Intent and Extent: Computer Science Concepts and Practices in Integrated Computing

LAUREN E. MARGULIEUX, YIN-CHAN LIAO, and ERIN ANDERSON, Georgia State University, Atlanta, GA, USA
MIRANDA C. PARKER, San Diego State University, San Diego, CA, USA
BRENDAN D. CALANDRA, Georgia State University, Atlanta, GA, USA

Integrated computing curricula combine learning objectives in computing with those in another discipline, like literacy, math, or science, to give all students experience with computing, typically before they must decide whether to take standalone CS courses. One goal of integrated computing curricula is to provide an accessible path to an introductory computing course by introducing computing concepts and practices in required courses. This study analyzed integrated computing curricula to determine which CS practices and concepts are taught, how extensively the curricula are taught, and, by extension, how they might prepare students for later computing courses. The authors conducted a content analysis to examine primary and lower secondary (i.e., K-8) curricula that are taught in non-CS classrooms, have explicit CS learning objectives (i.e., CS+X), and that took 5+ hours to complete. Lesson plans, descriptions, and resources were scored based on frameworks developed from the K-12 CS Framework, including programming concepts, non-programming CS concepts, and CS practices. The results found that curricula most extensively taught introductory concepts and practices, such as sequences, and rarely taught more advanced content, such as conditionals. Students who engage with most of these curricula would have no experience working with fundamental concepts, like variables, operators, data collection or storage, or abstraction in the context of a program. While this focus might be appropriate for integrated curricula, it has implications for the prior knowledge that students should be expected to have when starting standalone computing courses.

CCS Concepts: • **Social and professional topics → K-12 education**; **Computing literacy**; **Computational thinking**;

Additional Key Words and Phrases: Integrated computing, computational thinking, CS+X, content analysis, curricula analysis, K-12 computer science education, primary school computer science education

ACM Reference format:
Lauren E. Margulieux, Yin-Chan Liao, Erin Anderson, Miranda C. Parker, and Brendan D. Calandra. 2024. Intent and Extent: Computer Science Concepts and Practices in Integrated Computing. *ACM Trans. Comput. Educ.* 24, 3, Article 35 (August 2024), 23 pages.
https://doi.org/10.1145/3664825

For the data created from this analysis, please find the complete dataset at https://doi.org/10.5061/dryad.j6q573nnt
This work was supported by the National Science Foundation under grant #1941642.
Authors' Contact Information: Lauren E. Margulieux (Corresponding author), Georgia State University, Atlanta, GA, USA; e-mail: lmargulieux@gsu.edu; Yin-Chan Liao, Georgia State University, Atlanta, GA, USA; e-mail: yliao3@gsu.edu; Erin Anderson, Georgia State University, Atlanta, GA, USA; e-mail: eanderson20@gsu.edu; Miranda C. Parker, San Diego State University, San Diego, CA, USA; e-mail: mcparker@sdsu.edu; Brendan D. Calandra, Georgia State University, Atlanta, GA, USA; e-mail: bcalandra@gsu.edu.

ACM 1946-6226/2024/8-ART35
https://doi.org/10.1145/3664825

1 Introduction

Integrated computing applies **computer science (CS)** knowledge and skills, such as computational thinking and programming, in non-CS classrooms [29, 31, 46]. For example, students might create visualizations of fractions through programming as they learn about fractions in math [41]. A major motivation behind integrated computing is to increase access to computing education (e.g., [40, 46]). By integrating computing experiences into the school day, students no longer need to rely on informal experiences, like summer camps, after-school programs, or family and friends, for low-stakes introductions to CS. Equal access to these out-of-school resources is often examined as a barrier to equal participation in computing (e.g., [17, 28]). In addition, prior experience with programming is arguably the best predictor of performance in introductory CS classrooms [10, 36]. Integrated computing serves to give all students prior experience with programming to support their early learning of computing concepts.

Integrating computing into non-CS formal education, especially in primary and secondary (i.e., K-12) education, serves multiple goals. One potential goal is to make computing tools and solutions accessible to teachers and students in non-CS classrooms to improve instruction and learning in other disciplines [20, 30, 46]. Another potential goal is to give all students experience with computing, especially early exposure in primary/elementary school, so that they can develop computational literacy and make informed decisions about whether to pursue standalone CS classwork [29, 31]. To complement a previous analysis that analyzed how integrated activities served the first goal [20], the current analysis focused on the second goal by exploring how integrated computing teaches CS. The study examined integrated computing curricula to determine how they teach CS concepts and practices, as defined in the K-12 CS Framework, in primary and lower secondary (i.e., K-8) classrooms.

It is important to acknowledge that these are not the only two potential goals of integrated computing. Another goal is introducing basic computing ideas to new audiences to provide a new medium for discovery or self-expression. Yet another is to develop a CS identity in young learners and enthusiasm for future CS learning, inside or outside of standalone CS classrooms. Thus, the goals of the curricula analyzed in this study might not align with the goal of this study–to explore how integrated computing curricula teach CS. Still, these curricula are often treated as CS instruction by administrators, policymakers, and teachers of future standalone CS courses. This categorization persists despite the need for these curricula to primarily serve non-CS learning as part of instruction in other disciplines, which are often tied to standardized testing, like language arts and math. Thus, the point of this study is not to criticize the goals or products of curricula designers but instead to examine how well the curricula serve a major goal that others place on them.

2 Related Work

2.1 Expanding CS Education and Integrated CS

The past decade has seen fast growth in demand for CS education. The discipline has transformed from the "dismal state" that researchers found it in the 2010s to a significant increase in students taking CS courses, numerous teacher certification programs, and districts and states making CS a graduation requirement ([16], p. 146). Much of this comes from an increased understanding of the importance of CS in K-12 education. Vogel and colleagues [44] compiled 161 arguments for CS education in K-12 schooling, including those arguing that CS education equips "individuals with the specific skills needed to secure a job in the growing technology sector," promotes "social justice impacts and the dismantling of social hierarchies," and underlies "every industry and discipline" (p. 611). The researchers rooted these 161 arguments within seven core impact areas, which became the basis for the CS Vision Framework that motivates many current K-12 CS education initiatives [38, 44].

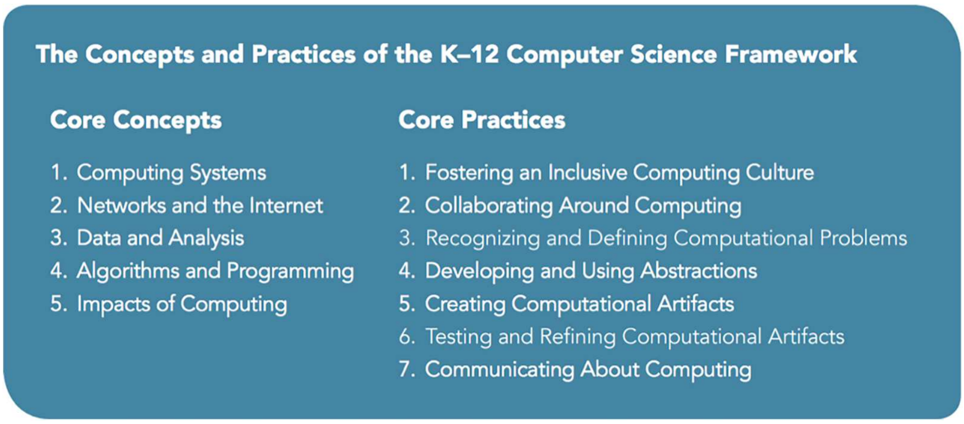
As the relevance of CS education has expanded, the prevalence of CS education has expanded throughout K-12 education [13]. Teaching CS to young students has particularly grown due to advances in the field, such as the development of block-based coding languages [4], evidence of children's cognitive development showing that students as young as four can understand computing concepts [8], and an awareness of the universality of computational thinking [47] and computational literacies [4, 16]. Broadening participation in computing also spurred primary grade initiatives, such as addressing the gender gap before it widens in middle school [42] or helping students develop an interest in CS before they start picking after-school activities and electives in lower secondary/middle school [6]. In fact, some researchers and educators believe that introducing CS in secondary school is too late to provide equitable access, which could affect students' choice of future CS careers [24]. To incorporate CS education into primary education and help all students engage with it, many schools choose to integrate CS education into existing, non-CS core classes, like language arts, math, or science [41, 46].

In countries that begin CS education in primary school, Oda and colleagues [25] found three broad approaches to CS education: (a) standalone CS courses, (b) integrated CS embedded within multiple subjects, or (c) integrated CS as an independent module with a cross-curricular approach. Many barriers prevent implementing standalone CS education in primary school (e.g., lack of curricular time, educator expertise, and inability to enact 1-1 computing), and integrated CS has been used to mitigate these barriers [15, 39]. Integrated CS in primary and lower secondary (i.e., K-8) education has been found to promote students' computational literacy in all classrooms without adding "one more thing" for teachers, who already have a packed teaching schedule [19].

To not be "one more thing," integrated computing must replace existing instruction and activities. Thus, as a part of instruction in non-CS classrooms, integrated CS activities often must serve non-CS learning objectives first, affecting the CS concepts and practices that are appropriate and the extent to which they are taught [14]. This prioritization is especially true for core classes tied to standardized testing, such as language arts and math, which often face pressure from administrators and policymakers to improve outcomes. Still, integrated CS is also commonly treated as CS education by the same administrators and policymakers and potentially by future CS teachers who might expect to build upon prior knowledge developed through these activities. For this reason, it is important to examine how integrated computing curricula teach CS concepts and practices, given that they are likely to be adopted only when they primarily serve non-CS learning objectives.

2.2 K-12 CS Framework

With numerous rationales driving CS education's expansion, educators, administrators, and policymakers have grappled with the goals of CS education for young learners. As Parker and DeLyser [27] noted, researchers and stakeholders have actively sought to identify the essential CS knowledge and skills students should acquire and the CS practices that should be integrated into K-12 CS pathways. In response to these needs, the K-12 CS Framework was developed to fill the gap and provide guidance for states, districts, schools, and industry in the development of standards and curricula ([27]). The K-12 CS framework promotes an understanding of CS in which all students can explore complex issues in CS to solve problems while developing personally and socially relevant computational artifacts [7]. The K-12 CS Framework's core CS concepts, i.e., "what students should know," and CS practices, i.e., "what students should do," are organized by four grade bands: grades K-2/ages 5–7, grades 3–5/ages 8–10, grades 6–8/ages 11–13, and grades 9–12/ages 14–18. These concepts and practices are designed to let students meaningfully engage with CS in ways that grow more sophisticated over time ([7], p. 3).



The Concepts and Practices of the K–12 Computer Science Framework

Core Concepts

1. Computing Systems
2. Networks and the Internet
3. Data and Analysis
4. Algorithms and Programming
5. Impacts of Computing

Core Practices

1. Fostering an Inclusive Computing Culture
2. Collaborating Around Computing
3. Recognizing and Defining Computational Problems
4. Developing and Using Abstractions
5. Creating Computational Artifacts
6. Testing and Refining Computational Artifacts
7. Communicating About Computing

Fig. 1. K-12 CS framework concepts and practices [7]. Reprinted under creative commons license CC BY-NC-SA 4.0, no changes made.

2.2.1 Development and Goals of the Framework. The development of the K-12 CS Framework was a community-led effort, spearheaded by a diverse team of 27 writers and 25 advisors of varying genders, types of experience, and ethnicities and from a variety of states, school districts, companies, and nonprofits (albeit primarily in the United States) [7]. This comprehensive development process spanned one year, from October 2015–2016, and included advisor workshops, writing workshops, stakeholder convenings, and public review sessions. This project overlapped with the CS Teacher Association's (CSTA) K-12 CS Standards revisions in 2015–2016, and some advisors and writers simultaneously served on both independent projects to align them [7]). Various content documents inspired the development of the K-12 CS Framework, such as the National Research Council's K-12 Framework for Science, the Common Core Mathematics and English Standards, and National standards from countries, such as the United Kingdom, Germany, Poland, New Zealand, and Israel [27]. Furthermore, this work builds upon A Model Curriculum for K-12 CS Standards and the CSTA K-12 CS Standards [7]. The team's rigorous development process resulted in the delineation of five core concepts and seven core practices (See Figure 1), each designed to be infused in a classroom environment where "practice and project-based learning reinforce conceptual knowledge" [27], p. 454).

In regards to implementing the K-12 CS Framework's concepts and practices with young learners, the framework's developers ensure that this is not only possible but necessary because it "builds foundational knowledge and understandings: which help children engage with CS in their later years" ([7], p. 184). The K-12 CS Framework was developed with an understanding that core ideas of socio-emotional learning, pattern recognition, problem-solving, representation, and sequencing are already found within early education in math, science, and literacy and that these ideas are also foundational to CS. For example, the idea of representation, in which humans represent concepts with symbols is found in both literacy and CS (and emphasized in the K-12 CS Framework's practices 4, 5, and 7 [7]). Given these features of the K-12 CS Framework, we felt it was an appropriate framework upon which to base our exploration of how CS is taught in integrated CS curricula.

2.3 The State of Integrated Computing in K-8 Schools

2.3.1 Integrated CS Curricula. Although the K-12 CS Framework suggests that integrated computing might be particularly compelling in primary schools, it is still a relatively new approach. Rich et al. [35] surveyed international trends in coding in K-8 schools and found that computing/coding was twice as likely to be taught in a standalone CS course than integrated. When CS education was integrated, it was most commonly integrated with math followed by science, language arts, engineering, and social science [35]. However, integrated CS is growing and increasingly recommended in primary schools. When Guo and Ottenbreit-Leftwich [11] surveyed CS standards across the United States, where most existing CS standards started in Kindergarten, they found some states recommended CS integration with other disciplines. They also highlighted the benefits of such a recommendation, including taking advantage of the similarity of CS content with other disciplines and capitalizing on the possibility of developing innovative CS curricula [11]. Franklin et al. [9] provided some practical recommendations from their experience implementing an integrated CS curriculum for 4th–6th graders/ages 9–11, such as using the integrated curricula to reinforce, but not depend on, other content knowledge. Furthermore, project-based approaches to CS integration in a variety of academic subjects were shown to successfully improve students' autonomy, learning, collaboration, and social engagement [5, 26]. While the application of CS skills can take many interdisciplinary forms, our current work focuses on how these integrated curricula teach CS concepts and practices.

2.3.2 CS Concepts and Practices in K-8 Curricula. Our understanding of what constitutes computational literacy and, thus, should be taught in integrated CS is still evolving, but the K-12 CS Framework seems to represent CS education policy well. Guo and Ottenbreit-Leftwich [11] found all five of the K-12 CS Framework concepts within the wording of US states' CS standards. As Oda and colleagues [25] surveyed international trends in K-12 CS education using the K-12 CS Framework, they found that most countries' primary curricula had students begin by engaging with the framework's concepts of algorithms and program development, the impact of computing, and the practice of developing of computational artifacts. For older students, common learning objectives also included concepts like networks and the Internet and practices like recognizing problems, developing abstractions, and testing and refining computational artifacts. Oda et al. also recommended future practitioners integrate concepts from existing subjects with CS, especially to help students develop their computational thinking as it is at "the heart of the CS practices," explicit in the K-12 CS Framework's practices 3–6 and complemented by the other practices ([7], p. 67).

Given that the K-12 CS Framework's concepts and practices align with initiatives and policies to teach CS education in primary/elementary and lower secondary/middle schools, the current analysis uses this framework to examine how CS is taught in integrated CS curricula for those students. Particularly, the analysis examined which concepts and practices are taught and to what extent. It builds upon a previous study that examined which programming concepts were taught in integrated CS activities (rather than full curricula) and concluded that the most commonly used programming concepts did not align well with those emphasized in introductory programming courses [20]. Besides the length of CS instruction (i.e., activities vs. curricula), the other main difference between studies was which field's learning objectives were primary. The former study included only integrated activities with non-CS disciplinary learning objectives while the current study included only integrated curricula with CS learning objectives. Because the former study focused on non-CS learning, the types of programming concepts used might have been appropriate, but the authors did note that "this finding does have implications for students who are entering introductory programming courses and how experience with integrated computing activities is treated as prior knowledge," ([20], p. 198). As noted above, prior experience with programming is one of the best predictors of future performance in introductory computing classrooms [10, 36]. If students do not get adequate exposure in early grades, they may not be prepared for future CS coursework. With this in mind, the current study explicitly explores the CS learning objectives and instruction in integrated CS curricula to examine how they teach students CS concepts and practices and prepare students for future learning in CS.

Our research questions were:

—Which CS concepts and practices are introduced in computing curricula that are integrated into other disciplines?
—How extensively are CS concepts and practices taught in integrated computing curricula?

3 Method

To examine the CS concepts and practices taught in integrated CS curricula and the depth at which they were taught, the researchers conducted a content analysis of curricula that met our inclusion criteria (see Section 3.1). We based our scoring of curricula on the K-12 CS Framework [7] with some modifications to capture concepts at a finer-grain or to better represent the data available (see Section 3.3). Tradeoffs in research design decisions are discussed throughout this section, and limitations and areas for future work are examined in the Discussion section.

3.1 Search and Inclusion Criteria

While the current analysis used many of the same tools as a systematic literature review to find curricula, it is not a systematic review. Unlike in literature reviews, there are no databases of integrated computing curricula to search systematically. Instead, we searched the literature for evidence-based curricula. We first searched the ACM Digital Library for articles with "(integration OR integrated) AND (computing OR 'computer science' OR CS) AND curriculum" to find curricula that had been studied. We repeated the search with Google Scholar in journals that include "(computing OR 'CS' OR computers) AND (education OR research)" in their title, such as *Computer Science Education, Computers & Education*, and *Journal of Educational Computing Research*. Last, we examined each entry in CSforAll's curriculum directory for curricula that matched our inclusion criteria.

We used four inclusion criteria to select curricula for analysis. The first two criteria are necessitated by the research questions. To answer the first research question, about which CS concepts and practices are introduced in integrated computing curricula, our first criterion was that curricula must include learning objectives related to programming concepts, non-programming CS concepts, or CS practices. These CS concepts and practices were based on the K-12 CS Framework [7], and many of the curricula included explicit connections to the K-12 CS Framework or to the CS Teacher Association's CS standards, which are based on the framework. More information about how the framework was used in the analysis can be found in the Framework Development section (Section 3.3).

To answer the second research question, about the extent of instruction about CS concepts and practices taught in integrated computing, our second criterion was that curricula must include at least five hours of instruction. This criterion was necessary to explore how CS instruction develops and evolves over several hours of instruction. It also serves to complement the previous analysis, which analyzed all activities regardless of length and ranged from less than an hour to multiple days [20].

Fortunately, many integrated computing curricula fit these criteria. To make the scope of work reasonable, we added two additional inclusion criteria: curricula must be free to access and curricula must be for primary and lower secondary school (i.e., grades K-8 in the United States). Because our goal was to examine how these curricula teach concepts that are foundational to standalone CS courses, which often start in upper secondary (i.e., high) school, these inclusion criteria made the scope of the project manageable while best serving the goals of the analysis and ensuring accessibility. For both criteria, we erred on the side of inclusion. For example, we included free

Table 1. List of Included Curricula

Curriculum Creator	Curriculum	Age Range	Programming language
MIT Media Lab	MIT AI Ethics Education Curriculum	10–13 years	None
Elementary Computing for All (ECforALL)	ECforALL Curriculum—Act 1	8–9 years	Scratch
	ECforALL Curriculum—Act 2	9–10 years	Scratch
youcubed.org	Data Science Lessons	Any	None
DevTech Research Group	Coding as Another Language (CAL)—KIBO (Pre-Kindergarten)	4–5 years	KIBO
	CAL—KIBO (Kindergarten)	5–6 years	KIBO
	CAL—KIBO (Grade 1)	6–7 years	KIBO
	CAL—KIBO (Grade 2)	7–8 years	KIBO
	CAL—Scratch Jr. (Kindergarten)	5–6 years	Scratch Jr.
	CAL—Scratch Jr. (Grade 1)	6–7 years	Scratch Jr.
	CAL—Scratch Jr. (Grade 2)	7–8 years	Scratch Jr.
Everyday Computing	Action Fractions (Grade 3)	8- 9 years	Scratch
	Action Fractions (Grade 4)	9–10 years	Scratch
CS+ Elementary	CS + Fables	8–9 years	Scratch
	CS + Data	8–9 years	Scratch
	CS + Earth	9–10 years	Scratch
	CS + Community	9–10 years	Scratch
Google CS First	CS First—Storytelling	8–12 years	Scratch
	CS First—Friends	8–12 years	Scratch
	CS First—Music and Sound	8–12 years	Scratch
	CS First—Fashion and Design	8–12 years	Scratch
	CS First—Art	8–12 years	Scratch
CSinSF.org	K-2 CS Curriculum—Red	5–7 years	Bee Bot, Scratch Jr
	K-2 CS Curriculum—Orange	5–7 years	Bee Bot, Scratch Jr
	K-2 CS Curriculum—Yellow	5–7 years	Bee Bot, Scratch Jr
Bootstrap	Bootstrap—Algebra	12–15 years	Pyret
	Bootstrap—Data Science	12–18 years	Pyret

MIT, Massachusetts Institute of Technology.

curricula that used robotics, such as Bee Bots or KIBO, because schools might receive grants or gifts to purchase robots. In addition, courses that are typically taught in upper secondary school were included if they are commonly taught earlier in an accelerated track, such as Algebra.

The final dataset included 27 curricula from 9 different curricula creators (see Table 1). Within these curricula, there were 519 lessons with over 600 hours of content. Some follow-up curricula to those listed below, such as Scratch encore or CSinSF's green and blue curricula, were excluded from the analysis because they focus exclusively on CS learning objectives and, thus, were not considered integrated computing. Other common integrated curricula, such as Project Growing Up Thinking Scientifically or University College London's Scratch math, were excluded based on our first criteria for not including CS learning objectives.

3.2 Unit of Analysis and Scorers

These 27 curricula were all organized as a compilation of lessons. Lessons were designed to be used during one class period, typically 30 minutes to an hour in length. Lessons were typically discrete, except that final projects would often stretch over multiple lessons. Lessons included multiple activities, such as reviewing previous concepts, introducing new concepts, or completing tasks. For this analysis, the unit of analysis was the lesson. This level of analysis allowed us to address our research questions and describe how concepts were taught and revisited over curricula. Further, each lesson typically included a lesson plan describing learning objectives and activities, instructional artifacts like presentations or worksheet activities, and computational artifacts like starter projects or sample projects. These materials provided rich and redundant information about the concepts being taught. Information about which practices were taught was often less explicit, which accounts for the lower interrater reliability discussed in Section 3.3.

The five scorers in this analysis were the five authors of this article. Four of the five scorers were professors who have studied CS education for at least nine years each. Their Ph.D.s are in the areas of CS, curriculum and instruction, instructional design, and instructional technology. The other scorer was a doctoral student studying CS education in an instructional technology Ph.D. program.

3.3 Framework Development and Scoring Training and Procedure

This section describes the process used to develop the frameworks used for scoring and the procedure used to train scorers and score the data. Details of the final frameworks can be found in the next section. The authors based the initial frameworks of codes on the K-12 CS Framework to analyze curricula for programming concepts (e.g., variables, loops, and conditionals), non-programming CS concepts (e.g., computing systems, data and analysis, and impacts of computing), and CS practices (e.g., fostering an inclusive computing culture, collaborating around computing).

All five authors applied these three frameworks to scoring the initial 10% of data (58 hours of lessons), working independently. The group then discussed the limitations of the frameworks based on the data, including features of the curricula that the frameworks did not capture, captured poorly, or captured redundantly. For example, after this discussion, the modularity concept was removed from the programming concepts framework because decomposition tasks were not often explicitly stated in the lessons (i.e., captured poorly), and when they were, they would be captured by the decomposition CS practice (i.e., captured redundantly).

After refining the frameworks based on this discussion, the scorers applied the revised frameworks to the same initial 10% of the data, working independently. From this point, the authors worked in pairs so that two people specialized in using the same framework. As a result, across all of the curricula, two authors applied the programming concepts framework, two authors applied the non-programming concepts framework, and two authors applied the CS practices framework (i.e., one author scored both non-programming concepts and CS practices).

From this second round of scoring, the pairs discussed any further refinements needed to the frameworks and began to compare interrater reliability to refine the scoring technique. In this stage, the scorers of the programming concepts recognized the need to implement additional notation to capture how programming concepts were being used within programming activities. They added notation based on the Use-Modify-Create model of learning programming to differentiate between when learners were using programs given to them, modifying programs, or creating their own [18]. These notations allowed the team to better address the second research question about the extent of instruction, and a full explanation of the notations used is in the next section.

After refining the frameworks and scoring techniques based on the second round of discussion, the frameworks had reached their final versions. The pairs then applied the final frameworks to 20% of the data (116 hours of lessons) with two scorers rating the same data using the same framework. Their scores were then compared to calculate interrater reliability.

— Programming concepts: Actual agreement = 98%; Cohen's Kappa = 0.92
— CS practices: Actual agreement = 80%; Cohen's Kappa = 0.68
— Non-programming concepts: Actual agreement = 97%; Cohen's Kappa = 0.79

Cohen's Kappa was used to evaluate interrater reliability because it is appropriate for nominal scoring techniques. It compares the actual agreement to the probability of agreement by chance, so both the actual agreement and Kappa values are provided. A Cohen's Kappa of 0.61–0.8 is considered high agreement, and above 0.81 is considered near perfect agreement [23]. Given all teams had at least high agreement, the remaining scoring was divided among authors and completed independently.

3.3.1 Programming Concepts Analysis Framework. The programming concepts framework started from the Algorithms and Programming section of the K-12 CS Framework. However, the team found the categories too high-level (i.e., algorithms, variables, control, modularity, and program development) to sufficiently address the research questions, so these categories were expanded into more detailed concepts based on the framework developed by Margulieux et al. [20]. For example, within the variables category, concepts for data types, initializing and calling variables, and using lists are differentiated. After the initial framework revision, the modularity and program development categories were excluded from the programming concepts framework. The modularity decision is already explained in the previous section (Section 3.3). Program development was excluded because the stages of program development were not often explicitly stated in the lessons, and the lessons typically involved all three levels of development–designing, implementing, and reviewing/debugging. Thus, the scorers would have had to infer the stages of program development in the lesson, reducing objectivity, and the lack of discrimination among lessons made this information unuseful. The final framework included concepts related to data input, data output, variables and lists, loops, events, conditionals, operators, functions, and multimedia components (see Table 2). While a detailed definition and example of each concept are too expansive to reprint here, they can be found along with justifications for their inclusion in Margulieux et al. [20].

Each lesson could receive different notations for how the concepts in the framework were used. These notations are mutually exclusive, and the most advanced interaction present was applied.

- Nothing (least advanced)—students do not use the concept, or they use a program that has the concept in the background but do not interact with it.
- Use—students are given code that they run to understand how it works or answer questions about it, but they do not modify it.
- Modify—students are given code and asked to change something about it, such as changing the order of code or modifying the parameters.
- Create—students create a new algorithm, which might involve dragging new blocks into the workspace, writing new code, or repurposing existing code to create a new algorithm.
- Unguided (most advanced)—an advanced version of create in which students are given specifications for the concepts to include in a program but not how to implement them, such as the program must include three sprites.

3.3.2 CS Practices Analysis Framework. The analysis framework for CS practices was based on the K-12 CS Framework [7]. The analysis framework includes seven core practices of CS as seen in Figure 1. Each core practice comprises three to four practice statements that outline the computing competencies students should possess by the end of Grade 12. Unlike the CS concepts, which are divided into subconcepts and statements by grade bands, the CS practices are written in a narrative format that emphasizes the progression of students' development of the CS practices.

The team used the seven core practices of CS and their practice statements as codes and sub-codes in the coding scheme. To align with the scope of the curriculum analysis (up to Grade 8/age 13), the knowledge and skills in the progression statements at the early or middle stages of the progression were more emphasized than the entire progression statements for up to Grade 12/age 18. For

Table 2. Frequency of Programming Concepts Taught in Integrated Lessons and Type of Instruction

Category	Concept	Use	Modify	Create	Unguided	Total (of 346)
Input	Input String	4	2	3	0	9 (3%)
	Input Variable	4	0	3	0	7 (2%)
Output	Output String	20	6	42	14	82 (24%)
	Output Variable	4	1	9	0	14 (4%)
	Output Combo	1	1	0	0	2 (<1%)
Data and Variables	Data Types	1	0	42	0	43 (12%)
	Initialize/Call Variable	11	5	21	1	38 (11%)
	List	0	0	2	0	2 (<1%)
Loops	For Loop	28	13	38	12	91 (26%)
	While Loop	2	0	5	2	9 (3%)
	Loop Index Variable	0	1	1	0	2 (<1%)
Events	Event	30	2	56	20	108 (31%)
Conditionals	If-Then Conditional	7	5	20	6	38 (11%)
	If-Else Conditional	3	3	7	0	13 (4%)
Operators	Arithmetic Operator	2	2	13	1	18 (5%)
	Relational Operator	10	3	16	1	30 (9%)
	Boolean Operator	1	1	3	0	5 (1%)
Functions	Define/Call Function	16	3	43	2	64 (18%)
	Function Parameter	4	16	46	16	82 (24%)
Multimedia	Movement	41	7	72	50	170 (49%)
	Properties	37	7	74	15	133 (38%)
	Sprite/Object	113	6	80	35	234 (68%)

The most common category and concepts are highlighted in dark green, while common, but less frequent, concepts are highlighted in a light green.

example, in Practice 4: Developing and Using Abstraction, the team included the first practice statement, "Extract common features from a set of interrelated processes or complex phenomena," as the description for Practice 4 in the coding sheet, which states that "Students at all grade levels should be able to recognize patterns. Young learners should be able to learn to identify and describe repeated sequences in data or code through analogy to visual patterns or physical sequences of objects. As they progress, students should identify patterns as opportunities for abstraction, such as recognizing repeated patterns of code that could be more efficiently implemented as a loop." However, the last part of the progress statement, "Eventually, students should extract common features from more complex phenomena or processes...When working with data, students should be able to identify important aspects and find patterns in related datasets, such as crop output, fertilization methods, and climate conditions," was not included in the coding scheme. A similar process was applied to identifying the descriptions of other practice statements in the coding scheme.

3.3.3 Non-Programming Concept Analysis Framework. The non-programming CS concepts comprise the remaining K-12 CS Framework's core concepts: Computing Systems, Networks and the Internet, Data and Analysis, and Impacts of Computing. The team adopted the K-12 CS Framework's subconcepts and concept statements to guide our analysis. Each subconcept is organized into three levels: by Grade 2 (age 7), by Grade 5 (age 10), and by Grade 8 (age 13), which the research team refers to as introductory, intermediate, and advanced levels.

During the initial coding phase, the scorers faced inconsistencies in scoring concepts and subconcepts. Primarily, these inconsistencies stemmed from whether lessons explicitly taught concepts given in their descriptions. To overcome this, the research team decided on scoring based on the learning objectives and goals for the lesson instead of relying solely on the topics and keywords in the lesson description. For example, the scorers would code the devices concept when a lesson introduces different components of a KIBO robot and how it follows instructions to perform actions. However, the scorers would not code the devices concept when a lesson uses a KIBO robot in a storytelling project, as the primary focus is on the application of KIBO rather than the concept of a device.

4 Results

For the data created from this analysis, please find the complete dataset at https://doi.org/10.5061/dryad.j6q573nnt

4.1 Programming Concepts Taught in Curricula

4.1.1 Frequency of Programming Concepts in Instruction. Each lesson was scored for which programming concepts were included and at which level they were taught (i.e., use, modify, create, unguided, see Table 2). For all curricula, we treated final projects that spanned multiple lessons as one lesson because the programming instruction and activities were not unique in each lesson. Thus, counting them independently would have resulted in overcounting concepts. Final projects are not collapsed for the other frameworks because each lesson provided unique instruction. After collapsing final project lessons, 519 lessons yielded 453 unique lessons, and 346 of them included programming activities.

The programming concepts used in these curricula primarily focused on multimedia components (i.e., sprites or other multimedia objects (68% of lessons with programming), their movement (49%), and their properties (38%)). Other common, but less frequent, concepts were events (31%), outputs (28%; especially string-only outputs—24%), and for loops (26%). The frequency of these concepts is most likely due to the overwhelming use of Scratch in these curricula. These lessons in Scratch foregrounded multimedia components to visualize non-CS concepts or processes. Further, the other common concepts primarily served to interact with the multimedia components (e.g., tap a sprite to trigger an event, communicate through a sprite using an output, or repeat a sprite's actions with a loop). Thus, the programming paradigm seemed to influence the concepts used.

Similarly, functions (18%) and function parameters (24%) were also common, though primarily in curricula that used Pyret, a functional programming language. 42 of 64 function uses are from Bootstrap curricula, and 42 of 81 function parameters are from Bootstrap curricula. The other uses of function parameters were primarily modifying them (16 of the remaining 39 cases). Across all curricula, the majority of functions used in student programs called a function that was defined by the curricula designers. These functions tended to allow students to complete complex tasks, such as create complicated visualizations. Functions that students defined on their own were much simpler, such as calculating a value based on given parameters.

Programming concepts that were used less commonly came from the input, data and variables, conditionals, and operators categories. Only 5% of lessons included an input from a user (i.e., data entered while the program was running). 11% of lessons used a variable, and less than 1% used a list. In Bootstrap curricula, data types were addressed frequently, but they were not addressed in other curricula. Some form of conditional was used in 15% of lessons as was some form of operator. A comparison of these frequencies with the previous analysis of integrated computing activities [20] is examined in the discussion.

4.1.2 Extent of Programming Concepts in Instruction. For our second research question, we explored how the concepts included in these curricula were taught with the Use-Modify-Create framework [18] and an additional Unguided category. These codes were mutually exclusive, so each lesson was scored for only the least scaffolded way that concepts were used to avoid overcounting concepts (i.e., `Use` < `Modify` < `Create` < `Unguided`). Table 2 shows the number of lessons teaching each concept across these categories.

`Create` was by far the most common way to engage with the concepts. For almost every concept, `Create` was the most popular category. `Use` was also common, especially when using sprites in existing Scratch programs or using advanced code segments created by the curriculum developers, such as using a pre-defined function. Surprisingly, `Modify` was not common as an endpoint for lessons, except in function parameters. However, given that codes were mutually exclusive, `Modify` was more common in the lessons than the data represents because within a lesson, students might start by using or modifying a program and be creating by the end. In this scenario, the lessons would be scored as `Create`. Curricula that followed the Title, Instructions, Purpose, and Play and Sprites, Events, Explore approach [37] typically advanced through the Use-Modify-Create cycle within a single lesson. From the CS practices data in the following section, modifying a program was part of about a third of the lessons.

Based on different curricula features or creators, the curricula followed a few different patterns for how concepts were taught. Curricula that used programmable robots or Scratch Jr (i.e., **Coding as Another Language (CAL)** and CSinSF) were limited in the concepts that they could use–multimedia components, loops, events, and string output primarily. These curricula contributed to these concepts being the most common concepts. They tended to emphasize more unguided use of concepts, perhaps due to the constrained problem-solving space afforded by these programming paradigms.

Another type of curriculum asked students to create programs, using scaffolded instruction, with the consistently same concepts across the curricula (i.e., ECforAll, Bootstrap, and CS First). For example, Scratch-based curricula consistently asked students to create programs using multimedia components, and Bootstrap curricula consistently asked students to create programs by calling functions across all lessons. Other concepts were applied a few times and usually within a unit or set of lessons. For example, a unit might ask students to create programs using conditionals or operators, but then the next unit would not continue to apply these concepts. When these curricula had final projects, they did not typically require the application of concepts beyond those that were well represented across the curriculum.

The last category of curricula applied a variety of concepts and applied them more consistently across lessons (i.e., Action Fractions and the CS+ curricula). In this category, the level of engagement was typically `Use` or `Modify`, and it less often included `Create` or `Unguided` use of the concepts. This approach resulted in students using more complex programs but not necessarily in creating them. The tradeoffs of these different strategies and other viable strategies are examined in the discussion section.

4.2 CS Practices Taught in Curricula

4.2.1 Frequency of CS Practices in Integrated Lessons. To examine which CS practices were integrated into the 27 curricula, we scored the 519 lessons using the seven core practices and practice statements for each. Based on the distribution, we categorized the frequency of each practice statement as frequent (> 50%), common (25–50%), infrequent (10–24%), and rare (< 10%; see Table 3). While the majority of practice statements were either infrequently (26.5%) or rarely (43%) found throughout the lessons, some were frequently or commonly found.

Table 3. Frequency of CS Practices Taught in Integrated Lessons

Core Practice	Practice Statement	Count	%
Practice 1. Fostering an Inclusive Computing Culture	1. Include the unique perspectives of others and reflect on one's own perspectives when designing and developing computational products.	**487**	**93.3**
	2. Address the needs of diverse end users during the design process to produce artifacts with broad accessibility and usability.	13	2.3
	3. Employ self- and peer-advocacy to address bias in interactions, product design, and development methods.	4	0.8
Practice 2. Collaborating Around Computing	1. Cultivate working relationships with individuals possessing diverse perspectives, skills, and personalities.	**321**	**61.5**
	2. Create team norms, expectations, and equitable workloads to increase efficiency and effectiveness.	53	10.2
	3. Solicit and incorporate feedback from, and provide constructive feedback to, team members and other stakeholders	50	9.6
	4. Evaluate and select technological tools that can be used to collaborate on a project.	89	17.1
Practice 3. Recognizing and Defining Computational Problems	1. Identify complex, interdisciplinary, real-world problems that can be solved computationally.	**157**	**30.3**
	2. Decompose complex real-world problems into manageable subproblems that could integrate existing solutions or procedures.	75	14.5
	3. Evaluate whether it is appropriate and feasible to solve a problem computationally.	6	1.2
Practice 4. Developing and Using Abstractions	1. Extract common features from a set of interrelated processes or complex phenomena.	**152**	**29.3**
	2. Evaluate existing technological functionalities and incorporate them into new designs.	18	3.5
	3. Create modules and develop points of interaction that can apply to multiple situations and reduce complexity.	21	4.0
	4. Model phenomena and processes and simulate systems to understand and evaluate potential outcomes.	13	2.5
Practice 5. Creating Computational Artifacts	1. Plan the development of a computational artifact using an iterative process that includes reflection on and modification of the plan, taking into account key features, time and resource constraints, and user expectations.	102	19.7
	2. Create a computational artifact for practical intent, personal expression, or to address a societal issue.	**154**	**29.7**
	3. Modify an existing artifact to improve or customize it.	**159**	**30.6**
Practice 6. Testing and Refining Computational Artifacts	1. Systematically test computational artifacts by considering all scenarios and using test cases.	52	10.0
	2. Identify and fix errors using a systematic process	57	11.0
	3. Evaluate and refine a computational artifact multiple times to enhance its performance, reliability, usability, and accessibility.	11	2.1

(Continued)

Table 3. Continued

Core Practice	Practice Statement	Count	%
Practice 7. Communicating About Computing	1. Select, organize, and interpret large data sets from multiple sources to support a claim.	35	6.7
	2. Describe, justify, and document computational processes and solutions using appropriate terminology consistent with the intended audience and purpose.	**272**	**52.4**
	3. Articulate ideas responsibly by observing intellectual property rights and giving appropriate attribution	10	1.9

Dark green indicates frequent (>50%) usage, green indicates common (25–50%) usage, and light green indicates infrequent (10–24%) usage.

Frequently found practice statements in the curricula were including the unique perspectives of others (93%), cultivating working relationships (62%), and describing, justifying, and documenting computational processes and solutions (52%). All of these involve a student sharing their perspective, reflecting on their computing learning experiences, and/or justifying their designs and decisions in the context of peer interactions and communications. In particular, most lessons across the curricula encourage students to include the unique perspectives of others when reflecting on learning and design.

Commonly found practice statements that appeared in CS curricula were modifying an existing artifact (31%), creating a computational artifact (30%), identifying complex, interdisciplinary, real-world problems (30%), and extracting common features from a set of interrelated processes or complex phenomena (29%). Many of the modify and create practice statements were found in programming activities in Scratch, especially when students worked from a pre-designed starter project that they modified and built upon. Many instances of the practice statement identifying complex, interdisciplinary, real-world problems that can be solved computationally involved comparing actions in students' everyday lives to a programming activity, such as comparing events in a Scratch program to their alarm clock (i.e., CS First—Art) or comparing KIBO's sound and light sensors to their ears and eyes (i.e., CAL KIBO—2nd Grade). Sometimes, this practice asked students to compare the activity to real jobs, like comparing the steps in a computational activity to the steps engineers take to iterate a prototype (CAL KIBO—Pre-K), animators take to create movies (CS First—Art), or authors take to write a story (CSinSF K-2—Red). Last, the extracting common features practice statement was seen in many activities asking students to notice how certain programming concepts were used, such as having students notice how conditionals or loops were used in projects.

Some infrequent practices found in the curricula were planning the development of a computational artifact (20%), evaluating and selecting technological tools (17%), decomposing complex real-world problems (15%), identifying and fixing errors (11%), creating team norms, expectations, and equitable workloads (10%), systematically testing computational artifacts (10%), and soliciting and incorporating feedback (10%). These practices align with higher-level program design skills and elements of project management. Given the complexity of most of the programs developed in these curricula (i.e., either completed within a single lesson or as an individual final project), implementing these practices in the curricula would likely not be a good tradeoff of effort and time. In the cases that students were designing programs, students used a planning guide to brainstorm their thoughts before creating or modifying a Scratch program, usually before a final project (i.e., CS+Community). Similarly, practices for fixing and testing programs were emphasized in a few of the lessons per curriculum.

The K-12 CS Framework's creators infused evaluative practices within five of the practice statements. However, many of the rarely found practices centered around evaluative practices, such as evaluating the appropriateness of solving processes computationally (1.2%), evaluating existing technological functionalities (3.5%), evaluating simulated processes' outcomes (2.5%), and evaluating and refining a computational artifact (2.1%). Again, this finding is likely a reflection of the complexity of programs created in the lessons. The following practice statements were also rarely found in the integrated curricula: selecting, organizing, and interpreting large datasets (6.7%), creating modules and developing points of interaction (4.0%), addressing the needs of diverse end users (2.3%), articulating ideas responsibly (1.9%), and employing self- and peer-advocacy (0.8%). The rare practice statements of selecting, organizing, and interpreting large datasets and creating modules and developing points of interaction were mainly found in the data science curricula. In general, though, these practices were also not well-aligned with the typical integrated computing paradigms in these curricula.

4.2.2 Extent of CS Practices in Integrated Lessons. All integrated K-8 CS curricula addressed CS practices, but the extent varied. Some curricula addressed all seven core practices across the CS lessons (i.e., CS First—Storytelling/Music and Sound/, Bootstrap—Algebra, K-2 CS curriculum, action fractions, CAL—KIBO and Scratch, **Massachusetts Institute of Technology's (MIT)** AI Ethics, Youcubed—Data Science), while others integrated more targeted practices.

Within frequently integrated CS core practices (i.e., Practices 1, 4, and 7), only one of the practice statements was addressed repeatedly. As with Practice 4, about developing and using abstraction, many lessons included activities in which students practiced extracting common features of complex processes and phenomena with real-world examples (4.1). Yet there were minimal instances of other aspects of abstraction skills, such as evaluating existing technological functionalities (4.2), creating modules and developing points of interaction (4.3), and modeling phenomena and processes and simulating systems (4.4). Following this pattern, most of the practices, except Practice 5, had some practice statements included more extensively than others.

This finding suggests more advanced applications of these practices are not being reached in integrated lessons. Because these lessons are taught in non-CS classrooms, this finding is not necessarily problematic. Yet for all of the integrated CS curricula, most of the CS practices were found within lessons that used programming, either within Scratch, robotics, or online data science environments. Despite the practices being CS practices, rather than programming practices, they were typically taught in the context of programming.

4.3 Non-Programming CS Concepts Taught in Curricula

4.3.1 Frequency of Non-Programming Concepts Taught in Integrated Lessons. In the 519 lessons, most did not address non-programming concepts, at least not at nearly the same rate as programming concepts or practices. Based on the distribution of scores, we categorized the frequency of sub-concepts as most frequent (> 10%), infrequent (5–9%), and least frequent (< 5%; see Table 4). All of the most frequent sub-concepts, devices, hardware and software, and culture, were taught at the introductory level (i.e., by Grade 2). This distribution is not necessarily unexpected given that a third of the curricula are for students younger than age 8, which is Grade 2.

The least frequently addressed non-programming CS concept was networks and the Internet. Fewer than 1% of lessons included the sub-concepts at each level of instruction, which is unsurprising given that learning about networks is not typically the goal of integrated computing lessons. Notably, troubleshooting and storage at the intermediate level (by Grade 5) and network communication and organization at the advanced level (by Grade 8) were not taught in any of the lessons. Overall, the advanced levels (by Grade 8) of non-programming concepts were rarely taught in the integrated

Table 4. Frequency of Non-Programming Concepts Taught in Integrated Lessons

Concept	Subconcept	Level	Count	%
Computing Systems	Devices	By Grade 2. Introductory	**78**	**15.0**
		By Grade 5. Intermediate	**33**	**6.4**
		By Grade 8. Advanced	1	0.2
	Hardware and Software	By Grade 2.Introductory	**59**	**11.4**
		By Grade 5. Intermediate	7	1.3
		By Grade 8. Advanced	2	0.4
	Troubleshooting	By Grade 2. Introductory	**30**	**5.8**
		By Grade 5. Intermediate	0	0.0
		By Grade 8. Advanced	1	0.2
Networks and the Internet	Network Communication and Organization	By Grade 2. Introductory	3	0.6
		By Grade 5. Intermediate	2	0.4
		By Grade 8. Advanced	0	0.0
	Cybersecurity	By Grade 2. Introductory	2	0.4
		By Grade 5. Intermediate	1	0.2
		By Grade 8. Advanced	1	0.2
Data and Analysis	Collection	By Grade 2. Introductory	17	3.3
		By Grade 5. Intermediate	15	2.9
		By Grade 8. Advanced	10	1.9
	Storage	By Grade 2. Introductory	5	1.0
		By Grade 5. Intermediate	0	0.0
		By Grade 8. Advanced	2	0.4
	Visualization and Transformation	By Grade 2. Introductory	**32**	**6.2**
		By Grade 5. Intermediate	**27**	**5.2**
		By Grade 8. Advanced	18	3.5
	Inference and Models	By Grade 2. Introductory	**26**	**5.0**
		By Grade 5. Intermediate	**29**	**5.6**
		By Grade 8. Advanced	22	4.2
Impacts of Computing	Culture	By Grade 2. Introductory	**60**	**11.6**
		By Grade 5. Intermediate	**30**	**5.8**
		By Grade 8. Advanced	8	1.5
	Social Interactions	By Grade 2. Introductory	11	2.1
		By Grade 5. Intermediate	18	3.5
		By Grade 8. Advanced	4	0.8
	Safety, Law, and Ethics	By Grade 2. Introductory	8	1.5
		By Grade 5. Intermediate	5	1.0
		By Grade 8. Advanced	3	0.6

Dark green indicates most frequent (>10%) use and green indicates less frequent (5–9%) use.

curricula, likely due to the majority of the curricula being developed for younger children. Curricula for older students, especially in lower secondary/middle school (i.e., Bootstrap, CS First, MIT Ethics, and YouCubed Data Science), included more non-programming concepts than other curricula.

4.3.2 *Extent of Non-Programming Concept Taught in Integrated Computing Curricula.* The results showed that the breadth and depth of non-programming concept integration varied depending on the curriculum. A few CS curricula addressed multiple non-programming concepts throughout the lessons, such as data science lessons (i.e., Bootstrap and Youcubed), all CS First curricula, and CS+Earth. For instance, all five CS First curricula emphasized the impacts of computing, especially how computing technology has impacted our daily lives and ways for social interactions, while also introducing how computing devices operate or how data are collected and interpreted. These CS First lessons provide many real-world examples, the history of CS, and the impacts on our culture and the industry.

The depth of integration of non-programming computing concepts was also affected by the topic of the curriculum. A few curricula that had an intensive integration of non-programming concepts were centered around data science (e.g., Bootstrap-Data Science, CS+Data, Youcubed-Data Science) and ethics (e.g., MIT AI Ethics Education). For instance, the concepts about data collection, visualization and transformation, and inferences and models were intensively integrated into the lessons in Youcubed-Data Science (100%), Bootstrap-Data Science curricula (93%), and MIT AI Ethics Education (63%) curricula. In other curricula, the data and analysis concept was rarely addressed.

Other curricula took a more targeted approach, focusing on a narrower range of concepts. Particularly, curricula that focus on programming concepts and CS practices typically briefly introduce non-programming concepts at the beginning (e.g., CSinSF K-2 CS Curriculum-Red and Yellow, CAL Scratch Jr.) or toward the end of the curriculum (e.g., ECforALL-Act 2, CS+Data). For example, the first few lessons of CAL Scratch Jr. and the CSinSF K-2 Red and Yellow curricula introduced computers and mobile devices, and later lessons focused on programming concepts and CS practices. Robotic-based curricula, unsurprisingly, emphasized the concepts of computing systems, and many lessons discussed how a computing device works, such as an explanation of how hardware and software work together as a computing system. For example, CAL KIBO, which used the KIBO robot, focused on devices and hardware and software concepts, which were continuously addressed across the curriculum. However, these sub-concepts were the only non-programming concepts introduced.

4.4 Crosscutting Findings across Frameworks

The research team examined patterns of data across the three frameworks to determine how programming concepts, CS practices, and non-programming CS concepts are taught together. Based on this examination, we found no consistent patterns across the frameworks to report. For example, we identified that curricula that use robotics to program consistently include non-programming concepts about devices and hardware, as would be expected. However, the devices and hardware sub-concepts are also frequently taught in other non-robotics curricula, so this trend is not unique. In addition, we considered the patterns across curricula designers identified in Section 4.1.2 for the extent of instruction on programming concepts to examine whether they matched patterns in other frameworks. Again, we found no consistent results. For the practices and non-programming frameworks, we found no similar patterns, even within curriculum designers, leaving nothing to report.

5 Discussion

5.1 Central Contributions to Integrated Computing

The primary goal of this analysis was to explore which CS concepts and practices are taught in integrated CS curricula at the primary and lower secondary (i.e., K-8) levels and the depth of computing instruction. For this analysis, the researchers were particularly interested in how these curricula prepare students for later, standalone introductory computing and programming courses. However, we also hope that this analysis contributes to the larger discussion of how CS can be integrated into other disciplines and a better understanding of computational literacy for all students, not only those pursuing CS education.

There were two main differences in selection criteria between a previous analysis of programming concepts taught in integrated CS [20] and the current analysis. First, the previous analysis required lessons to have non-CS learning objectives, while the current analysis required lessons to have CS learning objectives. In addition, the previous analysis had no minimum time frame for lessons, so it included mostly single or week-long lessons, while the current analysis required at least five hours of instruction.

For these reasons (i.e., more focus on CS learning and a longer time frame), we expected that lessons in the current analysis would include more programming concepts than those in the previous analysis, but that is not what we found. Instead, most concepts were taught at about the same relative frequency. As in the previous analysis, multimedia components (objects, properties, and movement), events, string outputs, for loops, and functions were commonly found in the current analysis. Similarly, inputs, conditionals, and operators were uncommon in both analyses. The only category that changed in relative frequency was variables but not in the expected direction. Variables were much more common in the previous analysis, similar in frequency to string outputs and for loops, than in the current analysis, in which they were similar in frequency to conditionals and operators. This difference might be due to the math- and science-based learning objectives that made variables useful in the lessons included in the previous analysis.

Based on these findings, we agree with the previous analysis' conclusion that integrated CS activities give students experience creating multimedia and animations but not necessarily automating information processing, except for some relatively basic concepts (i.e., using for loops and calling functions [20]). While we agree that in the absence of CS learning objectives, this pattern is not inherently problematic. However, it might be problematic when integrated CS curricula are expected to prepare students for standalone CS courses, especially programming. Many of the concepts taught in the integrated curricula are not those taught in introductory programming courses, except basic concepts like sequences, algorithms, string outputs, for loops, and calling pre-defined functions. Thus, it is important that the teachers of introductory programming courses do not expect students who have engaged in integrated curricula to have experience with variables, conditionals, operators, and defining functions. More importantly, CS teachers should not expect students to think of programming as a way to automate problem-solving, though that is often the process underlying programming instruction [20]. On the other hand, integrated CS is often students' first experience with programming, and this level of engagement with concepts and practices might be the most appropriate, especially at early ages. Thus, our primary recommendation is not to change the curricula but instead to not overestimate the prior knowledge of students in later CS courses.

Some people would argue that the goal of integrated computing is not to teach programming, or at least not primarily to teach programming (e.g., [12, 30, 46]. However, the current analysis found that these analyzed curricula introduced relatively few CS practices and non-programming concepts as well. Thus, the primary goal of these curricula does not seem to be teaching CS practices beyond the context of programming nor non-programming concepts, at least not those identified by the K-12 CS Framework. If one were to argue that the goal of integrated computing was to teach about computing more broadly than programming, like to teach about data and analysis, that argument is also not well-supported by these data, at least beyond the data science curricula.

The CS practices best represented in the curricula reflected social practices, i.e., practice 1.1 about including the unique perspectives of others, practice 2.1 about cultivating working relationships, and practice 7.2 about documenting solutions for an intended audience. These practices are aligned with other socio-emotional learning objectives in the classroom, especially in early grades where the development of personal character, emotional intelligence, and social skills is often explicitly included. Thus, their application to the development of computational products and collaborations can reinforce these objectives and practices, which was a deliberate goal of the K-12 CS Framework [7]. The K-12 CS Framework also aimed to reinforce objectives related to pattern recognition, problem-solving, representation, and sequencing [7]. The integrated curricula achieved basic practices related to these objectives about 30% of the time (i.e., practice 3.1 about identifying computational problems, practice 4.1 about extracting common features, and practice 5.2-3 about creating computational artifacts/representations). The more advanced CS practices, such as those related to developing and using abstracts, would be difficult to apply without developing more computationally complex products than were typical in the analyzed curricula.

The lack of non-programming CS concepts in these lessons is unsurprising given that many of those concepts are domain-specific to computing, such as networks, and would not easily fit into non-CS instruction. However, integrated computing curricula do provide opportunities for students to learn digital citizenship and computational literacy skills that would serve them well, whether they decide to pursue standalone CS classes or not. For example, concepts from the Impact of Computing or Networks and the Internet could be addressed by teaching students about responsible computing practices in the face of cybersecurity threats and the ethical development of artificial intelligence for societal benefit. These non-programming CS concepts are particularly well-suited for younger students to grasp through unplugged activities. This approach supports students' computational literacy and could make the learning experience more engaging and comprehensive.

5.1.1 Discussion of Programming Concept Instruction in Integrated Computing. From the analysis of the extent to which programming concepts are taught, we found three patterns of concept use across curricula: (1) curricula that were limited in concepts by the programming paradigm (i.e., robotics or Scratch Jr) and emphasized unguided application of few concepts, (2) curricula that prioritized learning a small set of concepts and creating new programs with them, situationally branching out to other concepts, and (3) curricula that consistently used a wide range of concepts, focusing on using or modifying existing programs. These patterns represent an arguably necessary tradeoff in integrated computing curricula, that the number of concepts taught is inversely related to the depth at which they are taught, given limited time for CS instruction. Depending on the goals of instruction and constraints within the classroom, any of these patterns could provide a viable strategy for integrating computing in non-CS instruction. For example, the pattern that uses unguided exploration is likely the best choice for a classroom of five-year-olds that aims to foster CS identity, introduce a new medium for self-expression, and is flexible in the disciplinary learning objectives achieved. In this case, the limited number of CS concepts taught is not consequential to the goals of the instruction. However, for older students who have already had regular CS experiences and whose teacher feels pressure to prepare them for standardized testing, the latter two patterns would be more appropriate.

In the first pattern, limiting the problem-solving space through a programming paradigm, such as programmable robots or Scratch Jr, provides a narrow problem-solving space that affords more unguided application. Thus, if minimally-guided activities and student exploration are the goal, such as in a guided discovery learning paradigm (e.g., [1]), then this pattern is a viable strategy. This pattern, however, did not introduce many concepts, and given the paradigms, most of the concepts were for animating robots or sprites. Thus, few of the programming concepts introduced in these curricula were those that would be taught in an introductory programming course. Given that they were also taught to the youngest students, this pattern is likely developmentally appropriate [41]. A similar paradigm that is for older learners but was not represented in our database is task-specific languages [12], which are languages with narrow functionality designed to accomplish a single task. They are similarly designed to allow students to explore deeply within a constrained problem-solving space, albeit their goal is typically to avoid the need to develop general programming skill [12].

In the authors' experience, school districts and teachers are becoming aware of how programming paradigms affect student learning. They value the constrained problem-solving space of paradigms like programmable robots or Scratch/Scratch Jr to both scaffold student learning and facilitate teacher development. However, as students advance to more computing-centric courses, they are also recognizing that these paradigms can limit students' engagement with programming concepts. Some districts are now focusing on the vertical alignment of computing instruction as students progress from primary to secondary education, which typically involves switching from integrated computing to standalone computing. Integrated computing curricula creators, thus, could help practitioners by suggesting how their learning objectives and paradigms could align with computing-centric curricula. For example, creators could include a unit about the progression from Scratch Jr to Scratch or Scratch to Snap!. Such a unit could highlight the affordances of more advanced concepts, such as how Scratch affords the creation of variables and how Snap! affords other data storage and handling opportunities, which are relevant to computing as well as advanced science or math objectives.

The second pattern provides more instruction on programming concepts while still providing opportunities for student creation in more structured tasks. Consistently using the same concepts throughout a curriculum, such as multimedia concepts or functions, allows students to repeatedly practice using those concepts. Building upon these recurring concepts, the curricula introduced a wider range of concepts more shallowly. Similar to the first pattern, the recurring concepts taught in many of these curricula focused on animation. In contrast to the first pattern, these curricula also include exposure to concepts focused on information processing, like variables, conditionals, and operators. Thus, this pattern allows students to experience how solving a problem with computing is different than solving a problem with other tools.

In the third pattern, students applied a variety of concepts consistently through `Use` and `Modify` levels of engagement. In this pattern, students had more experience with concepts that would be taught in introductory programming courses, but they less often practiced creating programs and, thus, the procedural knowledge of how to create programs. Given that the student is likely to use a different programming paradigm in introductory programming than in integrated computing curricula, this lack of procedural practice might not be problematic. Like the second pattern, students could experience how problems are solved with computing, which might prepare them conceptually, if not procedurally, for later computing courses.

5.2 Limitations and Future Work

Future work would be needed to compare these patterns and their effect on later CS education. The current analysis' goal was to examine the curricula itself, not their effects on students. Though all of these patterns seem like viable strategies for integrating CS, they might have different effects on students in terms of encouraging students to pursue standalone CS courses or improving their performance or experience in those courses. For example, in the first two patterns, the process of unguided application or `Creating` programs might help students normalize the process of receiving errors and fixing bugs, helping them overcome those challenges. This effect would be significant because errors and bugs often reduce novices' self-efficacy and their persistence in computing education (e.g., [3, 22]). In addition, in the third pattern, exploring how to automate information processing might encourage students to learn more about problem-solving through programming, even if they lack the procedural knowledge to implement their ideas.

Future work could also explore other instructional design frameworks for integrated CS. The researchers had expected curricula to include more features of project-based learning, which anchors instruction around a single, large, (semi-)authentic problem [2]. However, in many of the curricula, such as those created by ECforALL, CS+, and Everyday Computing, individual lessons were set in different contexts (i.e., they gave students a different Scratch starter project at the start of most lessons) rather than building additional features into a single context. Given that lessons were 45–75 minutes long, building upon a single project over multiple lessons might allow students to explore more concepts and practices or explore them in more depth. If the integrated context lent itself to a project-based approach, one effective instructional design model for this strategy could be the Four Components of Instructional Design model, which is specifically for complex learning in which knowledge, skills, and attitudes are developed simultaneously [43], like in integrated CS. This well-researched model explains how to support students through four components: learning tasks, part-task practice, supportive information, and procedural information [43]. It and other instructional design models and frameworks for complex learning should be applied when creating integrated computing curricula.

The current analysis focused on what was taught but not how it was taught. Because the unit of analysis in this study was the lesson, we did not have in-depth information about instructional approaches to individual activities, such as how CS concepts were introduced. A more detailed analysis of the content and activities within the lessons would be needed to examine how new concepts are introduced to students. For example, such an analysis could explore whether instruction follows the K-5 learning trajectories [32, 34] and how students build upon prior knowledge or build prerequisite knowledge to learn more advanced concepts. In addition, such an analysis could explore whether instruction applies the principles of semantic waves [21, 45] and how teachers can make analogies with everyday knowledge to develop technical knowledge. This type of analysis, while valuable, would address different research questions than those posed in this study.

Despite much work still needing to be done, the current analysis contributes to our understanding of the current role of integrated CS curricula in computing education. It is critical that these curricula not overload the domain in which they are integrated, either in terms of time spent on CS content or in terms of teacher CS knowledge and skill required. Either would result in the curricula not being adopted. Still, many CS educators might be surprised by the limited CS concepts and practices taught in integrated CS and should be cautious about their expectations for students' prior knowledge. Of course, this caution applies to educators of all domains, especially in the transition from primary to secondary school in which learning goals often shift from giving students experiences within domains to formally teaching concepts, as students' developmental capabilities increase. The authors hope that the current analysis provides evidence-based information about which CS concepts are taught in integrated curricula and to what extent so that researchers and educators can better understand how to support students in CS education.

References

[1] Arthur J. Baroody, David J. Purpura, Michael D. Eiland, and Erin E. Reid. 2015. The impact of highly and minimally guided discovery instruction on promoting the learning of reasoning strategies for basic add-1 and doubles combinations. *Early Childhood Research Quarterly* 30 (2015), 93–105.

[2] Brigid J. S. Barron, Daniel L. Schwartz, Nancy J. Vye, Allison Moore, Anthony Petrosino, Linda Zech, and John D. Bransford. 2014. Doing with understanding: Lessons from research on problem-and project-based learning. In *Learning Through Problem Solving*. Psychology Press, 271–311.

[3] Jens Bennedsen and Michael E. Caspersen. 2005. Revealing the programming process. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*. 186–190.

[4] Marina Umaschi Bers. 2019. Coding as another language: A pedagogical approach for teaching computer science in early childhood. *Journal of Computers in Education* 6, 4 (2019), 499–528.

[5] Thomas Brush, Anne Ottenbreit-Leftwich, Kyungbin Kwon, and Michael Karlin. 2019. Implementing socially relevant problem-based computer science curriculum at the elementary level: Students' computer science knowledge and teachers' implementation needs. In *Proceedings of the Society for Information Technology & Teacher Education International Conference*. Association for the Advancement of Computing in Education (AACE), 2257–2266.

[6] Lautaro Cabrera, Diane Jass Ketelhut, Kelly Mills, Heather Killen, Merijke Coenraad, Virginia L. Byrne, and Jandelyn Dawn Plane. 2023. Designing a framework for teachers' integration of computational thinking into elementary science. *Journal of Research in Science Teaching* (2023).

[7] K-12 Computer Science Framework Steering Committee. 2016. K–12 Computer Science Framework. 1–36. Retrieved from https://k12cs.org/

[8] Louise P. Flannery, Brian Silverman, Elizabeth R. Kazakoff, Marina Umaschi Bers, Paula Bontá, and Mitchel Resnick. 2013. Designing ScratchJr: Support for early childhood learning through computer programming. In *Proceedings of the 12th International Conference on Interaction Design and Children*. 1–10.

[9] Diana Franklin, Charlotte Hill, Hilary Dwyer, Ashley Iveland, Alexandria Killian, and Danielle Harlow. 2015. Getting started in teaching and researching computer science in the elementary classroom. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 552–557.

[10] Pawel Grabarczyk, Sebastian Mateos Nicolajsen, and Claus Brabrand. 2022. On the effect of onboarding computing students without programming-confidence or-experience. In *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research*. 1–8.

[11] Meize Guo and Anne Ottenbreit-Leftwich. 2020. Exploring the K-12 computer science curriculum standards in the US. In *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*. 1–6.

[12] Mark Guzdial and Tamara Shreiner. 2021. Integrating computing through task-specific programming for disciplinary relevance: Considerations and examples. *Computational Thinking in Education* (2021), 172–190.

[13] Fredrik Heintz, Linda Mannila, and Tommy Färnqvist. 2016. A review of models for introducing computational thinking, computer science and computing in K-12 education. In *Proceedings of the IEEE Frontiers in Education Conference (FIE '16)*. 1–9. DOI: https://doi.org/10.1109/FIE.2016.7757410

[14] Maya Israel and Todd Lash. 2020. From classroom lessons to exploratory learning progressions: Mathematics+ computational thinking. *Interactive Learning Environments* 28, 3 (2020), 362–382.

[15] Maya Israel, Ruohan Liu, Wei Yan, Heather Sherwood, Wendy Martin, Cheri Fancsali, Edgar Rivera-Cash, and Alexandra Adair. 2022. Understanding barriers to school-wide computational thinking integration at the elementary grades: Lessons from three schools. In *Computational Thinking in PreK-5: Empirical Evidence for Integration and Future Directions*. ACM, New York, NY, 64–71.

[16] Yasmin B. Kafai and Chris Proctor. 2022. A revaluation of computational thinking in K–12 education: Moving toward computational literacies. *Educational Researcher* 51, 2 (2022), 146–151.

[17] Maria Kallia and Quintin Cutts. 2021. Re-examining inequalities in computer science participation from a Bourdieusian sociological perspective. In *Proceedings of the 17th ACM Conference on International Computing Education Research*. 379–392.

[18] Irene Lee, Fred Martin, Jill Denner, Bob Coulter, Walter Allan, Jeri Erickson, Joyce Malyn-Smith, and Linda Werner. 2011. Computational thinking for youth in practice. *ACM Inroads* 2, 1 (2011), 32–37.

[19] Tom Liam Lynch, Gerald Ardito, and Pam Amendola. 2020. *Integrating Computer Science across the Core: Strategies for k-12 Districts*. CRC Press.

[20] Lauren Margulieux, Miranda C. Parker, Gozde Cetin Uzun, and Jonathan D. Cohen. 2023. Levels of programming concepts used in computing integration activities across disciplines. *Journal of Technology and Teacher Education* 31, 2 (2023), 167–202.

[21] Karl Maton. 2019. Semantic waves: Context, complexity and academic discourse. In *Accessing Academic Discourse*. Routledge, 59–85.

[22] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: A review of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92.

[23] Mary L. McHugh. 2012. Interrater reliability: The kappa statistic. *Biochemia Medica* 22, 3 (2012), 276–282.

[24] Elena Novak and Javed I. Khan. 2022. A research-practice partnership approach for co-designing a culturally responsive computer science curriculum for upper elementary students. *TechTrends* 66, 3 (2022), 527–538.

[25] Michiyo Oda, Yoko Noborimoto, and Tatsuya Horita. 2022. Analysis of K–12 computer science curricula from the perspective of a competency-based approach. In *Proceedings of the Society for Information Technology & Teacher Education International Conference*. Association for the Advancement of Computing in Education (AACE), 75–79.

[26] Zehra Ozturk, Caitlin McMunn Dooley, and Meghan Welch. 2018. Finding the hook: Computer science education in elementary contexts. *Journal of Research on Technology in Education* 50, 2 (2018), 149–163.

[27] Miranda C. Parker and Leigh Ann DeLyser. 2017. Concepts and practices: Designing and developing a modern k-12 cs framework. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 453–458.

[28] Miranda C. Parker, Amber Solomon, Brianna Pritchett, David A. Illingworth, Lauren E. Marguilieux, and Mark Guzdial. 2018. Socioeconomic status and computer science achievement: Spatial ability as a mediating variable in a novel model of understanding. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 97–105.

[29] John T Paxton, Rockford J Ross, and J Denbigh Starkey. 1994. A methodology for teaching an integrated computer science curriculum. *ACM SIGCSE Bulletin* 26, 1 (1994), 1–5.

[30] Burkhard Priemer, Katja Eilerts, Andreas Filler, Niels Pinkwart, Bettina Rösken-Winter, Rüdiger Tiemann, and Annette Upmeier Zu Belzen. 2020. A framework to foster problem-solving in STEM and computing education. *Research in Science & Technological Education* 38, 1 (2020), 105–130.

[31] Jake A. Qualls and Linda B. Sherrell. 2010. Why computational thinking should be integrated into the curriculum. *Journal of Computing Sciences in Colleges* 25, 5 (2010), 66–71.

[32] Kathryn M. Rich, T. Andrew Binkowski, Carla Strickland, and Diana Franklin. 2018. Decomposition: A K-8 computational thinking learning trajectory. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 124–132.

[33] Kathryn M. Rich, Carla Strickland, T. Andrew Binkowski, and Diana Franklin. 2019b. A K-8 debugging learning trajectory derived from research literature. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 745–751.

[34] Kathryn M. Rich, Carla Strickland, T. Andrew Binkowski, Cheryl Moran, and Diana Franklin. 2017. K-8 learning trajectories derived from research literature: Sequence, repetition, conditionals. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. 182–190.

[35] Peter J. Rich, Samuel F. Browning, McKay Perkins, Timothy Shoop, Emily Yoshikawa, and Olga M. Belikov. 2019a. Coding in K-8: International trends in teaching elementary/primary computing. *TechTrends* 63 (2019), 311–329.

[36] Anthony Robins. 2010. Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education* 20, 1 (2010), 37–71.

[37] Jean Salac, Cathy Thomas, Chloe Butler, Ashley Sanchez, and Diana Franklin. 2020. TIPP & SEE: A learning strategy to guide students through use-modify scratch activities. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 79–85.

[38] Rafi Santo, Sara Vogel, and Dixie Ching. 2019. *CS for What? Diverse Visions of Computer Science Education in Practice*. CSforALL.

[39] Umar Shehzad, Jody E. Clarke-Midura, Kimberly Beck, Jessica F. Shumway, and Mimi M. Recker. 2023. Rethinking integrated computer science instruction: A cross-context and expansive approach in elementary classrooms. In *Proceedings of the American Educational Research Association's Annual Meeting (AERA '23)*. 1.

[40] Carla Strickland, Andrea Ramírez-Salgado, Lauren Weisberg, LaToya Chandler, Jeanne Di Domenico, Elizabeth M. Lehman, and Maya Israel. 2023. Designing an equity-centered framework and crosswalk for integrated elementary computer science curriculum and instruction. *Journal of Computer Science Integration* 6, 1 (2023), 1–16.

[41] Carla Strickland, Kathryn M. Rich, Donna Eatinger, Todd Lash, Andy Isaacs, Maya Israel, and Diana Franklin. 2021. Action fractions: The design and pilot of an integrated math+ CS elementary curriculum based on learning trajectories. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 1149–1155.

[42] Amanda Sullivan. 2021. Supporting girls' computational thinking skillsets: Why early exposure is critical to success. In *Teaching Computational Thinking and Coding to Young Children*. IGI Global, 216–235.

[43] Jeroen J. G. Van Merriënboer. 2019. *The Four-Component Instructional Design Model*. Maastricht University.

[44] Sara Vogel, Rafi Santo, and Dixie Ching. 2017. Visions of computer science education: Unpacking arguments for and projected impacts of CS4All initiatives. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 609–614.

[45] Jane Waite, Karl Maton, Paul Curzon, and Lucinda Tuttiett. 2019. Unplugged computing and semantic waves: Analysing crazy characters. In *Proceedings of the 2019 Conference on United Kingdom & Ireland Computing Education* Research. 1–7.

[46] Changzhao Wang, Ji Shen, and Jie Chao. 2022. Integrating computational thinking in STEM education: A literature review. *International Journal of Science and Mathematics Education* 20, 8 (2022), 1949–1972.

[47] Jeannette M. Wing. 2006. Computational thinking. *Communications of the ACM* 49, 3 (2006), 33–35.

Received 10 November 2023; revised 1 May 2024; accepted 2 May 2024