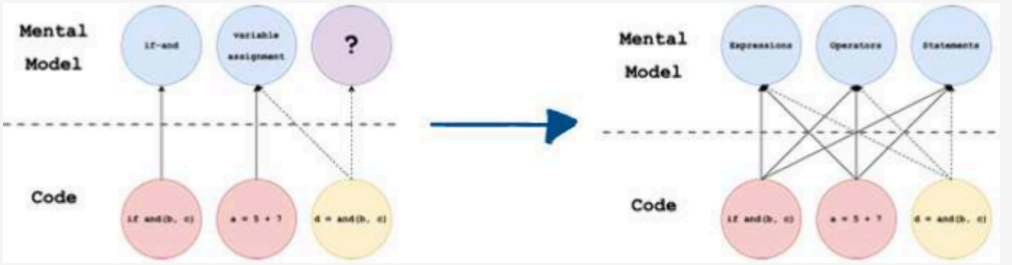
JOURNAL OF CHEMICAL EDUCATION

pubs.acs.org/jchemeduc
Chemical Education Research

Recommendations for Improving End-User Programming Education: A Case Study with Undergraduate Chemistry Students

William Fuchs, Ashley Ringer McDonald,* Aakash Gautam,* and Ayaan M. Kazerouni*

Cite This: https://doi.org/10.1021/acs.jchemed.4c00219
Read Online



ABSTRACT: Programming is widespread in multiple domains and is being integrated into various discipline-specific university courses where, like students in a typical introductory computing course, students from other disciplines face challenges with learning to program. We offer a case study in which we study undergraduate students majoring in either chemistry or biochemistry as they learn programming in a physical chemistry course sequence. Using surveys and think-aloud sessions with students, we conducted a thematic content analysis to explain the challenges they face in this endeavor. We found that students struggled to transfer their programming knowledge to new representations and problems, and they did not have strategies in place for solving problems with programming. These facts combine to lower students' confidence in their programming abilities, making it less likely that they will reach for computing to help solve domain-specific problems. We recommend that students in end-user programming contexts be explicitly taught the skills of abstraction, decomposition, and metacognitive awareness as they pertain to programming.

KEYWORDS: *Chemical Education Research, End-User Programming Education, Abstraction, Decomposition, Metacognitive Awareness*

INTRODUCTION

Computer programming is commonplace across disciplines (e.g., climate science[1] and chemistry[2]), carried out by *end-user programmers*, so-called because they are often the end users of their own software. They are estimated to vastly outnumber professional software engineers,[3] and they often work on critical software systems. Consequently, programming is being integrated into various non computer science courses[2,4,5] to produce disciplinary experts who are also proficient programmers.

At our university, students majoring in chemistry or biochemistry (two majors offered by the same department) are taught programming over a required sequence of four physical chemistry (PChem) courses. The full curriculum, including the programming aspects, is described in detail in a previous publication.[2] The goal of the curriculum is to impart a deeper understanding of the PChem concepts by enabling students to solve problems using programming that would be difficult or impossible to solve without programming.

Unfortunately, many students in the sequence struggle with learning programming and the vast majority opt out of elective computational chemistry experiences. We wish to more deeply understand their challenges and identify potential areas for improvement in the course sequence.

We present a case study about the challenges faced by these students learning programming through analysis of concept inventory and survey responses as well as interview transcript data. We used a thematic analysis to answer the following research question: *What are the challenges faced by students while learning programming in a discipline-specific end-user programming context?* On the basis of our findings, we offer recommendations for the teaching of programming in these types of contexts and a brief description of how our recommendations might help.

Received: February 27, 2024
Revised: May 9, 2024
Accepted: June 14, 2024

BACKGROUND

To motivate the learning of programming for students from other disciplines, educators have often turned to contextualized pedagogical approaches, where programming content is integrated into discipline-specific courses. This section sketches the relevant research, describing the benefits and drawbacks of highly contextualized computing education.

Contextualized Introduction of Computing for End-User Programmers

Computer programming is widespread in domains other than computer science and software engineering. Unlike professional software engineers, programmers in these domains are mostly not creating software for others to use—they themselves are the "end users" of their software, and they are referred to as *end-user programmers*.[6] They are typically not formally trained software developers but use various tools and platforms to write programs to accomplish particular domain-specific tasks.

End-user programmers often complete tasks by studying similar programs and modifying them for their own use cases.[3,7] This can be helpful for these learners, because their programs tend to be short and simple, and their goal is often to *get things done*, not to *learn programming*.[8,9] For example, studying a community of graphic designers, Dorn et al.[7] found that many of the programs produced by and for this community contained constructs that are typically found in introductory computing courses (hereafter referred to as "CS 1").

Educators have opted to introduce programming within the broader context of a given domain to motivate the learning of programming (e.g., media computation,[10] interactive entertainment,[11−13] analysis of personally meaningful data,[14,15] and chemistry[2,16]). These *contextualized* approaches often appear at the introductory level as a way to introduce computing to non-CS majors, or to CS majors with no prior programming experience. They have been successful at improving retention of students in computing degrees[13] and at broadening participation in computing.[10,14]

These efforts acknowledge the importance of computing as a tool to engage with a domain topic. This value aligns with end-user programmers' task-oriented engagement with computing. In these contextualized introductions, the primary emphasis shifts from *learning to program* to *programming to learn* the subject matter. However, we believe that there is a need to achieve a balance between learning to program and programming to learn when engaging with computing in a contextualized setting.

Supporting End-User Programmers in Knowledge Transfer

In end-user programming education contexts, an overly contextualized base of programming knowledge can lead to challenges that stifle growth in learning programming.

First, when students are mostly exposed to examples in one context, they can struggle to disentangle their knowledge from that context,[17] making it difficult to apply what they have learned to new contexts or problems. This constitutes a lack of *knowledge transfer*. Knowledge transfer does not occur simply by virtue of having the initial knowledge—we have to explicitly teach in ways that facilitate transfer. Students must learn the abstract rules of a system (e.g., a programming language) to be able to apply their knowledge of it in varied contexts.[18] In other words, they must form an accurate *mental model* of the program execution environment.

Second, while programming by modifying examples can be useful, it also erodes the opportunity for the student to learn more generalizable principles critical to programming. In a highly contextualized introduction, students can only learn or use concepts that are present in their current pool of related programs.[7,19] For example, learners are less likely to choose to define their own reusable functions or to aggregate data using objects if the pool of programs rarely involves these concepts.[7]

Finally, in contrast to end-user programming education, students learning programming to become software engineers are often taught explicit programming and problem-solving strategies, such as the Design Recipe[20] or Loksa et al.'s six-stage process.[21] Explicit guidance about these processes can increase a student's *metacognitive awareness* as they approach programming problems.[22,23] This knowledge can also improve a learner's ability to self-regulate during software development,[24] affording them the self-efficacy and strategies to tackle larger or unfamiliar programming problems. End-user programming education has not typically devoted focus to these self-regulatory programming strategies.[3]

Mental Models and Notional Machines

A *mental model* is a structure held by an individual representing their beliefs about how a particular system works.[25] They can update over time as the individual's understanding evolves, and they can be incomplete, inconsistent, or incorrect.

Mental models held by experts differ from those held by novices in important ways.[26] Experts' mental models are based on a strong understanding of the abstract rules of a system (e.g., of a program execution environment), which allows them to gracefully handle unanticipated situations. Conversely, novices' mental models are often based on superficial features of a system (e.g., on keywords or symbol names in a *program* as opposed to the execution environment). Novices' mental models are, therefore, liable to undergo ad hoc changes during problem-solving, making them unstable and possibly internally inconsistent.

A *notional machine*[27] is an abstract model of a program runtime environment,[28] used to explain program execution. It is crucial for learners to form an understanding of the program runtime that matches a normative model.[29] This "model" does not need to be the actual machine; it can be a *notion of the machine* that operates just beneath the abstractions that the learner will primarily use.[28] That is, it can elide certain details depending on the level of the learners or the tasks being performed. For example, a notional machine in an introductory Java course might describe program execution only in terms of Java's memory model and control flow rules. For more advanced courses or learners, the notional machine might include facilities for multithreaded programs. Every programming course involves a notional machine, either implicitly in the programming language and features used or explicitly taught by the instructor.[28] It has been argued that notional machines ought to be explicitly taught in introductory programming courses (e.g., Sorva,[28] Dickson et al.[30]).

Learners will inevitably form a mental model of the notional machine as they work.[29] Without explicit instruction, ad hoc models will be formed based on surface-level characteristics of programs and not on the underlying execution model. These types of models may work for a limited subset of programs, but they will eventually lead to misconceptions and hinder the transfer of knowledge to new contexts. An important goal of programming courses is therefore to help students form an

https://doi.org/10.1021/acs.jchemed.4c00219
J. Chem. Educ. XXXX, XXX, XXX−XXX

accurate mental model of the notional machine.[28] We argue that courses involving end-user programming are no exception.

Teaching Programming in the Context of Chemistry

There is a lack of consensus among chemistry educators on whether or not teaching programming should be considered a goal in a chemistry curriculum. According to a report by the LABSIP group,[31] some chemistry faculty argue that PChem coursework should involve programming, while others believe that it would be an unnecessary barrier to student success. While the group did not make a recommendation about including programming, it emphasized that faculty should introduce students to the idea that many PChem problems are complex enough that computer-based approaches are superior to calculations by hand, and that many computer-based approaches (programming, using spreadsheet programs, etc.) could meet this learning goal.

In agreement with scholarship from as early as the 1960s,[32,33] we take the position that programming *should* be included in chemistry curricula, grounded in the context in which it is expected to be used. Doing so would carry with it all the benefits of contextualized computing education that we have described above. These benefits might be lost if chemistry students are placed in CS courses that are divorced from the chemistry context.

However, despite the growing incidence of teaching programming in the chemistry curriculum, there has been limited assessment of the transferability of students' programming skills to new contexts and no work examining chemistry students' mental models about programming. We address this gap in this paper.

STUDY OVERVIEW

We write from a medium-sized primarily undergraduate public institution in the United States of America. It has a large and moderately selective college of engineering where class sizes rarely exceed 35 students. We studied undergraduate students learning programming in a PChem course sequence. The sequence is taken by undergraduates majoring in either chemistry or biochemistry during their third year (in a typically four-year program) at our university. The primary focus of these courses is physical chemistry, *not* programming. At the time the study was conducted, the programming in the course was taught using MATLAB, though it has now switched to Python. However, our findings and recommendations are not tied to any particular programming language.

The programming portion of this sequence is described in a previous paper[2] and is summarized in Table 1.

Table 1. MATLAB Topics Covered in the PChem Curriculum[a]

PChem course	major MATLAB topics
First course	Symbolic math in MATLAB
	Using and writing scripts in MATLAB
	Arrays and plotting
Second course	Write and use functions
	Conditional logic (including relational operators)
	Differential equations and plotting
Third course	Matrix manipulation
	Advanced control flow (nested conditionals, loops, nested loops)

[a]Paraphrased from McDonald and Hagen.[2]

While the PChem curriculum has been implemented for 10 years, the programming component of the sequence has been challenging for many students. Students struggle to learn the programming content and seem to desire to learn only enough to complete their required assignments. This sentiment was expressed numerous times on course evaluations for all seven instructors who have taught the courses. Following this sequence, the students are eligible to take an advanced elective computational chemistry course. Perhaps unsurprisingly, the vast majority of students—generally more than 75%—self-select out of the advanced elective course. Those who *do* grasp the programming content go on to successfully use that knowledge in their research groups.[2] We would like to increase the incidence of this less common scenario.

We conducted two studies to learn about students' attitudes toward and abilities with computing. First, our *preliminary study* used two surveys to quantitatively measure these items. Results from these surveys motivated our *core study*, in which we conducted a series of one-on-one interviews with students enrolled in the PChem sequence. Our research protocol and materials were approved by our Institutional Review Board (IRB) prior to conducting the study. The following sections describe our data collection and analysis processes for each of the studies.

PRELIMINARY STUDY: SURVEYS

We used a pair of surveys to measure students' attitudes toward computing and MATLAB knowledge, both administered to students enrolled in the second (n = 32) and third (n = 36) PChem courses in the spring term of 2021.[78] All students were required to take the surveys outside of class time, and responses from consenting students (100%) were included in our analysis. The two surveys were given to students as a single questionnaire, and students were given 75 min to complete it. The survey was administered to a section of each course, resulting in a roughly even split of responses between each course.

Demographic details about the survey respondents are provided in Table 2. Though we did not base any analyses on these fields, they are provided for the sake of context.

Table 2. Summary of Survey Participant Demographics (n = 68)

Race	
White	56%
Asian	24%
Latinx/Hispanic	11%
Two or more	6%
Prefer not to answer	3%
Gender	
Woman	63%
Man	33%
Nonbinary	3%
Major	
Biochemistry	55%
Chemistry	45%
Do you use CS outside the classroom?	
No	76%
Yes	24%

https://doi.org/10.1021/acs.jchemed.4c00219
J. Chem. Educ. XXXX, XXX, XXX−XXX

Attitudes toward Computing

We used the Attitudes Toward Computing (ATC) scale[34] to measure students' *confidence* with and *enjoyment* of computing, their perception of computing as *important and useful*, and the strength of their *belongingness*[79] in computing. Together, these constructs can explain in large part one's motivation to persist in an academic discipline.[35]

All questions were answered on a five-point scale from *strongly disagree* to *strongly agree*. Scores for each construct were obtained using the mean of the items for that construct.

Result: *Students reported negative attitudes toward computing.* On average, students reported low confidence (μ = 2.25, σ = 0.95), enjoyment (μ = 2.5, σ = 0.87), and belonging (μ = 2.3, σ = 0.84). However, computing was perceived as marginally useful (μ = 3.66, σ = 0.82).

Programming Knowledge

Next, we used a pilot exam used to gain an initial understanding of students' MATLAB knowledge. We used the MATLAB Computer Science 1 (MCS1)[36] concept inventory, a MATLAB assessment that is based on the "SCS1", an existing validated assessment of programming knowledge.[37]

Result: *Students performed poorly on the MCS1 (μ = 30%, σ = 15%).* Prior work suggests that the SCS1 is a difficult test and not suitable as an introductory-level CS pretest.[38] Upon further review of the test, we found that many of the MCS1 questions tested concepts that had not been covered in class. We therefore elected to take a more refined qualitative approach to understanding students' MATLAB knowledge, described in the following section.

CORE STUDY: INTERVIEWS

Following the surveys, we conducted interviews in the fall 2021 and winter 2022[80] academic terms to gain a richer understanding of students' attitudes and abilities. A total of eight interviews were conducted, and participants were enrolled in either the first (n = 4) or second (n = 4) PChem course. The interviews were in person or over video conferencing software, and the audio was recorded and transcribed. Interviewees were recruited using in-class announcements, and they were compensated for their participation with online retail gift cards worth $25. No students who took the surveys are likely to have participated in the interviews (unless they retook the second PChem course and happened to be interviewed). We cannot say for certain since survey responses were anonymized per our IRB-approved research protocol.

Interview Procedure

We administered a modified MCS1 as a think-aloud interview instead of a multiple-choice exam. The first and last authors independently labeled each MCS1 question with the learning objectives that it assessed and then collaboratively discussed and refined their labels until they were in agreement. We then selected a subset of MCS1 questions that targeted PChem learning objectives such that interview participants had previously been exposed to all the concepts that appeared in the questions.

Borrowing from the grounded theory methodology,[39] we continuously analyzed interview transcripts and adjusted interview questions to explore emergent concepts of interest. We made two major types of changes to the interview questions.

1. *Questions were rewritten to only use syntax or concepts that were previously taught in class.* Some questions involved concepts that the students had simply never seen before. For example, one problem asked students to explain a function that squares all of the odd numbers in an array and returns the array. The function used the modulo operator (%) to determine a number's parity. Since PChem students are not taught this operator, we modified the problem to test for parity by comparing the results of division by 2 and integer division by 2. The students were generally able to trace this modified line, which allowed us to gain insight into their knowledge of the problem's target concepts of *functions* and *control flow*.

2. *Multiple-choice questions were turned into short-answer questions.* The original MCS1's multiple-choice questions allow students to solve problems through a process of elimination of choices, making it difficult to gauge the student's understanding of MATLAB. For example, one problem lists a series of variable assignment statements and, given a set of initial values, asks the student to order the statements such that a given variable ends up with a particular value (see Box 1). As a multiple-choice problem, this can be solved by testing each of the available orderings until the correct one is reached. As a short-answer problem, the student must understand the relationships between statements, identify and ignore unnecessary statements, and construct a plan to achieve their goal. This allowed us to distinguish between students who were only able to mechanically trace the function one statement at a time and students who were able to explain the purpose of the function at an abstract level. This gave us richer insight into students' understanding of MATLAB and their problem-solving strategies.

Box 1. Given initial values for all these variables, this problem asks students to reorder these assignment statements such that a given variable ends up with a particular value.

```
a = b / z;
b = c + x;
y = y * b;
x = c * a;
y = z - y;
```

We also asked students the following three questions and subsequent follow-up questions as part of a semistructured interview.

- How do you feel about MATLAB?
- Do you use MATLAB if it is not specifically asked for in a question or problem?
- How do you feel about your computing abilities?

Analysis

CS educators have adapted Bloom's taxonomy into a computing-specific taxonomy (the *matrix taxonomy*) to describe stages of programming knowledge.[40] The key insight behind the matrix taxonomy is that the ability to *interpret* a program and the ability to *produce* a program are semi-independent skills.[41]

In the matrix taxonomy, levels of program interpretation (*remember*, *understand*, *analyze*, *evaluate*) are on the horizontal axis, while levels of program production (*apply*, *create*) are on the vertical axis (e.g., see Figure 1). Fuller et al.[40] argue that this framing allows one to categorize various programming tasks in terms of the code-reading and code-writing skills they

https://doi.org/10.1021/acs.jchemed.4c00219
J. Chem. Educ. XXXX, XXX, XXX−XXX

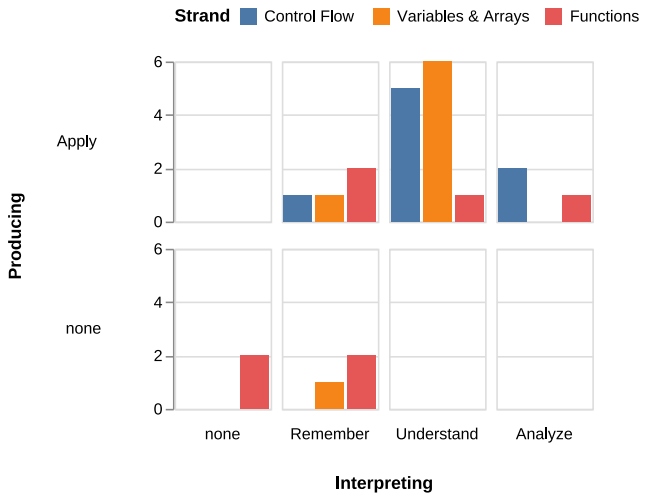


Figure 1. Number of students in each cell of the matrix taxonomy for the Control Flow, Variables & Arrays, and Functions strands.

require. For example, modifying an existing program to solve a new problem involves *understanding* the existing code and *applying* one's programming knowledge to make the necessary modifications.[40] Higher-level tasks like refactoring involve a deep understanding of the problem and program and sit at the highest levels on both dimensions, i.e., *evaluate* and *create*.

We set out to use the matrix taxonomy to categorize students' abilities with various MATLAB programming concepts. Learning to program involves understanding multiple computing concepts that are not directly related, and mastery of one does not imply mastery of others. Following Castro and Fisler,[42] we used a *multistrand* taxonomy to separately categorize students' abilities with different programming concepts. We settled on three strands:

- *Control Flow* includes statement ordering, conditionals, and loops.
- *Variables & Arrays* includes variable assignment and mutation, as well as array operations. We chose to combine variables and arrays since all MATLAB variables are arrays.
- *Functions* includes creating custom functions, as well as calling functions and scope of the variables and parameters local to the function.

After each interview, the researcher who conducted the interview created a memo describing the student's location on each strand of the taxonomy. Each memo included quotes and incidents from the interview that indicated the student's ability level for each strand (see Figure 1). Misconceptions were placed outside the producing−interpreting grid, in the bottom-left cell.

Two researchers collaboratively discussed the memos for the first three interviews, making adjustments and clarifications until a shared understanding of each level of the taxonomy was reached. Following this, the first author individually prepared memos for the remaining interviews, locating students on the matrix taxonomy.

Results

We begin by summarizing the students' programming abilities based on our interviews, following which we describe recurring patterns we observed as students worked through programming problems.

Overview of Programming Abilities among PChem Students. *Control Flow.* Five out of eight students were able to reach the Understand/Apply level in Control Flow (blue in Figure 1).[81] For example, one problem in the interview provided students with a set of variable assignment statements (Box 1) and initial values. We asked them to order the statements such that a given variable would end up with a required value. The problem evaluated students' understanding of assignment statements, their code-tracing ability, and their mastery of organizing statements in a program. The problem required students to understand the mechanics of the assignment operator and, critically, understand the relationships between the different variables.

Students at the Understand/Apply level *understand* that lines of code are executed sequentially and *apply* that knowledge to construct and test different orderings. However, they were unable to gauge the higher-level relationships between the variables. They often used a "brute force" approach by testing seemingly random orderings until they found one that worked.

In contrast, students at the Analyze/Apply level (two out of eight) began by decomposing the problem, first identifying the statements that would affect the variable of interest (`x`), demonstrating higher-level code interpretation skills, one that moved them beyond understanding the variable assignment syntax to work with relationships between variables. While solving the problem and thinking aloud, one student said:

> *So the thing that I'm looking for first is which one will actually affect* `a`*, because that's the one that I'm trying to change, and then I just need to set the variables that are in the one that affects* `a` *be the the correct form. − P7*

Like P7, some students were able to draw out the variables and the relationships between them.

Variables & Arrays. Most students (six out of eight) were able to reach the Understand/Apply level in Variables & Arrays (orange in Figure 1). Students at the Understand level typically did not express misconceptions about variables. The student at the Remember/Apply level was able to recognize the variable assignment syntax but displayed a flawed understanding of how variables work in MATLAB. They displayed the so-called "parallelism bug",[43] where they expected the runtime to be *reactive*. That is, if a variable was given a new value, they expected the values of prior expressions using that variable to also be updated. The student at the Remember/None level could recognize a variable assignment statement or the use of a variable but was unable to effectively trace or explain code that used variables.

Functions. Functions proved to be difficult for most students (red in Figure 1). Students also expressed the most varied levels in this strand. Students at the Remember/Apply (two out of eight) level could be expected to *remember* some facts about functions and *apply* them to programming. However, they often expressed important misconceptions (discussed in the section Recurring Patterns) that sidelined their attempts to apply their knowledge. Students at this level tend to use a trial-and-error approach, which can lead to frustration as they encounter errors.[40]

In a problem that asked students to explain a function, the student at the Understand/Apply level explained it line by line. While technically correct, this suggests that they considered functions as a collection of lines of code, not as a modular entity that accomplishes a task, one that could be reused as a building block to solve other problems. On the other hand, the student at the Analyze/Apply level was able to read the

https://doi.org/10.1021/acs.jchemed.4c00219
J. Chem. Educ. XXXX, XXX, XXX−XXX

function and describe it as a high-level abstraction, in terms of its inputs, outputs, and purpose, demonstrating "task-level thinking"[44] and "plan knowledge".[45]

Recurring Patterns. We now discuss recurring patterns and misconceptions we observed during the interviews.

Difficulties with `and` Expressions. In MATLAB, `and(A, B)` and `A & B` are two ways to express a logical AND of two Boolean expressions `A` and `B`. Students in the PChem sequence had been taught the `and` syntax. Many students (P2, P3, P4, P5, P6, P8) could correctly trace a program with conditional branching (e.g., `if and(a, b) ...`), but were stymied by a statement like `c = and(true, and(true, false))`. Students who held this misconception were placed in the Understand/Apply cell of the Control Flow strand (Figure 1).

For example, when asked to solve a Boolean logic problem containing a standalone and expression, one student said:

> *We've never gone over a problem like this, or the logic behind this ... I would really just be guessing on this. − P5*

Then, while solving another problem, the same student correctly traced an `and` inside an `if` statement:

> *[In a snippet where `a`, `b`, and `c` are numeric variables] `a` is greater than `b` and `b` is greater than `c`, so that's not true because 3 is not greater than 7. − P5*

These students understood "if-and" as its own MATLAB construct. Lacking an abstract understanding of the concepts of *statements* and *expressions*, they combined the notions of `if` and `and` into a single abstraction based on superficial characteristics of programs they had seen before. Only one student (P1) recognized the `and` syntax in a variable assignment after seeing Boolean logic used within an `if` statement, highlighting the challenge of abstracting out similarities and transferring knowledge from one context to another.

Inability to Recognize Matrix Concatenation. When asked to explain code like Box 2, only one student (P7) was able to

Box 2. One way to append values to a matrix in MATLAB.

```
x = [''];
x = [x 'B'];
x = [x 'A'];
```

recognize that it would create the matrix `['B' 'A']` by appending items to the empty matrix. This works by assigning `x` the value of *a new matrix*, created from the old values of `x` and an additional value.

The other students expressed some familiarity with the matrix creation syntax, but they did not grasp that values were being appended to the matrix. When it was suggested that the code in Box 2 will create the matrix `['B', 'A']`, one student said:

> *I don't think that is true because if you want to write in matrix format ... you have to do like x, say the first one, then the second one. − P1*

This student is familiar with the matrix creation syntax, but they do not recognize that `x = [x 'B']` is a valid example of matrix creation. They are confused by the use of a variable, instead of only literal values. They appear to believe that the matrices can only be created with a matrix literal, using literal values for matrix elements, such as `x = ['B' 'A']`. Although appending to a matrix uses the same matrix creation operation as initializing the matrix with literal values, students were unable to recognize the matrix creation operation. This suggests that the student struggled to comprehend abstract representations, which in this case involved seeing variables as an abstract representation of values.

Students at this level would land in the Understand/Apply cell in the Variables & Arrays strand of the taxonomy. They understand how to work with arrays in one way, but they do not yet have an understanding of the abstractions used to analyze alternative solutions, such as appending with the method used in Box 2. This limits their ability to transfer their knowledge beyond the particularities of the programming language-level artifacts that they have encountered before.

Difficulties with Functions. As can be seen in Figure 1, functions were more difficult for students than either of the other two strands, with five students (P3, P4, P5, P6, P8) expressing misconceptions about functions. We describe some of the salient misconceptions below.

Two students (P4, P8) appeared to have a general lack of knowledge about functions—they did not seem to recognize the syntax for calling functions, which meant they were unable to correctly trace code that used user-defined or existing MATLAB functions.

For example, consider line 2 in Box 3. The line in question uses the preexisting MATLAB function `zeros` to create an array of zeros. Referring to this line, one student said:

Box 3. A complex function involving conditionals, prints, and looping. This question asked students to predict the value of the `output` variable after executing the snippet.

```
1  function [ ret ] = square_evens(numbers)
2      ret = zeros(1, length(numbers));
3      for i=1:length(numbers)
4          val = numbers(i);
5          if(floor(val / 2) == (val / 2))
6              val = power(val, 2);
7              disp(val);
8          end
9          ret(i) = val;
10     end
11 end
12
13 in_numbers = [1, 2, 4, 3, 5];
14 output = square_evens(in_numbers);
```

> *And the `numbers` are also a set of the zeros. − P8*

The student believes that the `zeros` function is going to modify the `numbers` variable. This implies that they did not recognize that `length(numbers)` is itself a function call that returns a value, which in turn is passed to the `zeros` function. Moreover, the student struggled to explain the function operation, leading to the gap in their understanding that even if `zeros` were passed the `numbers` matrix directly, `numbers` cannot be modified since MATLAB uses pass-by-value semantics. The rest of this student's explanation of the `square_evens` function (Box 3) continued similarly.

Some students held more nuanced misconceptions. Referencing code similar to lines 2 and 3 of Box 4, one student said:

> *Calculating `x` & `y` based off of each other ... I would think that we would need a value to start with. − P6*

While partially correct, the student struggled in decomposing the function and recognizing that `x` and `y` are defined as parameters that get assigned values when the function is called (9 and 6, respectively). From this, we get a glimpse of how students struggle with the abstraction involved in functions: they saw the function as a series of statements that were not necessarily bound within a holistic unit of operation.

https://doi.org/10.1021/acs.jchemed.4c00219
J. Chem. Educ. XXXX, XXX, XXX−XXX

Box 4. A short function that does not return anything. The question asks the student to predict the values of `x`, `y`, `a`, and `b` after the snippet has executed.

```
function [ res ] = compute(x, y)
    x = x - y;
    y = x / y;
end
a = 9;
b = 6;
compute(a, b)
```

The challenge of decomposing a function to understand its operation varied. For example, some students correctly mapped arguments to parameters but did not understand the scope of the parameters. When solving a problem similar to Box 5, one student said:

Box 5. An example problem that assesses understanding of scope. The question asks the student what will be printed after execution.

```
1 function [solution] = calculate(a, b, c)
2     solution = a * c + 0.5 * b * c * c;
3 end
4
5 x = 4;
6 y = 16;
7 z = 12;
8 solution = calculate(x, y, z);
9 disp(b); % displays the value of a variable
```

`a` would be like `x` and then `b` would be `y` and `c` would be `z` so `b` would probably be like 16. — P5

This demonstrates that they correctly mapped the arguments to parameters, but they incorrectly believed that the parameters to `calculate` are still defined on line 9, outside the scope of the function.

Indeed, only P7, who was pursuing a CS minor and had completed several prior programming courses, could explain that line 9 would produce an error because `b` is undefined outside the scope of the `calculate` function. Considering that PChem involves teaching students to "write and use custom functions",[2] the gap highlights the broader challenges in teaching end-user programmers to grasp the abstractions and decomposition required to transfer their knowledge to different problems.

Attitudes toward Programming. Patterns also emerged in the students' attitudes toward programming. Overall, most students viewed their programming skills as specific and limited. One student (P7) was pursuing a CS minor and displayed higher self-efficacy. However, unless otherwise mentioned, the following observations hold true for the rest of the students interviewed.

Students displayed an apparent sense of confidence with the particular MATLAB elements that they had encountered in the class but low self-efficacy about using those skills in a more general context. PChem students tended to treat MATLAB as an advanced graphing calculator, using it to solve difficult equations and create high-quality plots. All students except P7 and P8 expressed this sentiment, exemplified in P1's comment:

So I literally threw my calculator away and started doing everything on MATLAB. — P1

In contrast, students reported low self-efficacy for general-purpose programming. In fact, they did not believe that those skills could transfer to general programming. When asked "How do you feel about your computing abilities?", one student responded simply, "I feel like they're pretty weak" (P3). We heard this clearly in P5's concern:

Within the limits of chemistry, I think I feel like very good about my MATLAB skills ... But if you gave me something like outside of that I wouldn't really know what I'm doing. — P5

Similarly, others found it daunting to transfer their skills outside the of PChem class:

I don't think I'd be able to take these skills outside of class and use them elsewhere. — P2

This sentiment seemed to emerge from the students' perception that their skills were closely tied to the particular objectives that were established for them in the PChem class.

I want to say it's ... from my perspective at least ... as a calculator, so while calculating huge derivatives or taking huge, you know, differentials, right? ... It's a lot of copy and paste code. — P1

The fact that they mentioned copying and pasting code suggests that they were unable to grasp and work with the fundamental ideas of MATLAB programming. The lack of movement to more general programming problems seems to hinder the students' self-efficacy in transferring their knowledge.

I know how to do explicitly what I've learned, and not much else. — P6

Indeed, we heard similar concerns of not being able to move beyond the particulars of MATLAB in PChem context throughout our interviews:

I think with chemistry they do a really good job of teaching us. But there's no like okay we're only going to be doing like MATLAB and like learning the very basics of the basics. — P5

The challenge in learning the fundamentals of (MATLAB) programming did not seem to emerge from students' lack of motivation. In fact, students acknowledged that programming is a growing part of research in many domains.

I should probably understand everything computationally little bit more because that's the way that the research is moving and I think that's, you know, an important part of the research future. So I want to develop these skills more but I'm like I don't know what specific applications really look like. — P3

The lack of movement to more generalizable knowledge appeared to lead to the perception that MATLAB in and of itself was limiting. Indeed, a student viewed the PChem programming curriculum as nonauthentic because they were using MATLAB and not Python:

We're not learning Python or anything which I know is what's generally used on industry and a lot more out in the field. — P8

Students had a sense that programming is important, but they did not believe that the programming *they were doing* was broadly useful or authentic. Such mismatch in perception can degrade motivation, leading to worse learning outcomes.[35]

DISCUSSION

To deepen understanding of programming in contextualized contexts like ours, we believe that instructors need to move back and forth between higher-level ideas that are fundamental in computing and contextualized encounters with examples specific to the domain. Computing education literature posits fundamental ideas in computing (e.g., refs 46–48): abstraction, decomposition, and metacognitive awareness. While the

https://doi.org/10.1021/acs.jchemed.4c00219
J. Chem. Educ. XXXX, XXX, XXX−XXX

ideas that are considered "fundamental" vary, there is a wide agreement that these three are critical.[49−51] In our findings, we note that students differed in their abilities to engage with these ideas. These three ideas are *not* exclusive to computing. As we illustrate below, they are applied in different domains, including PChem, thus allowing for synergistic presentation, one where instructors can build on their domain expertise to scaffold student learning.

The discussion in the following subsections is distilled into a concise set of recommendations in Table 3.

Table 3. Recommendations for Teaching Programming to Chemistry Students

topic	recommendations
Abstraction	Explicitly teach students about a chosen notional machine.[28]
	Use multiple representations of programming concepts to promote abstraction and transfer learning.[78]
	Relate abstractions used to solve programming problems to those used in chemistry problems.[52]
Decomposition	Use subgoals to divide programming problem-solving into small steps.[46]
	Show examples of code and function reuse.
Metacognitive awareness	Teach an explicit programming problem-solving process, such as the Design Recipe.[20]
	Relate iterative problem solving in programming to iterative problem-solving in chemistry.

Abstraction

Abstraction allows us to focus on salient details of a concept, object, problem, or process while ignoring others.[47,52] In the task of learning programming, the goal is for learners to gain an abstract understanding of programming concepts that can be brought to bear on previously unseen programs or problems. Lacking abstractions for programming concepts can lead to brittle knowledge that is easily stymied by novel problems or programs that look different from previously seen examples.

Students faced difficulties with abstraction in our end-user programming context. For example, consider the students whose mental models combined `if` and `and` into a single programming construct. Encumbered by mental models that are tightly bound to a limited set of examples, these students are forced to assimilate unfamiliar-looking code into their mental models "on-the-fly".

Despite its importance, challenges remain in teaching abstraction in early computer programming.[53,54] For example, Koppelman and Van Dijk[53] argue that novice programmers are asked to focus on the flow of control in programs, and this hinders their ability to see abstraction being used in different parts of the program. They posit the need for explicitly introducing abstraction early in programming classes. Their recommendation echoes other scholars' arguments on the importance of making moves between levels of abstraction consciously.[47,54,55] This challenge remains in courses that provide contextualized introductions to programming.[52]

As we formulate strategies to introduce abstraction in the context of learning programming, we acknowledge that abstraction is also a critical skill in chemistry. Exploring an integration of computer programming and chemistry education, Gautam et al.[52] demonstrate the need to build a connection between abstractions used in chemistry with abstractions used in programming. They demonstrate the need to move between levels of abstraction, both *laterally* and *vertically*. For example, consider chemical equations, which are abstract representations of real-world phenomena, and the similar abstraction skill necessary for computer programming where modeling chemical reactions requires representations using variables and expressions. They argue that such an integration would enable students to learn to use abstraction, and to more deeply learn *through* abstractions. We ought to build on the strengths of instructors and students in end-user programming who already have domain expertise leveraging abstraction. Chemistry courses that include programming should focus on helping students build the computing abstractions necessary to engage more deeply with new problems or programs.

There exist numerous strategies to help learners acquire this abstract understanding. One strategy is to explicitly teach a *notional machine*,[27,29,30] i.e., an abstract model of the program execution environment. Another strategy is to teach using *many* relevant examples, each showing different representations of the same concepts.[56,57] Exposure to many representations or orientations of the same concept can help students perceive the abstract principles that underlie the surface-level characteristics of any single example. In terms of program comprehension, different example usages of Boolean expressions (e.g., used in `if` conditions, variable assignment statements, or function arguments), coupled with explicit instruction that "an expression is a syntactic entity that evaluates to a value", would help students develop an abstract understanding of the notion of Boolean expressions. This, for example, would dispel the "if-and" misconception that we observed in our interviews.

Decomposition

Closely related to abstraction is the skill of *decomposition*. This is the ability to apply knowledge of the particular features of a system (e.g., the notional machine) to break down a problem into manageable parts, solve those parts using familiar and reusable patterns if available, and compose the solutions for each part into a solution for the larger problem.[58−62] Programming students' skills with abstraction must be augmented with skills for decomposition.

While decomposition is widely accepted as a critical practice in learning programming,[51,63] with a few notable exceptions,[58,64,65] few studies have explicitly explored *how* we can enable students to learn decomposition. Rich et al.[58] conducted a literature review to identify 13 consensus goals including "complex problems can be broken into smaller parts" and "often existing code from other programs can be used to solve parts of a decomposed problem". Novice programmers face challenges in learning and working on decomposition.[44,66,67] For example, Lee et al.[66] highlight that students struggle to understand the difference between creating a function and using a function through a function call. They found that novice students tend to ignore reusing functions, focusing mostly on creating new ones even when existing functions could help. Indeed, our findings highlight similar challenges faced by end-user programmers who struggled to identify or use functions.

The practice of decomposition, that of breaking down a problem into smaller manageable parts and reusing existing solutions to solve the decomposed parts, is not unique to computer science.[68−73] For example, early chemistry education scholars have posited the importance of scientific modeling skills which requires breaking down a complex system into

https://doi.org/10.1021/acs.jchemed.4c00219
J. Chem. Educ. XXXX, XXX, XXX−XXX

smaller elements and mechanisms.[72] More recently, systems thinking practices have gained greater attention in science education, which too has led to an emphasis on problem decomposition.[68,73] As we explore opportunities to support undergraduate chemistry students in end-user programming, we argue for building upon the students' existing familiarity with decomposition by drawing parallels between decomposition in science and programming. In doing so, we echo Rich et al.[58] in arguing that decomposition ought to be a central practice, and not just a content topic, in learning computer programming.

Metacognitive Awareness

The final component, *metacognitive awareness*, refers to the student's ability to evaluate their own progress toward a programming task (e.g., understanding a problem; planning, writing, testing, and debugging a solution).

Consider the problem in Box 1. It asks the student to order a series of variable assignment statements so that the variable x ends up with a specified value. Some students, having the facilities to engage in "planful" programming,[23,45] began by omitting the assignment statements that did not have any direct or indirect effect on the variable of interest. Then they ordered the rest of the statements based on dependencies within them, ignoring more statements if necessary, before arriving at a satisfactory sequence. They were able to accomplish this task by tracing partial solutions while keeping track of known intermediate information; this helped them maintain an awareness of whether or not they were on the right track toward solving the problem. Other students, lacking the metacognitive skills needed to monitor their progress, resorted to a brute-force approach to solving the problem, considering permutations of assignment statements in a seemingly random fashion until they arrived at a satisfactory ordering.

These aspects of metacognitive awareness and planful programming are prevalent throughout a typical CS education, but not in end-user programming education.[3] To encourage increased monitoring of one's programming process, computing educators have advocated for teaching explicit problem-solving strategies like the Design Recipe[20] or Loksa et al.'s six-step process.[21] These strategies involve explicit steps for confirming one's understanding of the problem *before* one starts programming a solution and for outlining a solution by decomposing the problem into pieces that can be solved using familiar patterns.[41] Edwards et al.[74] suggest using software testing to move students away from "trial-and-error" styles of programming (like some students in our case study) toward more reflective, metacognitively aware programming problem-solving.

This process of metacognitive awareness in problem-solving is also common in chemistry. For example, consider the problem of identifying a chemical compound's structure from its spectroscopic data. A student might begin with IR spectroscopy and, observing a broad peak in the low 3000s wavenumber range, would identify the presence of an O−H stretch. Noting that multiple functional groups contain an O−H stretch, the student might then turn to NMR spectroscopy to narrow down the possibilities. Chemistry students already have considerable facilities with this type of iterative problem-solving process; we ought to leverage this in teaching programming problem-solving.

CONCLUSION

We believe that would-be end-user programmers—such as the students in our study—would benefit from instruction about the fundamental ideas we have outlined: abstraction, decomposition, and metacognitive awareness in problem-solving. Difficulties with abstraction and planful programming are commonly reported and—to an extent—addressed in CS 1 courses.[75] They should also be addressed in end-user programming contexts. The chemistry faculty teaching these courses are updating the curriculum based on these recommendations. Future work will evaluate the impact of these changes on course performance, students' confidence with programming, and their intent to continue to learn and use programming.

The three ideas discussed above are not an exhaustive list of fundamental ideas in computing. Neither are they mutually exclusive: mastering decomposition necessitates an understanding of abstraction, and evaluating one's progress on a problem requires the practices of abstraction and decomposition. Nonetheless, we believe these ideas are necessary to support students in transferring their knowledge of programming to different contexts. Importantly, these ideas are not unique to computer science. Drawing on non-CS students' existing strengths in these areas would allow us to introduce computing topics in a way that makes it easy for novices to start learning but also provides them with room to develop their skills beyond the initial coursework and work on more sophisticated tasks.[76,77]

Non-CS majors are learning to program with increasing frequency, and they will go on to conduct important work in science and other domains. Though their goals are not to become software engineers, they *will* engineer software, potentially of a critical nature. Their programming education should account for this.

AUTHOR INFORMATION

Corresponding Authors

Ashley Ringer McDonald − *Department of Chemistry and Biochemistry, Cal Poly, San Luis Obispo, California 93405, United States;* orcid.org/0000-0002-4381-1239; Email: armcdona@calpoly.edu

Aakash Gautam − *Department of Computer Science, University of Pittsburgh, Pittsburgh, Pennsylvania 15260, United States*; Email: aakash@pitt.edu

Ayaan M. Kazerouni − *Department of Computer Science and Software Engineering, Cal Poly, San Luis Obispo, California 93405, United States;* orcid.org/0000-0002-6574-1278; Email: ayaank@calpoly.edu

Author

William Fuchs − *Department of Computer Science and Software Engineering, Cal Poly, San Luis Obispo, California 93405, United States;* Present Address: Ridgeline, Inc

Complete contact information is available at: https://pubs.acs.org/10.1021/acs.jchemed.4c00219

Notes

The authors declare no competing financial interest.

https://doi.org/10.1021/acs.jchemed.4c00219
J. Chem. Educ. XXXX, XXX, XXX−XXX

■ ACKNOWLEDGMENTS

A.R.M. acknowledges financial support from National Science Foundation Award No. CHE-2136142.

■ REFERENCES

(1) Easterbrook, S. M. Climate Change: A Grand Software Challenge. *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*; Association for Computing Machinery: New York, NY, USA, 2010; pp 99−104.
(2) McDonald, A. R.; Hagen, J. P. Using Computational Methods To Teach Chemical Principles. *ACS Symposium Series* **2019**, *1312*, 195−210.
(3) Ko, A. J.; Abraham, R.; Beckwith, L.; Blackwell, A.; Burnett, M.; Erwig, M.; Scaffidi, C.; Lawrance, J.; Lieberman, H.; Myers, B.; Rosson, M. B.; Rothermel, G.; Shaw, M.; Wiedenbeck, S. The State of the Art in End-User Software Engineering. *ACM Comput. Surv.* **2011**, *43*, 1.
(4) Guzman, L. M.; Pennell, M. W.; Nikelski, E.; Srivastava, D. S. Successful Integration of Data Science in Undergraduate Biostatistics Courses Using Cognitive Load Theory. *CBE—Life Sciences Education* **2019**, *18*, ar49.
(5) Valle, D.; Berdanier, A. Computer programming skills for environmental sciences. *Bulletin of the Ecological Society of America* **2012**, *93*, 373−389.
(6) Nardi, B. A. *A Small Matter of Programming: Perspectives on End User Computing*, 1st ed.; MIT Press: Cambridge, MA, USA, 1993.
(7) Dorn, B.; Tew, A. E.; Guzdial, M. Introductory Computing Construct Use in an End-User Programming Community. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*; IEEE: 2007; pp 27−32.
(8) Dorn, B.; Guzdial, M. Discovering Computing: Perspectives of Web Designers. *Proceedings of the Sixth International Workshop on Computing Education Research*; Association for Computing Machinery: New York, NY, USA, 2010; pp 23−30.
(9) Dorn, B.; Guzdial, M. Learning on the Job: Characterizing the Programming Knowledge and Learning Strategies of Web Designers. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*; Association for Computing Machinery: New York, NY, USA, 2010; pp 703−712.
(10) Guzdial, M. A Media Computation Course for Non-Majors. *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*; Association for Computing Machinery: New York, NY, USA, 2003; pp 104−108.
(11) Repenning, A.; Webb, D.; Ioannidou, A. Scalable Game Design and the Development of a Checklist for Getting Computational Thinking into Public Schools. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*; Association for Computing Machinery: New York, NY, USA, 2010; pp 265−269.
(12) Basawapatna, A.; Koh, K. H.; Repenning, A.; Webb, D. C.; Marshall, K. S. Recognizing Computational Thinking Patterns. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*; Association for Computing Machinery: New York, NY, USA, 2011; pp 245−250.
(13) Wood, Z. J.; Clements, J.; Peterson, Z.; Janzen, D.; Smith, H.; Haungs, M.; Workman, J.; Bellardo, J.; DeBruhl, B. Mixed Approaches to CS0: Exploring Topic and Pedagogy Variance after Six Years of CS0. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*; Association for Computing Machinery: New York, NY, USA, 2018; pp 20−25.
(14) Bart, A. C.; Whitcomb, R.; Kafura, D.; Shaffer, C. A.; Tilevich, E. Computing with CORGIS: Diverse, Real-World Datasets for Introductory Computing. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*; Association for Computing Machinery: New York, NY, USA, 2017; pp 57−62.
(15) Kazerouni, A. M.; Lehr, J.; Wood, Z. Community-action Computing: A Data-centric CS0 course. *Proceedings of the 55th ACM Technical Symposium on Computer Science Education*; Association for Computing Machinery: New York, NY, USA, 2024.
(16) McDonald, A. R.; Roberts, R.; Koeppe, J. R.; Hall, B. L. Undergraduate structural biology education: A shift from users to developers of computation and simulation tools. *Curr. Opin. Struct. Biol.* **2022**, *72*, 39−45.
(17) Gick, M. L.; Holyoak, K. J. Schema induction and analogical transfer. *Cognitive psychology* **1983**, *15*, 1−38.
(18) Bransford, J. D.; Brown, A. L.; Cocking, R. R.; et al. *How People Learn*; National Academy Press: Washington, DC, 2000; Vol. *11*.
(19) Burridge, J.; Fekete, A. Teaching Programming for First-Year Data Science. *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education*; Association for Computing Machinery: New York, NY, USA, 2022; Vol. 1, pp 297−303.
(20) Felleisen, M.; Findler, R. B.; Flatt, M.; Krishnamurthi, S. *How to Design Programs: An Introduction to Programming and Computing*; MIT Press: 2018.
(21) Loksa, D.; Ko, A. J.; Jernigan, W.; Oleson, A.; Mendez, C. J.; Burnett, M. M. Programming, problem solving, and self-awareness: Effects of explicit guidance. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*; Association for Computing Machinery: 2016; pp 1449−1461.
(22) Prather, J.; Pettit, R.; McMurry, K.; Peters, A.; Homer, J.; Cohen, M. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. *Proceedings of the 2018 ACM Conference on International Computing Education Research*; Association for Computing Machinery: New York, NY, USA, 2018; pp 41−50.
(23) Prather, J.; Pettit, R.; Becker, B. A.; Denny, P.; Loksa, D.; Peters, A.; Albrecht, Z.; Masci, K. First Things First: Providing Metacognitive Scaffolding for Interpreting Problem Prompts. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*; Association for Computing Machinery: New York, NY, USA, 2019; pp 531−537.
(24) Prather, J.; Becker, B. A.; Craig, M.; Denny, P.; Loksa, D.; Margulieux, L. What Do We Think We Think We Are Doing? Metacognition and Self-Regulation in Programming. *Proceedings of the 2020 ACM Conference on International Computing Education Research*; Association for Computing Machinery: New York, NY, USA, 2020; pp 2−13.
(25) Norman, D. A. In *Mental Models*; Gentner, D., Stevens, A. L., Eds.; Psychology Press: New York, 1983; pp 15−22.
(26) de Kleer, J.; Brown, J. S. In *Cognitive Skills and Their Acquisition*; Anderson, J. R., Ed.; Psychology Press: 1981; Chapter 1.
(27) du Boulay, B.; O'Shea, T.; Monk, J. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies* **1981**, *14*, 237−249.
(28) Sorva, J. Notional Machines and Introductory Programming Education. *ACM Trans. Comput. Educ.* **2013**, *13*, 1.
(29) Ben-Ari, M. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching* **2001**, *20*, 45−73.
(30) Dickson, P. E.; Brown, N. C. C.; Becker, B. A. Engage Against the Machine: Rise of the Notional Machines as Effective Pedagogical Devices. *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*; Association for Computing Machinery: New York, NY, USA, 2020; pp 159−165.
(31) Baiz, C. R.; Berger, R. F.; Donald, K. J.; de Paula, J. C.; Fried, S. D.; Rubenstein, B.; Stokes, G. Y.; Takematsu, K.; Londergan, C. Lowering Activation Barriers to Success in Physical Chemistry (LABSIP): A Community Project. *J. Phys. Chem. A* **2024**, *128*, 3.
(32) DeTar, D. F. A computer program for making steady state calculations: Notes on effective programming techniques. *J. Chem. Educ.* **1967**, *44*, 193.
(33) Kim, H. Computer programming in physical chemistry laboratory: Least-squares analysis. *J. Chem. Educ.* **1970**, *47*, 120.
(34) Wanzer, D.; McKlin, T.; Edwards, D.; Freeman, J.; Magerko, B. Assessing the Attitudes Towards Computing Scale: A Survey Validation Study. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*; Association for Computing Machinery: New York, NY, USA, 2019; pp 859−865.

https://doi.org/10.1021/acs.jchemed.4c00219
J. Chem. Educ. XXXX, XXX, XXX−XXX

(35) Jones, B. D. Motivating students to engage in learning: the MUSIC model of academic motivation. *International Journal of Teaching and Learning in Higher Education* **2009**, *21*, 272−285.
(36) Barach, A. E.; Jenkins, C.; Gunawardena, S. S.; Kecskemety, K. M. MCS1: A MATLAB Programming Concept Inventory for Assessing First-year Engineering Courses. *2020 ASEE Virtual Annual Conference Content Access* [Virtual Online]; 2020. https://peer.asee.org/34958.
(37) Parker, M. C.; Guzdial, M.; Engleman, S. Replication, Validation, and Use of a Language Independent CS1 Knowledge Assessment. *Proceedings of the 2016 ACM Conference on International Computing Education Research*; Association for Computing Machinery: New York, NY, USA, 2016; pp 93−101.
(38) Parker, M. C.; Guzdial, M.; Tew, A. E. Uses, Revisions, and the Future of Validated Assessments in Computing Education: A Case Study of the FCS1 and SCS1. *Proceedings of the 17th ACM Conference on International Computing Education Research*; Association for Computing Machinery: 2021; pp 60−68.
(39) Strauss, A.; Corbin, J. M. *Grounded Theory in Practice*; Sage: 1997.
(40) Fuller, U.; Johnson, C. G.; Ahoniemi, T.; Cukierman, D.; Hernán-Losada, I.; Jackova, J.; Lahtinen, E.; Lewis, T. L.; Thompson, D. M.; Riedesel, C.; Thompson, E. Developing a computer science-specific learning taxonomy. *ACM SIGCSE Bulletin* **2007**, *39*, 152−170.
(41) Xie, B.; Loksa, D.; Nelson, G. L.; Davidson, M. J.; Dong, D.; Kwik, H.; Tan, A. H.; Hwa, L.; Li, M.; Ko, A. J. A theory of instruction for introductory programming skills. *Computer Science Education* **2019**, *29*, 205−253.
(42) Castro, F. E. V.; Fisler, K. Designing a Multi-Faceted SOLO Taxonomy to Track Program Design Skills through an Entire Course. *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*; Association for Computing Machinery: New York, NY, USA, 2017; pp 10−19.
(43) Pea, R. D.; Soloway, E.; Spohrer, J. C. The buggy path to the development of programming expertise. *Focus on Learning Problems in Mathematics* **1987**, *9*, 5−30.
(44) Castro, F. E. V.; Fisler, K. *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*; Association for Computing Machinery: New York, NY, USA, 2020; pp 487−493.
(45) Pennington, N. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* **1987**, *19*, 295−341.
(46) Margulieux, L. E.; Morrison, B. B.; Decker, A. Design and pilot testing of subgoal labeled worked examples for five core concepts in CS1. *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*; Association for Computing Machinery: 2019; pp 548−554.
(47) Kramer, J. Is abstraction the key to computing? *Communications of the ACM* **2007**, *50*, 36−42.
(48) Bennedsen, J.; Caspersen, M. E. Programming in context: a model-first approach to CS1. *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*; Association for Computing Machinery: 2004; pp 477−481.
(49) Dijkstra, E. W. The humble programmer. *Communications of the ACM* **1972**, *15*, 859−866.
(50) Hazzan, O. How students attempt to reduce abstraction in the learning of mathematics and in the learning of computer science. *Computer Science Education* **2003**, *13*, 95−122.
(51) Lee, I.; Martin, F.; Denner, J.; Coulter, B.; Allan, W.; Erickson, J.; Malyn-Smith, J.; Werner, L. Computational thinking for youth in practice. *Acm Inroads* **2011**, *2*, 32−37.
(52) Gautam, A.; Bortz, W.; Tatar, D. *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*; Association for Computing Machinery: New York, NY, USA, 2020; pp 393−399.
(53) Koppelman, H.; Van Dijk, B. Teaching abstraction in introductory courses. *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*; Association for Computing Machinery: 2010; pp 174−178.
(54) Hazzan, O. Reflections on teaching abstraction and other soft ideas. *ACM SIGCSE Bulletin* **2008**, *40*, 40−43.
(55) Victor, B. *Up and down the ladder of abstraction*. 2011. http://worrydream.com/LadderOfAbstraction/ (accessed 2023-03-04).
(56) Ainsworth, S. *Visualization: Theory and Practice in Science Education*; Springer: 2008; pp 191−208.
(57) Kozma, R. The material features of multiple representations and their cognitive and social affordances for science understanding. *Learning and instruction* **2003**, *13*, 205−226.
(58) Rich, K. M.; Binkowski, T. A.; Strickland, C.; Franklin, D. Decomposition: A K-8 computational thinking learning trajectory. *Proceedings of the 2018 ACM Conference on International Computing Education Research*; Association for Computing Machinery: 2018; pp 124−132.
(59) K-12 Computer Science Framework Steering Committee. *K-12 Computer Science Framework*; ACM: 2016.
(60) Seehorn, D.; Carey, S.; Fuschetto, B.; Lee, I.; Moix, D.; O'Grady-Cunniff, D.; Owens, B. B.; Stephenson, C.; Verno, A. *CSTA K−12 Computer Science Standards: Revised 2011*; ACM: 2011.
(61) Ioannidou, A.; Bennett, V.; Repenning, A.; Koh, K. H.; Basawapatna, A. Computational Thinking Patterns. Presented at the 2011 Annual Meeting of the American Educational Research Association (AERA).
(62) Schanzer, E.; Fisler, K.; Krishnamurthi, S.; Felleisen, M. Transferring skills at solving word problems from computing to algebra through Bootstrap. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*; Association for Computing Machinery: 2015; pp 616−621.
(63) Brennan, K.; Resnick, M. New frameworks for studying and assessing the development of computational thinking. *Proceedings of the 2012 Annual Meeting of the American Educational Research Association*; AERA: 2012; p 25.
(64) Keen, A.; Mammen, K. Program decomposition and complexity in CS1. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*; Association for Computing Machinery: 2015; pp 48−53.
(65) Muller, O.; Ginat, D.; Haberman, B. Pattern-oriented instruction and its influence on problem decomposition and solution construction. *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*; Association for Computing Machinery: 2007; pp 151−155.
(66) Lee, M. J.; Bahmani, F.; Kwan, I.; LaFerte, J.; Charters, P.; Horvath, A.; Luor, F.; Cao, J.; Law, C.; Beswetherick, M.; et al.Principles of a debugging-first puzzle game for computing education. *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*; IEEE: 2014; pp 57−64.
(67) Meerbaum-Salant, O.; Armoni, M.; Ben-Ari, M. Habits of programming in scratch. *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*; Association for Computing Machinery: 2011; pp 168−172.
(68) Gregg, C.; Tychonievich, L.; Cohoon, J.; Hazelwood, K. EcoSim: a language and experience teaching parallel programming in elementary school. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*; Association for Computing Machinery: 2012; pp 51−56.
(69) Jacobson, M. J.; Wilensky, U. Complex systems in education: Scientific and educational importance and implications for the learning sciences. *Journal of the learning sciences* **2006**, *15*, 11−34.
(70) Voss, J. F.; Greene, T. R.; Post, T. A.; Penner, B. C. *Psychology of Learning and Motivation*; Elsevier: 1983; Vol. *17*, pp 165−213.
(71) Polya, G. *How to Solve It: A New Aspect of Mathematical Method*; Princeton University Press: 2004.
(72) Nersessian, N. J. How do scientists think? Capturing the dynamics of conceptual change in science. *Cognitive Models of Science*; Giere, R., Ed.; Minnesota Studies in the Philosophy of Science 15; University of Minnesota Press: 1992; pp 3−44.
(73) Hmelo, C. E.; Holton, D. L.; Kolodner, J. L. Designing to learn about complex systems. *journal of the learning sciences* **2000**, *9*, 247−298.

https://doi.org/10.1021/acs.jchemed.4c00219
J. Chem. Educ. XXXX, XXX, XXX−XXX

(74) Edwards, S. H. Using software testing to move students from trial-and-error to reflection-in-action. *ACM SIGCSE Bulletin* **2004**, *36*, 26.
(75) Sorva, J. *Visual Program Simulation in Introductory Programming Education*; Aalto University Publication Series Doctoral Dissertations 61/2012; School of Science, Aalto University: Espoo, 2012.
(76) Papert, S. A. *Mindstorms: Children, Computers, and Powerful Ideas*; Basic Books: 2020.
(77) Resnick, M. Falling in love with Seymour's ideas. Presented at the American Educational Research Association (AERA) Annual Conference.
(78) This took place during online instruction due to the COVID-19 pandemic.
(79) We omitted "I consider myself as a scientist, technologist, engineer, or mathematician", since we are interested in students' *computing* identities, disentangled from their identities as chemists and biochemists.
(80) These terms constituted our institution's return to in-person instruction during the COVID-19 pandemic.
(81) Two levels from Fuller et al.'s taxonomy are not depicted: the *evaluate* level on the interpreting axis, because no students displayed that ability level, and the *create* level on the producing axis, because our interviews did not ask students to create any code from scratch.

https://doi.org/10.1021/acs.jchemed.4c00219
J. Chem. Educ. XXXX, XXX, XXX−XXX