Good (and Bad) Reasons to Teach All Students Computer Science

Colleen M. Lewis

Abstract Recently everyone seems to be arguing that all students should learn computer science and/or learn to program. I agree. I see teaching all students computer science to be essential to counteracting our history and present state of differential access by race, class, and gender to computer science learning and computing-related jobs. However, teaching computer science is not a silver bullet or panacea. The content, assumptions, and implications of our arguments for teaching computer science matter. Some of the common arguments for why all students need to learn computer science are false; some do more to exclude than to expand participation in computing. This chapter seeks to deconstruct the many flawed reasons to teach all students computer science to help identify and amplify the good reasons.

Keywords Computer science · Education · CS4All · Equity · Computational thinking · Programming · Interdisciplinary

Introduction

I am a computer scientist. I love computer science! I see computer science concepts in the world around me not only because computing is ubiquitous, but because computer science concepts allow me to see the non-computing world in new and interesting ways.

When I graduated from college and started working as a software engineer, I sought out opportunities to teach people computer science as a way to share my passion. At the time, I unconsciously assumed that because I loved computer science, everyone should learn computer science. My view has since changed. There are plenty of good reasons for all students to learn computer science; my passion for computer science is not one of them.

C.M. Lewis (✉)
Harvey Mudd College, Claremont, USA
e-mail: lewis@cs.hmc.edu

S.B. Fee et al. (eds.), *New Directions for Computing Education*,
DOI 10.1007/978-3-319-54226-3_2

In this chapter I discuss nine possible motivations for teaching all K-12 students (i.e., students in Kindergarten through twelfth grade) computer science. For each, I use research from education and psychology to identify the extent to which it motivates and justifies teaching all K-12 students computer science and what limitations exist for this justification.

Some of the motivations rely on teaching students computer science, broadly defined. Some of the motivations rely on teaching students to program, which I see as a subset of computer science. While there appears to be constant debate about the definition of computer science (e.g., Denning 2005), I define computer science and programming broadly. In my computer science training I never learned to fix computers, but in my broad definition of computer science I happily include this expertise. Similarly, I happily include writing equations in a spreadsheet in my definition of programming. In general, but particularly for the purposes of this chapter, I am disinclined to say "X isn't real programming" or "Y isn't real computer science."

The idea of teaching K-12 students computer science and/or programming is not new. In the 1980s "Thousands of schools taught millions of students to code with Seymour Papert's Logo programming language. But the enthusiasm did not last" (Resnick 2014, p. xi). My advocacy for computer science education began with unbridled enthusiasm and a bit of ignorance. I hope this chapter provides nuance to the debates about computer science education for policy makers, educators, and individuals interested in providing all students access to educational opportunities. Perhaps we can sustain the enthusiasm this time.

Motivations Premised on Immediate Benefits

I separate the supposed benefits of teaching all K-12 students computer science into immediate and long-term benefits. By immediate benefits I mean those benefits that students can receive without further engagement with computer science after their K-12 education. In contrast, long-term benefits require engagement beyond K-12.

Can Programming Teach Students to Think Logically?

> I think everybody in this country should learn how to program a computer. Should learn a computer language, because it teaches you how to think—(Jobs 1995).

There is a long history of proposing that particular intellectual activities, such as learning Latin, will teach children to be "rigorous thinkers" (Kafai and Burke 2014). The current form of this argument (Kafai and Burke 2014; Koschmann 1997) is that programming will teach students to think (Jobs 1995) or be logical

thinkers.[1] The long history of isomorphic claims (i.e., that X teaches logical thinking or general problem solving skills), provides ample evidence that this intuitive hypothesis is false (Ceci 1991; Tricot and Sweller 2014). In the context of science, Tricot and Sweller (2014) explain that "A person who is able to reason logically in science may show no such ability in his or her personal life or in any area outside of his or her areas of science" (Tricot and Sweller 2014, p. 273). Instead, they explain that "expertise in complex areas can be fully explained by the acquisition of domain-specific knowledge" (Tricot and Sweller 2014, p. 276).

Consider the complex task of learning to play chess. It too appears that this should improve an individual's logical thinking ability and that this logical thinking would explain expert chess performance. However, this is not the case. Tricot and Sweller (2014) summarize:

> [Chess] Masters were superior to lower-ranked players not because they had acquired complex, sophisticated general problem solving strategies, nor general memory capacity, but rather, because they had acquired an enormous domain-specific knowledge base consisting of tens of thousands of problem configurations along with the best move for each configuration (Simon and Gilmarti 1973). No evidence, either before or after De Groot's work has revealed differential, general problem solving strategies, or indeed, any learned, domain-general knowledge, that can be used to distinguish chess masters from lower ranked players. The only difference between players that we have is in terms of domain-specific knowledge held in long-term memory (Tricot and Sweller 2014, p. 274).

While there is no evidence that we can teach individuals to be logical thinkers independent of a specific domain (Tricot and Sweller 2014), it is reasonable to assume that engaging in intellectually demanding tasks is important for students' cognitive development. Research suggests that attempts to teach programming are less than effective (McCracken et al. 2001) suggesting it is difficult to learn and can reasonably be described as intellectually demanding. I propose that it is accurate that programming "teaches you how to think" (Jobs 1995) to the extent that programming, like (even low-quality) schooling, "may be sufficient to maintain and develop IQ and related cognitive abilities" (Ceci 1991, p. 717). This means that including programming as a way of engaging students in intellectually demanding tasks is reasonable. However, this does not motivate replacing existing content with computer science. Computer science is my favorite discipline, but that does not cloud my judgment to believe it is the most (or only) intellectually demanding discipline. It seems to primarily be a matter of depth. Likely, the less superficial the engagement within a domain, the more intellectually demanding it becomes.

It may be possible to teach programming as a way to have students engage in intellectually demanding tasks. However, related claims that programming is uniquely qualified to introduce students to think logically are unsupported and appear to reinforce the idea that programming is the domain of only the intellectual elite. I see the argument that computer science and programming teach people to

[1]This is also one dimension of the current argument for teaching all students "computational thinking" (Wing 2006). Other dimensions of this argument cut across the majority of the eight arguments discussed in this chapter.

think as exceptionally problematic for two reasons. First, it seems to imply that only computer scientists are thinking or thinking logically. That seems arrogant at best and trivially false. Second, the idea that only computer scientists are "thinking logically" becomes racist, sexist, and classist when considering current demographics of computer scientists (Camp 2012). This may appear to be a straw-person critique, but is consistent with the cultural phenomenon of valuing activities in relation to the individuals who are performing those activities (Ashcraft and Ashcraft 2015).

Can Programming Help Students Develop Persistence?

Having rejected the argument that programming teaches students to think, we can move to a similar argument that programming helps normalize failure. Programming has been speculated to be uniquely qualified to help normalize failure and thus encourage productive learning strategies. Given the strict requirements of programming languages, it is uncommon to write a program that works immediately. Instead, people develop programs iteratively while frequently checking for and fixing errors. This mode of interaction is possible because, when working with a programming language, students can often tell if their program works. This contrasts with students' experiences in other subjects where a teacher, or the back of the book, are the only source for whether or not their answer is correct. This self-directed iterative process is often described as having the potential to develop students' persistence.

Helping students recognize that failure is an inevitable part of the learning process appears to be important (Dweck and Leggett 1988; Dweck 2008). Dweck and colleagues have documented that when students believe that their ability is innate and perceive failure as evidence of a lack of their ability, they tend to pursue ineffective learning strategies (Dweck and Leggett 1988; Dweck 2008). Conversely, when students believe that ability is malleable, and can grow with effort, they are more likely to pursue effective learning strategies, like asking questions. Dweck and colleagues refer to these mindsets as fixed and growth, respectively.

Normalizing failure and developing a growth mindset may plausibly result from learning to program, but some researchers argue that frequently receiving error messages may instead encourage students to develop a fixed mindset (Cutts et al. 2010; Simon et al. 2008). At the college level, one intervention to encourage students to adopt a growth mindset in a computer science course was successful (Cutts et al. 2010), and a similar intervention was not (Simon et al. 2008).

This pattern of iterative development and expectations of failure also relate to the construct of "grit," which was introduced by Duckworth and colleagues:

> We define grit as perseverance and passion for long-term goals. Grit entails working strenuously toward challenges, maintaining effort and interest over years despite failure, adversity, and plateaus in progress (Duckworth et al. 2007, pp. 1087–1088).

However, arguments for universal computer science education on the grounds of improving grit should be viewed with caution. The grit narrative sometimes emphasizes personal responsibility and perseverance in ways that ignore and deny systems of power and privilege that promote and prevent success (Golden 2015). Additionally, Credé et al. (2016) provide a meta-analysis of grit research claiming that the presentation of statistical results overstates the relevance of grit.

Whether describing this benefit as developing a growth mindset or developing grit, this argument relies on students transferring this normalization of failure or persistence to new contexts. Credé et al. (2016) argue that grit may be difficult to encourage students to transfer across domains. According to Dweck (2008) people can have a growth mindset about some domains while having a fixed mindset about others, which supports the idea that this mindset must be transferred across domains. And transfer across domains is notoriously difficult (Barnett and Ceci 2002).

While programming certainly requires frequent incremental failures, it is not clear that this helps develop a growth mindset (Cutts et al. 2010; Simon et al. 2008), it is unlikely that this normalization of failure can be transferred outside of the programming context (Credé et al. 2016), and the grit narrative can do harm by ignoring social inequities (Credé et al. 2016; Golden 2015). While this argument that programming can help students develop persistence is insufficient for justifying teaching all K-12 students to program, these ideas can still inform teaching practice. For example, due to the nature of programming, you can expect students to face setbacks. As an educator I make a point to frame these setbacks as expected. I try to connect this experience to other iterative learning processes that students have experience with and connect it to what they should expect in their later learning.

Can Programming Help Students Learn Science and Math?

The previous two sections evaluated claims about what students could learn within a computer science context and then later apply to a new context. A similar claim exists for how programming can help students learn science and math. Hopefully, at this point you are wary of this argument. Transfer of programming knowledge itself to (or from) mathematics or science knowledge is unlikely. While many colleges require students to have a particular level of mathematical proficiency to take computer science courses, adults' mathematics performance is not correlated with success in learning to program (Robins 2010).

However, programming instruction can be designed to teach math or science by integrating the teaching and learning of programming with math or science. Rather than claiming that programming knowledge transfers to math or science, researchers claim that there are benefits from integrating programming into mathematics and science education. That is, mathematics or science that is presented in the context of programming can be transferred to a non-programming context. It is

not reasonable to assume that without intentional curriculum design that learning to program would have any bearing on a student's math or science performance.

Programming has a long history of being used to teach math. For example, early activities for teaching the programming language Logo had students create activities for their peers to learn about fractions (Kafai and Burke 2014). In 1986, Abelson and diSessa published a book demonstrating how the programming language Logo can be used for "mathematical discovery" and as a medium to teach advanced mathematics. The programming language Logo was the foundation for the programming language Scratch (Kafai and Burke 2014), which provides access to some of the same mathematical ideas (Lewis 2010; Lewis and Shah 2012). Lewis and Shah (2012) documented ways in which content in a Scratch programming course for elementary-school students aligned with California elementary-school mathematics standards. Clements et al. (2001) summarize the results of a set of interrelated studies using Logo to teach specific ideas in geometry to K-8 students. They identify ways in which Logo was and was not successful in supporting students' geometric understanding, which can be applied to the Scratch environment because it has the same geometric programming primitives.

More recently, researchers affiliated with Bootstrap World (www.bootstrapworld.org) have developed a curriculum that is aligned with algebra standards (Schanzer et al. 2015). The curriculum is based upon the programming language Racket (racket-lang.org) and the creators argue that most programming languages, unlike Racket, actually introduce properties of variables that are inconsistent with algebraic variables. This suggests the alternative hypothesis that teaching programming could actually lead to negative transfer (Barnett and Ceci 2002).

The Bootstrap curriculum has been shown to improve specific aspects of students' mathematical performance (Schanzer et al. 2015). Compared to a control group, students who completed the Bootstrap curriculum showed larger improvement from pretest to posttest in answering state algebra test questions regarding function application and function composition. Additionally, students who completed the Bootstrap curriculum showed larger improvements on researcher-written word problems. In all cases, these differences in performance between the control and treatment groups were statistically significant. However, the researchers were unable to control for whether students in the control or treatment groups were receiving other mathematics instruction between the pretest and posttest.

Within science education, researchers at the Center for Connected Learning and Computer-Based Modeling at Northwestern University have developed NetLogo, a free and open-source programming environment for agent-based modeling (ccl.northwestern.edu/netlogo).

> Agent-based modeling provides the means to build on intuitive understandings about individual agents acting at the micro level in order to grasp the mechanisms of emergence at the aggregate, macro level (Wilensky et al. 2014, p. 26).

This process, of understanding how a micro-level activity can produce an emergent, macro-level phenomenon, has been identified as a persistent area of misconceptions (Wilensky and Resnick 1999). These emergent phenomena occur across social

science and the sciences. To date, NetLogo has been used by hundreds of thousands users (Wilensky et al. 2014), which has helped users explore emergent phenomena. While the NetLogo project provides an example of how programming can be used to support understanding of emergent phenomena, this focus on science learning has been intentional and it appears unreasonable to assume that computer science and/or programming will improve students' science knowledge without explicit design of curricula.

Again, the fact that programming instruction can be aligned to reinforce or introduce mathematical and scientific ideas does not imply that all programming instruction will provide this benefit. Instead, it is reasonable to assume that there are opportunities for negative transfer from programming to math and that programming instruction may displace important math and science content. The intuition that programming could help students learn science and math is likely based upon the argument that programming inherently requires students to use a type of logical reasoning present in mathematics and science. However, this argument suffers from the same lack of evidence seen in the appeal to programming as an opportunity to learn to think logically.

Can Programming Provide Students Emotional Value, Agency, and Motivation?

Programming can be thought of as a medium for creation, communication, and creative expression. These ideas were most notably introduced in Papert's (1980) book, *Mindstorms: Children, computers, and powerful ideas*. These ideas have been expanded upon and reframed within the current context of computing by Kafai and Burke (2014). In their book, *Connected Code: Why Children Need to Learn Programming*, Kafai and Burke present programming not just as an abstract discipline, but as an activity that can provide students emotional value, agency, and motivation. This is distinct from arguments about how students will transfer their knowledge from computer science. Instead, Kafai and Burke highlight the activities that programming allows children to participate in, because this participation is important in and of itself.

Their argument builds upon constructionism (Papert and Harel 1991), which argues generally for the potential for learning within activities where students are creating. However, Kafai and Burke's argument also extends beyond constructionist claims about cognition. "Programming is not just a cognitive skill that is used to design code. It also is a social and cultural skill that is used to participate in groups" (Kafai and Burke 2014, p. 28).

Kafai and Burke highlight the potential of programming to allow students to create things that they can share. "When code is created, it has both personal value and value for sharing with others" (p. 24). This is particularly evident in the programming language Scratch, which was co-designed by Kafai. Scratch has a

website (scratch.mit.edu) that allows programmers to share, remix, and discuss Scratch projects. As of June 2016, the website hosted more than 15 million shared projects. Building upon the work of Lave and Wenger (1991), Kafai and Burke argue "we need to move beyond seeing programming as an individualistic act and begin to understand it as a communal practice that reflects how students today can participate in their communities" (2014, p. 128).

By articulating how participating in creating is a way to create and support community, Kafai and Burke contrast with critiques of the broader "maker" movement:

> The cultural primacy of *making*, especially in tech culture—that it is intrinsically superior to not-making, to repair, analysis, and especially caregiving—is informed by the gendered history of who made things, and in particular, who made things that were shared with the world, not merely for hearth and home… The problem is the idea that the alternative to making is usually not doing *nothing*—it's almost always doing things for and with other people (emphasis original, Chachra 2015).

Kafai and Burke show the interconnected nature of making and "doing things for and with other people" (Chachra 2015).

Kafai and Burke present a compelling case for the benefits of enabling students to use programming to both create and connect through sharing those creations. This argument is stronger than others that require students to transfer competencies outside of the programming context. However, there may be other equally effective opportunities for enabling students to create and connect. My love of computer science might bias me to believe that programming is the best medium for this, but identifying the best medium is an empirical question and may vary per student or community.

Can Computer Science Learning Help Students Understand the World Around Them?

Computing is ubiquitous and computer science education is important for helping students understand the world around them and for basic citizenship. Labaree (1997) provides a framework for understanding the goals of education. From Labaree's definition of democratic equality we can provide motivation to teach all students computer science.

> From the *democratic equality* approach to schooling, one argues that a democratic society cannot persist unless it prepares all of its young with equal care to take on the full responsibilities of citizenship in a competent manner (emphasis original, Labaree 1997, p. 42).

This argument has been applied within computer science education to suggest that computer science skills are necessary for citizenship.

> Thinking effectively about and with computational processes is a broad-based literacy needed by all citizens to support their effective social, economic, and political participation (Wilensky et al. 2014, p. 28).

For example, all people need to understand enough computer science to be able to make decisions about their digital privacy and security. It is relatively easy for adults to model good practices for protecting physical property, e.g. locking doors or closing curtains. However, it is difficult for children to observe adults using good practices for selecting passwords or safely browsing the Internet (D. Wagner, personal communication, September 2007).

I see this citizenship argument as the superset of the goal of enabling all people to understand the world around them:

> Although few of us will become computer scientists who write the code and design the systems that undergird our daily life, we all need to understand code to be able to examine digital designs and decisions constructively, creatively, and critically (Kafai and Burke 2014, p. 135).

Central to the issue of democratic equality is ensuring that *all* students have this access, which is currently far from the case (Margolis et al. 2008; College Board[2] 2015; Parker and Guzdial 2015). It is reprehensible that not all K-12 students have the opportunity to learn how the world around them works, which in the current age includes understanding computing devices and computing infrastructure. And this access to computer science instruction cannot only be the responsibility of after-school programs or optional courses. If learning to understand the computational world is not integrated into K-12 classrooms, some students will be left behind. Before a student has been introduced to computer science they only have stereotypes of computer science and computer scientists to decide if they will like it. However, these stereotypes are not neutral; they are gendered, racialized, and associated with particular personality characteristics (Ensmenger 2012). If we let students opt-out based solely on these stereotypes we will perpetuate existing patterns of unbalanced participation in computer science careers (Camp 2012).

This argument for universal K-12 computer science instruction is only valid if that instruction helps students understand the world or more generally "take on the full responsibilities of citizenship in a competent manner" (Labaree 1997, p. 42). When K-12 students learn to program, they might not realize that they are learning computer science or how what they are learning connects to the world around them. For example, in a previous study when asked if a screenshot of an unfamiliar programming environment was "programming" a sixth-grade student said "that

[2]We can use the College Board's Advanced Placement Computer Science A exam (AP CS A exam) test-taking rates as a proxy to measure differential access to K-12 computer science instruction. Test-takers who select one of the following demographic options are underrepresented in AP CS test-taking in comparison to their proportion of the US population: female, American Indian, Black, Mexican American, Puerto Rican, and Other Hispanic. While these demographic options provided by the College Board are idiosyncratic, they demonstrate a clear pattern of underrepresentation.

looks like a game you could play online, and when you're playing a game, you're not programming, you're just playing a game" (Lewis et al. 2014, p. 131). Based upon research regarding the transfer of learning (Barnett and Ceci 2002), it may be more difficult for students to transfer their understanding of computer science outside of the original learning context (viz., school) if they do not perceive that they have been learning computer science.

Providing opportunities for all students to understand the world around them is essential, but requires that teachers and students pursue learning goals specific to understanding the computational world around them. It is not sufficient to teach students something that can be applied outside of the classroom without making that connection explicit.

Motivations Premised on Long-Term Benefits

The previous arguments were focused on benefits to K-12 students that were accessible without additional engagement in computer science learning. However, we may want to use K-12 computer science instruction to encourage, enable, or facilitate later learning. The following arguments have a narrower focus by emphasizing these opportunities and potential benefits for students who pursue additional computer science learning such as computer science major in college.[3]

Can Teaching Students Computer Science Help Fill Jobs?

Possibly the most common argument for teaching K-12 students computer science and/or programming is that there are projected to be a lot of computer science-related jobs.[4] The prevalence of computer-science jobs and particularly the projection of unfilled jobs, can motivate teaching K-12 students computer science from the perspective of social efficiency (Labaree 1997). "The social efficiency approach to schooling argues that our economic well-being depends on our ability to prepare the young to carry out useful economic roles with competence" (Labaree 1997, p. 42). If we see the fundamental goal of education as preparing workers, then these job projections appear to motivate teaching K-12 students computer science.

[3]By "college" I mean post-secondary education such as community college and 4-year colleges and universities.

[4]"Exactly how many is a lot?" is a tricky question. Even given detailed projections by the United States Department of Labor: Bureau of Labor Statistics (www.bls.gov), the job classifications do not map clearly to computer science or programming jobs. As an estimate, the National Center for Women and Information Technology (NCWIT) reports that by 2024 there will be 1.1 million computing-related job openings in the USA (2016, ncwit.org/bythenumbers).

However, this motivation relies on the assumption that our efforts to teach computer science will in fact prepare workers. I have seen no evidence that any of the plans for K-12 computer science instruction will directly make students "job ready." Instead, this motivation only becomes credible if we expect that K-12 computer science instruction will encourage and support students in pursuing computer science in college. Later sections of this chapter summarize the cultural and structural barriers that nearly preclude students from pursuing computer science in college if they do not have K-12 experience with computer science. Only because of these barriers is teaching all students computer science important to help fill these expected unfilled jobs.[5]

In addition to benefiting students who pursue computer science in college, it is plausible that K-12 computer science instruction provides relevant preparation across a variety of fields. Wing (2006) identified the many ways in which research across domains relies on computation. Wing (2006) argues that many practitioners need to understand programming and computer science topics related to the efficient execution of their programs. Much like many fields require fluency with statistics, many fields now require, or at least benefit from, fluency with programming.

While programming skills are useful in some, but not all fields, the potential benefits of programming to a field have been described as more foundational, akin to the development of algebra (diSessa 2001). In the book *Changing Minds* diSessa (2001) chronicles the ways in which the development of algebra provided new expressive power to scientists. diSessa describes the idea of "computational literacy" (2001) to capture the ways in which computer programing, like algebra, provides a new representational tool that can expand the expressive power of scientists.

Programming is certainly relevant to a subset of careers and its impact may be transformational (diSessa 2001). This context can motivate teaching computer science in K-12 schools again by working toward the educational goal of social efficiency (Labaree 1997). It may be unnecessary to state, but this does not imply that all careers, or even all scientists, rely on programming skills. However, we can justify teaching all students as a way enable them to pursue a wide set of options.

Broadening access to K-12 computer science instruction is even more essential when considering that despite initiatives to broaden access (see initiatives at www.cdc-computing.org/resources/ and www.yeswecode.org/), access to computer science instruction at the K-12 level in the USA is unevenly distributed by race (Margolis et al. 2008; College Board 2015), class (Kafai and Burke 2014, p. 10), and gender (Margolis and Fisher 2003; College Board 2015). Without increased and improved access, it is unreasonable to expect the demographics of the tech sector to match the US population. This lack of diversity in the tech industry is well documented (Camp 2012) and a cause for concern because individuals have unequal access to lucrative careers.

[5]It is reasonable to direct our attention to removing these barriers, which I believe must be done in addition to providing all K-12 students computer science instruction.

Can Diversity Improve the Tech Industry?

In addition to the argument to expand access and broaden participation to help fill jobs, people make the argument that this will benefit the industry itself. Ibarra and Hansen (2011) in the Harvard Business Review summarize that: "Research has consistently shown that diverse teams produce better results" (Ibarra and Hansen 2011). In line with Ibarra and Hansen's summary, there are great reasons for broadening participation in computer science, but in tandem with each there are narratives that argue for diversity in ways that may be counter productive to the goals of broadening participation. Arguments for broadening participation require nuance.

The appeal to diversity as a tool for more effective teams is sometimes misunderstood or misused to imply that individuals who are members of groups who are underrepresented in computer science will provide a specific, and predictable contribution to a team. For example, consider the claim that having women on a team will help the team be more collaborative. This can lead to differential expectations of people (Gutek and Cohen 1987) and a meta-review suggests that, contrary to the stereotype, women are not more collaborative than men (Balliet et al. 2011). Even if women were, on average, more collaborative than men, there would be women who were less collaborative than most men and men who were more collaborative than most women. Because of significant overlaps in distributions across cognitive, communicative, social, psychological, and motor dimensions, a person's gender identity is likely a poor predictor for most characteristics (Hyde 2005).

Instead, it is important to recognize that all individuals have a variety of identities and experiences and that these identities and experiences shape an individual's contribution to a team in diverse and unpredictable ways (Ashcraft and Ashcraft 2015). I see the benefit of diversity in the tech industry as enabling tech companies to develop products that would not have been conceived of without the perspectives and experiences of the team members. Additionally, while software engineers may have some input into the direction of software features and products, product managers or other decision makers within a company may have the best opportunities to contribute these benefits.

Another version of the argument for the benefit of diversity in the tech industry is to prevent tech companies from developing explicitly racist products. A great example of the racism we should avoid is Snapchat's blatantly racist blackface (King 2016) and yellowface (Zhu 2016) filters. As a White person, I recognize that my experiences are shaped by my race and that this limits my perspective and experiences (McIntosh 1989). I do not think this prevents me from detecting or avoiding blatant racism. It is likely true that if people of color were in positions of power within Snapchat these blatantly racist features would not have been released. However, we cannot use this argument to distract from the responsibility and capability of all humans to avoid blatant racism. I argue that all people need to understand the historical context of racism and other forms oppression. At the college level, I happily integrate this into my computer science teaching.

In helping software companies avoid making racist software products, The Hampshire Halloween Checklist: Is your costume racist? (n.d.) may be a helpful starting point. When replacing "costume" with "software feature," as shown in the following list, it appears the questions could be helpful for avoiding blatant racism in software:

- "Would I be embarrassed or ashamed if someone from the group I'm portraying saw [this software feature]?
- Is my [software feature] supposed to be funny? Is the humor based on making fun of real people, human traits, or cultures?
- Does my [software feature] represent a culture that is not my own?
- Does my [software feature] reduce cultural differences to jokes or stereotypes?
- Does my [software feature description] include the words 'traditional', 'ethnic', 'colonial', 'cultural', 'authentic', or 'tribal'?
- Does my [software feature] perpetuate stereotypes, misinformation, or historical and cultural inaccuracies?"

It seems essential that all employees have the skills to recognize and prevent both blatant and subtle racism. Focusing on these egregious offenses may distract from more prevalent and more difficult to detect forms of bias that influence the design and impact of software (Garratt 2016).

Another variant of the previous argument for the benefit of diversity in the tech industry is to prevent tech companies from developing products that cannot be used by all people. Examples of blunders include developing facial recognition software that does not detect Black people's faces (Rose 2010) and a long history of voice recognition software that does not work for women or girls (Tatman 2016; Rodger and Pendharkar 2004; Nicol et al. 2002). It is reasonable to assume that when characteristics of individuals (e.g., gender identity, skin color, height, ability status) are afforded dominance in society, individuals with those privileged characteristics may be frequently unaware of those characteristics in themselves (McIntosh 1989). This provides a mechanism by which a person could unintentionally make a product not usable by someone who does not share the same characteristics as them. However, it is not clear that previous blunders were the result of this lack of perspective; they may primarily result from cost cutting measures that prioritize benefiting individuals with privileged identities and characteristics.

It is interesting to consider the parallel between these software inaccessibility issues and the history of car safety testing. Manufacturers originally considered variations in height and weight when testing the safety of cars (Shaver 2012; Vinsel 2012). However, in the creation of present-day test dummies, cost-cutting measures were taken to use only a single test dummy, which was modeled after the 50th percentile height and weight for a man: 5 ft 9 in. and 172 lb (Shaver 2012; Vinsel 2012). Only in 2011 did safety tests with results that are accessible to consumers begin including a smaller test dummy. These cost-cutting measures have serious consequences for the health and safety of drivers. Analyzing crash data from 1998 to 2008, Bose et al. (2011) found that "The odds for a belt-restrained female driver to sustain severe injuries were 47% (95% confidence interval = 28%, 70%) higher

than those for a belt-restrained male driver involved in a comparable crash" (p. 2638). These results are unsurprising given the pattern of designing cars to be safe for the median man. This appears to be the result of a calculated decision and not a lack of awareness of the bias in their design.

The inadequacy of a single test-dummy maps clearly onto a common form of bias in software that can explain the inaccessibility of facial recognition software by Black people (Rose 2010) and voice recognition software by women and girls (Tatman 2016; Rodger and Pendharkar 2004; Nicol et al. 2002). In these software cases, the testing data, faces and voices, respectively, were not representative of the possible users of the software. For example, Tatman (2016) describes popular speech databases in which recordings of men's voices are overrepresented. However, these forms of bias and other possibilities for software systems to discriminate are known (Barocas 2014; Crawford 2013) and should become common knowledge for all software engineers.

Again, there are benefits to diverse teams (Ibarra and Hansen 2011), but these benefits are not easily predicted by a single dimension of a team member's identity (Ashcraft and Ashcraft 2015). And our training of software engineers must include an understanding of the historical context of racism and other forms of oppression so that they can push back against cost-cutting measures that will result in unusable, biased, or explicitly racist software.

Can K-12 Computer Science Help Students Feel Like They Belonging in Computer Science Courses in College?

K-12 computer science may confer an advantage by fostering a sense of belonging within computer science. Belonging is important to students' academic success (Yeager et al. 2013) and students' decisions to major in computer science (Lewis et al. 2016). A Google study (2014) reported that "young women who had opportunities to engage in Computer Science coursework were more likely to consider a Computer Science degree than those without those opportunities" (p. 6).

Belonging is likely particularly important in computer science because the idea that computer science requires innate ability is common (Lewis 2007; Robins 2010). This suggests that some people belong in computer science because of that innate ability while others do not. Through ethnography, Barker et al. (2002) documented the patterns of conflating pre-college experience with being "smart" in college computer science classes (2002). Across domains the idea that ability is fixed is inconsistent with research and leads students to pursue less productive learning strategies (Dweck and Leggett 1988). Additionally, a belief that ability within computer science is fixed appears to unproductively shape students' interpretations of their grades, time to complete programming tasks, and their previous experience (Lewis et al. 2011).

One strategy to respond to differences in pre-college experience is by encouraging students to skip the first computer science course. The variant of this strategy used at Harvey Mudd College of providing a different introductory course for students with different levels of previous experience (Dodds et al. 2008) has gained traction as a best practice. This is partially because the practice helps to avoid conflating experience with intelligence and attempts to provide all students the opportunity to have a course designed for students with their level of previous experience. (McGrath Cohoon 2010). However, this goal can be subverted if competitive admissions processes for the major encourage students with pre-college experience to still take the course designed for students without experience (Lewis et al. 2011).

Harvey Mudd College has three different levels of introductory courses labeled with the school colors. The "gold" version of the course is the default for students without previous experience. The "black" version of the course covers the same content as gold for students with some programming experience. This content, predictably, requires less of the semester's instructional time. The remaining time is filled with material of interest to the faculty member. The goal is for this material to be interesting computer science content, but to not confer an additional advantage for students in the lower-division courses. The content typically comes from an upper-division elective. A third version of the course is for students with a lot of previous experience and condenses the content of two courses into a single semester.

Harvey Mudd College is particularly known for this design because it was one of a suite of strategies the computer science department used to try to increase the participation of women in computer science (Alvarado et al. 2012). The department saw a rise in the participation of women in computer science from a typical average of 12% through 2006 to consistently above 40% (Alvarado et al. 2012).

Belonging in computer science appears particularly relevant because persistent stereotypes exist about the identities of typical or ideal computer science practitioners (Ensmenger 2012; Ashcraft and Ashcraft 2015). Ashcraft and Ashcraft (2015) explain how these stereotypes create expectations that serve to exclude individuals from participation. Ashcraft (2012) introduced the construct of a "glass slipper" which, in the spirit of the glass ceiling, captures the ways in which stereotypes of the ideal practitioners exclude participation: Individuals are seen as responsible for adapting to a hostile or unwelcoming environment; the glass slipper does not fit and it is their responsibility to make it fit. One thing we can do to help more students feel like they belong in college computer science courses is to provide all students access to K-12 computer science.

Can K-12 Computer Science Help Students Compete in Computer Science Courses in College?

A lack of K-12 opportunities within computer science may block pathways to pursue computer science in college. Computer science experience likely provides students

an advantage in college computer science courses through higher grades or through requiring less effort. This may be particularly impactful for women who are reported to leave computing majors with higher grades than men (Barker et al. 2009).

It is reasonable to assume that students who had pre-college experience will have an academic advantage over their peers without pre-college experience, particularly if they have experience with the programming language used in their college computer science course (Lewis et al. 2012). For example, a 2012 study showed that although a second-semester computer science course at the University of California, Berkeley did not require prior experience with the programming language Java, "students with Java experience performed better than their peers without Java experience on all outcome measures" (Lewis et al. 2012, p. 86).

At the University of California, Berkeley, there is an implicit prerequisite of Java experience. This barrier is replicated at Harvey Mudd College where the second, but not the first required computer science course is taught in Java. It is likely that many colleges have similar implicit prerequisites that create unintentional barriers to students without previous experience. The benefit of previous programming experience is likely conferred even if the student has to transfer their knowledge from a programming language they learned before college to a programming language they are learning in college. The advantage conferred within college computer science courses by Java experience is particularly problematic because of the differential participation on the AP CS A exam, which tests students' understanding of the programming language Java (College Board 2015). Only 22.0% of AP CS A test-takers identified themselves as female. The following percentages of test-takers identified their race as: American Indian: 0.4%, Black: 3.8%, Mexican American: 3.6%, Puerto Rican: 0.6%, Other Hispanic: 5.0%, Asian: 29.2%, White: 51.7%, Other: 3.6%.

Research regarding college computer science course structures suggest that K-12 computer science experience provides students an advantage in college computer science courses. The unfilled computer science jobs become relevant to teaching all K-12 students computer science through the supporting claim that K-12 computer science helps students compete in computer science courses in college. Competitive admissions processes for the major heighten the relevance of this academic advantage for students (Lewis et al. 2011). With increasing enrollments in computer science departments, (Roberts 2011; Kurose 2015) it is likely that even more colleges will institute competitive admissions policies as a way to limit enrollments to a number of students that is feasible for the department to serve. This could serve to reinforce these structural barriers for students without, or with less, K-12 computer science access.

Conclusion

The goal of this chapter has been to provide nuance to the arguments about why K-12 students should learn computer science. This nuance is important because it shapes what we teach and how we teach.

When I graduated from college, my arguments for K-12 computer science instruction were some of the poorly reasoned or false arguments that I deconstructed within this chapter. I have continued my advocacy for K-12 computer science instruction guided by the following supportable arguments:

- Computing is ubiquitous. Universal, high-quality K-12 computer science instruction could provide all student the opportunities they need and deserve to understand the world around them.
- Cultural and structural barriers block students from pursuing computer science at the college level. Universal, high-quality K-12 computer science instruction could serve as a protective factor for students.
- Computer science jobs are high-paying and high status. Universal, high-quality K-12 computer science instruction could increase access to these high-paying, high-status jobs and push back against current forms of oppression.

I see this work as timely and essential to the broader goals of fighting injustice and inequity. I echo the sentiment of Kamau Bobb, a researcher and advocate for K-12 computer science instruction (Bobb 2016) who argues: "the goal of STEM education work is the acquisition of power and the ability to write the American story" (Mariama-Aruthur 2016).

Acknowledgements This work was partially funded by National Science Foundation grant #1339404. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

Alvarado, C., Dodds, Z., & Libeskind-Hadas, R. (2012). Increasing women's participation in computing at Harvey Mudd College. *ACM Inroads, 3*(4), 55–64.

Ashcraft, K. (2012). The glass slipper: 'Incorporating' occupational identity in management studies. *Academy of Management Review*.

Ashcraft, K. L., & Ashcraft, C. (2015). Breaking the "glass slipper": What diversity interventions can learn from the historical evolution of occupational identity in ICT and commercial aviation. In *Connecting women* (pp. 137–155). Springer International Publishing.

Balliet, D., Li, N. P., Macfarlan, S. J., & Van Vugt, M. (2011). Sex differences in cooperation: A meta-analytic review of social dilemmas. *Psychological Bulletin, 137*(6), 881.

Barker, L. J., Garvin-Doxas, K., & Jackson, M. (2002, February). Defensive climate in the computer science classroom. *ACM SIGCSE Bulletin, 34*(1), 43–47 (ACM).

Barker, L. J., McDowell, C., & Kalahar, K. (2009, March). Exploring factors that influence computer science introductory course students to persist in the major. *ACM SIGCSE Bulletin, 41*(1), 153–157 (ACM).

Barnett, S. M., & Ceci, S. J. (2002). When and where do we apply what we learn?: A taxonomy for far transfer. *Psychological Bulletin, 128*(4), 612.

Barocas, S. (2014). Data mining and the discourse on discrimination. In *Data Ethics Workshop, Conference on Knowledge Discovery and Data Mining*.

Bobb, K. (2016, March 9). Why teaching computer science to students of color is vital to the future of our nation. *The Root*. Retrieved from http://www.theroot.com/articles/culture/2016/03/why_teaching_computer_science_to_students_of_color_is_vital_to_the_future/

Bose, D., Segui-Gomez, M., & Crandall, J. R. (2011). Vulnerability of female drivers involved in motor vehicle crashes: An analysis of US population at risk. *American Journal of Public Health, 101*(12), 2368–2373.

Camp, T. (2012). Computing, we have a problem…. *ACM Inroads, 3*(4), 34–40.

Ceci, S. J. (1991). How much does schooling influence general intelligence and its cognitive components? A reassessment of the evidence. *Developmental Psychology, 27*(5), 703.

Chachra, D. (2015, Jan 23). Why I am not a maker: When tech culture only celebrates creation, it risks ignoring those who teach, criticize, and take care of others. *The Atlantic*. Retrieved from http://www.theatlantic.com/technology/archive/2015/01/why-i-am-not-a-maker/384767/

Clements, D. H., Battista, M. T., & Sarama, J. (2001). Logo and geometry. *Journal for Research in Mathematics Education. Monograph*, *10*, i–177.

College Board. (2015). Program Summary Report 2015. Retrieved from http://media.collegeboard.com/digitalServices/misc/ap/national-summary-2015.xlsx

Crawford, K. (2013). Think again: Big data. *Foreign Policy, 9*.

Credé, M., Tynan, M. C., & Harms, P. D. (2016). Much ado about grit: A meta-analytic synthesis of the grit literature. *Journal of Personality and Social Psychology*.

Cutts, Q., Cutts, E., Draper, S., O'Donnell, P., & Saffrey, P. (2010). Manipulating mindset to positively influence introductory programming performance. In *Proceedings of the 41st ACM technical symposium on Computer science education* (pp. 431–435). ACM.

Denning, P. J. (2005). Is computer science science? *Communications of the ACM, 48*(4), 27–31.

diSessa, A. A. (2001). *Changing minds: Computers, learning, and literacy*. MIT Press.

Dodds, Z., Libeskind-Hadas, R., Alvarado, C., & Kuenning, G. (2008). Evaluating a breadth-first CS 1 for scientists. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, Portland, OR* (pp. 266–270).

Duckworth, A. L., Peterson, C., Matthews, M. D., & Kelly, D. R. (2007). Grit: Perseverance and passion for long-term goals. *Journal of Personality and Social Psychology, 92*(6), 1087.

Dweck, C. S. (2008). *Mindset: The new psychology of success*. Random House Digital, Inc.

Dweck, C. S., & Leggett, E. L. (1988). A social-cognitive approach to motivation and personality. *Psychological Review, 95*(2), 256.

Ensmenger, N. L. (2012). *The computer boys take over: Computers, programmers, and the politics of technical expertise.* MIT Press.

Garratt, P. (2016, May 6). *How do algorithms perpetuate discrimination and what can we do to fix it?* Retrieved from http://www.paige-garratt.com/algorithms

Golden, N. A. (2015). There's still that window that's open" the problem with "grit". *Urban Education,* 1–28.

Google for Education. (2014). *Women who choose computer science—What really matters*. Retrieved from https://docs.google.com/file/d/0B-E2rcvhnlQa1Q4VUxWQ2dtTHM/edit

Gutek, B. A., & Cohen, A. G. (1987). Sex ratios, sex role spillover, and sex at work: A comparison of men's and women's experiences. *Human Relations, 40*(2), 97–115.

Hampshire's Community Advocacy Union. (n.d.). *Hampshire Halloween Checklist: Is your costume racist?* Retrieved from https://www.hampshire.edu/sites/default/files/culturalcenter/files/Halloween.jpg

Hyde, J. S. (2005). The gender similarities hypothesis. *American Psychologist, 60*(6), 581.

Ibarra, H., & Hansen, M. T. (2011). Are you a collaborative leader? *Harvard Business Review, 89*(7/8), 68–74.

Jobs, S. (1995). Lost interview. Retrieved from https://youtu.be/IY7EsTnUSxY

Kafai, Y. B., & Burke, Q. (2014). *Connected code: Why children need to learn programming*. MIT Press.

King, H. (2016, April 20). Snapchat's new Bob Marley lens sparks 'blackface' outrage. *CNN Money*. Retrieved from http://money.cnn.com/2016/04/20/technology/snapchat-blackface/index.html

Koschmann, T. (1997). Logo-as-latin redux. *The Journal of the Learning Sciences, 6*(4), 409–415.
Kurose, J. (2015). Booming undergraduate enrollments: A wave or a sea change? *ACM Inroads, 6*(4), 105–106.
Labaree, D. F. (1997). Public goods, private goods: The American struggle over educational goals. *American Educational Research Journal, 34*(1), 39–81.
Lave, J., & Wenger, E. (1991). *Situated learning: Legitimate peripheral participation.* Cambridge University Press.
Lewis, C. (2007). Attitudes and beliefs about computer science among students and faculty. *ACM SIGCSE Bulletin, 39*(2), 37–41.
Lewis, C. M. (2010, March). How programming environment shapes perception, learning and goals: Logo vs. scratch. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 346–350). ACM.
Lewis, C. M., Anderson, R. E., & Yasuhara, K. (2016). "I don't code all day": Fitting in computer science when the stereotypes don't fit. In *Proceedings of the International Computer Science Education Research Workshop*.
Lewis, C. M., Esper, S., Bhattacharyya, V., Fa-Kaji, N., Dominguez, N., & Schlesinger, A. (2014). Children's perceptions of what counts as a programming language. *Journal of Computing Sciences in Colleges, 29*(4), 123–133.
Lewis, C. M., & Shah, N. (2012, February). Building upon and enriching grade four mathematics standards with programming curriculum. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 57–62). ACM.
Lewis, C. M., Titterton, N., & Clancy, M. (2012, September). Using collaboration to overcome disparities in Java experience. In *Proceedings of the ninth annual international conference on International computing education research* (pp. 79–86). ACM.
Lewis, C. M., Yasuhara, K., & Anderson, R. E. (2011). Deciding to major in computer science: a grounded theory of students' self-assessment of ability. In *Proceedings of the seventh international workshop on Computing education research* (pp. 3–10). ACM.
Margolis, J., Estrella, R., Goode, J., Jellison-Holme, J., & Nao, K. (2008). *Stuck in the shallow end: Education, race, & computing*. Cambridge, MA: MIT Press.
Margolis, J., & Fisher, A. (2003). *Unlocking the clubhouse: Women in computing*. MIT press.
Mariama-Aruthur, K. (2016, May 26). Dr. Kamau Bobb Talks Leadership and Diversity in STEM and Computer Science Education (Part I). *Black Enterprise*. Retrieved from http://www.blackenterprise.com/career/dr-kamau-bobb-talks-leadership-and-diversity-in-stem-and-computer-science-education-part-i/
McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B. D., … Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin, 33*(4), 125–180.
McGrath Cohoon, J. (2010). *Harvey Mudd College's Successful Systemic Approach (Case Study 2)*. Retrieved from https://www.ncwit.org/sites/default/files/resources/howdoesengagingcurriculumattractstudentstocomputing_1.pdf
McIntosh, P. (1989). White privilege: Unpacking the invisible knapsack. *Independent School, 90*(49), 2.
National Center for Women in Information Technology. (2016, March 10). By the numbers. Retrieved from http://www.ncwit.org/bythenumbers
Nicol, A., Casey, C., & MacFarlane, S. (2002). Children are ready for speech technology-but is the technology ready for them. *Interaction Design and Children, Eindhoven, The Netherlands*.
Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas.* Basic Books, Inc.
Papert, S., & Harel, I. (1991). Situating constructionism. *Constructionism, 36,* 1–11.
Parker, M. C., & Guzdial, M. (2015, August). A critical research synthesis of privilege in computing education. In *Research in Equity and Sustained Participation in Engineering, Computing, and Technology (RESPECT), 2015* (pp. 1–5). IEEE.
Resnick, M. (2014). Forward. In Y. B. Kafia & Q. Burke (Eds.), *Connected code: Why children need to learn programming* (pp. xi–xiii). MIT Press.
Roberts, E. S. (2011). Meeting the challenges of rising enrollments. *ACM Inroads, 2*(3), 4–6.

Robins, A. (2010). Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education, 20*(1), 37–71.
Rodger, J. A., & Pendharkar, P. C. (2004). A field study of the impact of gender and user's technical experience on the performance of voice-activated medical tracking application. *International Journal of Human-Computer Studies, 60*(5), 529–544.
Rose, A. (2010, January 22). Are face-detection cameras racist? *TIME*. Retrieved from http://content.time.com/time/business/article/0,8599,1954643,00.html
Schanzer, E., Fisler, K., Krishnamurthi, S., & Felleisen, M. (2015, February). Transferring skills at solving word problems from computing to algebra through bootstrap. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 616–621). ACM.
Shaver, K. (2012, March 25). Female dummy makes her mark on male-dominated crash tests. *The Washington Post*. Retrieved from https://www.washingtonpost.com/local/trafficandcommuting/female-dummy-makes-her-mark-on-male-dominated-crash-tests/2012/03/07/gIQANBLjaS_story.html
Simon, B., Hanks, B., Murphy, L., Fitzgerald, S., McCauley, R., Thomas, L., et al. (2008). Saying isn't necessarily believing: Influencing self-theories in computing. In *Proceedings of the Fourth International Workshop on Computing Education Research* (pp. 173–184). ACM.
Tatman, R. (2016, July 12). Google's speech recognition has a gender bias. Making noise and hearing things. https://makingnoiseandhearingthings.com/2016/07/12/googles-speech-recognition-has-a-gender-bias/
The coalition to diversify computing. (n.d.) *Resources*. Retrieved from http://www.cdc-computing.org/resources/
Tricot, A., & Sweller, J. (2014). Domain-specific knowledge and why teaching generic skills does not work. *Educational Psychology Review, 26*(2), 265–283.
Vinsel, L. J. (2012, August 22). Why carmakers always insisted on male crash-test dummies. *Bloomberg View*. Retrieved from https://www.bloomberg.com/view/articles/2012-08-22/why-carmakers-always-insisted-on-male-crash-test-dummies
Wilensky, U., Brady, C. E., & Horn, M. S. (2014). Fostering computational literacy in science classrooms. *Communications of the ACM, 57*(8), 24–28.
Wilensky, U., & Resnick, M. (1999). Thinking in levels: A dynamic systems approach to making sense of the world. *Journal of Science Education and technology, 8*(1), 3–19.
Wing, J. M. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33–35.
Yeager, D., Walton, G., & Cohen, G. L. (2013). Addressing achievement gaps with psychological interventions. *Phi Delta Kappan, 94*(5), 62–65.
Zhu, K. (2016, August 10). I'm deleting Snapchat, and you should too. *Medium*. Retrieved from https://medium.com/@katie/im-deleting-snapchat-and-you-should-too-98569b2609e4#.40x2u9str

Author Biography

Colleen M. Lewis is Assistant Professor of Computer Science at Harvey Mudd College who specializes in computer science education. At the University of California, Berkeley, Lewis completed a Ph.D. in science and mathematics education, a M.S. in computer science, and B.S. in electrical engineering and computer science. Her research seeks to identify effective teaching practices for creating equitable learning spaces where all students have the opportunity to learn. Lewis curates CSTeachingTips.org, a NSF-sponsored project for disseminating effective computer science teaching practices.

http://www.springer.com/978-3-319-54225-6

New Directions for Computing Education
Embedding Computing Across Disciplines
Fee, S.B.; Holland-Minkley, A.; Lombardi, Th.E. (Eds.)
2017, XI, 308 p. 11 illus., Hardcover
ISBN: 978-3-319-54225-6