Thinking About Computational Thinking

James J. Lu
Mathematics and Computer Science
Emory University, Atlanta, GA, USA
jlu@mathcs.emory.edu

George H. L. Fletcher
School of Engineering and Computer Science
Washington State University, Vancouver, USA
fletcher@vancouver.wsu.edu

ABSTRACT

Jeannette Wing's call for teaching Computational Thinking (CT) as a formative skill on par with reading, writing, and arithmetic places computer science in the category of basic knowledge. Just as proficiency in basic language arts helps us to effectively communicate and in basic math helps us to successfully quantitate, proficiency in computational thinking helps us to systematically and efficiently process information and tasks. But while teaching everyone to think computationally is a noble goal, there are pedagogical challenges. Perhaps the most confounding issue is the role of programming, and whether we can separate it from teaching basic computer science. How much programming, if any, should be required for CT proficiency?

We believe that to successfully broaden participation in computer science, efforts must be made to lay the foundations of CT long before students experience their first programming language. We posit that programming is to Computer Science what proof construction is to mathematics, and what literary analysis is to English. Hence by analogy, programming should be the entrance into higher CS, and not the student's first encounter in CS. We argue that in the absence of programming, teaching CT should focus on establishing vocabularies and symbols that can be used to annotate and describe computation and abstraction, suggest information and execution, and provide notation around which mental models of processes can be built. Lastly, we conjecture that students with sustained exposure to CT in their formative education will be better prepared for programming and the CS curriculum, and, furthermore, that they might choose to major in CS not only for career opportunities, but also for its intellectual content.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

Theory

Keywords

computational thinking, language, K-12 education

1. INTRODUCTION

Since the dot-com bubble, the conundrum we face in computer science is how such a useful discipline can have such difficulties attracting students, despite continuing growth of the IT industry. We blame student disinterest on career instability, but similar and even stronger arguments have long existed for other disciplines with little impact on enrollment. Recent data from the National Center for Education Statistics show that computer and information sciences conferred fewer degrees than either the visual and performing arts or the social sciences and history[1] – hardly the stuff that ironclad career guarantees are made of. Not surprisingly, the number of students majoring in CS lags far behind those majoring in other practically-perceived disciplines such as education or business. Through the years, despite our best efforts to articulate that CS is more than "just programming," the misconception that the two are equivalent remains. This equation continues to project an overly narrow and misleading image of our discipline [6] – and this directly impacts both the size and character of the body of students we attract.

Jeannette Wing's call for teaching Computational Thinking (CT) [10] as a formative skill on par with reading, writing, and arithmetic (the three R's) places the core of CS, we believe correctly, in the category of basic knowledge. Very briefly, the key points of Computational Thinking[2] are that 1) it is a way of solving problems and designing systems that draws on concepts fundamental to computer science; 2) it means creating and making use of different levels of abstraction, to understand and solve problems more effectively; 3) it means thinking algorithmically and with the ability to apply mathematical concepts to develop more efficient, fair, and secure solutions; and 4) it means understanding the consequences of scale, not only for reasons of efficiency but also for economic and social reasons. CT is *not* about getting humans to think like computers [10], but rather about developing the full set of mental tools necessary to effectively use computing to solve complex human problems [8].

To be sure, information representation, abstraction, efficiency, and heuristics are recurring themes that arise in ordinary human activities on a daily basis. Dealing with these themes does not require specialized career skills. On the contrary, just as proficiency in basic language arts helps

[1] `http://nces.ed.gov/programs/digest/d06/figures/fig_15.asp`
[2] `http://www.cs.cmu.edu/~CompThink/`

us to effectively *communicate* and in basic math helps us to successfully *quantitate*, proficiency in computational thinking – what we synonymously call basic computer science – helps us to systematically, correctly and efficiently *process* information and tasks.[3]

But while teaching everyone to think computationally is a noble goal, there are pedagogical challenges. It is not enough to simply repackage CS1, or CS0, and teach it at an earlier stage, as many high schools and even lower-level grades already do. Perhaps the most confounding issue is the role of programming, and whether we can separate it from teaching basic computer science. How much programming, if any, should be required for CT proficiency? To answer this question, we believe it is necessary to consider more carefully the implications of aligning CT with the three R's.

In this essay we explore this alignment and argue that to successfully broaden awareness of the depth, breadth, and beauty of computer science, efforts must be made to lay the foundations of CT long before students experience their first programming language.

2. PROGRAMMING: DESCRIBING COMPUTATIONAL PROCESSES

For those students interested in pursuing higher-level English and mathematics, there exist milestone courses to help shift the focus from the development of useful skills to the academic study of these subjects. In English, courses in literary analysis pave the way for students to read texts critically and to argue theoretically. In mathematics, a course on proof understanding and construction is the gateway into higher mathematics. These courses make critical intellectual leaps. And while being educated implies proficiency in basic reading, writing, and quantitative skills, it does not imply knowledge of or the ability to understand and carry out scholarly English and mathematics. Analogously, we believe the same dichotomy exists between computational thinking, as a skill, and computer science as an academic subject. Our thesis is this.

> *Programming is to CS what proof construction is to mathematics and what literary analysis is to English.*[4]

Logically, a program is a succinct, finite description of many computation instances, each embodying a possibly infinite process. Pragmatically, understanding a program execution on a computer requires, in addition to the program logic, an understanding of how the logical artifact is constrained by concrete physical limitations (e.g., an integer is no longer an ordinary integer in the context of 16-bit representation). Mastery of these and related insights is essential for students interested in higher-level CS, and cannot be achieved without intense immersion in crafting programs. Only by fretting over details of data representation, algorithm design and analysis, programming languages, and systems will students be prepared to tackle CS on a higher plane. In this respect, the traditional programming-first curriculum is appropriate.

Programming should not, however, be essential in the teaching of computational thinking, nor should knowledge of programming be necessary to proclaim literacy in basic computer science. Just as math students come to proofs after 12 or more years of experience with basic math, and English students come to literary analysis after an even longer period of reading and writing, programming should begin for *all* students only after they have had substantial practice thinking computationally. Missing in primary and secondary school curricula are the equivalents of math courses such as algebra, probability, and calculus, and English courses such as language arts, literature, and composition. Missing also are the buffet of service courses which college mathematics and English departments offer non-majors to enhance their reading, writing, and quantitative skills. Currently, the setting in which students are introduced to CT is also where they first learn programming. This is true not just of the conventional CS1/CS2 courses, but also of the many service-level CS0 courses. This pedagogical approach is akin to teaching basic arithmetic alongside proof construction, and elementary reading and writing with linguistics and discourse analysis. It can be done, but perhaps not optimally. Writing descriptions in unfamiliar formal languages cannot be easy when one does not yet have a solid grasp of the processes that these descriptions are designed to capture. A corollary to our thesis, then, is the following.

> *Substantial preparation in computational thinking is required before students enroll in programming courses.*[5]

3. A LANGUAGE FOR THINKING COMPUTATIONALLY

We need to start teaching computational thinking early and often. But what does this entail in the absence of programming? The emphasis should be on understanding (and being able to manually perform) computational processes, and not on their manifestations in particular programming languages. Gaining familiarity with algorithmic notions such as basic flow of control is important. Also central is the development of skills for abstracting and representing information, and for evaluating properties of processes.

In grade school mathematics, we already speak of *representing* word problems, and applying algebraic rules to *derive* simpler forms. But we can also describe the *search space* of possible algebraic simplifications, induced by the *initial state*, the *final state*, and the set of applicable *operations*. We may explore the process of finding a derivation through a *blind* or a *heuristic* search, and compare the *efficiency* of two derivations.

As a glue for connecting these concepts, a common language – a *computational thinking language* (CTL) – must permeate the pedagogy. Again, this is not a programming language, but rather vocabularies and symbols that can be

[3]cf. the notion of *procedural epistemology* introduced in the preface to Abelson and Sussman's classic textbook [1].

[4]Clearly, programming poses intellectual challenges similar to those faced in proving theorems and analyzing texts (cf. [2]). Indeed, D. Knuth has stated in interview that "writing software was much more difficult than anything else I had done in my life" [9]. Furthermore, he notes "I didn't realize how much more bandwidth of my brain was being taken up by that [software] work than it was when I was doing *just* theoretical work."

[5]This preparation should, of course, be coupled with efforts to rethink the ways in which we transition students into programming, as has been recently suggested [7].

used to annotate and describe computation and abstraction, suggest information and execution, and provide notation around which semantic understanding of computational processes can be hung [4, 8].

As Wing and others have pointed out, there are countless opportunities to integrate computational thinking into existing middle and high-school courses, or even at the primary school level [5]. The most obvious occur in math courses. But just as researchers in the biological and physical sciences have long realized, and scholars in the social sciences and humanities are discovering, computing processes are ubiquitous. An appropriate set of these opportunities should be identified, enunciated explicitly in the CTL, and integrated into pre-college curricula.

4. LEARNING ABOUT COMPUTATIONAL PROCESSES

A CTL should include many familiar and basic notions for data representation and transformation. The notation and concepts that are presented in association with the CTL must, of course, be grade-appropriate. In the rest of this section, we present a series of examples to show possible ways of integrating computational thinking into primary and secondary curricula. For each, we focus on the basic computer science concepts and ideas that it conveys, and highlight relevant CTL vocabularies and notations. Some of our examples are adapted from materials found on the popular website `edHelper.com`, which also provides us guidance on current U.S. curricula.

4.1 CTL Vocabularies

A student's initial encounter with computational thinking is likely to be around Grade 3, when multi-step calculations and small combinatorial problems are first encountered. At this stage, strategic introduction of vocabularies can create awareness of computational processes. We next consider a few illustrative opportunities to introduce CTL vocabulary.

Example 1 (Introduction to Multiplication). Current curricula introduce multiplication in Grade 3. Two common concepts are "multiplication is repeated addition" and "the result of multiplication is the same no matter which number you write first."

The use of repeated addition as a definition for multiplication is an opportunity to introduce two computational concepts: *iteration* and *efficiency*. We may explain that each application of the symbol + is an iteration, and that while the operation is commutative, the efficiency of the two forms of expression may be different. Some useful exercises may be the following.

1. For each multiplication, write it as repeated addition and then the answer. Also write down the number of *iterations* that are required.

2. Write the multiplication by switching the two numbers, and compare the number of iterations required. Which one is more *efficient*?

Example

expression	numbers switched	which is more efficient?
3×6	6×3	6×3 needs 3 iterations, 3×6 needs 6, so 6×3 is more efficient.

Example 2 (Reading Comprehension). As part of developing their reading comprehension skills, students in Grade 3 are often given the task of putting a set of simple sentences into chronological order. Consider an example of such an exercise: Given the four sentences

```
1:  I don't want pizza again for a long time.
2:  I ate ten pieces of pizza.
3:  Later that night, I got sick.
4:  I felt very full.
```

which of the following sentence orderings is correct?

a)	1, 3, 4, 2	b)	4, 3, 2, 1
c)	2, 3, 1, 4	d)	3, 1, 4, 2
e)	2, 4, 3, 1		

We may explain to students that each ordering of the four sentences is a *state*, and the five possibilities a) through e) make up the *search space* of the problem. To solve the problem, we may verify each state individually, but we can also use *divide-and-conquer* to *prune* incorrect answers. The following are potentially suggestive homework questions.

1. What is the correct ordering between 2 and 3?

2. Which of the states in the search space have 2 and 3 in the wrong order? Can these answers be correct?

3. What are some other possible states not listed?

Later in middle school, when *permutation* is introduced, the search space concept may be revisited and broadened to those sentence orderings that result from applying the permute *operation* to some initial sentence ordering. □

Example 3 (Charting Information). Students are introduced to a variety of visual display techniques for *data representation* in grade school: pie chart, bar graph, table. In addition to teaching students how to read each type of display, which is currently the goal of most exercises on this topic, a comparison of computational advantages can also be discussed. For example, suppose a pie chart labels each slice as a ratio of the slice to the whole, and for the same set of data, a bar graph labels each bar as an absolute number of the corresponding category. Then determining the the percentage of each category with respect to the total is "easier" in the pie chart than in the bar graph, while determining the ratio between any two categories is more difficult in the pie chart. We can make the notion of *easy* and *hard* more precise by asking students how many arithmetic operations are required to compute the ratio of each category to the total. □

Example 4 (Automobile Assembly Line). In Grade 6 Social Studies, the assembly line concept is concurrently introduced with the history of the mass production of automobiles. The concept may be introduced much earlier, however, through the use of concrete illustrations and activities. A good example from the Columbia Education Center website is an art project for Grades 2-3 students.[6] Students individually complete projects that involve tracing and cutting variously shaped patterns, and then repeat the same

[6]`http://www.col-ed.org/cur/sst/sst180.txt`

task as an assembly line project with each student *specializing* on producing one pattern or on assembling the pieces into the final art work. Recording the time that it takes to complete an equal number of projects by the two processes (along with some discussion and calculation) will help to demonstrate improvements in *throughput*. Comparisons of the final products and the number of restarts due to mistakes will help to convey that by reducing the complexity of each person's or machine's task, we make the job easier and less error prone. □

4.2 CTL Notation

As students begin to encounter more complex calculations in middle school, helpful CTL notation should be introduced. In particular, basic tuple notation for structuring data and state representation, together with a simple rewrite system to annotate state changes can assist in clarifying the computational aspects of many problems.

Example 5 (Finding Square Roots). A quick Web search shows that the most common advice for finding the square root of a number n, apart from using a calculator, is by repeated guess and check. The process for obtaining each successive guess is to divide the current guess g into n, and averaging the result with g. This is the estimate-divide-average algorithm (EDA). Denoting by $\Rightarrow$ the above calculation, we may annotate the computation process for finding the square root of 60, for example, as follows, using a (poor) initial guess of 2.

$$\begin{array}{lllll} 2 & \Rightarrow & 16 & \Rightarrow & 9.875 \\ & \Rightarrow & 7.975 & \Rightarrow & 7.749 \\ & \Rightarrow & 7.746 & & \end{array}$$

We may explain that $\Rightarrow$ is an *abstraction* of the function $\lambda g.(g/60 + g)/2$, and compare its efficiency to a naive calculation, such as a simple linear search (e.g., $\lambda g.g + 0.1$). Depending on the precision desired, the concept of *correctness* may be noted. For example, linear search may be unable to achieve results that are within an acceptable range; even worse, it may result in an *infinite loop*.

Another alternative is to describe a binary search in which a range is explicitly represented and successively narrowed until its mid-point is sufficiently close to the answer. The comparative computation provides a chance to discuss *representation* and *decision*. For EDA, a single number captures the complete state since subsequent state is a unary function of the current state. For the binary search, a pair representation is necessary as each subsequent state is a conditional on both bounds of the current range.[7] □

Example 6 (Diagramming Sentences). Parsing and diagramming sentences using the Reed-Kellog system or as trees is often employed in language arts classes for learning grammar. Starting with the simplest examples, the first lesson in diagramming is typically to identify a given sentence's subject (phrase) and verb (phrase). Each sub-phrase is subsequently further identified with other linguistic categories.

Regardless of the representation system, such exercises are useful for familiarizing students with different notations. Moreover, the representation of context-free grammars and the process of derivation are ideal opportunities for emphasizing the power of *recursion* and *non-determinism*.

$$\begin{array}{lcl} \texttt{sentence} & \to & \texttt{noun-phrase verb-phrase} \\ \texttt{noun-phrase} & \to & \texttt{modifier noun} \mid \texttt{noun} \\ & \vdots & \\ \texttt{verb-phrase} & \to & \texttt{verb noun-phrase} \end{array}$$

The second rule shows two possible rewrites of `noun-phrase`. Together with the last rule, the grammar illustrates recursion. The diagramming process may be notated in the familiar parse tree form, or, to show non-deterministic computational processes, as a derivation through recursive application of grammar rules.

$$\begin{array}{lcl} (\texttt{sentence}) & \Rightarrow & (\texttt{noun phrase}, \texttt{verb phrase}) \\ & \Rightarrow & ((\texttt{modifier}, \texttt{noun}), \texttt{verb phrase}) \\ & \vdots & \\ & \Rightarrow & ((\texttt{the}, \texttt{summer}), (\texttt{is}, \texttt{over})) \end{array}$$

□

4.3 Revisiting, Advancing, and Integrating

As students mature, previously introduced ideas may be revisited at a more advanced level and with more rigor.

Example 7 (Equational Reasoning). As students become adept with basic algebra in middle school, they are gradually introduced to the use of equations in a variety of courses such as physics, chemistry, business, astronomy, and biology. This is an excellent opportunity to present the uses and power of *functional abstraction* and *procedural problem solving*, both of which are core CT skills.

For example, consider the standard physics textbook equation $a = \frac{\Delta v}{\Delta t}$ used in computing the acceleration a of an object given a change in velocity v over a period of time t. In essence, acceleration a is the output type of the function

$$\lambda v, v', t, t'.(v' - v)/(t' - t)$$

which is used (i.e., called as a *sub-procedure*) in a variety of other basic physical equations, such as Newton's second law of motion for computing force, $F = ma$. Force F, in turn, is a sub-procedure in a host of other physical equations. Such abstract reasoning about functions, data typing, and function composition is intuitive, simplifying, and powerful. □

Integration of concepts helps students broaden their appreciation for the universality and ubiquity of computational thinking, and, furthermore, conveys the point that CT skills are essential in a wide variety of endeavors.

Example 8 (Interdisciplinary Projects). An interesting interdisciplinary project is making travel brochures. The problem requires students to apply language arts, math, and social studies (e.g., geography), and may be posed as an exercise as early as 5th grade. A good example can be found on the Columbia Education Center website.[8] In high school, an *optimization* version of the problem may be discussed, in the development of "good" tour packages over a geographic region (e.g., Eastern Europe). This is a scheduling problem that must take into account *constraints* such as distance, time, expense, levels of interest of possible destinations, and others. Rather than turning in just the finished

[7] $\lambda a, b.\text{if } (a + b)/2 > 60, (a, (a + b)/2);\ \text{else } ((a + b)/2, b)$

[8] `http://www.col-ed.org/cur/sst/sst114.txt`

product, students may also be asked to demonstrate computational thinking by showing how the constraints may be represented, and how (near) *optimal* solutions with respect to some *objective function* may be computed. An example objective function might be to maximize profit for the travel agency while meeting some "fun" threshold for the travelers. □

In addition to the explicit integration of CT and CTL in traditional subjects, it might also be useful to investigate opportunities for introducing information processing into ordinary classroom activities.

Example 9 (Group Projects). Group work is common in science courses. Relationships in the group are typically divided along tasks: each group member takes sole responsibility for one or two tasks (e.g., data recording, report write-up). Data-exchange interactions are ideal situations for formally introducing notions of *interface* and *encapsulation*. By restructuring the interactions differently, interesting new challenges arise. For example, if the project report must be written collaboratively (either simultaneously or *asynchronously*), then concepts such as *locking* and *message passing* must be discussed and implemented among group members. □

In all the examples of this section, note that computers are not explicitly part of the discussion. Indeed, during the introduction and development of CT and CTL, *students* are the computing agents. The emphasis here is on helping students, as computers, to become more knowledgeable, skilled, and effective. Of course, software tools can be used to assist in this, just as they are used in teaching basic language skills and math.

In contrast to other major computer science education efforts, such as [5], our suggestion is not so much to teach about how computation is implemented, but rather to permeate the collective knowledge and lessons of computer science research into the discussion and development of all subjects that involve (information) processing. To that end, we need a team of computationally aware thinkers to fundamentally reexamine current pre-college curricula, to identify opportunities and to create educational material for introducing and integrating CT and CTL.

5. DISCUSSION

Through practice and repeated encounters, thinking and communicating in the CTL will become second nature by the time students reach their final year in high school. A culminating AP course or equivalent introductory college courses, such as the excellent "Great Theoretical Ideas In Computer Science" by Steven Rudich and colleagues at CMU,[9] will help to integrate students' experiences and prepare them for exposure to programming. Such courses may formally address computational properties, such as convergence, efficiency and limits of computation, again without necessarily referring to a specific computing agent. College-level service courses may be offered on domain-specific computational thinking (e.g., bioinformatics, chemical informatics).

For students that follow up with more advanced CS courses, starting with programming, the challenge will no longer be in learning to think computationally, but in learning the nuances of new languages, how to formally describe computations in these languages, and in subsequent courses on how such descriptions are executed on a von Neumann machine. For students that do not pursue CS further, their background in computational thinking will be of substantial benefit in their professional careers and in everyday life. Indeed, in this age of information, it is crucial to have a solid understanding of the uses and limitations of CT.

On the issue of CS enrollment, we suspect that most students major in mathematics, English, and the humanities not for the abundance of career opportunities in these fields, but more for intellectual interests, born out of gradual and sustained exposure. We conjecture that students with similar development in computational thinking will also be more likely to choose CS based on intellectual motivations (cf. [3]), and, furthermore, that they will be better prepared for programming and the major curriculum. Exposure to basic computer science will raise an awareness in students of what CS (and, more broadly, informatics) might be as a field of inquiry, leading to a broader participation of a wider variety of students in our discipline.

To truly integrate computational thinking into current primary and secondary curricula undoubtably presents significant challenges. It will necessarily be a gradual and evolutionary process, and requires concerted efforts and coordination among many constituents of the wider education community. We see concrete efforts towards achieving broader CT literacy as some of the most exciting, challenging, and necessary next steps in the maturation of our discipline.

Acknowledgments. Our (computational) thinking has benefited greatly from discussions with and comments from colleagues: Dwight Duffus, Mark Guzdial, Ken Lambert, Ken Mandelberg, Eileen Peluso, Dave Reed, Sebastien Siva, Allen Tucker, and Scott Wallace. We also thank the reviewers for their helpful comments and criticism.

6. REFERENCES

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs, 2nd ed.* MIT Press, Cambridge, 1996.

[2] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Trans. Program. Lang. Syst.* 7(1):113-136, 1985.

[3] L. Carter. Why students with an apparent aptitude for computer science don't choose to major in computer science. SIGCSE 2006, Houston, pp. 27-31.

[4] A. Cohen and B. Haberman. Computer science: a language of technology. *SIGCSE inroads* 39(4):65-69, 2007.

[5] CS Unplugged. `http://csunplugged.com`.

[6] P. J. Denning and A. McGettrick. Recentering computer science. *CACM* 48(11):15-19, 2005.

[7] M. Guzdial. Paving the way for computational thinking. *CACM* 51(8):25-27, 2008.

[8] S. Reges. The mystery of "b := (b = false)." SIGCSE 2008, Portland, pp. 21-25.

[9] L. Shustek, ed. Donald Knuth: a life's work interrupted, part 2. *CACM* 51(8):31-35, 2008.

[10] J. M. Wing. Computational thinking. *CACM* 49(3):33-35, 2006.

[9] `http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15251-s04/Site/`