Design Principles behind *Beauty and Joy of Computing*

Paul Goldenberg
EDC
Waltham, MA, USA
pgoldenberg@edc.org

June Mark
EDC
Waltham, MA, USA
jmark@edc.org

Brian Harvey
UCB
Berkeley, CA, USA
bh@cs.berkeley.edu

Al Cuoco
EDC
Waltham, MA, USA
acuoco@edc.org

Mary Fries
EDC
Waltham, MA, USA
mfries@edc.org

ABSTRACT

This paper shares the design principles of one Advanced Placement Computer Science Principles (AP CSP) course, Beauty and Joy of Computing (BJC), both for schools considering curriculum, and for developers in this still-new field. BJC students not only *learn about* CS, but *do* some and analyze its social implications; we feel that the job of enticing students into the field isn't complete until students find programming, itself, something they enjoy and know they can do, and its key ideas accessible. Students must feel invited to use their own creativity and logic, and enjoy the power of their logic and the beauty and elegance of the code by which they express it. All kids need genuine challenge and sensible support so all can have the joy of making—seeing themselves as creators, not just consumers, and seeing that it is their own intellect, not just our instructions, that is the source of that making. Framework standards are woven into a consistent social and intellectual storyline to give the curriculum integrity.

Principles guide even our choice of programming language. Learners should focus on the logic and structure of their thinking, not on misplaced semicolons; attention to such syntactic detail is antithetical to broadening participation. We feature recursion and higher-order functions because they beautifully exemplify abstraction, a key idea in CS and the CSP framework.

BJC also places significant emphasis on the social implications of computing, balancing fundamental optimism about computing technology with a critical view of specific uses of technology.

KEYWORDS: CS education, Curriculum Design, Advanced Placement, Computer Science Principles

ACM Reference format:

Paul Goldenberg, June Mark, Brian Harvey, Al Cuoco, Mary Fries. 2020. Design principles behind *Beauty and Joy of Computing*. In *The 51st ACM Technical Symposium on Computer Science Education (SIGCSE'20), March 11–14, 2020, Portland, OR, USA*. ACM, NewYork, NY, USA, 7 pages. ACM, New York, NY, USA, 7 pages https://doi.org/10.1145/3328778.3366794

1 Introduction

The National Science Foundation (NSF) and College Board (CB) introduced the Advanced Placement Computer Science Principles (AP CSP) course to broaden participation in CS by appealing to high school students who didn't see CS as an inviting option—especially female, black, and Latinx students who have been typically underrepresented in computing. The AP CSP course was the centerpiece of an NSF-led initiative (CS10K) to train 10,000 high school teachers to teach CS [1, 2]. After pilots at college and high-school levels, CSP became an official AP course during 2016–17 with the first CSP exam given in May 2017 [3, 4, 5]. The CSP framework specifies six computational thinking practices and seven conceptual Big Ideas central to the study of computer science [6]. To support adoption of CSP, the NSF and other organizations funded several projects to develop curriculum materials aligned to the framework and to provide professional development for teachers to learn about curricular options and about pedagogical practices that support equitable CS instruction, and to plan for implementation and use in their schools [7].

The Beauty and Joy of Computing (BJC) curriculum was designed as a version of AP CSP that would meet these goals with strategically more emphasis on programming than the framework requires. CB-endorsed, BJC has been revised over the past four school years as part of an NSF-funded partnership among Education Development Center (EDC); University of California, Berkeley (UCB); the NYC Department of Education (NYCDOE); CSNYC (now CSforAll); and NCSU (North Carolina State University). BJC began as an undergraduate introduction to CS for non-majors at UCB [8]. High school teachers who knew of it saw its appeal and potential for their students, and some adapted it for their own use. But to spread successfully in high school, the college structure (e.g., hour lectures and unlimited lab time with TAs) and resources (e.g., lack of teacher guide, assessments, differentiated learning) required change.

The NSF's call for developing instances of such a course was an opportunity to revise the undergraduate BJC for wide use in high schools. To this end, UCB partnered with EDC, NYCDOE, NCSU, and CSNYC to redesign the student materials; create professional development, guides for teachers, and support for implementation; research the implementation of the materials and programs; and periodically report back to the field.

Given the range of options—now, a dozen curricula, varying in approach, all aligned with the framework [9] and endorsed by CB—we want to share with schools, in the same spirit as [10], the underlying design principles as schools consider these programs.

When we originally proposed this work, we believed that most of our attention would be on the structural elements and supplementary resources while course-content changes would only be needed for compliance with the AP CSP framework [7]. We also anticipated cleaning up writing and style, but only in a small way. In fact, things turned out quite differently.

By the time we were examining the student materials closely enough to fine-tune lesson by lesson, we began to see ways in which the epistemology did not reflect our principles, and we began to focus as much on issues of pedagogy and emphasis as on structure, resources, and compliance. The result is now one of the CB-endorsed curricula for the AP CSP course and exam.

What's to be learned from curriculum designers? By the time a curriculum reaches schools, the result often hides the thought, discussion, decisions about content and pedagogy, initial crafting, trials, revision, editing, and so on that went into its development, and the many contributors—content specialists, writers, classroom teachers, students, advisors, page designers, illustrators, editors, evaluators, and others—who tug in different directions. The process elements—having a vision and articulating it clearly, brainstorming, outlining, writing, designing format and layout, editing, field testing, revising, striving for equity, teaching in the classroom, assuring usability by teachers, assessing student progress, meeting required standards,accounting for sustainability, and more—interact in complex ways. A myriad of decisions must be made, all of which collectively determine the final product.

What *guides* the decisions? There are non-negotiables: content matter must not be factually wrong, text must be readable by the intended audience, and standards/frameworks can't be ignored. But otherwise, there are few absolutes. While there are certainly *wrong* ways to proceed, there are many right ways. Our aim here is to clarify what kinds of decisions must be made, illustrate how we came to particular ones, and show how philosophy, assumptions, and "high-level" decisions can affect curriculum design, in some cases right down to matters of "low-level" page-craft. The paths from vision to page and from principle to implementation are messy. We doubt that many visions survive whole by the time they've gone through the meat-grinder of what, for now, we'll just call "practicalities." The result often imperfectly represents the principles that were to guide it, but examining the principles can help guide other development.

All the principles we list below derive from our fundamental belief that c*urriculum inevitably teaches more than its list of contents* [11, 12, 13, 14]. Its organization and pedagogy also teach a point of view. For example, explanation followed by practice teaches a very different way of thinking than experience and experimentation followed by formalization and consolidation. Content can be arranged to emphasize theory or application, or the historical development or other features of a discipline. Without close attention to message, a curriculum can make everything from small details to major ideas seem equally important.

In BJC, our absolute top-level goal is *broadened participation* in *computer science.* All by itself, that goal dictates several things. At a minimum, such a curriculum must avoid biases in culture, intellectual or social interest, or accessibility that exclude students even implicitly. It must also be computer science. It must present real but manageable challenge so that students feel the fun and exhileration of *competence* and confidence that they can "do" computer science. That is, our way to serve the top-level goal is to entice students with the pleasure and sense of agency programmers feel when they use their own creativity and logic to make things, and to let them enjoy the power of their own logic and the beauty and elegance of the code they create to express it.

To accomplish this, BJC emphasizes programming more than the AP CSP standards require. Simple code and basic CS ideas can do part of that but, in our experience, big ideas (e.g., recursion and higher order functions) and their power can be presented accessibly. These *ideas*, not just coding or its products, can fascinate students and catch their interest. Of course, there are *many* big CS ideas—abstraction, algorithms, parallel processing, distributed processing, object-oriented programming, the concept (and techniques) of debugging, and so on. Because it is impossible to teach them all equally, we had to make decisions both about *what* to teach and about *how* to teach.

2 Pedagogical design principles

To broaden participation, we must consider *how* we teach as well as *what* we teach: believe in students, build experience, organize around big ideas, let them learn by doing, provide beauty and joy.

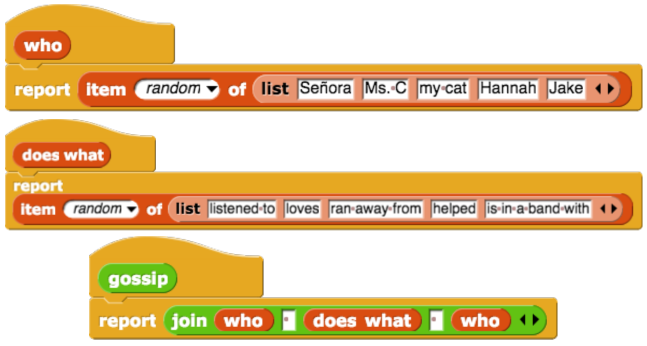
Design with conviction that all kids can do challenging things. Reaching a historically excluded audience can tempt curriculum writers to assume that reduced contact renders the new audience less capable than those who have already joined the club without invitation or accommodation. That assumption is destructive. Yes, people who have been excluded from CS (whether from external bias or from their own expectations) have often been implicitly or explicitly excluded from advantages in other subjects, too, with consequent background gaps. And, yes, many underrepresented kids are also students whose native language is not English and, therefore, find heavy text a barrier. So we try to keep text light and limit prerequisite special knowledge, but *without limiting challenge or depth.* One principle of our curriculum development is that all kids can do challenging things, and that all can figure out a great deal on their own. Because pace inevitably differs—sometimes because deep interest slows kids down to experiment more or speeds them up to find the next surprise, and sometimes because of hurdles that change how kids work—we must provide approaches that are genuine challenges to all kids; we must also provide sensible support so all students can experience the joy of making, and the joy of seeing that it is their own intellect, not our instructions, that is the source of that making.

Experience before formality. This epistemological principle has guided our curriculum development in mathematics and in programming for decades [15, 16, 17]. Roughly, it involves giving students enough varied experience with a concept to let them create the abstraction, the concept, themselves, saving technical details like vocabulary, definition, variations, and so on until students have built the basic concept. BJC employs this use-modify-create [18] paradigm. For example, very early in BJC, students load a fully-functioning project with three pieces of code:

who, does what, gossip.

They click on each, *using* each to see what it does. The **gossip** block reports a simple sentence. Then students open the blocks up to examine their structure (fig. 1). The structure is simple and the behavior they've seen in their experiments tells them what the code means. This is analogous to how children learn their native language, from use in context. Students *modify* the code—initially in an almost trivial way—personalizing it with their own content.

A second gossiper (fig. 2) with a different vocabulary, *not* yet seen by the students, sparks curiosity and leads to new learning but is still simple and clear. Some students use the pattern they see to make the second gossiper invoke the first again, leading to a recursive back and forth. Despite having deliberately introduced a call back to the first gossiper, some students are surprised by the continued behavior; some anticipate only *one* extra step, and not a recurring one. Long before we formally "teach" recursion, students (possibly just by accident) invent it themselves, though not yet with a way to control it. A lot has happened in just this one early lesson. Students encounter lists (arrays), multiple agents each with its own script, some control structure and event handling. They can modify these purposively before receiving formal instruction. And they have opportunities for surprise that build curiosity. Consolidation, extension, and explanation—formalizing the knowledge—all come later after students have had time to explore ideas in the context of remixing an established project.


```
who
report item (random) of list Señora Ms. C my cat Hannah Jake

does what
report item (random) of list listened to loves ran away from helped is in a band with

gossip
report join who [ ] does what [ ] who
```

Figure 1: The definitions of who, **does what**, and **gossip**.

Then they try a two-block script.
The behavior is a surprise (fig 2).

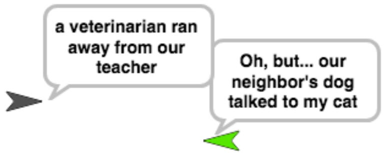

a veterinarian ran away from our teacher
Oh, but... our neighbor's dog talked to my cat

Figure 2: Two sprites gossiping with different vocabulary.

Ideally, each tool/technique is encountered in a more than one context, partly to feed the diversity of student interests—language, art, mathematics, building games or quizzes.... Encountering the same tool/technique in multiple contexts also shows that it is not single-purpose; it lets students abstract out the utility of the tool precisely so that they *can* extend it to their own purposes. For example, in this first stage of the **gossip** project, the use of lists is basic: at this point, a list is just a repository for content selected at random; no indexing, no construction of the list, no mapping over the list. But the idea grows to handle more complex language and to patch together parts of words. Students create a block that appends *s* to the end of a noun, then improve it so that it does a better job of making plurals than just adding *s.* (Optionally, students use the same techniques to conjugate a verb in Spanish or another language of their choice.) The higher order function **map** lets them apply their pluralizer to a test-list of nouns. Lists keep appearing in different ways—lists of coordinate pairs, lists of embedded lists, lists of runnable blocks—and the powerful tools that process lists, like **map**ping a function over a list, or applying a predicate to all items of a list, and **keep**ing only those items that fit—appear in semantically clear, syntactically simple contexts.

Students encounter recursive processes early before studying the *structure* of recursion in any formal way. They may have seen the back-and-forth caused by introducing **broadcast** to the second sprite. They definitely replace the second **who** in **gossip** with **more complicated who** (fig. 3) which introduces a conditional, and then they replace one of the first two **who**s inside **more complicated who** with **more complicated who**, try **gossip** a few more times, and describe the surprising result.

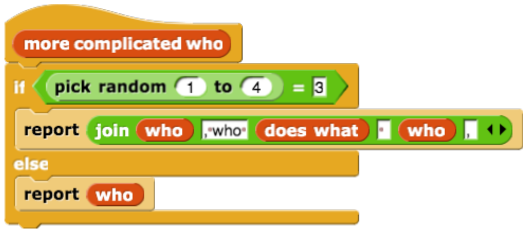

```
more complicated who
if pick random (1) to (4) = 3
    report join who [, who ] does what [ ] who [,]
else
    report who
```

Figure 3: Introducing a condition and an informal opportunity for recursion

Preserve all required details, but organize around big ideas.

Curriculum teaches more than content. When a constraint (e.g., a framework requirement) requires what feels like a loose factoid or a distraction from our goals (e.g., broader participation, personal power, important and beautiful ideas) we must find a way to meet the constraint in an intellectually or socially worthy context. Ignoring frameworks is no option; a curriculum that schools can't accept is ineffective; but weaving each standard into a consistent social or intellectual storyline gives a curriculum integrity.

The same principle guides the choice of programming language. If students are to experience the power and intellectual beauty of programming, then logic, not syntactic detail, must be the focus. In many text-based languages, a program can be completely logical in structure but fail to work because of a misplaced semicolon. Though professionals may have to develop skill at searching for syntax errors, we believe that is not where *any* beginners' attention should be, and is especially antithetical to broadening participation. Choosing Snap*!* as our language for instruction and work focuses students' coding and debugging time on the logic and strategies of programming, not syntax. Snap*!* is essentially Scheme [19] disguised as Scratch [20]. On the one hand, it is a sophisticated programming language with advanced logic and power—recursion, higher-order functions, complex data structures (including lists that can contain numbers, words, other lists, and even blocks of code), object- oriented programming, and lambda—and has been applied even in commercial settings. On the other hand, its visual, blocks-based interface nearly eliminates syntactic fussiness; its visual metaphors make powerful ideas accessible even to beginners. And research supports such a choice: students in blocks-based classes outperform students in otherwise comparable text-based classes and express greater interest in future computing courses [21, 22].

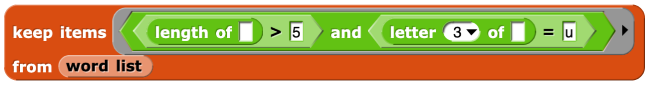
Learning by doing. To a first approximation, CS is a body of knowledge and ways of thinking that help people create hardware and algorithms for solving problems like programming a machine to sort mountains of data or understand natural language or learn from "experience." CS also builds languages that let people program to create things, tools for productivity, health, games, science, art.... Programing is *inherently* creative, and learning to program is, by nature, hands on. That experience of making things, therefore, should be a big part of our students' experience.

Designing learning around projects doesn't mean that the objective is the *project.* If that were the goal, Lego-like instructions, explicit steps to get a lovely result, would suffice. Instead, we want students to learn how to do their own projects—ones we have not thought of—to experience "I can create," "I can solve problems," "I can program." That means they need to learn tools and techniques that let *them* improvise. To us, project-based means learning important tools and techniques not by doing exercises on them but by building things that *need* them. That helps us *define* "important techniques": ones that are re-usable. Some things *we* think are extremely cool can't be taught because they can't be re-used within the limited time we have. Showing many wonderful things, but giving students no time to assimilate and *own* them seems fruitless to us.

Students see that their own work can always be extended and improved. *Some* projects appear in stages. You can do this part now, with what you know. Later, when you know more, you can add to it. A more general message is that work is never "done, but bad"; work that is not yet as one likes is simply not yet done. And there's *no* perfect state; things can *always* be revised.

Project orientation does not mean there are never etudes along the way. Though we may generally strive for introducing new techniques or tools in the context of *needing* them for some purpose we are already engaged in, sometimes *learning* the new ideas needs to be uncluttered with the camouflage of context and other techniques or tools. In those cases, we try to present puzzles for kids to solve using the new tool. This one, inspired by Vaniček [23], teaches predicates by giving students code and a result to try to understand (fig 4), and then new designs to try to make by using and modifying other predicates (fig 5).

BJC uses the same strategy to introduce the higher order function **keep**, as students comb through a massive dictionary to create a list of words that match clues for a word puzzle.



```
keep items (length of [ ] > 5 and letter (3) of [ ] = u)
from (word list)
```

Puzzles need not be artificially crafted. They can be part of a project, but abstracted from the whole so that the focus is on the new element. Ultimately, to teach students how to think on their own, the projects *we* present are incomplete, going only as far as the raw functionality of the new elements we are teaching so that students get the sense of what's possible and the power they have to create it, but leaving room for them to add their own features creatively. We assure that they have all the essential tools (blocks), give them some example, and then let them go.

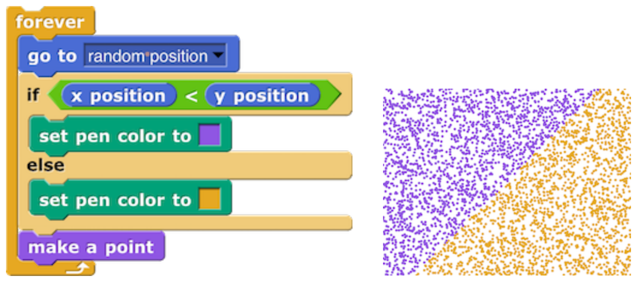
Beauty and joy: helping students recognize, respect, grow, and enjoy their own logic and creativity in CS. The esthetic of programming is not just in its products; programs, themselves, can have intellectual beauty. The classic text, *Structure and Interpretation of Computer Programs* [24] says it this way:

> "To appreciate programming as an intellectual activity in its own right [...] *you must read and write computer programs—many of them.* It doesn't matter much what the programs are about.... What does matter is how well they perform and how smoothly they fit with other programs in the creation of still greater programs. The programmer must seek both perfection of part and adequacy of collection." (Emphasis ours.)

The CSP framework emphasizes creativity. To serve this goal and build a sense of *competence* to pursue CS, our students experience "doing CS" through more programming than the CB requires. To help them feel "CS-smart," students need to see their code not just as a means to an end with an effect they like but as "poetry," code with structure, elegance and power. Programs represent students' thinking, so the code, itself, should feel beautiful to students.

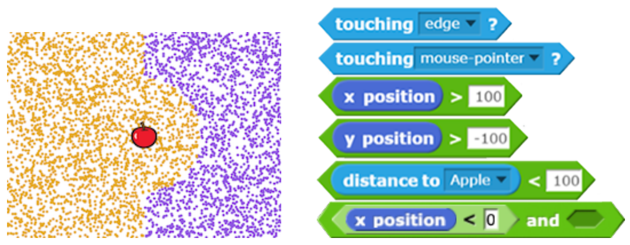
Achieving and appreciating beautiful code isn't about being clever. Cleverness is a local phenomenon that appears differently in each student. Our goal is to help students express the beauty in their insights, without having to worry about computerish details. We want to let programming help them refine and add precision both to their insights and to their expression. For example, there are many ways to define a function that takes a list of numbers as input and returns a list containing the squares of those numbers.

Figure 6 shows three methods for a **squares of** block. Two closely mimic how students often describe their thinking; the third uses a style that few students articulate spontaneously. While attracting students to the field, we don't push particular techniques—students will, in time, develop their own style and esthetic—but to develop a personal style, or even to notice that there *can* be a poetry in programming, they need at least some exposure to varied approaches to a single problem.



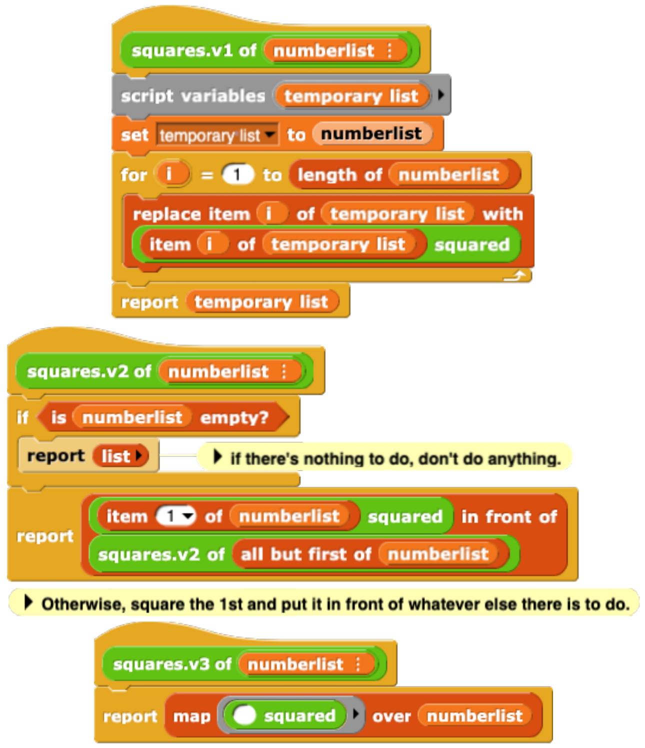
```
forever
  go to random position
  if x position < y position
    set pen color to
  else
    set pen color to
  make a point
```

Figure 4: A script and its result.



```
touching edge ?
touching mouse-pointer ?
x position > 100
y position > -100
distance to Apple < 100
x position < 0 and
```

Figure 5: A design and some predicates that might help.



```
squares.v1 of numberlist
  script variables temporary list
  set temporary list to numberlist
  for i = 1 to length of numberlist
    replace item i of temporary list with
        item i of temporary list squared
  report temporary list

squares.v2 of numberlist
  if is numberlist empty?
    report list          ▸ If there's nothing to do, don't do anything.
  report item 1 of numberlist squared in front of
         squares.v2 of all but first of numberlist
▸ Otherwise, square the 1st and put it in front of whatever else there is to do.

squares.v3 of numberlist
  report map ( squared) over numberlist
```

Figure 6: Iterative, recursive, and higher-order function approaches to processing a list.

Design for diversity by offering diversity and personalization. A curriculum cannot be completely bias-free. The tone, contexts, even the content one chooses to highlight, all evince a point of view that is not universal. Some curricula attempt inclusivity by using culture-stereotypical names, contexts or activities—pop-social images of "boy" and "girl" or "white" and "black" culture, or culture-associated names. We think this is *not* a good way to be inclusive, reduce bias, or broaden appeal. While such techniques explicitly signal acknowledgment of diverse groups and sensibilities, they do so by invoking stereotypes: *this* project is here for girls; *this* name (ostentatiously) reflects *your* ethnicity.

So? Avoid imagery, style, or content that implicitly targets a group pro or con. But there are also more subtle biases. We recall Barbara Janson [25] describing her visit with a teacher group who *knew* that the math text they had was rote, non-thinking junk, but saw no help from the new materials they were shown because of the materials' strong (liberal) cultural bias: boys doing dishes, "rain forest math," and so on. Though this bias probably *wasn't* accidental (surely it reflected a conscious concern of the university R&D teams that built the curricula), its consequences included one that the developers surely didn't intend: It left some groups unserved. But even a naïve accident can create bias. "Mary went to the clothing store with $200 and wanted to...." And of course cultural/political neutrality is also a bias—a choice *not* to take certain stands, resulting in a bias toward the status quo.

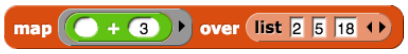
We take the deliberately *non*-neutral stand of aiming to attract and serve the most underrepresented groups in CS. We actively *recruit* to get kids in, then let word of mouth draw in more. The BJC principle (not fully realized) is to appeal to a breadth of personal, social and intellectual interests that cross race, gender, and economic lines (mathematics, language, games, art, science...) and leave room for students to put their own stamp on their work.

3 Content principles: Programming

Helping students see that they can "do" CS and enjoy it is one part of our strategy. We also invite students beyond the entry points, experiencing recursion and higher-order functions because of the powerful and beautiful way they exemplify abstraction, a key idea in CS and in the AP CSP framework. Such "advanced topics" are often seen by others as too difficult for students, but Snap*!*'s explicit visual representations make them more accessible. Seeing the complexity of a fractal tree (or being taken by surprise by an astonishingly long **gossip**) and seeing the simplicity of the recursive procedure that produces it is an "aha!" that you don't get from a Google search, a video, making a poster in Photoshop, or even writing programs with no control structure more powerful than a loop. We've mentioned some of these "extra content" choices before, so this section will be light, but we want to clarify why we think these *matter* even in an introductory CS experience.

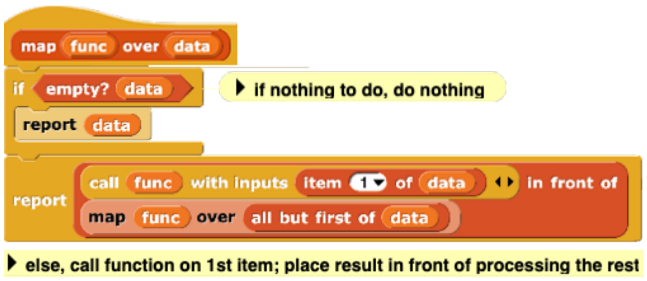
The power of recursion. One important face of abstraction occurs when you notice that in some problems, parts mirror the same steps as the whole. For example, in this design, seen very early in BJC without complex trappings, whatever process draws the red triangle and its blue children could conceivably let blue triangles draw green children. After multiple early experiences that clarify the *idea*, we teach recursion formally late in the course, analyzing what enables it (the recursive call) and what stops it (base case) using both graphical (e.g., fractal trees) and list or number processing (with memoization, if needed). Modeling the reasoning with a recursive program helps develop such thinking.

Function as data. Another important face of abstraction occurs when methods become objects. In mathematics, one engine for progress is that the methods of one generation become the objects of study for the next. Like Scheme, Snap*!* implements "first class functions." Students first see functions as inputs with semantically clear examples. Using **map** to test **plural** is one; here is another.



```
map (( ) + (3)) over (list 2 5 18)
```

When students study recursion formally, they learn to *build* **map**.



```
map (func) over (data)
if (empty? (data))          ▸ if nothing to do, do nothing
    report (data)
report (call (func) with inputs (item (1) of (data)) in front of (map (func) over (all but first of (data))))
▸ else, call function on 1st item; place result in front of processing the rest
```

The new idea here is that one of the inputs is a *reporter* (a function), and its treated just like any other piece of data.

Mathematics as a tool. CSP is not a mathematics course, but mathematical thinking is a valuable tool in programming. Basic algebra can help make algorithms more efficient.

For example, in compounding interest, the double recursion algorithm

```
balance ← balance + rate • balance
```

is inefficient. Elementary factoring increases efficiency. Similarly, the use of coordinates and **mod** are often needed when working with time or screen graphics. The perceived difficulty of some of these ideas, we think, is partly an artifact of the languages and metaphors chosen and the contexts in which they are encountered.

4 Content principles: Social implications

A major element of BJC and of the AP CSP framework focuses on the social implications of computing. To build students' sense of agency, we balance a fundamental optimism about the future of computer technology with a critical stance toward each specific use, focusing not just on the facts of the social implications, but also on establishing particular perspectives, which include:

Social implications differ for different groups of people. To talk broadly of technology's "benefits and harms" papers over the question of *who* benefits, *who* is harmed. We encourage students to read critically, to ask themselves who wrote a text, and who benefits if the text persuades them to a particular point of view.

Everyone can participate in developing technology policy. Even apart from being the programmers, students learn that they can control the development of new technologies simply by being aware voters and consumers. BJC students read, discuss, and write about issues of computing in society using the *Blown to Bits* book [26] and regularly engage in "Computing in the News" activities, in which students present a recent article about technology, and the class asks clarifying questions and discusses implications.

Teaching social implications is not "teaching ethics." Much of the teaching about social topics, especially in K–12 but even at the university level, takes the form of shalt-nots: Don't download illegal copies of movies and music. Don't cyberbully. We prefer to respect students as thoughtful social agents and to inform them about the implications of technology for different stakeholders rather than to lay down rules.

For example, students work in groups to try to develop a way for authors to make a living from their work while still allowing the unlimited downloading rights that many students want, and while respecting the original public interest purpose of copyright. They also consider how making movies, with a cast and crew of thousands, needs a different financial model from making music, which may be created by a single person.

5 Conclusion

Given the growth in interest in introducing and engaging K–12 students in computer science, particularly among high school students, many teams have designed materials to serve this goal, resulting in diverse curricular options for teachers and schools to learn about and choose among, all aligned with the new AP CSP framework and endorsed by the College Board, but they vary in content and approach, and critically, in the guiding principles that undergird their design. We share the guiding principles underlying one of these AP CSP endorsed curricula to illuminate the rationale and craft behind it, and to spark reflection and discussion about critical principles for AP CSP curricula. We encourage all those considering AP CSP curricula to investigate the guiding principles of the different curricular options, and to share these principles with teachers at PD to inform teacher implementation. We also share positive results. To date, 800 teachers have participated in BJC professional development, with over 150 teachers trained in NYC. In addition, over 4400 BJC students nationally sat for the 2018 AP CSP exam. BJC4NYC findings from '16–17 and '17–18 show that diversity increased, teachers made statistically significant pre/post gains in content knowledge, self-efficacy, self-rated programming ability, preparation/effectiveness, and knowledge/fluency, and students showed significant pre/post gains on a content assessment, with small to medium effect sizes [27, 28, 29]. Students' 2017 AP CSP passing rates also show encouraging results [30].

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grants 1441075 and 1837280. Opinions, findings, and conclusions or recommendations expressed here are those of the author(s) and do not necessarily reflect views of the Foundation.

REFERENCES

[1] Jan Cuny. 2012. Transforming high school computing. ACM Inroads. 3, 2 (June 2012), 32-36.

[2] Owen Astrachan, Jan Cuny, Chris Stephenson, & Cameron Wilson. 2011. The CS10K Project: Mobilizing the community to transform high school computing. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11). ACM Press, New York, NY, 85-86.

[3] Owen Astrachan & Amy Briggs. 2012. The CS Principles project. ACM Inroads. 3, 2 (June 2012), 38-42.

[4] Lawrence Snyder, Tiffany Barnes, Dan Garcia, Jody Paul, & Beth Simon, 2012. The First Five Computer Science Principles Pilots: Summary and comparisons. ACM Inroads. 3, 2 (June 2012), 54-71.

[5] Richard Kick, 2012. Computer Science Principles at Newbury Park High School, ACM Inroads, 3, 2 (June 2012), 75-77.

[6] College Board. 2017. AP Computer Science Principles: Including the Curriculum Framework (Fall 2017). Retrieved August 30, 2018 from https://apcentral.collegeboard.org/pdf/ap-computer-science-principles-course-and-exam-description.pdf

[7] Marie desJardins. 2015. Creating AP CS Principles: Let many flowers bloom. ACM Inroads 6, 4 (Dec. 2015), 60-66. DOI: 10.1145/2835852

[8] Brian Harvey. 2012. The Beauty and Joy of Computing: Computer science for everyone. In Constructionism: Theory, Practice, and Impact Conference Proceedings. Athens, Greece, 33-39.

[9] College Board. 2018. Computer Science Principles: Course details. Retrieved Aug 30, 2018, https://advancesinap.collegeboard.org/stem/computerscience-principles/course-details

[10] George Veletsianos, Bradley Beth, Calvin Lin, & Gregory Russell. 2016. Design Principles for *Thriving in Our Digital World*: A high school computer science course. *J. Ed. Comp. Research.* 54(4), 443-461.

[11] Albert A. Cuoco, E. Paul Goldenberg & June Mark. 2010. Organizing a curriculum around mathematical habits of mind. Mathematics Teacher. 103(9) pp. 682-688.

[12] Goldenberg, E. P., Mark, J., & Cuoco, A. (2010). Contemporary curriculum issues: An algebraic-habits-of-mind perspective on elementary school. Teaching Children Mathematics, 16(9), 548–556.

[13] Mark, J., Cuoco, A., Goldenberg, P. & Sword, S. (2010). Contemporary curriculum issues: Developing mathematical habits of mind in the middle grades. Mathematics Teaching in the Middle School. 15(9) pp. 505-509.

[14] Al Cuoco & E. Paul Goldenberg. 2011 Beyond Topics: Benchmarks for Judging a High School Curriculum. *Mathematics Teacher.* 104(7), 486-488.

[15] EDC. CME Project: Curriculum. Retrieved from http://cmeproject.edc.org/cme-project.

[16] E. Paul Goldenberg, June Mark, Jane M. Kang, Mary Fries, Cynthia J. Carter, and Tracy Cordner. 2015. Making Sense of Algebra (1st. ed.). Extended Investigations for Students, Ch. 4. Heinemann, Portsmouth, NH.

[17] Bowen Kerins, Benjamin Sinwell, Darryl Yong, Al Cuoco, and Glenn Stevens. 2015. Mathematics for Teaching: A Problem Based Approach, Vol. 2: Applications of Algebra and Geometry to the Work of Teaching (1st. ed.). American Mathematical Society.

[18] Irene Lee, Fred Martin, Jill Denner, Bob Coulter, Walter Allan, Jeri Erickson, Joyce Malyn-Smith, & Linda Werner. 2011. Computational thinking for youth in practice. ACM Inroads 2, 1 (Mar. 2011), 32-37. DOI: https://doi.org/10.1145/1929887.1929902

[19] Brian Harvey & Matthew Wright. 1999. Simply Scheme: Introducing Computer Science (2nd. Ed.). The MIT Press, Cambridge, MA.

[20] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, & Yasmin Kafai. 2009. Scratch: Programming for All. Commun. ACM 52, 11 (Nov. 2009), 60-67. DOI: https://doi.org/10.1145/1592761.1592779

[21] David Weintrop, Heather Killen, & Baker Franke. 2018. ICLS 2018 Proceedings. 328–335.

[22] David Weintrop & Uri Wilensky. 2017. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Trans. on Comput. Educ.* 18, 1, Article 3 (October 2017), 25 pages.

[23] Jiří Vaníček. 2018. Concept-building Oriented Programming Education. In Constructionism 2018: Constructionism, Computational Thinking and Educational Innovation Conference Proceedings. Vilnius, 495-503.

[24] Harold Abelson, Gerald Jay Sussman, & Julie Sussman. 1996. *Structure and interpretation of computer programs* (2nd. Ed.). MIT Press, Cambridge, MA.

[25] Barbara Janson, personal communication, NSF Gateways Conference, 1997.

[26] Harold Abelson, Ken Ledeen, & Harry R. Lewis. 2008. *Blown to Bits: Your Life, Liberty, and Happiness After the Digital Explosion.* Addison-Wesley, Upper Saddle River, NJ.

[27] Mary Fries. 2019. Beauty and Joy of Computing AP Computer Science Principles for All. Presented at the 3rd. Computer Science Teacher Association New England Regional Conference, (CSTA NERC '19). https://cstanewenglandregionalconfe2019.sched.com/event/Vrso/beauty-and-joy-of-computing-ap-computer-science-principles-for-all

[28] Education Development Center. 2017. Bringing a Rigorous Computer Science Principles Course to the Largest School System in the United States, 2017 annual project report submitted to the National Science Foundation.

[29] Education Development Center. 2018. Bringing a Rigorous Computer Science Principles Course to the Largest School System in the United States, 2018 annual project report submitted to the National Science Foundation.

[30] June Mark and Kelsey Klein. 2019. Beauty and Joy of Computing: 2016–17 Findings from an AP CS Principles course. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Minneapolis, Minnesota USA, February-March 2019 (SIGCSE'19), 7 pages. DOI: 10.1145/3287324.3287375