Jl. of Technology and Teacher Education (2023) **31**(2), 167–202

Levels of Programming Concepts Used in Computing Integration Activities across Disciplines

LAUREN MARGULIEUX
Georgia State University, USA
lmargulieux@gsu.edu

MIRANDA C. PARKER
San Diego State University, USA
mcparker@sdsu.edu

GOZDE CETIN UZUN
Georgia State University, USA
gcetin1@gsu.edu

JONATHAN D. COHEN
Georgia State University, USA
jcohen@gsu.edu

Educators across disciplines are implementing lessons and activities that integrate computing concepts into their curriculum to broaden participation in computing. Out of myriad important introductory computing skills, it is unknown which—and to what extent—these concepts are included in these integrated experiences, especially when compared to concepts commonly taught in introductory computer science courses. Thus, it is unclear how integrated computing activities serve the goal of broadening participation in computing. To address this deficit, we compiled a database of 81 integrated computing activities, constructed a framework of fundamental programming concepts, and scored each activity in the database for the presence of each concept. We also analyzed frequency and patterns of scores across different activity features, in-

cluding academic discipline, programming language, student age, and duration of activity. Analysis showed that concepts that appear most frequently in integration activities (i.e., concepts for animation and visualization) largely did not align with concepts taught most frequently in introductory programming courses (i.e., concepts that automate problem-solving processes). We argue that our findings can inform the way teacher educators frame integrated computing activities for their students and the decisions they make when determining the types of computing integration activities they introduce to their students. We also discuss implications for treating integration activities as prior knowledge in introductory programming courses.

The computing education community has begun to advocate for integrating computing into other disciplines as a sustainable and equitable method of introducing all students to computing, a skillset that is increasingly incorporated into our personal and professional lives (Margulieux et al., 2022; Mouza et al., 2017; Yadav et al., 2021; Yadav, Gretter, et al., 2017). For example, elementary/primary English language arts teachers could teach an integrated computing activity in which their students create digital stories using the block-based programming language Scratch, achieving learning outcomes in both English and computer science (CS). Because every student takes English language arts courses, this approach gives all students the opportunity to experience CS and learn fundamental concepts. This widespread interest in integrated computing from both researchers and teachers is evidenced by a new journal on the topic, the *Journal of Computer Science Integration* and a summit held in February 2022 called "CS Across the Curriculum" for both CS and non-CS teachers, organized by the Computer Science Teachers Association. While many integrated computing initiatives began to provide an on-ramp to standalone CS education, they now also target the role of computing in non-CS education and as a general literacy (Lee et al., 2014; Waterman et al., 2020).

With growing interest and applications of CS, the CS community is still working towards a concrete definition of general computational literacy for people who will not become computer scientists. This definition is central to integrated computing, as it can inform the work of teacher educators who must balance computing learning objectives in integration activities with learning objectives for their primary discipline.

CS educators tend to define the computing learning objectives of an integration activity with one of two top-down approaches. They either em-

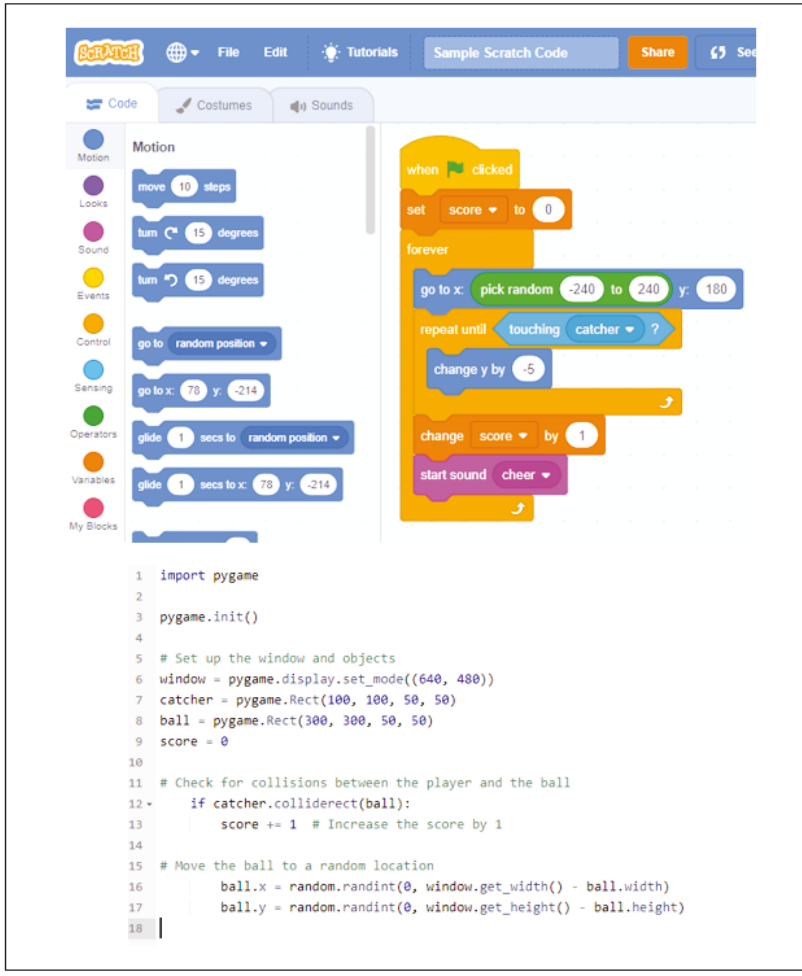
phasize computational thinking (CT) as a systematic problem-solving strategy (e.g., Palts & Pedaste, 2020), or they focus on previewing concepts that would be taught in an introductory programming course, such as variables, operators, loops, conditionals, and functions (e.g., Brennan & Resnick, 2012). Accordingly, they design learning experiences for students to achieve computing learning objectives. However, these top-down designs from CS educators do not always best serve the non-CS educators who will implement the activities. Because teachers' decisions to integrate (or not) technology is driven by their values (Kopcha et al., 2020), teachers in a disciplinary context will address other, non-CS-related factors, such as access to resources, time, and their own expertise, when designing learning experiences that include integrated computing.

Because the top-down approaches that center CS learning objectives are already well-studied, the current analysis aimed to examine integration activities that were designed primarily to teach non-CS learning objectives. Specifically, the analysis examined with a bottom-up approach which programming concepts emerge as the most used across integration activities that do not prioritize computing learning objectives. Our project analyzed existing integrated computing activities from a variety of disciplines, student ages, designers, and languages to determine which programming concepts they employ. The result was ratios of how commonly each programming concept appears in activities. This analysis focused on programming concepts because much work already examines CT concepts in integration activities (e.g., Lee et al., 2014; Lye & Koh, 2014; Weintrop et al., 2016), and these concepts are pervasive in all activities and well-defined, such as in the competencies outlined by the Computer Science Teachers Association and International Society for Technology in Education (ISTE). Our goal for examining the frequency of programming concepts was to make progress towards a definition of general computational literacy, especially to 1) serve as a comparison to concepts typically taught in introductory computing courses, and 2) inform the work of educators who are including computing integration into their practice so that they can better prepare their students for an increasingly technological world.

To make the scope of this project manageable, only integration activities based on block-based programming languages were included in the analysis currently. Block-based languages, such as Scratch (scratch.mit.edu) and Snap! (snap.berkeley.edu) are visual coding environments in which users create programs not by typing commands, but by selecting jigsaw puzzle-like blocks that perform actions when fit together properly (Figure 1).

Figure 1

Block-based Coding and Text-based Coding Samples



Note. The top image is an example of block-based coding in Scratch. The code (i.e., the shape on the right) is created by clicking-and-dragging blocks that snap together from the menu on the left. The bottom image is text-based code which achieves a similar objective to the Scratch blocks.

Restricting the scope to block-based languages created a bias in the data that has benefits and limitations. One benefit is that a touted asset of block-based languages is the emphasis on concepts over syntax and semantics (Grover & Basu, 2017; Papadakis et al., 2014), affording an emphasis on programming concepts (i.e., the focus of the analysis) in learning activities. Likewise, block-based activities are typically designed for learners with little to no programming experience (Kelleher & Pausch, 2005), allowing us to capture concepts used at the most introductory level. Similarly, block-based activities are more likely than text-based activities to be made by non-CS teachers, especially in ScratchEd's extensive database, providing authenticity to the integration activities for achieving non-CS learning objectives. Some of the limitations, especially when considering activities designed by programming novices, are that concepts in block-based languages are predetermined by the blocks menu, which can affect how they are used. On one hand, the order of blocks in a menu or grouping of blocks affects whether concepts are considered for inclusion (Weintrop & Wilensky, 2015, 2018). On the other hand, block menus allow novices to scroll through various options that they might not recall in a text-based environment, increasing the number of concepts used by novices in block-based programming compared to text-based programming (Weintrop & Wilensky, 2015, 2018).

In recognition of this bias, we offer the current analysis as one step towards a goal that cannot be encapsulated by a single analysis: defining general computational literacy. We also recognize that our bottom-up analysis was likely affected by top-down design approaches. However, we have attempted to mitigate these biases by intentionally including activities that varied on key features, such as applied discipline and student age, to provide a new perspective on the relevance of programming concepts in other disciplines. Our research questions were:

- How frequently are different programming concepts used in computing integration activities that feature non-CS disciplinary learning objectives, and
- How do features of computing integration activities, such as discipline, student age, programming language, and duration, affect the concepts used?

LITERATURE REVIEW

In this paper, we aimed to move towards a general computational literacy outside of the context of standalone CS education. As such, this lit-

erature review focuses on programming concepts from a general education perspective rather than from a CS education perspective. Thus, the literature review draws from but does not attempt to thoroughly discuss research related to teaching introductory programming (for a current summary, see Luxton-Reilly et al. 2018), nor features of programming languages designed for novices, such as those described by

Guzdial (2004) and Kelleher (2005) and including the recent landscape of block-based languages by Lin and Weintrop (2021). We recognize that work on learning trajectories for computing concepts (e.g., Fields et al., 2016; Luo et al., 2022; Rich et al., 2018) is relevant to applying the findings of our current analysis, but the analysis focused on identifying which concepts are taught in various contexts, rather than how to teach them. We also did not thoroughly discuss research on CT education, which tends to focus on instructional strategies, assessment, and tools for CT learning objectives (e.g., Kong & Abelson, 2019; Palts & Pedaste, 2020), and instead focus on research about CT *in* education to examine the role of CT in supporting other disciplinary learning objectives.

Computing as a Tool for Teaching and Learning

Starting in the 1960s, Papert (1980), DiSessa (2000), and others have called for a universal computational literacy that empowered learners to apply computational tools to problem-solving of all sorts. In addition to fields that manage mounds of data or require extremely precise calculations, a prominent area of application is education. In education, computing is commonly used as a tool to support constructivist or constructionist learning environments that enable students to create digital stories, collect and analyze data, or simulate scientific phenomena (Lee et al., 2014). For example, in science education, simulating phenomena in a computational model allows students to tinker with variables and visualize the results, affording confrontation of misconceptions, instantaneous data collection to evaluate hypotheses, and infinite concrete cases from which to induce scientific theories (Waterman et al., 2020; Wilensky et al., 2014). Applying computational tools, like any technical tool, to education requires professional development for teachers. At least in the United States, integrated computing and CT are almost always introduced in the educational technology portion of non-CS teacher preparation (Mouza et al., 2017; Yadav et al., 2014; Yadav, Stephenson, et al., 2017).

Because integrated computing is typically categorized as educational technology in teacher preparation programs, it is commonly conceptualized

in the Technological Pedagogical and Content Knowledge (TPACK) framework (Grover, 2021). TPACK is a widespread framework for the integration of technology in educational practice that adds technological applications to pedagogical content knowledge (PCK; Koehler & Mishra, 2009). The PCK framework posits that effective teachers must know content knowledge within their discipline, pedagogical knowledge about how to teach generally, and pedagogical content knowledge about how to teach within their discipline (Shulman, 1986). For example, PCK would argue that the most effective methods for teaching social science are not the same as the most effective methods for teaching science, though some general education methods are shared. TPACK adds technological knowledge as an equal component to this framework, including standalone technological knowledge about how to use technology and its interaction with content, pedagogical, and pedagogical content knowledge (Koehler & Mishra, 2009).

Teacher education programs have largely found CT and integrated computing useful as a technological tool in the classroom. Kale et al. (2018) found that CT, framed in terms of TPACK, provided a useful approach for teachers to develop students' problem-solving strategies and to implement other pedagogical content knowledge, such as modeling, guided discovery, structured methods, scheme activation, and load-reducing. In addition, teachers appreciated that computing allows students to be creative and constructive in pursuit of learning objectives in other disciplines, such as geometry, Spanish, and financial literacy (Kale et al., 2018). Further, Saritepeci found that teachers who had learned CT had higher TPACK overall and better classroom management in the context of technology-enriched classrooms from a higher technological knowledge (2021). One of the benefits of computing integration, especially framed as a technological activity, is that the activities support learning environments that often provide immediate feedback to learners (Margulieux et al., 2022; Saritepeci, 2021). In addition, programming activities require students to formalize and externalize their thought processes to create the program. This approach results in the student essentially teaching the computer and can facilitate communication between students or with teachers as they share their work or if they get stuck by having their thought process externally formalized into a program (Margulieux et al., 2022).

Despite the benefits of integrated computing, learning programming concepts (i.e., technological knowledge) takes considerable time in an already crowded teacher preparation curriculum. Kong and Lai (2021) developed a TPACK-based CT unit for teachers that used block-based languages to minimize time spent learning technological knowledge. They found this design decision to minimize time learning programming worthwhile be-

cause teachers found the explicit discussion of all seven TPACK components (CK, PK, TK, CPK, CTK, PTK, TPACK) valuable to understanding how CT connects to the disciplinary learning objectives and practices in their classroom. Several researchers have made the same decision to minimize programming instruction and found that a couple of hours of instruction about integrated computing activities allow teachers to implement predesigned lessons in their classrooms, but it does not allow them to design their own integration activities (da Silva et al., 2020; Margulieux et al., 2022; Mouza et al., 2017). To add a substantive programming element that enabled teachers to independently create computing integration activities, Kong et al. (2020) found that two 39-hour courses that included sustained instruction and practice of programming concepts were necessary to learn sufficient technological knowledge, similar to yearlong sequence for introductory programming for CS majors. This amount of time and effort is not realistic in the context of teacher preparation programs. The teacher preparation programs that emphasize integrated computing typically give a maximum of 10 hours to CT and computing integration, and it is only one area of TPACK that contends with all the other technologies teachers use.

Feasible Programming Instruction in Teacher Preparation

It is not currently viable to teach programming to non-CS teachers the same way as we would teach programming to CS teachers. This section discusses US-based teacher preparation programs, but this statement is likely true everywhere. Even if we considered the opportunity for non-CS teachers to take introductory programming courses as an elective, like other non-CS undergraduates, preparing all teachers for computing instruction is not feasible. Teacher preparation programs are typically among the most rigid undergraduate and graduate majors, with few opportunities for electives. Take a future biology teacher as an example. That teacher candidate must take general education courses, a sequence of biology courses and education courses, and teach in an actual biology classroom during their final year while submitting portfolios to certification agencies for accreditation. Biology teachers essentially dual-major in biology and education with a mandatory yearlong internship and certification. The requirements for elementary teachers, who do not need the disciplinary sequence of courses, are no more flexible because they must prepare to teach literacy, math, science, and humanities, all within an emphasis on child development.

Informed by the requirements of teacher preparation programs, the CS education community needs to make strategic decisions about which pro-

gramming concepts are most useful in integrated computing activities. Several researchers have responded to this integration challenge by eschewing programming instruction and focusing on CT as a problem-solving strategy. While reasonable, this strategy is akin to teaching mathematical thinking as a way to solve problems involving data in science classes without enabling students to use calculators or spreadsheets to analyze data. It does not introduce the technical skills required to use computing as a tool to solve an array of problems. In addition, many teachers and students want to learn to program but do not know how to start or where to look (Lye & Koh, 2014). Before experiencing programming, many teachers think as consumers of technology who must use existing tools as they were designed (Margulieux et al., 2022). After a brief introduction to programming, they change their perception to a producer of technology who can, perhaps not create their own program independently, but begin to recognize the possibilities for modifying tools and automating tasks (Margulieux et al., 2022). The authors argue that this shift in perception is critical for achieving general computational literacy, making the introduction to programming critical for computational literacy. The current analysis aims to describe the current state of computing integration activities used in classrooms so that teacher educators can consider the inclusion of computing integration practices in their curricula in a more targeted way.

METHOD

Selection Criteria: Features and Limitations

Non-CS Disciplinary Learning Objectives

Given the current analysis' focus on programming concepts used outside of standalone computing, the first selection criterion for activities to include in the analysis was the inclusion of learning objectives in a discipline other than computing. No restrictions were placed on which other disciplines qualified, and we found activities from language arts, math, science, art, music, foreign language, history, social studies, and even spatial skill development for young children.

One indirect benefit of requiring non-computing disciplinary learning objectives was that many included activities have substantive lesson plans. These lesson plans make the activities more accessible to teachers by including TPACK-related information, such as disciplinary learning objec-

tives for the activity. As discussed in the literature review, teachers can often apply pre-designed computing-integration activities in their class but not create them themselves. As a result, the authors recognize the limitations of requiring non-computing learning objectives but also that it provides a level of authenticity and accessibility for the included activities.

One of the major sources of computing integration activities affected by this requirement was the ScratchEd website. Scratch is a popular language for computing integration activities, aided by an extensive repository of student- and teacher-created projects that users are encouraged to remix into their own projects. The thousands of programs in this repository are of widely varying complexity and quality, and most of them are listed with a topic but without explicit learning objectives. To draw from this wealth of activities without comprehensively including projects, we identified lists of vetted computing integrated activities using Scratch to include in the analysis. These lists were "Integrated Scratch Programming in the Curriculum," "Scratch Projects Across the Curriculum," "From Music to Math: Scratch Across Every Subject," and "Scratch Cross-Curricular Integration Guide." Similarly, resources related to the Snap! language had plentiful examples of projects across disciplines with limited explicit non-CS disciplinary learning objectives.

Block-Based Programming Languages

Because computing integration activities are becoming popular, an initial search revealed too many activities to score in one analysis. For example, in the Exploring Computational Thinking repository originally created by Google and now managed by ISTE, there are 141 activities. To narrow the scope of the analysis, the next selection criterion was that the activity had to use a block-based programming language. The authors intend to analyze other types of programming environments in later analyses. As discussed in the introduction, this criterion has benefits and limitations. One of the main benefits for the goal of the current analysis was that block-based activities include a range of concepts, regardless of their syntactic or semantic difficulty (Grover & Basu, 2017; Papadakis et al., 2014). This benefit means that concepts that best serve the activity can be included for learners with little to no programming experience (Weintrop & Wilensky, 2018). The associated limitation, however, was that concepts are also restricted by the blocks that are built into the language. Most popular languages use a low-floor, high-ceiling design that includes blocks for all concepts that would be taught in an introductory programming course, though (Grover, 2021; Wein-

trop & Wilensky, 2015). Thus, this limitation was not expected to substantially affect the results. Another limitation was that prominent, text-based integration activities, such as Bootstrap's curricula in Algebra and Physics, are excluded, though they are planned to be included in future analyses.

This selection criterion also notably excluded commonly used science simulation platforms, like NetLogo and PhET. These platforms include an extensive range of simulations for scientific phenomena and other models beyond science. While the simulations allow users to easily access the source code, the primary interface does not include the program used to create the simulation. In addition, the source code, except for some adapted NetLogo simulations, is text-based. Though the programs are heavily commented to make them understandable, they do not meet the inclusion criteria for the current analysis. More programming-centric and block-based options for scientific simulations, like StarLogo Nova, were included.

Access

Because the goal of the current analysis was towards a general computational literacy, the accessibility of the activities was the final criterion for inclusion. Following the accessibility criteria used by Lin and Weintrop (2021), we included activities only if they could be found online, were free of cost, did not require a physical device like robotics toolkits, and were updated recently enough that it ran on current versions of languages and operating systems. The requirement to be found online is not expected to substantially narrow the analysis because Lin and Weintrop found that 90% of block-based programming languages ran in a web browser. Exclusion for use of physical devices is a corollary to the requirement to be free of cost. We felt that these criteria would result in a dataset that had the broadest and most equitable applications because many public schools in low-income areas in the US cannot afford physical computing or robotics kits.

Search Criteria

It is important for readers to recognize that the current analysis was based on a review of computing integration activities but not a *systematic* review. Unlike systematic literature reviews of scholarly work on a given keyword or topic area, there are no databases of indexed computing integration activities that span our inclusion criteria. Some repositories for certain languages exist, such as ScratchEd's repository of Scratch projects and

the Exploring Computational Thinking repository of Pencil Code and Python activities. However, computing integration activities are not published through a central organization, so they can be difficult to find.

In lieu of a systematic review, we attempted to build a database that represented activities from a variety of disciplines, student ages, designers, and languages. To create this database, we included any activities that we were already aware of, such as Action Fractions, links from lists of computing integration activities, such as "Scratch Projects Across the Curriculum," links from CSforAll's curriculum directory, and a general Google search for "'integrated computing' activities" and "'computational thinking' + programming" or "'computational thinking' + coding." We examined the first 100 returns for these searches. However, many of the activities found through Google search were excluded based on our criteria, primarily for not including non-CS learning objectives.

We included activities as whole units, whether they were single-class lessons or extended curricular units that included multiple lessons, like Coding as Another Language. Treating individual lessons from curricular units as individual activities would have created over-representation of extended units (e.g., 72 lessons for the kindergarten, 1st, and 2nd grade curricular units from Coding as Another Language instead of three activities). Our database included 81 activities from the following sources:

- CANON Lab
- Code.org's CS Connections
- Code.org's Hour of Code
- Coding as Another Language curriculum
- CS+ units from University of California San Diego
- CSforALL's Curriculum Repository (including 144 curricular units at the time of searching)
- CT4Edu
- Everyday Computing
- Exploring Computational Thinking
- Google search
- Google's CS First
- Integrated computing activities from Georgia State University
- Project GUTS
- ScratchEd
- The Tech Interactive
- TVO Learn
- UCL Scratch Maths

We analyzed the distribution of these activities' characteristics based on primary discipline, student age, programming language, and minimum time to complete (see Table 1). Based on discipline, we recognized that we had only two from history or social studies and searched for additional activities. While we found many projects on ScratchEd's website, they did not meet the selection criteria. Required courses, including language arts, math, and science had a sufficient number of activities, matching their representation in the school day. "Other" disciplines in Table 1 refer to foreign language, spatial skill practice for young children, and an any-discipline vocabulary learning activity. We also had a wide range of activities based on student age and minimum time to complete, so we did not search for any additional activities based on these characteristics.

Table 1

Characteristics of Computing Integration Activities Included in The Database

Discipline	*n*	Grade	*n*	Language	*n*	Minimum Time	*n*
Math	29	K-2	10	Scratch	36	<1 hour	37
Language Arts	22	3-5	46	PencilCode	23	1-3 hours	20
Science	16	6-8	14	AppLab	13	3-8 hours	15
Art/Music	6	9-12	3	StarLogo Nova	5	>8 hours	9
History/Social Studies	2	All	8	ScratchJr	4		
Other	6						

To explore the representation in our database based on programming languages, we used the categories identified in Lin and Weintrop (2021) to ensure coverage of several types of block-based languages. The database has activities from Pencil Code (i.e., block-based implementation of a text-based language), Scratch (i.e., multimedia focused on animations and storytelling), AppLab (i.e., mobile app development), StarLogo Nova (i.e., simulations), and ScratchJr (i.e., pre-reading language). We decided against requiring languages from Lin and Weintrop's other categories for data science, physical computing, and task-specific languages because they did not match our inclusion criteria. We explored other common languages to include, like Alice, Snap!, and App Inventor, but we did not find activities that matched our criteria.

Programming Concept Framework

Initial Programming Concept Framework

Integrated computing activities are often theoretically framed in a CT framework, especially for science and math activities for which national standards (e.g., the Next Generation Science Standards and Common Core) explicitly include CT. Thus, we explored CT frameworks for the programming concepts that they include (Aho, 2012; Armoni, 2016; Barr & Stephenson, 2011; Brennan & Resnick, 2012; Denner et al., 2012; Denning, 2017; Grover & Pea, 2013; Lye & Koh, 2014; Palts & Pedaste, 2020; Tang et al., 2020; Weintrop et al., 2016; Wing, 2010; Yadav et al., 2014). As Tang et al. (2020) highlight in their systematic review of CT frameworks aimed at assessment, many of the frameworks focus on problem-solving competencies rather than programming concepts. The main exception to this trend is Brennan and Resnick's (2012) framework, which includes a category for computational concepts that lists sequences, parallelism, loops, events, conditionals, operators, and data. We used these concepts as the initial foundation for our scoring scheme. In addition, Weintrop et al.'s (2016) definition of CT for math and science integration includes specific data practices, such as creating, manipulating, and visualizing data. We transformed these practices into programming practices, such as inputting variable values, using operators to calculate variables, and various visualization concepts.

To supplement the CT frameworks, we also considered how different block-based programs grouped blocks into menus. These groupings were recently analyzed in Lin and Weintrop's (2021) landscape report of block-based programming languages. Their analysis of 36 languages grouped blocks based on concepts. The conceptual groupings included: variables/data structures, logic operators, sprite/character appearance, operators (numerical, textual, color), customize block (i.e., function), movement, sound, sense/input, interacting with the physical/virtual environment, output, debug, comment, and extension. These categories were added to our initial conceptual framework.

From this theoretical basis, we began an iterative scoring of computing integration activities. Our goal was to refine the framework based on concepts that were missing and differentiate large categories into more specific concepts, such as separating the concept of loops into *for* and *while* loops.

Revisions to Programming Concept Framework

Our full, revised programming concept framework can be found in Table 2, with definitions and examples for each concept that we considered when analyzing the integrated computing activities. In this section, we discuss the evolution of the initial concept framework into this final version. When considering each concept definition, it is important to differentiate the role of the programmer and of the user, which can often be the same person in integrated activities. The programmer is the person creating the program by interacting with the blocks, such as initializing a variable. The user is the person interacting with the program as it runs, such as responding to prompts to change the value of variables.

Certain concept categories were minimally modified from the initial concept framework. The *algorithms* category is higher order than the other concept categories because it does not refer to particular code blocks but instead the nature of the entire program. The category includes Brennan and Resnick's (2012) concepts of sequences and parallelism, which can be used to describe the quality of a program used in an activity. Namely, it describes whether the program involved sequential steps and/or parallel components operating in tandem. We also maintained the *operator* category, which differentiates between arithmetic, relational, and Boolean/logic operators for processing and analyzing data. The last minimally modified category, *functions*, includes using parameters and defining and calling a function, which is sometimes called creating or customizing a code block in the block-based paradigm.

Table 2

Definitions of Programming Concepts Used in Integrated Computing Activities with Examples

Concepts	*Definitions*	*Examples*
Algorithms – nature of the program		
Sequence	The order of blocks was critical to correct execution	Defining the variables before using them to calculate other variables
Parallelism	Used multiple code sequences in the program	Multiple sprites programmed with their own behaviors

Concepts	*Definitions*	*Examples*
Variables (and Data Structures) – how data is stored or managed		
Variable	Used a variable	Assigned a string to the variable "adjective" and used that variable later in the program
sub-concept: +=/ change	Used a += or change block to calculate a new value for a variable	Increase the value of variables to visualize the Fibonacci sequence
sub-concept: calculated variable	Used arithmetic operators or the value of other variables to calculate the value of the variable	Subtract the value of other angles from 180 to find the value of the last angle in a triangle
List	Used a list to assign multiple values to a variable	Used a list to provide multiple options for the variable "adjective"
Operators – how data is processed or analyzed		
Arithmetic	Used +, -, *, /, or other operators to calculate numerical values	Divide a wavelength in half to draw the first half of the wave
Relational	Used <, <=, >, >=, or ==	Compare the value of different fractions
Boolean/Logical	Used and, or, not, true, false, or other logical operators	Determine the quadrant of a graph based on the x- and y-axis value
Loops – how code is repeated		
For loop	Used a loop that repeated for a specific number of times entered by the user or a forever loop that repeated infinitely	A program that repeats a sequence multiple times to draw a symmetrical shape
Sub-concept: loop index	Used an index to count or control how many times to loop	An index within a loop that counts the number of times a loop has run
While loop	Used a loop with a termination condition	Loops until a variable becomes negative
sub-concept: nested loop	Used a loop within another structure, like a loop	Repeat a drawing sequence within a larger repeated sequence to draw symmetrical shapes
Conditionals – how the program makes decisions		
If-then	Used a statement to determine whether a condition was met to determine whether code should be executed	Compare a variable to a predetermined value to determine whether to display a message
If-else	Used a series of statements to determine which matched a given condition	Compare a variable to a predetermined value to determine which message to display
sub-concept: nested conditional	Used a conditional within another structure, like a loop	Repeat a conditional statement for the changing value of a variable within a loop
Functions – chunks of code that are easy to reuse		
Define/call	Called a function defined by the user	Define a function that draws a shape

Concepts	*Definitions*	*Examples*
Parameter	Included a parameter as part of the function definition	Define a function that draws and shape using a parameter for the size of the shape
Visualization – how components move around the screen		
Movement	Moved sprites or objects to visualize a process	Characters in a story move throughout the story
sub-concept: pen up/down	Used the pen up or pen down blocks	Using pen up and down to differentiate between different shapes
Cartesian	Used Cartesian coordinates to determine the location of a sprite or object	Start a character in the bottom left corner by setting their location to negative x and y values
Angles	Changed the direction of sprites or objects using angles	Draw a triangle based on the angles of vertices
Components – audiovisual components to be visualized		
Multimedia	Added a multimedia component to the program such as a sprite, sound, pen, or stamp	Add two characters and a background to a digital story
sub-concept: button	Added a button for the user to click to trigger an event	Add a button that says "Click here when ready"
sub-concept: counter	Added a counter for the number of times an event has occurred	Count how many times a sprite has touched another sprite
Properties	Changed the default properties	Change the color of a pen
Output – the program communicating with the user		
String	Displayed a string entered by the programmer	A character is programmed to say a line
Variable	Displayed the value of a variable, either string or numeric	The program prints the final value of a variable
String and Variable	Displayed a string and value of a variable in the same output statement	A character says, "And you picked," + a variable selected by the user
Input – the user communicating with the program		
Event	Triggered the execution of code based on the action of the user, sensor, or internal feature	A character in a story will not talk until the user clicks them or a message is broadcasted
Variable: String	The user entered a string into the program	A poem generator asks for a verb
Variable: Numerical	The programmer entered a starting variable value or asks the user for input	The user picks a number
Cleaning/ transforming	Used a cleaning or transforming process on user-inputted data	Change all text to lowercase

The category *variables (and data structures)* evolved from data concepts that originated in all three source frameworks related to data storage and handling (Brennan & Resnick, 2012; Lin & Weintrop, 2021; Weintrop et al., 2016). This category primarily revolved around data being passed through the program as variables. During our iterative review of integrated activities, we found recurring use cases that we added as sub-concepts for using variables. In our framework, *sub-concepts* are particular ways of applying a concept within an activity that would not be taught separately but also are not necessary for applying the concept. In this case, the first is using += or *change*, which is often included in the base set of blocks, to update a variable based on the current value. The second is using operators to dynamically calculate the value of a variable, compared to the programmer entering a static value for a variable. In this category, we put "and data structures" in parentheses because the use of multifaceted data structures is rare in the activities we analyzed. Except in one case of importing a spreadsheet, the most activities would do, occasionally, is use a list.

We adopted the *loop* category from Brennan and Resnick's framework to describe how programs repeat code. This category does not appear in Lin and Weintrop's (2021) framework because loops are often included in the control category with conditionals (Brennan & Resnick, 2012). We added a distinction between *for* and *while* loops to mark a difference in repeating part of the program for a specified number of times, including forever, as opposed to while a continuation condition is met. One area of discussion among scorers was how to categorize a *for* loop that repeated for a dynamic value, such as the length of a list (i.e., *for each*). We decided to categorize this as a *for* loop and added a sub-concept for a loop index variable to indicate cases when the number of repetitions was not static. We also added a sub-concept for nested loops (i.e., when a loop is inside another control structure) as a recurring, though somewhat rare, use case.

The category *conditions* describe how programs make decisions and had the same origination as loops. We added a differentiation between *if-then* and *if-else*. These are typically represented as different blocks, but we also wanted to mark the conceptual difference between them. The *if-then* conditionals were those that checked for a specific condition and executed code only when it was met. The *if-else* conditionals were those that served as a decision tree, affecting the flow of the program regardless of which condition was met. Like in loops, we also added a sub-concept for nested conditionals for when conditionals were inside another control structure.

The *visualization* category stemmed from data concepts from all three frameworks but focused primarily on visualizing data (Brennan & Resnick,

2012; Lin & Weintrop, 2021; Weintrop et al., 2016). The initial version of this category was later split between visualization and components. The visualization category includes concepts that are not necessarily programming concepts but are common enough in integrated activities to warrant discussion, like the movement of a component to show a process. A particular use case we highlight as a sub-concept is using the pen up or down blocks, which are often in the base set of blocks. The other two concepts are math concepts commonly applied in the integrated activities – use of coordinates in the Cartesian plane and use of angles.

The other category that stemmed from visualization, *components*, is more directly related to programming concepts for multimedia elements. The first concept is creating a multimedia object, such as a sprite, sound, or pen. The second concept is changing the properties of that object, such as adding a file or picking an option from a preset list. We also included two unique recurring options as sub-concepts: buttons and counters.

The last two categories manage the input and output data for a program, which describe how users communicate with the program and how programs communicate with users. In the *output* category, we included outputting a string that is predetermined by the programmer, such as a line of dialogue. The other type of output was printing the current value of a variable. We also specifically scored activities for whether they output both a string and variable in the same line of code, which includes a different procedure than outputting them on separate lines.

The final category was for *inputs*, which included inputs from the programmer and user. One type of input was a variable, and it was either a string or numerical value except in rare cases. The other type of input was an event, which is explicitly included in Brennan and Resnick's (2012) framework. Events could be triggered by sensors, internally programmed as information is passed, or based on input from users. Input from users included either pressing a button to trigger an effect, like the arrow key to move a sprite, or entering data that was stored as a variable. In the case of entering data, we included a sub-concept for any sort of automatic cleaning or transformation of that data, like removing punctuation.

Scoring Activities for Concepts

Activities were scored by two of three raters for whether they included a concept. We decided to score activities for whether they included a concept rather than the number of times a concept is used. Our goal was to de-

termine which concepts are more commonly used across computing integration activities, and scoring for the number of times a concept was used in an activity might have skewed the results based on incidental features of the activities. For example, if an activity used five variables, the conceptual knowledge required is not necessarily different than if an activity used two variables. Thus, scoring for times a concept was used was more likely to result in over-representation of the concept rather than information about which concepts were most commonly used in computing integration.

The raters were three CS education researchers. Two have a Ph.D. in areas related to CS education, one from educational psychology and one from CS and have worked and published in computing education for at least five years. The other has a master's degree in CS and is earning a Ph.D. in instructional technology with an emphasis on CS education. Each activity was scored by at least one rater with a degree in education and at least one rater with a degree in computer science.

To determine interrater reliability, we used Cohen's Kappa. Cohen's Kappa is appropriate to compare binary data from two raters, such as whether a concept is used in an activity. Cohen's Kappa compares the observed agreement, in this case 96%, to the chance of agreement, calculated from the base rate of each binary option from each rater. For example, if rater one marked 80% of concepts as present in each activity on average while rater two marked 20% of concepts as present, then the chance of agreement would be low. A higher Kappa indicates higher agreement, with 0.81 to 1.00 indicating nearly absolute agreement (McHugh, 2012). For the current study, Kappa was 0.92, indicating nearly absolute agreement.

RESULTS

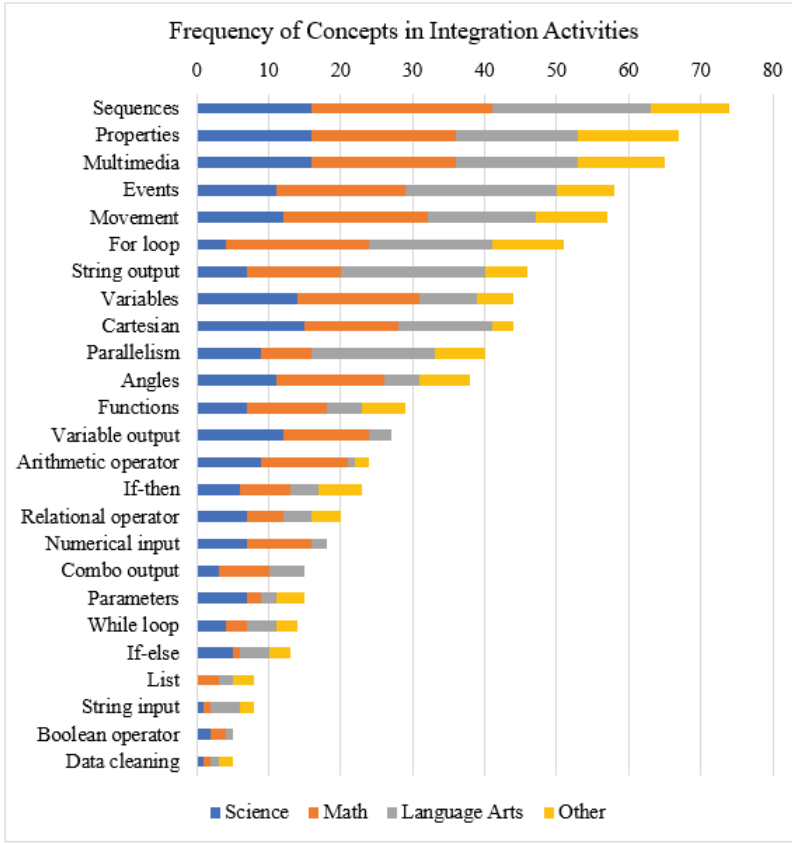
By scoring the concepts used in computing integration activities included in our database of 81 activities, we can look at bottom-up trends across activities to address our first research question. Our results describe how common concepts, defined in Table 2, were used across activities. The percentage of activities including each concept can be seen in Figure 2.

For each concept, to address our second research question, we explore the distribution within unique features of each activity, including discipline, student age, programming language, and minimum time on task. in regard to discipline, only language arts, math, and science had substantial representation, so any differences in distribution in art/music, history/social studies, or other disciplines are not likely to be reliable. As such, we focus on only

these three disciplines for this analysis, which are also represented in Figure 2. For student age, we examined the earliest age at which an activity could be completed. Language was similar to discipline in that there was little representation in ScratchJr and StarLogo Nova. Within languages, we highlight cases when common concepts are never used, especially if there was no block for the concept. Uneven distributions across these activity features will be called out in the results. Otherwise, the reader can assume an even, relative (i.e., percentage-based) distribution across categories.

Figure 2

Frequency Counts of Concepts Identified Within the 81 Integrated Computing Activities in Our Database



The last distribution that we considered was the minimum time to complete the activity, but we found no notable differences in concept frequency across this feature. We expected that less common programming concepts would appear only in the longer duration activities because activities that included less common concepts typically also included the more common concepts. Thus, activities that included less common concepts generally included more concepts in total, which was expected to result in longer activities. However, this was not true. All concepts were evenly distributed among activities of different durations.

Most Frequent Programming Concepts

Based on the distribution of frequencies for each concept, the most frequent programming concepts are defined as those that are present in more than half of the activities. They include 9 of the 25 concepts in the framework (excluding sub-concepts). No sub-concepts, which are specific applications of concepts that recur in the data, were used in over half of the activities.

The most frequent concept—sequences—came, unsurprisingly, from the algorithms category, which were found in 91% of activities. Earlier ages were more likely to not use this concept, i.e., not need the program to be in a particular order to work. All activities from 4th grade/age 9 and later included sequences. The simplicity of programs for the younger age group was likely a contributing factor, such as sprites doing only one thing. In addition, sequences were only absent from the activities in AppLab and Scratch that focused on digital storytelling.

The next most frequent concepts were multimedia components, 80%, and their properties, 83%, from the components category. In a couple of cases, multimedia components, like a background picture, were pre-created for learners, so they only had to set properties. These concepts were distributed evenly across all activity features.

About 70% of the activities included events or movement. Events, such as clicking on a character in an animation, were more frequent in language arts than in math or science. They were also less common in Pencil Code than in the other languages, likely a result of the use of events in digital storytelling activities. The next concept, movement, was evenly distributed across all features. Movement is not a programming concept but blocks for movement were used frequently enough to include in this analysis, such as to draw with a pen or move a character.

The last four concepts in this category neared 50% use. *For* loops (used to repeat code for a set number of times), 63%, were less common in science than language arts or math. String outputs (used to show a predetermined set of characters, often text), 57%, were most common in language arts, perhaps unsurprisingly given the use of dialogue. Variables (used to store data values), 54%, were least common in language arts, again unsurprisingly given the use of variables in math and science. Last, another non-programming topic, the use of Cartesian coordinates, 54%, was more common in science than language arts or math. In addition, Cartesian coordinates were more common for older students, used in about half of activities starting at 3rd grade/8 years old.

Frequent Concepts Summary

Perhaps the most striking takeaway from these results is that the most frequently used concepts in integrated computing activities are not the concepts that are typically the focus of instruction in introductory programming courses. Concepts of sequences, multimedia components and their properties, and events might be discussed, but they are not typically featured. Further, non-programming concepts tied to specific blocks are included in the most frequent list: movement and Cartesian coordinates. With these concepts aside, only *for* loops, variables, and string outputs appear in the most frequently used concepts. Each of these represents, most likely, a day of instruction in an introductory programming course. This difference is likely due to a difference in goals and knowledge, which will be expanded upon in the discussion section.

Common Programming Concepts

The next level of programming concepts used in integrated computing activities is those found in 25% to 50% of activities. We found 7 of 25 concepts were commonly used. Again, no sub-concepts were found with this frequency range. In this intermediate frequency category, the most common concept was again from the algorithms category. Parallelism, which allows multiple things to happen at once in a program, was in 49% of activities and most common in language arts, especially for digital storytelling activities with multiple sprites. It also did not appear in any of the Pencil Code activities. Pencil Code can support parallelism, but as a Logo/turtle-based

programming language, it is not commonly used to support multiple turtles. The next concept was another non-programming concept, the use of angles, 47%. Unsurprisingly, angles were less common in language arts than in math or science activities.

The next two concepts were in about 35% of activities. Functions, which allow programmers to more easily reuse code, were distributed evenly across activity features. The other concept, an output that showed the value of a variable, shifted from rare to more common around 5th grade/10 years old, and it was most common in science.

The last group of concepts came from the conditional and operator categories. *If-then* conditionals, 28%, were evenly distributed across features. Arithmetic operators (e.g., + or -), 30%, were rare before 3rd grade/8 years old, which makes sense given students are still learning arithmetic at this age. They were also less common in language arts than in math or science, again unsurprisingly. Relational operators (e.g., <, >, =), 25%, were most common in science. They did not appear in activities that used AppLab or ScratchJr.

Common Concepts Summary

Compared to the most frequently used concepts, these commonly used concepts are more representative of concepts that comprise the bulk of instruction in introductory programming courses. However, the most common concepts within this category are again concepts that appear briefly in a standalone computing course. Not until we get to concepts in about a third of activities do we see concepts like functions, conditionals, or operators. One implication for this result is that even students engaged in integrated computing activities might have little exposure to concepts that would be the focus of instruction in standalone introductory programming courses.

Uncommon Programming Concepts

The last level of programming concepts was those found in fewer than 25% of activities. We found 9 of 25 concepts were used in less than a quarter of activities, plus all of the eight sub-concepts. Within uncommonly used concepts, the most frequent concepts were related to the input and output of data. Users inputting numerical variables, 22%, were evenly distributed among most features but absent in AppLab and ScratchJr activities, which

had no blocks to support this functionality. For output, the combination of variable and string within the same output, 19%, had no instances before 3rd grade/8 years old, but was otherwise evenly distributed. Given that this code commonly includes an arithmetic operator, +, this aligns with the relatively low use of arithmetic operators.

The next few concepts represent more advanced programming concepts in a programming course. Parameters, which are used with functions, were found in 19%. *While* loops, which repeat code while a condition is true, were found in 17%. *If-else* conditionals, which tell the program how to make decisions, were found in 16% of activities. Despite being more advanced programming concepts, they appeared evenly over disciplines, languages, and surprisingly, student ages. Though perhaps the more appropriate interpretation of these data is that they were equally uncommon across these features.

The last four concepts all appeared in less than 10% of activities. Users inputting non-numerical variables and the use of lists were both evenly distributed across activity features. Boolean operators (e.g., AND, NOT) were equally common among student ages starting at 3rd grade/8 years old. Lastly, code used to clean or transform input data was used in only five activities, primarily in Pencil Code activities. Given the low frequency, these distributions might be an artifact of the database rather than important distinctions.

Uncommon Concepts Summary

Most of these uncommon concepts are those that represent more advanced programming concepts or at least concepts that do not appear outside of computing or related fields. For example, it is uncommon to use Boolean operators outside of computing or technology use, so it makes sense that they would not be used for activities that primarily serve learning objectives in another discipline. The other commonality in these uncommon concepts is that many of them handle data inputs or outputs. Other disciplines handle data frequently, but concepts like cleaning and transforming data are not often automated outside of computing, even if they could be because the process of cleaning data follows algorithmic rules. The infrequency of these input and output concepts is also likely due to those concepts being more user-facing, which is not commonly part of integration activities.

Sub-concepts

All eight sub-concepts were used infrequently, which is expected given that they are particular applications of the concepts. For all sub-concepts, they were evenly distributed across activity features. Again, this distribution mostly means that they were uncommon across all features. The most common sub-concept was calculating a variable, 23%. This concept is the only one with an uneven distribution because it was rare before 3rd grade/8 years old. Also related to the value of variables, the += or change blocks were used in 14% of activities.

Nested loops and conditionals were a couple of the more common sub-concepts with nested loops in 15% of activities and nested conditionals in 9%. Loop index variables were also used in 9% of the activities. None of these sub-concepts represent difficult programming concepts, but they increase the complexity of the program, especially for novice programmers. The remaining three sub-concepts were related to visualization and multimedia. The pen up/down, button, and counter blocks were each used in 11% of activities. These sub-concepts, based on the integration activities, seem like the type of concept that might not be used within a block-based language, except that there are blocks to support them. Many of the activities did not seem like these features were essential to the program.

DISCUSSION

The goal of the current analysis was to provide a bottom-up examination of the programming concepts used in integrated computing activities that focus on non-CS learning objectives. This perspective serves as a complement to top-down approaches that ensure certain CS concepts are included. Our main finding suggests there are significant differences in the most frequently taught concepts in the analyzed integration activities compared to standalone programming instruction, which is often a basis for top-down approaches. This finding is important because it contradicts a narrative in CS education that integration activities preview concepts that students learn in standalone programming courses, at least in the subset of activities included in this analysis.

While the authors expected more alignment between bottom-up and top-down designs of integration activities, the selection criteria to require non-CS disciplinary learning objectives biased our database towards activities created primarily from the perspective of the non-CS discipline. Thus,

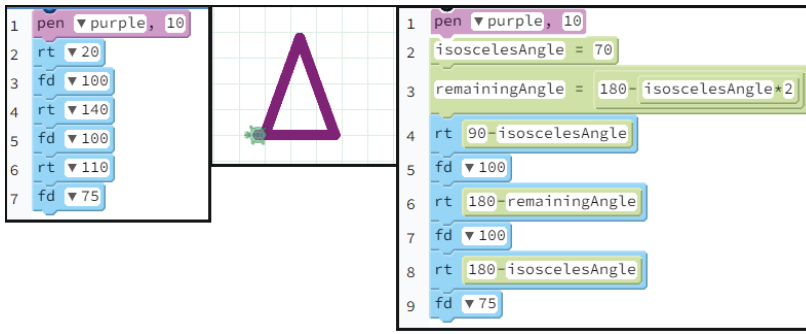
the activities in the current analysis do not have a primary goal of teaching programming. Instead, the selection criteria were designed to identify integration activities in which the goal was to support disciplinary learning. In this context, it is unsurprising that many of the most frequent concepts support visualization of processes and the use of multimedia to express ideas through animation.

These visualization and multimedia concepts are valuable in other disciplines, otherwise they would not be the most frequent concepts, but they do not harness the automated information processing potential of programming. Much of the visualization and multimedia concepts in integrated computing activities are simply a different medium in which to express ideas. As an example of a computing integration activity that does or does not use automated information processing, consider students learning about different types of triangles (see Figure 3). In a computing integration activity, instead of students drawing triangles using a pencil, paper, and protractor, they might program a turtle. With no automation concepts (left-hand side), they must manually calculate and enter the value of each angle. Thus, the programming environment is a different medium, but conceptually no different, than paper. To add automation (right-hand side), if students are drawing isosceles triangles, then two of the angles will always equal each other. In a programming environment, one variable can be used to represent both the matching angles. Furthermore, that variable can be used with arithmetic operators to calculate the value of the remaining angle. As a result, they must only enter one angle, and the program calculates the rest. In this case, the student is automating information processing, which requires them to formalize principles of geometry in the program, reinforcing their knowledge of geometry and teaching them how to enable automatic calculations with programming.

Of the activities we analyzed, 12% included no programming concepts that enable automated information processing (i.e., variables, operators, loops, conditionals, or functions). While this represents a small percentage of the activities, the authors question to what extent these activities support computational literacy compared to any other kind of technology-enhanced learning activity. Potential benefits include exposing students to programming languages, which helps to demystify programming (Margulieux et al., 2022) and introduce algorithms. However, within integrated computing, there is an open question about the effects of this type of automation-free activity and those that include automated information processing. This question is particularly important because students tend to easily apply multimedia concepts, such as sprites and loops, but struggle to apply deeper automation concepts (Franklin et al., 2017).

Figure 3

Information Processing Concepts in Code



Note. Two programs that can draw an isosceles triangle (middle panel). The left example uses no information processing concepts and must be manually updated throughout to change the angles. The right example automates information processing so that only the first variable must be updated to change the angles.

Programming Instruction Recommendations for Teacher Preparation

In the current analysis, most activities used an automated information processing concept. In the most frequent category, the two automation concepts were variables and *for* loops. Many activities also used an event, but in most cases the event was when the program was run, so that concept will not be discussed as a common automation concept. Variables allowed for a level of abstraction, a common CT concept, which enabled a value to be calculated or called throughout a program. They were used more frequently in math and science activities, likely due to the familiarity of variables in those fields, even though programming variables are conceptually different than variables in math or science. *For* loops allowed for processes to be repeated with no additional effort, drawing upon the strengths of a computer. In this way, loops were frequently used to reduce sustained effort in cyclical activities, such as for young students learning about patterns.

These concepts are basic from a programming perspective, but they provide an entry point to thinking about the computer as a tool for automation. In addition, these concepts are analogous to concepts that students and teachers already understand. Even before students learn about variables in math or science, they learn about abstractions. For example, they learn

that dogs, cats, and fish are all animals, and variables in activities for young students are often used in this way, as an abstract category for a concrete value (e.g., animal = dog, animal = cat, or animal = fish). In the case of loops, students know how to repeat a procedure and recognize that if they can articulate the procedure as an algorithm, then the computer can repeat it instead. Other automation concepts that are familiar outside of computing are conditionals for making decisions, arithmetic operators or calculating numbers, and relational operators for comparing values, but they were used less frequently in our database of activities.

This finding has interesting implications for one of our expected outcomes—to identify which concepts are most frequently used in integration activities and, thus, should be prioritized in teacher professional development. Because many of the most frequently used concepts were basic from a computing perspective, perhaps teachers need instruction in only basic concepts to apply integrated computing activities in their classrooms. Further, in terms of differences among disciplines, activities tended to include computing concepts that have analogs in the discipline, such as variables and arithmetic operators in math. Within disciplines, it could be particularly worthwhile to explore the differences between these concepts in each domain (e.g., how is a variable in science different than a variable in CS) because they are likely to be used in an integration activity.

Conditionals and arithmetic and relational operators were expected by the authors to be more frequent than in 25-30% of activities. *If-then* conditionals are analogous to decision trees, which are applicable to any discipline, but perhaps are more appropriate at higher levels of complexity than is common for integrated computing activities. For example, in Physics, learners often need to select an appropriate formula based on available values. While all possible formulas could be represented in a program with a conditional to select between them, that type of application would be somewhat complex. The complexity might also explain why arithmetic and relational operators were not used. Even less frequently used (about 17%) were *if-else* conditionals and *while* loops, which should be no more conceptually inaccessible than their *if-then* conditional and *for* loop counterparts.

Perhaps one direction for the design of integrated computing activities, then, is as end-of-unit projects that require a higher level of complexity than the activities we reviewed. A project like this would encourage students to consolidate all that they have learned about a topic into an algorithm, or parallel algorithms, that explicitly define the components and the relationships among them. This consolidation would serve disciplinary learning objectives (i.e., summarizing knowledge) and programming learning objectives

(i.e., learning to automate information processing). In addition, because students have already been introduced to the disciplinary concepts, they might be better suited to learning new programming concepts than learning them concurrently. The use of block-based languages might be particularly appropriate for an assignment like this because students would be able to search and implement various programming concepts that would be useful. This kind of project is commonly seen on ScratchEd, but more teacher support documentation, like lesson plans, would help broaden their use.

While functions can be used for automation, we argue that functions, as used in integrated computing activities, are different than other automation concepts. Though they were used in about a third of activities, they were often defined and used exactly once. This use pattern contradicts functions' intended purpose—to be used multiple times throughout a program to limit redundancy. Further, only half of these cases used a parameter, which would make the function operate differently than the same code outside of a function. Functions are a core component of computer science because they are used multiple times in an application, and they use parameters to handle variations of processes. However, this functionality is currently not used in the paradigm of integrated computing activities. Within the current paradigm, functions seem to be added, in the opinion of the authors, to introduce the concept of functions but outside of a context in which the usefulness of functions can be demonstrated. Thus, it often adds an unnecessary level of complexity that might be prone to confusing students and teachers. That hypothesis would need to be addressed with future empirical work.

Data in Integrated Computing Activities

Only in rare cases did the integration activities include data concepts more advanced than using variables. While many activities used variables and gave an output that was either a string or a variable, most activities did not handle data input, use lists, arrays, spreadsheets, or databases. Data is used in every discipline in one way or another, so it was surprising that data concepts were not more prevalent in integration activities.

There are two likely explanations for this trend. The first is that many of the programs had no user interaction element, so there was no user to input data or require an output. If our activity database had included more user-centric applications, like those created in App Inventor, the result would likely be different for input and output concepts. The other explanation is that many block-based languages have very few blocks to support data fea-

tures. For example, Pencil Code is able to read a spreadsheet or use a list because it is a hybrid block/text language, but there are no blocks for those features. Many languages include a block for a list, but not more advanced data structures. Snap! can be used for more advanced data concepts, but we could not find activities that used Snap! and met our search criteria.

Our search criteria directly affected this finding, which is not expected to be replicated in other databases of activities. For example, the Data Science curriculum from Bootstrap, which follows a top-down design approach and uses a text-based language, would clearly include more data concepts. In the context of rising interest in data science as an area for computing integration, however, it is important to recognize that block-based languages are still highly popular for novice programmers. Thus, the limitations of many block-based languages regarding data have important implications, especially for importing or exporting data.

CONCLUSION

Computing integration activities are a promising tool to 1) support disciplinary learning through computing-powered tools, and 2) introduce a broader range of students to computing by applying it outside of standalone CS courses. As the CS education community strives to increase computational literacy globally, the role of integration activities deserves a thorough examination. The goal of this paper was to complement top-down approaches to design these activities based on concepts taught in introductory programming courses with a bottom-up analysis of integration activities designed primarily within non-CS disciplines. The current analysis provides a new perspective of computing integration activities based on the frequency with which programming concepts were used.

The main finding from the current analysis is that the most frequent programming concepts used in integration activities do not match programming concepts typically emphasized in introductory programming courses. While programming courses emphasize concepts that support the automation of information processing (i.e., variables, operators, loops, conditionals, and functions), the activities we analyzed emphasized concepts that supported visualization and animation. Part of this mismatch might be caused by the use of block-based languages, which are visual in nature and almost always automatically include a sprite or turtle to program. However, the authors argue that programming visual elements is the easiest connection to other disciplines without a major paradigm shift, which drives this finding.

This connection might even be behind the popularity of block-based languages for computing integration activities.

In contrast to visualization concepts, the use of automation concepts was relatively light in our database of activities. While many activities used a variable or *for* loop, these are relatively simple programming concepts. More advanced concepts like *while* loops, *if-then* or *if-else* conditionals, functions, parameters, and data structures like lists were all relatively rare. The authors are not placing a judgment on the frequencies of these concepts and whether they should be included in integration. However, 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. In addition, it can inform the choices teacher educators who are infusing computing integration into their own disciplinary curriculum make when preparing their students to use integration activities in their own practice.

As stated in the methods section, finding computing integration activities can be difficult, especially from the perspective of a teacher in a non-CS discipline. One result of our search is creating a new database to centralize links to computing integration activities that are designed for teachers to use. This database includes all activities reviewed in the current analysis and an option to submit new activities. It categorizes activities based on the primary discipline, student age, programming language, and minimum time to complete the activity. Each entry also provides information about the topic being taught, the source, and links to the program and teacher support materials. The database can be accessed at integratedcomputing.org. We hope that the community will help us in building a robust database of activities for teachers to adopt in their classrooms and support computational literacy for all learners.

ACKNOWLEDGEMENTS

This work is funded in part by the National Science Foundation under grant #1941642.

REFERENCES

Aho, A. v. (2012). Computation and computational thinking. *The Computer Journal*, *55*(7), 832–835.

Armoni, M. (2016). COMPUTING IN SCHOOLS Computer science, computational thinking, programming, coding: The anomalies of transitivity in K–12 computer science education. *ACM Inroads*, *7*(4), 24–27.

Barr, V., & Stephenson, C. (2011). Bringing computational thinking to K–12: What is involved and what is the role of the computer science education community? *ACM Inroads*, *2*(1), 48–54. https://doi.org/10.1145/1929887.1929905

Brennan, K., & Resnick, M. (2012, April). New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 Annual Meeting of the American Educational Research Association* (Vol. 1, p. 25).

da Silva, D., Kurtz, F. D., & Paludo Santos, C. (2020). Computational thinking and TPACK in science education: A southern-Brazil experience. *Paradigma*, *41*(2).

Denner, J., Werner, L., & Ortiz, E. (2012). Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts? *Computers & Education*, *58*(1), 240–249.

Denning, P. J. (2017). Remaining trouble spots with computational thinking. *Communications of the ACM*, *60*(6), 33–39.

DiSessa, A. A. (2000). *Changing minds: Computers, learning, and literacy*. MIT Press.

Fields, D. A., Quirke, L., Amely, J., & Maughan, J. (2016). Combining big data and thick data analyses for understanding youth learning trajectories in a summer coding camp. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (pp. 150–155). Association for Computing Machinery. https://doi.org/10.1145/2839509.2844631

Franklin, D., Skifstad, G., Rolock, R., Mehrotra, I., Ding, V., Hansen, A., ..., & Harlow, D. (2017). Using upper-elementary student performance to understand conceptual sequencing in a blocks-based curriculum. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 231–236). Association for Computing Machinery. https://doi.org/10.1145/3017680.3017760

Grover, S. (2021, April 13). *'CTIntegration': A conceptual framework guiding design and analysis of integration of computing and computational thinking into school subjects*. https://doi.org/10.35542/osf.io/eg8n5

Grover, S., & Basu, S. (2017). Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and Boolean logic. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 267–272). Association for Computing Machinery. https://doi.org/10.1145/3017680.3017723

Grover, S., & Pea, R. (2013). Computational thinking in K–12: A review of the state of the field. *Educational Researcher*, *42*(1), 38–43.

Guzdial, M. (2004). *Programming environments for novices*. Citeseer.

Kale, U., Akcaoglu, M., Cullen, T., Goh, D., Devine, L., Calvert, N., & Grise, K. (2018). Computational what? Relating computational thinking to teaching. *TechTrends*, *62*(6), 574–584.

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, *37*(2), 83–137.

Koehler, M., & Mishra, P. (2009). What is technological pedagogical content knowledge (TPACK)? *Contemporary Issues in Technology and Teacher Education*, *9*(1), 60–70.

Kong, S.-C., & Abelson, H. (2019). *Computational thinking education*. Springer Nature.

Kong, S.-C., & Lai, M. (2021). A proposed computational thinking teacher development framework for K–12 guided by the TPACK model. *Journal of Computers in Education*, 9(3), 379–402. https://doi.org/ 10.1007/s40692-021-00207-7

Kong, S.-C., Lai, M., & Sun, D. (2020). Teacher development in computational thinking: Design and learning outcomes of programming concepts, practices and pedagogy. *Computers & Education*, *151*, 103872.

Kopcha, T. J., Neumann, K. L., Ottenbreit-Leftwich, A., & Pitman, E. (2020). Process over product: The next evolution of our quest for technology integration. *Educational Technology Research and Development*, *68*(2), 729–749. https://doi.org/10.1007/s11423-020-09735-y

Lee, I., Martin, F., & Apone, K. (2014). Integrating computational thinking across the K–8 curriculum. *ACM Inroads*, *5*(4), 64–71.

Lin, Y., & Weintrop, D. (2021). The landscape of block-based programming: Characteristics of block-based environments and how they support the transition to text-based programming. *Journal of Computer Languages*, *67*, 101075.

Luo, F., Israel, M., & Gane, B. (2022). Elementary computational thinking instruction and assessment: A learning trajectory perspective. *ACM Transactions on Computing Education.*, *22*(2), 1–26. https://doi.org/10.1145/3494579

Luxton-Reilly, A., Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., Ott, L., …, & Szabo, C. (2018). Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (pp. 55–106). Association for Computing Machinery. https://doi.org/10.1145/3293881.3295779

Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K–12? *Computers in Human Behavior*, *41*, 51–61.

Margulieux, L. E., Enderle, P., Junor Clarke, P. A., King, N., Sullivan, C., Zoss, M., & Many, J. (2022). Integrating computing into preservice teacher preparation programs across the core: Language, mathematics, and science. *Journal of Computer Science Integration, 5*(1), 1–16. https://doi.org/10.26716/jcsi.2022.11.15.35

McHugh, M. L. (2012). Interrater reliability: The Kappa statistic. *Biochemia Medica*, *22*(3), 276–282.

Mouza, C., Yang, H., Pan, Y.-C., Ozden, S. Y., & Pollock, L. (2017). Resetting educational technology coursework for pre-service teachers: A computational thinking approach to the development of technological pedagogical content knowledge (TPACK). *Australasian Journal of Educational Technology*, *33*(3). https://doi.org/10.14742/ajet.3521

Palts, T., & Pedaste, M. (2020). A model for developing computational thinking skills. *Informatics in Education*, *19*(1), 113–128.

Papadakis, S., Kalogiannakis, M., Orfanakis, V., & Zaranis, N. (2014). Novice programming environments. Scratch & app inventor: A first comparison. In *Proceedings of the 2014 Workshop on Interaction Design in Educational Environments* (pp. 1–7). Association for Computing Machinery. https://doi.org/10.1145/2643604.2643613

Papert, S. A. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books.

Rich, K. M., Strickland, C., Binkowski, T. A., Moran, C., & Franklin, D. (2018). K–8 learning trajectories derived from research literature: sequence, repetition, conditionals. *ACM Inroads*, *9*(1), 46–55.

Saritepeci, M. (2021). Modelling the effect of TPACK and computational thinking on classroom management in technology enriched courses. *Technology, Knowledge and Learning*, *27*(4), 1155–1169.

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

Tang, X., Yin, Y., Lin, Q., Hadad, R., & Zhai, X. (2020). Assessing computational thinking: A systematic review of empirical studies. *Computers & Education*, *148*, 103798.

Waterman, K. P., Goldsmith, L., & Pasquale, M. (2020). Integrating computational thinking into elementary science curriculum: An examination of activities that support students' computational thinking in the service of disciplinary learning. *Journal of Science Education and Technology*, *29*(1), 53–64.

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology*, *25*(1), 127–147.

Weintrop, D., & Wilensky, U. (2015, June). To block or not to block, that is the question: Students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children* (pp. 199–208). Association for Computing Machinery. https://doi.org/10.1145/2771839.2771860

Weintrop, D., & Wilensky, U. (2018). How block-based, text-based, and hybrid block/text modalities shape novice programming practices. *International Journal of Child-Computer Interaction*, *17*, 83–92.

Wilensky, U., Brady, C. E., & Horn, M. S. (2014). Fostering computational literacy in science classrooms. *Communications of the ACM*, *57*(8), 24–28.

Wing, J. M. (2010). Computational thinking: What and why? *The Link.* https://www.cs.cmu.edu/link/research-notebook-computational-thinking-what-and-why

Yadav, A., DeLyser, L. A., Kafai, Y., Guzdial, M., & Goode, J. (2021). Building and expanding the capacity of schools of education to prepare and support teachers to teach computer science. In C. Mouza, A. Yadav, & A. Ottenbreit-Leftwich (Eds.) *Preparing pre-service teachers to teach computer science: Models, practices, and policies* (pp. 191–204). Information Age Publishing.

Yadav, A., Gretter, S., Good, J., & McLean, T. (2017). Computational thinking in teacher education. In P. J. Rich & C. B. Hodges (Eds). *Emerging research, practice, and policy on computational thinking* (pp. 205–220). Springer.

Yadav, A., Mayfield, C., Zhou, N., Hambrusch, S., & Korb, J. T. (2014). Computational thinking in elementary and secondary teacher education. *ACM Transactions on Computing Education (TOCE)*, *14*(1), 1–16.

Yadav, A., Stephenson, C., & Hong, H. (2017). Computational thinking for teacher education. *Communications of the ACM*, *60*(4), 55–62. https://doi.org/10.1145/2994591