent parts.

How early can children learn CT skills? One study looked at teaching kindergarten

students (n = 53) CT skills (Bers et al., 2014). These researchers developed the *TangibleK Robotics* curriculum (http://ase.tufts.edu/DevTech/tangiblek/) which included 20 hours of instruction and one final project to measure students' development of CT in terms of debugging, sequencing, loops, and conditionals. However, repeated measures between tasks did not reveal any linear CT development in kindergarten students. The researchers speculated that either (a) kindergarten students are simply too young to develop CT skills, or (b) the robotics intervention was too difficult for them (Bers et al., 2014). Therefore, it is important to teach CT concepts in a way suitable for students' developmental stages.

3.2.3. Research using game design and other intervention tools

AgentSheets is an authoring tool that uses game design to teach CT, as well as to promote students' interest and skills in computer science (see Ioannidou et al., 2011 for details). AgentSheets provides a low entry bar for inexperienced students, yet does not restrain students with advanced abilities from creating complex game systems. In one study, teachers and community college students in two summer workshops learned how to animate interactions among objects via programming, using five AgentSheets design tasks (Basawapatna, Koh, Repenning, Webb, & Marshall, 2011). Unlike other studies, this research measured CT based on students' knowledge of relevant patterns, defined as object interactions, such as collision and absorption. These CT patterns represent the intersection of games and science simulations. After two weeks of intervention, the authors tested the participants' CT skills with eight questions, each representing a phenomenon sharing CT patterns across the five games (e.g., sledding collision). Participants needed to identify and justify usage of the CT patterns to create

simulations of the phenomena. Generally, all participants did very well in correctly identifying CT patterns, as indicated by the percent correct of their responses. However, the authors noted that the results do not necessarily mean that students could transfer CT knowledge from game design to simulations of scientific phenomena.

Teaching CT does not necessarily require digital tools. A paper-and pencil programming strategy (PPS) was created for non-computer science majors (n = 110) (Kim, Kim, & Kim, 2013). It enabled participants (n = 55) in one group to choose their preferred way to visualize and represent programming ideas (e.g., via concept maps, tables, words, or symbols). Its effects were compared to another group, who engaged in typical course instruction (n = 55) for 15 weeks. Typical instruction in this case refers to the regular programming course, using LOGO (Softronics, Inc., 2002). Pre- and posttests revealed that students using the paper-and-pencil programming strategy showed significantly better understanding of CT and more interest in CS, than those learning via typical course instruction. The authors therefore argued that teaching CT without computers might be effective and efficient for non-computer science majors.

A lecture-based CT instructional module could also change pre-service teachers' understanding of and attitudes toward integrating CT in their classrooms (Yadav, Mayfield, Zhou, Hambrusch, & Korb, 2014). The researchers tested a 100-minute module focused on CT concepts such as decomposition, abstraction, algorithms, and debugging. After training, participants completed questionnaires with open-ended questions related to CT and its connections with various disciplines. Compared to the no-treatment control group (n = 154), a higher percentage of those in the experimental group (n = 141) who received the CT training

considered CT a way of thinking to solve problems, with or without the help of computers. Moreover, they realized that CT was not restricted within the field of computer science. Conversely, the control group viewed CT narrowly—as using computers to complete specific computing tasks and restricted to subjects involving computers, like computer science and math.

3.2.4. CT skills undergirding programming, robotics, and game design

The rationale for using programming, robotics, and game design to improve CT skills is because each one of these areas emphasizes various CT components. For instance, writing computer programs requires the analysis of the problem (e.g., determining the goal to achieve), and then breaking the problem down to its constituent processes (e.g., identifying sub-goals and associated steps to achieve the goal). Writing efficient programs requires abstraction and generalization. For instance, if one step needs to be repeated four times, an efficient solution involves looping the step rather than writing the same code four times. Additionally, part of the programming code may be reused within similar problems, with minor adjustments, rather than rewriting a new program from scratch. And to test the correctness and efficiency of a program, debugging is necessary. That is why programming is frequently used to promote CT skills.

Similarly, robotics provides learners with tactile experiences to solve problems via CT skills. Learners need to identify the general problem/goal for the robot, then decompose the problem (e.g., figuring out the number of steps or sub-goals to accomplish the goal). Towards this end, learners develop algorithms for the robot so that it can follow the instructions and act accordingly. When the robot does not act as expected, debugging comes into play. Debugging requires the iterative processes of systematic testing and modifying.