A New Pedagogy to Address the Unacknowledged Failure of American Secondary CS Education

By Scott R. Portnoff, *Downtown Magnets High School*

THE NEW STATUS QUO

As the new decade begins, by most accounts, it would appear that CS education in American public high schools is doing well. Visit classrooms and you'll find students working with robotic sensors, writing games and animations in Scratch, interfacing with Arduino microcontrollers, constructing websites, and building apps with MIT App Inventor. Groups like CSTA (Computer Science Teachers Association) and CSforAll, whose missions are to support and expand high-quality, rigorous and demographically equitable secondary CS education, proudly announce that more students, particularly young women, have taken AP Computer Science[1] than ever before, ascribing the uptick to the new AP CS Principles course, e.g., "Recent Good News on Participation and Opportunities for Young Women Studying Computer Science." [24]

Look underneath the celebratory and self-congratulatory remarks, however, and you'll find that, although contemporary secondary education is quite good at generating initial student interest, it has had much less success at sustaining that engagement beyond a few weeks or months, and has frankly been ineffectual in terms of (a) measurable learning for the majority of students; (b) boosting the number of students who take a second CS course, either in high school or college; and (c) adequately preparing students for CS college study.

In stark contrast to the positive picture painted on the CSTA blog cited above, Girls Who Code CEO Reshma Saujani noted that "two-thirds of states with computer science programs report seeing no increase in participation from girls." [28] In an opinion piece entitled 'Don't Rely on Cute Apps and Games to Teach Coding,' SummerTech founder and CEO Steven Fink has described the situation as follows [5].

> Today, K-12 coding education offerings often include edutainment: some brand's robotic-design, game-design and app-design program, packaged as engaging, entertaining experiences for students…

[1] Most such announcements conflate the two AP Computer Science courses—A (Java) and Principles—as if they were of equal value, neglecting to mention that the former is a rigorous college-level introductory programming curriculum while the latter, despite the AP branding, is a simplified survey course.

> …In many schools, few teachers can teach coding. So, having no idea where to begin a curriculum, they turn to so-called experts: companies that sell them products to meet new coding or STEM requirements. To be clear, this isn't a case of lazy teaching or administration. It's a knowledge deficiency that can lead schools—and in turn, students—in the wrong direction. Unfortunately, dragging and dropping is not coding, and don't even think about taking on robotics until you've learned to code…
>
> …In 2002, I founded a summer coding camp, SummerTech, located on the campus of Purchase College, SUNY. For the first couple of years, I confess, we used edutainment programs. The kids had plenty of fun. For most of them, however, that wasn't their real motivation; learning was, and we didn't feel we were making much progress. They wanted to become real-life coders and they weren't headed in that direction.

Consider the tongue-in-cheek internship posting in Figure 1. That a company would be willing to take on a student without portfolio evidence of even basic programming proficiency is farfetched. The ad does, however, accurately reflect the outcomes typical of the most common high school course offerings: Exploring Computer Science (ECS) and AP Computer Science Principles (AP CSP), both of whose foundational premise is that programming is just one of many topics in the field of CS. Up until a decade ago, introductory high school computer science classes were synonymous with programming instruction, period. No longer. The new courses comprise a range of CS topics—and in many instances math or sociology topics (e.g., Implications of Computing Innovations, The Internet, Data Analysis), some with only tangential connections to the core CS topics studied in an undergraduate major—none of them taught to any degree of depth, which is why I term them *survey* courses.

PAID CS INTERNSHIP
Requires exposure to CS topics (one of which may be coding). Proficiency in text-based programming languages optional. Drag-n-drop language (e.g., Scratch) experience a plus.

Figure 1

How did this new status quo in secondary CS education establish itself? As it happens, the public-school system in which I teach—Los Angeles Unified School District (LAUSD)—is ground zero for this educational model. In 2004, a group of educators calling themselves the Computer Science Equity Alliance (CSEA) worked to augment the absolute enrollments of LAUSD students in the AP Computer Science A (Java) course over a 2-year period—impressively, it seemed, nearly tripling overall enrollments from 225 to 611, and increasing the participation numbers of female, Hispanic and African-American students four-fold, five-fold and two-fold, respectively. The students in these new classes, however, overwhelmingly failed the end-of-year AP exam. Had CSEA educators been familiar with both CS education and the literature, they would have recognized that they had run up against the **Novice Programmer Failure problem** (NPFP), a decades-old phenomenon of high student failure and dropout in introductory CS1 college courses, but greatly exacerbated at the secondary level. Instead, these educators made several inaccurate assumptions, misdiagnosed the problem, and devised a "solution" that did nothing to address the poor learning outcomes [9].

> [U]nderlying these increased enrollment numbers was a tension that the programming-centric focus of the AP course and the advanced, college-level status was not an accessible point of entry for most students. Three years into this work, we recognized the need for a foundational high school course introducing students to the major concepts of the field of computer science, and a course that had the potential to engage the diverse populations of Los Angeles schools.

CSEA was correct in noting that the college-level APCS-A course was too advanced for most high school students and that an introductory prerequisite was needed.

However, the contention that programming was what made the course inaccessible—as opposed to, say, an uninspiring or pedagogically ineffective version of that particular curriculum, or a poorly prepared instructor—was baseless speculation. Moreover, the very assertion is as absurd as arguing that Spanish language instruction denies students access to the subject of Spanish, or that math teachers should abandon algebra and replace it with general math. In retrospect, it's hard to understand how many school districts fell in line behind this nonsensical anti-programming ideology, because the universal post-secondary consensus is that programming has been, and continues to be, the core skill underlying the entire discipline [2].

> Many introductory programming courses have programmability as a core activity and learning goal, and for good reasons since programmability is the defining characteristic of the (digital) computer. This is also echoed in the ACM/IEEE curriculum recommendations the programming-first model is likely to remain dominant for the foreseeable future.

Section 2.5 Computer Science Can Engage All Students of the revised 2011 version of *CSTA K-12 Computer Science Standards* [25] similarly echoed the centrality of programming in CS education.

> Pedagogically, computer programming has the same relation to studying computer science as playing an instrument does to studying music or painting does to studying art. In each case, even a small amount of hands-on experience adds

> immensely to life-long appreciation and understanding, even if the student does not continue programming, playing, or painting as an adult. Although becoming an expert programmer, a violinist, or an oil painter demands much time and talent, we still want to expose every student to the joys of being creative. [p. 5]

However, the contention that programming was what made the course inaccessible—as opposed to, say, an uninspiring or pedagogically ineffective version of that particular curriculum, or a poorly prepared instructor—was baseless speculation.

The introductory passage to section 2 explicitly conflated computing and computer science with programming, using the terms interchangeably, and spoke of the capacity of programming for motivating and engaging learners.

> Children of all ages love computing. When given the opportunity, young students enjoy the sense of mastery and magic that programming provides. Older students are drawn to the combination of art, narrative, design, programming, and sheer enjoyment that comes from creating their own virtual worlds. Blending computer science with other interests also provides rich opportunities for learning. Students with an interest in music, for example, can learn about digital music and audio. This field integrates electronics, several kinds of math, music theory, computer programming, and a keen ear for what sounds beautiful, harmonious, or just plain interesting. [p. 2]

In the same publication, however, section 4.2 Strands, articulated a polar opposite position—unprecedented in CS education—that revealed a growing challenge by anti-programming educators influenced by the CSEA group.

> Almost since its inception, computer science has been hampered by the perception that it focuses exclusively on programming. This misconception has been particularly damaging in grades K–12 where it often has led to courses that were exceedingly limited in scope and negatively perceived by students. It also fed into other unfortunate perceptions of computer science as a solitary pursuit, disconnected from the rest of the world and of little relevance to the interests and concerns of students. [p. 9]

CSEA's proposed remedy for students failing to learn how to program—changing the focus from programming to an emphasis on "the major concepts" in CS—was a non sequitur: exactly how does *less* programming instruction help? Moreover, no rationale was given to justify why such a course might be an improvement, how it might align with future study or how it might provide some other tangible benefit. Finally, the crucial issue of how these so-called "major" introductory concepts might be determined was not even addressed, leaving the task wide open for deciding the matter arbitrarily.

Unfortunately, what this newly hatched agenda successfully accomplished over the next decade was the dissemination of a now widely accepted myth: the demotion of programming from its central place in the discipline. This fictitious ideology enabled two developments: (a) it conveniently let secondary instructors off the hook from their responsibility to, at the very least, attempt to remedy their overwhelming failure to teach students how to program; and (b) it provided these instructors—and more importantly their school districts—cover by replacing that instruction with so-called "CS" courses comprised of simplistic non-programming material, giving the *appearance* that progress was being made in the attainment of equity in secondary CS education. Note that such developments were only possible because of widespread ignorance about the fundamentals of the field among public K-12 stakeholders, including most secondary instructors of CS themselves.

Like a conspiracy theory that persists seemingly with a life of its own, the distorted view that programming is just one of many coequal CS topics contains just enough truth to sound reasonable. While it is true that programming is one of many topics taught in an undergraduate CS curriculum—along with algorithms and data structures, operating systems, automata theory, artificial intelligence and the like—what this account strategically omits is that programming provides the conceptual foundation and skill set needed to study virtually any other CS topic in any degree of depth, hence its crucial role in the introductory course. The only kind of "CS" course that students who lack basic proficiency in programming could pass would have to be severely simplified and stripped of fundamental content.

Last, why the phrase "engage the diverse populations of Los Angeles schools" was invoked is puzzling, as if an intent to rectify educational inequities (a) is enough to substitute for the lack of any sound rationale for the proposed survey course and (b) suffices to discredit the objectives of the traditional programming course as incorrigibly elitist. My reading, however, is that the phrase is at best patronizing and condescending, and at worst bigoted, as if economically disadvantaged minority and first-generation students in urban settings need special handling or are somehow less acculturated, assimilated to, or conversant in majority American culture—or participate in it less—than their suburban non-minority counterparts. That is certainly not what I've found in my fourteen years of teaching CS in this setting; to the contrary, my students are indistinguishable from other Americans in that regard—like

my first-generation grandparents. If the particular version of the APCS-A Java course taught in the CSEA experiment had been an interest-killer, all students would have found this to be a problem, regardless of their ethnicity.

To underscore the point, since 2013, I have been teaching high school freshmen from this same demographic[2] an original UCOP-approved contextualized pre-APCS-A Computer Programming course that uses models and simulations for exploring real-world cross-curricular problems from subjects across the academic spectrum [23]. The course uses Processing, a simplified dialect of Java with a built-in graphics layer, whose IDE allows students to do graphics programming on Day 1. In ongoing self-assessments and community circles, every student has reported finding the projects in the course engaging, including even those with little or no interest in programming itself. Interest/engagement, however, is only half the battle in the quest to retain students. Teachers need to also facilitate in their students a genuine sense of self-efficacy vis-à-vis conceptual understanding and ever-increasing proficiency in programming skills, through the crafting of instructional strategies that can counter the effects of the NPFP.

Two years following the CSEA project, then, CSEA's flawed analysis saw its fruition in the UCLA-based Exploring Computer Science (ECS), a survey course that several of the group's members went on to write and then propagate within LAUSD. Today, the course is taught widely throughout the United States—including the six other largest underperforming urban school districts serving large minority populations, like New York City and Chicago—with estimated enrollments in the tens of thousands. Development by a much larger national group of educators of what was to become APCSP, a more advanced survey course, but one that shared the same faulty foundational philosophy as ECS, also started around this same time period, but was launched in full nearly a decade later.

CONSEQUENCES OF THE SURVEY COURSES

What's the harm in teaching a survey course? To contend that superficial exposure to simplified non-programming topics—even were they to include limited coverage of short programs—will later enable students to surmount the NPFP and magically acquire competence in programming is like arguing that watching French films, eating Parisian cuisine, and learning a few tourist phrases would adequately prepare students for a second-year French language course. Rather, these courses simply postpone the difficulties that the vast majority of these students will encounter should they later take a subsequent CS class that will—without exception—require that they acquire proficiency in fundamental programming concepts and skills.

Rather, these [survey] courses simply postpone the difficulties that the vast majority of these students will encounter should they later take a subsequent CS class that will—without exception—require that they acquire proficiency in fundamental programming concepts and skills.

Furthermore, not only is programming central to the study of CS, it is the gatekeeper skill for economic opportunity, both for pursuing a CS college major and for high-paying computing jobs. Sixty-five percent of urban public-school students in the U.S. attend high-poverty (40%) or mid-high poverty (25%) schools [11]. According to Girls Who Code: "Computing jobs are among the fastest-growing in the U.S. economy. These jobs pay more than double the average U.S. salary. And in the coming years, they will be key drivers of national economic growth and mobility." [8]

This is not just a talking point—rigorous pre-college programming instruction has a profound positive impact in terms of access to these opportunities. Longitudinal studies have confirmed that two programs provide students sufficient preparation and motivation for pursuing a CS college major:

- young women who are alumni of Girls Who Code complete majors in CS-related fields at 15-16 times the national rate [7];[3] and
- students who score 2 or higher on the APCS-A (Java) exam (the course which was the subject of the CSEA experiment) choose a CS college major at rates of 16-27% [17,19].

The negative corollary is also true. Not providing programming instruction to students attending mid- to high-poverty schools has detrimental consequences, denying all of them—including those inclined to study CS—access to both the instructional preparation that would greatly facilitate their success as college CS majors, and the ensuing economic opportunities that a computing career can provide.

The UCLA-based ECS group long ago decided *not* to conduct similar longitudinal studies to validate its course, offering the following rationale [9].

---

[2] Approximate demographic composition of classes: 70% male, 30% female. 50% East-Asian/Filipino, 30% Hispanic, 10% African American, 10% other (Middle Eastern, Indi-an/Pakistani/Bangladeshi, Russian, white).

[3] Unpublished studies: Girls Who Code tracked its "alumni longitudinally into college at the student level using the National Student Clearinghouse. For this research, [Girls Who Code] compared college outcomes among our alumni population to the national trends using publicly available data [from the] National Center for Education Statistics, degrees awarded in 2016-17." (Girls Who Code personal email to author).

> Disrupting the traditional secondary curriculum with the introduction of computer science education can entrap reformers into proving that … students are more likely to choose to enroll in computer science classes in the future. These are incredibly high, if not impossible standards for any one high school class to obtain, and the high possibility of confounding variables makes it difficult to ultimately make any conclusions. Additionally, these types of measures dilute the importance of computer science as essential knowledge for 21st century students in its own right… As a community of computer science educators, we must allow access to learning computer science education in high school serve as a fundamental right to learn, rather than a stepping-stone for a future purpose.

The assertion that it is impossible to "prove" future outcomes did not dissuade either Girls Who Code or the College Board from demonstrating robust positive correlations, as noted above. And although misleadingly touted as a research-based, high school intro-level computer science curriculum," the ECS web page "The Research Behind ECS" lists only Margolis' 2008 book about inequitable access to the APCS-A course in LAUSD schools that ECS was supposed to address. Over a decade later, there is still nothing to demonstrate how ECS might be mitigating that problem.

As outlined earlier, the predicament that propelled the entire ECS enterprise was the 2004-06 CSEA effort that, although increasing enrollments of young women and traditionally under-represented minority students in the APCS-A course, resulted in massive failures in actual student learning per College Board standardized assessments. The ECS group has replicated the first half of this experiment by achieving more equitable enrollments. But it has jettisoned any effort to verify that students have even learned its watered-down content, *scrapping accountability entirely*. The obvious question is how can this in any way be considered an improvement? Another point: although the group has maintained that its curriculum contains "essential" CS topics, the year I attempted to teach the course and pointed out deficiencies in lessons, or when others complained that units were proving too difficult for students, the group simply dropped these materials and arbitrarily replaced them, oftentimes with lessons containing only tenuous links to CS [23].

Several years into the project, the ECS group also discarded its original pipeline goal in favor of a new objective [16].

> Our mission goes beyond the "pipeline" issue of who ends up majoring in CS in college. Rather, our mission is to democratize CS learning and assure that all students have access to CS knowledge.

What "democratize" might actually mean in the absence of increased access to future educational or economic opportunities in the field, however, is anyone's guess.

The group also attests that its curriculum is at a college preparatory level. However, based on my attempts to teach the course as well as anecdotes collected from my colleagues in the district, there is little doubt (a) that were assessments to be done, they would reveal that large groups of students are not even learning the simplistic content of the course; (b) that longitudinal studies would show that virtually no ECS students take a subsequent CS course; and (c) that students who do learn the content learn nothing that will help them succeed in subsequent courses should they enroll, making the year spent a meagre use of limited instructional time.[4]

Regarding the newer offering of APCSP, there is actually data to gauge its effectiveness. There is little reason to celebrate, however, if district-wide 2018 and 2019 AP exam results for LAUSD are typical of other large urban school districts. The failure rate for African American students is about 70% and for Hispanics about 64-69%, as compared to 16-30% for white and Asian students, vs. 30% national average. Among just those students who passed the AP exam, the mean passing score for African American and Hispanic students for these two academic years has also been somewhat lower than for white and Asian students. Note that letter grades for the course appearing on student transcripts have no relationship to the College Board exam score (assessable learning). Although all ethnic groups showed grade inflation, the gap between letter grades and AP exam pass scores was particularly wide for African American and Hispanic students, about 84% and 85-89% of whom, respectively, received letter grades of C or higher for the 2017-2018 and 2018-2019 courses. (Tables 1 and 2).

[4] For an extensive critique of ECS' deeply flawed content, including the relevance of the course's topics to CS and its preparatory value should students wish to take additional coursework, see [23], 19-34 and its Appendix A, 198-211.

Table 1: LAUSD 2018 AP CSP Exams by Ethnicity.

Ethnicity	Exams Taken	Exams Passed	Exam Pass Rate	% A, B or C Letter Grade	Mean Score	Mean Pass Score
African American	70	21	30.0%	84.3%	2.13	3.57
Asian	117	81	69.2%	96.6%	3.21	3.81
Filipino	52	32	61.5%	100.0%	2.75	3.47
Hispanic	649	203	31.3%	84.7%	2.07	3.31
White	138	107	77.5%	95.7%	3.31	3.78
Other	10	7	70.0%	—	3.20	3.43
Total LAUSD	1036	451	43.5%	—	2.41	3.53
National	72187	51383	71.2%	—	3.11	3.69

Table 2: LAUSD 2019 AP CSP Exams by Ethnicity.

Ethnicity	Exams Taken	Exams Passed	Exam Pass Rate	% A, B or C Letter Grade	Mean Score	Mean Pass Score
African American	80	25	31.3%	83.8%	2.10	3.36
Asian	180	129	71.7%	97.8%	3.24	3.82
Filipino	68	47	69.1%	92.6%	2.93	3.47
Hispanic	946	341	36.0%	89.3%	2.11	3.23
White	159	134	84.3%	97.5%	3.36	3.66
Other	30	26	86.7%	—	3.30	3.62
Total LAUSD	1463	702	48.0%	—	2.45	3.46
National	95105	69059	71.9%	—	3.11	3.68

It gets worse. In theory, one would think that the students who passed the College Board exam had at minimum mastered content that could prove useful in subsequent coursework. However, a look at the 22 sample multiple-choice exam questions in the course description suggests otherwise. For example, three pseudo-code programming questions (Sample Questions 5, 8 and 14) ask about the movement of a robot in a grid, simple tasks that elementary and middle school students often do and that traditional programming courses would have finished covering by the end of the first week or two.

Tellingly, Sample Question 2 asks students to identify an algorithm that would change all occurrences of "goats" to "sheep" and vice-versa by using the intermediary word "foxes." The answer is:

> First, change all occurrences of "goats" to "foxes;"
>
> Then, change all occurrences of "sheep" to "goats";
>
> Then, change all occurrences of "foxes" to "sheep." [4]

Although technically an algorithm, this is a text-editor task, not even close to one of the "big ideas" in CS that the course claims to be teaching. More likely the question was meant to echo the in-place swapping of the values in two variables using a 3rd variable that first saves off one of the original values. Although the end results for both tasks are comparable, the solutions are unrelated: swapping would utilize a variable/memory-location to save off "goats," whereas the text-editor changes "goats" to "foxes," trashing it without saving and requiring the user to "save" the word "goats" in his/her own brain. The question provides zero insight into even this "little idea" in programming.

The simplistic course content of which these questions are representative inspires little confidence of longitudinal outcomes similar to APCS A or the Girls Who Code program—or even that a significant number of students will enroll in and pass a subsequent CS course. It is not only that the content is not rigorous enough, but that it fails to give students a solid programming foundation.

Just as concerning, most APCSP instructors come to the job with limited, if any, subject-matter competence. University of Alabama CS Professor Jeff Gray was the principal investigator for a multi-year NSF-funded proposal "to develop and evaluate a model for the scalable deployment and sustainable persistence of the new CS Principles course across a statewide network of teachers." [20] He reported the following in [10].

> The assumption of all current APSI (AP Summer Institute) offerings is that the attendees already possess most of the desired content knowledge. This is not true for CS ... Applying the current AP training model for new CS teachers is similar to asking a teacher with no mathematics background to initiate a new AP Calculus course with just one week of training. This situation would seem absurd to most administrators in the mathematics context, but it is the common expectation for promoting new AP CS courses (i.e., APCSP).

"Applying the current AP training model for new CS teachers is similar to asking a teacher with no mathematics background to initiate a new AP Calculus course with just one week of training."

To underscore the point, Code.org's web page for APCSP [3] states that "no prerequisites [are] required for students or for teachers new to computer science!" and that teachers need not have "any prior computer science experience to get started." Note that the College Board maintains that its Advanced Placement exams reflect mastery of college-level subject matter. If this is the case, how credibly can it vouch for the rigor of a course taught by teachers whose subject matter competence is limited to a week or two—if that—of "professional development?"[5]

There is further fallout from a cadre of secondary CS instructors, most of whom—some estimates are as high as 90%—have had no formal CS education and no experience working in the field. Because these teachers lack subject-matter competence, they are prone to accepting without question the anti-programming rhetoric that has effectively demonized what the ECS group has termed "programming-centric" curricula. Thus, there is little bandwidth for these instructors to appreciate how central programming is to the discipline, or to recognize its gatekeeper role in determining access to high-paying computing jobs. As a consequence, these instructors are unwittingly misleading their students in a classic "bait-and-switch" that subsequent coursework will resemble the material they are studying, when the reality is that there is practically no overlap.

Another part of the problem is that, unlike in the core subjects, CS curricula that are actually penned by those secondary instructors having subject matter competence generally stay local and are rarely taken seriously or promoted widely. Such educators work in the classroom day-in and day-out, are able to observe what lessons and pedagogies do not work, and ultimately are the only ones who might have any insights on instructional strategies that might be effective with their students. Although the broad outline for APCSP was developed by both secondary and college-level instructors, the most prevalent versions of the full courses come from organizations like Code.org or Project Lead the Way, and target instructors with little to no subject matter competence who have little choice but to accept these curricula as they are, in effect serving as guinea pigs for untested material. Unfortunately, these courses also tend to employ pedagogic styles extrapolated from ones traditionally used in a college setting. This is problematic because in broad terms, college courses in effect weed out students—making little effort to accommodate strugglers—while secondary courses, in theory at least, focus on helping all students learn. The low AP exam pass rates of traditionally underrepresented minorities (URMs) in Tables 1 and 2 are indicative of this pedagogic difference.

The net effect of all of these factors and misconceptions has been to normalize a perverse assumption up-and-down the entire 9-12 bureaucratic structure that not only is programming an option that can be covered briefly and superficially—if not sidelined completely—but that rigorous programming instruction by definition automatically excludes traditionally underrepresented demographic groups, and that its replacement by the simplistic content of the survey courses is the most ethical and responsible choice. As an example, the USC-based Math for America Los Angeles program has recently added a Master Teacher Fellowship track to support LAUSD computer science instructors. A math professor on the steering committee who has helped to administer the program had this to say about proponents of programming curricula.[6]

> [This] perspective also runs counter to current trends in computer science education. The field as a whole recognizes that computer science does not consist solely of coding and to view it in that light is actually one of the main barriers to broadening access to computer science. This is particularly true because the field has very stereotypical views about who does computer science, and that prior computer science experience is very uneven and highly correlates with race, gender and family income… [To criticize ECS] shows a lack of understanding of the true reasons behind the lack of representation in computer science.

Apparently then, ensuring that students from underrepresented groups NOT learn how to program is the solution to increasing representation, according to this Mad-Hatter-like reasoning. This line of argument is based on an incomplete and distorted presentation of

[5] College Board AP "professional development" workshops are set up to deliver effective teaching strategies and instructional resources, not to perform the impossible task of teaching instructors an entire body of content knowledge.
[6] Testimony from an investigation into a discrimination complaint filed with the Office of Equity and Diversity, University of Southern California, against its *Math for America Los Angeles* program (March 2019).

the research about the factors that perpetuate demographic inequities, in order to make it fit a confused narrative. What makes it all the more disturbing is that it was expressed, not by a bureaucrat in K-12 administration, but by an academic from a preeminent university.

To be clear, Margolis' research in LAUSD high schools did NOT determine that programming ("coding") was one of the "main barriers to broadening access." Rather it faulted two broad categories: "the high school educational environment" and "psychological and cultural factors." Furthermore, multiple passages in her book deal with systemic factors that hamper students enrolled in the APCS-A course, all with the goal of increasing the numbers of URMs and females in that college-level programming course. Examples range from minority APCS-A students attending the monthly UCLA AP Readiness program; to the dropping of APCS-A courses by principals in low-resourced "program improvement" schools; to how "norms and structures widen or narrow the computer science pipeline for underrepresented minority and female high school students;" to students being locked out of programming classes they wanted because of constraints on scheduling [15].

> *Interviewer:* If you had a choice between floristry and another semester of programming, what would you choose?
> *Suzanna* [a student taking precalculus]: Programming … the flower thing is not really me.

The statement "…the field has very stereotypical views about who does computer science, and that prior computer science experience is very uneven and highly correlates with race, gender and family income" may be correct. But used to imply that this should somehow discredit the goal of teaching programming to all high school students flies in the face of Fisher's and Margolis' findings that these factors reduce minority students' chances of success in introductory college CS classes *precisely because* those students had limited access to programming experience prior to college. The intent of the researchers in this case was the exact opposite: to find ways to help those at a disadvantage surmount the barrier of less programming experience. To suggest, then, that programming instruction be avoided in the introductory course because students come equipped with different levels of experience is as absurd as throwing out the academic content of an Algebra 1 course because of varying proficiencies in prerequisite math skills. What good teachers do in the latter circumstance is find ways to shore up the mathematics foundations of students who struggle.

The net effect of all of these factors and misconceptions has been to normalize a perverse assumption up-and-down the entire 9-12 bureaucratic structure that not only is programming an option that can be covered briefly and superficially—if not sidelined completely—but that rigorous programming instruction by definition automatically excludes traditionally underrepresented demographic groups …

Although Fisher's and Margolis' research on students in CS classes at Carnegie-Mellon University (CMU) found that prior programming experience did give boys an advantage, they also found that it could be mitigated in several ways—at least in this elite college setting. One was to accommodate women's motivation for "computing with a purpose" by contextualizing the curriculum within real-world situations and adopting an interdisciplinary approach—although it should be noted that "enjoyment of computing" (63% of women and 70% of men) was still the most frequently cited factor by far in a student's choice of major. Another was to give students options for four different versions of the introductory course, based upon level of experience. This change also functioned as a counterweight to the tendency for such courses to "weed out" students, some of whom may actually be bright and committed but turned off by courses that catered to "nerd" aspects of CS student "culture."

Crucially, Fisher's and Margolis' research found that prior experience was ultimately not a predictor of eventual success [14]. And to the matter at hand, nothing in these studies ever came close to suggesting that the central role of programming in the introductory curriculum was a problem.

To the contrary, the impressive outcomes of the Girls Who Code program unequivocally belie the contention that a focus on programming is the cause of the "lack of representation in Computer Science." The assertion is also contradicted by efforts such as EngageCSEdu, a project of the National Center for Women in Technology (NCWIT) that has amassed an online collection of introductory college-level (CS1, CS2) programming assignments informed by research-based practices for engaging and retaining all students—but particularly women and URMs—in the study of CS [18].

All that said, my own experience with teaching urban secondary students is that even when students have had prior programming experience, such as middle school *Scratch* courses, the skills and concepts learned are very elementary, don't transfer to a rigorous introductory programming course using a text-based language, and don't give these students any technical advantage—although it may have heightened their internal motivation. That some students, regardless of prior programming experience, gender, or ethnicity, seem to be more adept at learning how to program is undeniable. However, the literature to date has not found any

student characteristics that can consistently be used as a predictor of success in the introductory class. As I discuss further on, I have found that the gap in learning differences within a class of students, and particularly students who struggle, can be narrowed and mitigated by instruction that is informed by principles underlying second language-acquisition.

The current state of secondary CS education is reminiscent of the push to offer college-level classes in math and science at secondary schools in the post-Sputnik era, except that this is the opposite: offering CS courses that have been dumbed-down. All the more remarkable, though, is that even as this new, confused, and self-defeating educational "philosophy" has played itself out over the last decade in restructuring the world of urban public secondary CS education—all in plain sight—policy watchers scratch their heads and continue to wonder why the new survey courses have failed to make a dent in the stubbornly low percentages of women and minorities in higher education and industry.

At the same time, there is a push by several states in the opposite direction to ensure that secondary instructors have a minimum familiarity with the discipline through each state's Department of Education's certification process. Since 2016, California, for example, has had a supplementary authorization in Computer Science, as a first major step towards an eventual full Computer Science certification. CSU campuses throughout the state provide support by offering 4-course or 5-course sequences—in programming, data structures and algorithms, software design, devices and networks, and impacts of computing—that satisfy the requirements for the authorization.

The problem is that—because the survey courses are now viewed as the new normal in "rigorous" and "effective" secondary CS education, and because nothing in their content requires instructors to have even the limited expertise that the supplementary authorization would give them—the result has been to disincentivize (a) instructors from learning the very fundamentals of the subject they teach, and (b) school districts from requiring that CS instructors earn the new authorization.

Advocates for the survey courses make an ethical argument that they are addressing issues of equity, diversity, and participation. In practice, though, by not urging that all students have access to rigorous programming instruction, the effect is to reinforce—rather than dismantle—the existing inequitable two-track system. The upper track in this system comprises wealthier schools or those academically-selective in some other way, where subject-matter competent instructors teach rigorous programming courses (APCS-A, often preceded by a pre-AP introductory programming course) that research has demonstrated prepares students (mostly white and Asian males) for success should they pursue CS study in college. The lower tier consists of high or mid-high poverty urban schools where students who previously learned the MS-Office suite and keyboarding now have teachers with no particular expertise in CS teaching simplified courses of unproven value (APCSP and ECS) that leave students (for the most part URMs and females) no better prepared for subsequent CS coursework. It could even be argued that lower-tier students were better off when they were taught the MS-Office suite, because at least then they might earn MS Office Specialist Certifications—in Word, PowerPoint, Excel, Access, etc.—industry-recognized learning that is validated by standardized exam. Ironically then, the survey courses have recapitulated—and continue to perpetuate—the very historical inequities that they were intended to eliminate. Engendered by familiar discriminatory educational practices like inferior instructional materials and the soft bigotry of low expectations, lower-tier students in this "new" reconstituted status quo still remain "stuck in the shallow end." [15]

Recalling the fictitious paid internship ad from Figure 1, a former APCS-A student (whose AP exam score was 2)—now a college freshman double-majoring in CS and Ethnic/Latin-American Studies—recently had an interview for a second Google summer internship. The company advised her to prepare as follows.

> **Coding:** Use C, C++, Java, Javascript or Python. You may be asked to:
> Construct / traverse data structures
> Implement system routines (e.g., string compare)
> Work with and distill large data sets to single values
> Transform one data set to another
>
> **Algorithms:** You will be expected to know the complexity of an algorithm and how you can improve/change it.
> Examples of algorithmic challenges:
> Big-O analysis (particularly important)
> Sorting and hashing
> Handling large amounts of data
>
> **Data Structures:** You will need to know about:
> Basic Tree construction, traversal and manipulation algorithms
> Hash tables
> Stacks, Arrays, Linked Lists, Priority Queues

These topics comprise the content typically taken by CS majors in their first full year or year-and-a-half—at minimum, the Introductory Programming course, and an Algorithms and Data Structures course. It's telling that much, if not most, of this content used to be covered in the College Board's now defunct APCS-AB high school course, discontinued in 2009 because of persistently low enrollments—and now replaced by the APCSP survey course. Proponents of the survey courses have argued that simply taking

either ECS or APCSP amounts to "increased participation," but it is hard to see such claims as anything other than cynical dissembling to excuse dead-end courses that do nothing to even lay the programming groundwork for this foundational CS material.

Nothing about the current academically, gendered, and racially segregated secondary status quo is particularly revelatory—the situation has been clear for years to many secondary and post-secondary CS educators, who will privately acknowledge serious concerns with the low educational quality of the survey courses, but nevertheless decline to openly challenge the upbeat public relations narrative. In this respect, allusions to the Emperor having no clothes are not an exaggeration. Conversely, some may even champion these courses, perhaps rationalizing that lowering the bar for the sole purpose of encouraging interest is better than doing nothing. After all, there has never been a credible diagnosis of—much less a viable solution to—the NPFP, so there is little doubt that the vast majority of grade-level high school students would fail a rigorous programming class.

To this last point, however, the next section discusses one potential solution to the NPFP, a pedagogic framework that has allowed all of the grade-level high school freshmen in my year-long pre-APCS-A/introductory programming course to make and sustain progress in learning how to program. Were enough instructors to adopt and validate—and extend, improve, and research—this approach, and the methodology scaled, the direction of secondary CS might finally shift away from the sub-grade-level goals of the last decade's survey courses. It should also be noted that this would not be the first time in the history of American education that a simplistic curriculum with little value was broadly adopted only to later fall out of favor as its flaws and detrimental educational effects became impossible to deny.[7]

A NEW PEDAGOGY FOR ADDRESSING THE NOVICE PROGRAMMER FAILURE PROBLEM (NPFP)

CSEA's initial instincts to augment the number of minority and female students in the APCS-A Java course were correct. 2019 LAUSD data (Table 3) bear out the severe and ongoing inequities in both participation (gender and ethnicity) and learning (ethnicity):

Table 3: LAUSD 2019 AP CS A (Java) Exams by Ethnicity / Gender.

Ethnicity / Gender	LAUSD %	% of Exam Takers	Exams Taken	Exams Passed	Exam Pass Rate	Mean Score	Mean Pass Score
African American	8.2%	3.5%	12	3	25.0%	1.58	3.00
Asian	4.2%	21.9%	75	53	70.7%	3.28	4.09
Filipino	2.1%	4.1%	14	8	57.1%	2.79	3.88
Hispanic	73.4%	58.0%	199	43	21.6%	1.67	3.53
American Indian	< 1%	0.3%	1	1	100.0%	3.00	3.00
White	10.5%	12.2%	42	32	76.2%	3.50	4.16
Female	—	31.5%	108	38	35.2%	2.16	3.95
Male	—	68.5%	235	102	43.4%	2.35	3.87
Total LAUSD	—	100.%	343	140	40.8%	2.29	3.89
National	—	—	69685	48518	69.6%	3.26	4.08

The proper goal, however, should have been to augment the enrollments in APCS-A of well-prepared students in these demographic categories. Consider also that in 2019 overall national enrollments in the course were just under 70,000 compared to over 300,000 taking the Calculus AB exam, an academically comparable group and a ratio of less than 1:4. This suggests that in addition to depressed enrollments of students from traditionally underrepresented groups, there are also low enrollments from the traditionally dominant demographic groups, and that consequently, those who take APCS-A are primarily from a small subgroup of white/Asian males. A second goal, perhaps, might therefore be to boost absolute enrollments overall, ensuring that all students have sufficient academic preparation and that all instructors possess subject matter competence.

That academic preparation, and the logical, commonsense solution for CSEA's massive student failure—and the NPFP in general—would not be a survey course, but rather a CS0-level pre-APCS-A programming course that identifies and addresses head-on the difficulties that novice programming students encounter. Such a course would help to level the playing field between the traditional demographic and URMs in a subsequent CS1-level APCS-A course. Such a course would deliver *measurable* learning objectives—a base set of program-

[7] The survey course phenomenon has uncanny parallels to the avowedly and unapologetically anti-academic Life Adjustment curriculum of the cold-war era. Life Adjustment advocated a high school curriculum that claimed to be relevant to the social needs of teenagers, including areas such as dating, learning to be an effective consumer, social relationships, family living, and the like. Courses included School and Life Planning, Preparation for Marriage, Boy-Girl Relationships, Learning to Work, and How to Make Friends and Keep Them. The movement secured a place in educational policy by professing to be a solution for a perceived dropout crisis based on regional dropout rates as high as 30-45%. The curriculum was backed by the National Association of Secondary-School Principals, and at one time, the U.S. Department of Education. Life Adjustment education eventual-ly came under heavy attack in the early 1950s by academics who had long viewed the school's job as the development of the intellect. By the time of Sputnik and the subsequent 1958 National Defense Education Act, the Life Adjustment movement had lost most of its credibility among the general public [12].

ming skills and concepts—to all grade-level students, the way a first-year foreign language class lays a foundation for the second-year class, or an Algebra-1 course prepares students for Algebra-2. Note that computer science departments at elite public magnet high schools, such as Thomas Jefferson High School of Science and Technology, Alexandria, Virginia, have long offered a pre-APCS-A programming course, recognizing that even academically gifted students benefit from a year of preparation prior to APCS-A.

Such courses are not meant only for those students who intend to study CS in college. After all, all high school students are required to take coursework in history, English, science, and math, regardless of their future choice of college major. These courses give students basic, foundational knowledge in these subjects, and are designed to align vertically, i.e., be college preparatory courses in substance and not just in name. Stakeholders would never countenance, for example, an English literature course that relied on movies and SparkNotes to teach *To Kill a Mockingbird*, under a guise that most students will not pursue a college English major. Among the countless reasons arguing against such a scheme, the principal one is that students "taught" in this way would be wholly unprepared for college, having never learned to read, analyze, reflect on, and write coherently about complex material. Unfortunately, the primary objective of much of CS education in urban public schools is simply to generate interest at the expense of preparing urban youth for a competitive academic and working world. It is hard to conceive exactly how—divorced from the study of programming and its applications to other CS concepts—learning about sociological aspects of CS will be of much use.

Instructors at the secondary level, and especially in the public sector, are also tasked with ensuring that all students learn the course content of the subjects they teach. Although this is an idealized target, the goal does prod teachers to dig deeper and try to surmount impediments that prevent every student from reaching her potential. Long-time K-12 instructors have learned to recognize that when most students fail to learn a concept, that reflecting on and identifying where the pedagogic stumbling blocks are and then modifying instruction often works. Unfortunately, programming instruction has historically had a terrible record in this regard, and teachers have had few, if any, clues how to help the large number of students who struggle.

There now exist three fMRI studies confirming that comprehension of computer programs occurs in those regions of the brain that process natural languages—not math and not logic [6,26,27]. This cognitive-physiological evidence indicates that programming languages, despite being artificial languages, are alive in the brains of programmers in much the same way as any natural language that those programmers speak. One logical implication that springs immediately to mind is that the primary mode for learning programming languages may be that they are implicitly *acquired* like second languages—and not explicitly *studied* as are mathematics, science, history, or literature. This would be a profound paradigm shift for thinking about how all students might benefit were instructors to include instructional pedagogies that teach a programming language as a second language per se.

Instructors might assume that, because the syntactic footprints of programming languages are quite small, that therefore learning to use their grammars is a trivial matter. The relatively small number of syntactic structures, though, are semantically broad—as compared to a natural language's numerous syntactic elements, each with its very specific semantics and narrow range of uses. This creates an entirely new challenge because, unlike the way natural languages work, these structures are used individually and in combination to handle a multitude of specific situations.

I've previously described a curricular framework that focuses on the teaching of programming languages using supplemental instructional strategies adapted from principles underlying foreign language instruction, also known as the field of Second Language Acquisition (SLA)—and that, importantly, work. The set of strategies and the programming curriculum I have been crafting on an ongoing basis—based largely upon observations of errors with which students struggle—consistently allow all of my grade-level high school freshmen, non-gifted as well as gifted, to make assessable gains in programming proficiency. This is particularly true for those who would have formerly learned little, suggesting the viability of such an approach if scaled [22,23].

Based on the above, I contend that the root cause of the NPFP is specifically a pedagogic gap: we want students to use logic to solve problems, but we have neglected to provide a curricular framework and the instruction that will help them acquire *the very language which mediates that logic*. If we want to include more students who will succeed in APCS-A (and by extension CS1), and in particular those who have historically been underrepresented, it is imperative to find ways to provide effective instruction.

To my knowledge, the three fMRI studies are the only brain research to date on programming. Though they do not decisively prove the case that language instruction should be a part of CS0 or CS1, they do suggest that this approach be seriously explored. In contrast, there is zero cognitive evidence to validate and justify the use of current introductory teaching approaches, whose deficient and exclusionary outcomes have long been documented, and whose only argument for con-tinued use is inertia. Common sense then, and not just the science, would argue for a wholesale reorientation of the way we think about our pedagogies, including how we take our current instructional strategies—oftentimes and reflexively the ways we were taught—for granted, and add a serious role for language instruction in CS0, and perhaps in CS1, as a way to reach groups of learners who have historically struggled.

HOW DOES INSTRUCTION INFORMED BY SLA STRATEGIES DIFFER FROM TRADITIONAL CS PEDAGOGIES?

Human language is an innate, highly specific cognitive ability quite distinct from general intelligence; that is, language is *acquired*, not *learned*. The pedagogic model guiding foreign language instruction for several decades now has been that second languages

are acquired primarily like first languages, i.e., implicitly and through subconscious processes—although there is a corrective or supporting role for explicit cognitive strategies like grammar instruction. In this model, the role of teachers is to provide the conditions, situations, and experiences that facilitate a second language's gradual acquisition implicitly through two primary activities: (a) receptively via repetitive exposure to language data—vocabulary, syntactic patterns, and paradigms—in meaningful contexts; and (b) intentional expressive use of the language with implicit feedback.

When applying an SLA model to programming language instruction, two curricular modifications immediately come to mind.

For most students, the start of meaningful automaticity and basic programming fluency—like the timeline for foreign languages—emerges after two to three years. The timeline of a curriculum should therefore be lengthened from a single course to a multi-year course sequence, with realistic and reasonable expectations for what programming skills and language features can be acquired at each stage.

A third foreign language is also easier to learn after having acquired proficiency in a second. A current and common practice in CS education is to teach a different programming language as often as every semester, a practice that assumes that programming concepts practiced in one language—after a period that can only provide an exposure—will easily transfer to another. Studies measuring such transfer have debunked this assumption [1]. Therefore, one programming language should be taught throughout the entire multi-year sequence until proficiency is acquired.

These two changes would provide the time in a year-long CS0-level pre-APCS-A course for students to develop a basic level of familiarity and proficiency—automaticity, really—with the Java language, and build their programming "toolkits" with a base set of language patterns and their most common usages in procedural and object-oriented programming.[8] In the subsequent CS1-level APCS-A course, students would expand this collection of tools with more nuanced, optimal language patterns, and a more varied, complex, and in-depth study of solutions for a range of problems typically covered in that course.

These two suggestions follow from the similarities between learning natural languages (NLs) and programming languages (PLs). However, there are significant differences between the grammars of NLs and PLs that often confound student learning, a principal one being hierarchy. The discussion that follows illustrates where and how SLA-informed instruction can address language-related learning difficulties that current pedagogies assume are trivial and gloss over.

Unlike NLs, PLs utilize hierarchical layers that one might imagine are situated along a 3rd dimension, much like parallel planes above the main x-y coordinate plane. Method signatures and bodies and class definitions occupy these layers, ultimately executing in a program's main() method via a chain of method calls and constructor instantiations, respectively.

The connections of method calls/class constructors to their signatures/definitions are not at all obvious to novice programmers. Because there is no grammatical counterpart in NLs, students have nothing to reference. Unlike the explicit syntax of an assignment statement where the assignment operator places an R-value into an L-value, the assignment of argument values in a method call or class constructor to their corresponding parameter variables in the signature/definition happens implicitly. Nowhere in the explicit surface syntax of a program do students see this connection.

This hidden or background processing causes a host of student misunderstandings, leaving students with no option but to guess. Even the best students have initially misunderstood that (a) the number and order of the arguments and parameters must be the same; (b) the types of arguments must match those of its corresponding parameters; and (c) parameter variables are assigned the val-ues of the corresponding arguments. One practice exercise that can illustrate this mechanism is a linguistic "transformation" that posits an underlying intermediate syntactic structure (Figure 2).

Problem: Assignment of argument values in constructor call to parameter variables is implicit.

// main.pde
```
c = new Comet( 15, 30, 1, -1, 5, 7 );
```

// Comet.pde
```
Comet( float x, float y, int dirX, int dirY, float speedX, float speedY ) {...}
```

Exercise that makes the assignments explicit.

#1: Start with assignment statements in the constructor body with no parameters:
```
Comet() {
    float x=15;
    float y=30;
    int dirX=1;
    int dirY=-1;
    float speedX=5;
    float speedY=7;
}
```

#2: Imagine an intermediate structure by promoting these assignment statements into the constructor parentheses / signature:
```
Comet( float xIn=15, float yIn=30, int dirX=1, int dirY=-1, float speedX=5,
          float speedY=7) {...}
```

#3: Move the values from the intermediate form's signature to a constructor call.
```
c = new Comet( 15, 30, 1, -1, 5, 7); // constructor call

Comet( float x, float y, int dirX, int dirY, float speedX, float speedY ) {...}
```

Figure 2: Transformational Model: Posit Intermediate Structures that clarify assignment of argument values to parameters (...and how parameters behave like local variables).

[8] For the types of issues considered in choosing this base set, please see [22].

I assess my CS0 students on these concepts using word problems that ask students, given a constructor call, to write the corresponding class (Figure 3). They continue to work on these problems until all misunderstandings have resolved and the PL pattern has been acquired.

Given the constructor call:

```
Star s;
s = new Star(RED, 1000, 7.3);
```

in which

```
RED is the color of the star
1000 is its size
7.3 is its brightness
```

write the **Star class** and its **constructor.**

ANSWER:

```
class Star {
  color clr;
  int size;
  float brightness;

  Star(color clrP, int sizeP, float brightnessP) {
    this.clr = clrP;
    this.size = sizeP;
    this.brightness = brightnessP;
  } // constructor
} // class
```

Figure 3: Word problem exam assessing student understanding of the connection between arguments in a constructor call and the corresponding parameters in the constructor definition.

A second syntactic feature whose action is implicit is Iteration. In this case, the surface syntax can be thought of as a contraction or abbreviation for generating multiple statements that the program will execute. The most common usage of the for-loop in my CS0 course uses the counter variable to traverse individual members of an array. The exercise in Figure 4 demonstrates a pattern of two successive for-loops, whose expansion results in two distinct, non-overlapping groups of state-ments.

For-loop (abbreviated) version

```
Song[] songList = new Song[3];
for (int i = 0; i < 3; i = i + 1) {
  songList[i] = new Song();
} // for
for(int j = 1; j < 3; j = j + 1) {
  songList[j].sing();
} // for
```

Expanded version

```
Song[] songList = new Song[3];
songList[0] = new Song();
songList[1] = new Song();
songList[2] = new Song();
songList[3] = new Song();
songList[1].sing();
songList[2].sing();
```

Figure 4: Two For-loops in succession, whose statements do not overlap.

Students practice procedures to (1) calculate the set of counter-variable numbers that the for-loop generates from the loop's initialization, condition and update statements; (2) copy one complete set of statement(s) in the for-loop body for each number in that list (corresponding to the number of iterations); and (3) substitute those numbers for the counter-variables.

Figure 5 illustrates a second pattern in which one or more statements in the for-loop's body is governed by a conditional statement. The IF-statement functions as a filter on the statements in its body. The final result is an interspersing of the statements.

For-loop (abbreviated) version

```
Film[] filmList = new Film[4];
for (int k = 0; k < 4; k = k + 1) {
  filmList[k] = new Film();
  if (k % 2 == 0) {
    filmList[k].writeReview();
  }
} // for
```

Expanded version

```
Film[] filmList = new Film[4];
filmList[0] = new Film();
filmList[0].writeReview();
filmList[1] = new Film();
filmList[2] = new Film();
filmList[2].writeReview();
filmList[3] = new Film();
```

Figure 5: A Single For-loop containing an IF statement, and its Expanded Form.

Figure 6 shows a multi-step procedure for solving problems utilizing this second pattern.

A common novice error is that prior to instruction, students who are given expansion problems for both patterns are not able to

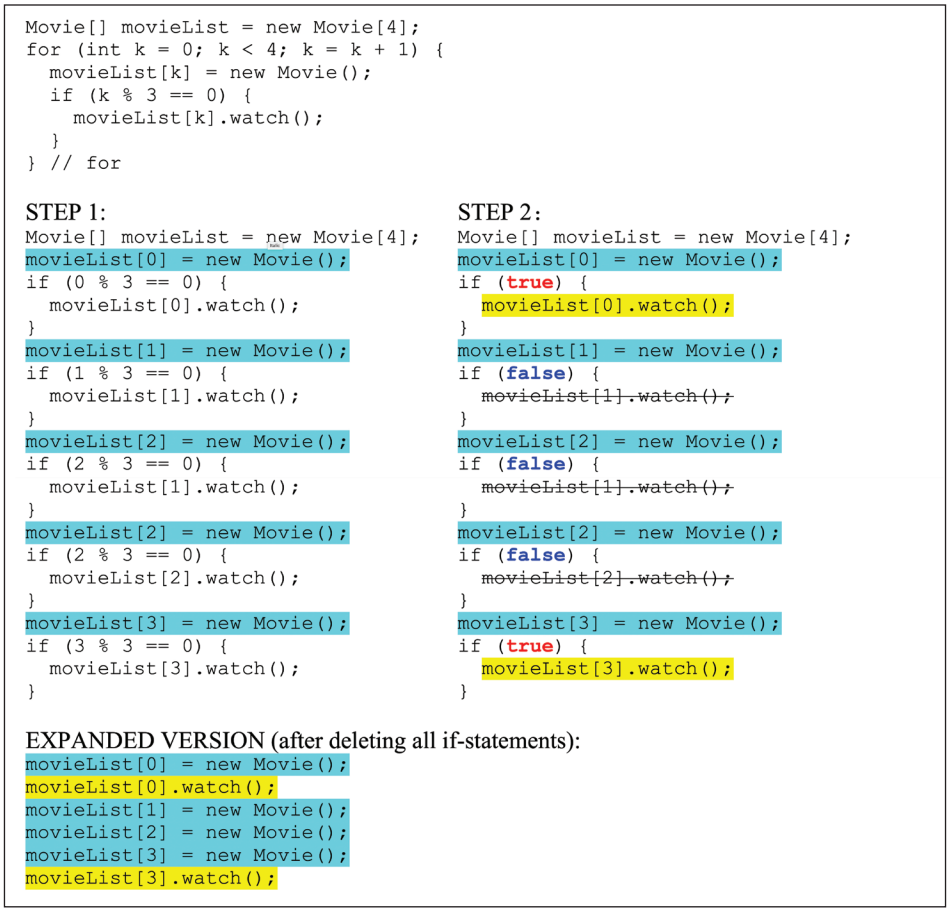

Figure 6: A Step-by-Step Procedure to expand Single For-loops containing IF statements.

distinguish the patterns from each other, and simply default to guessing. It is only through repeated practice that the syntactic rules that govern iterative control structures and their interactions with conditional statements become automatic and second nature.

One pedagogic strategy that actually argues for a language-based cause of the NPFP is memorization. Teachers of the introductory programming course universally observe sizeable groups of frustrated students unable to compile and run their programs because they continue to make the same syntax mistakes week after week. This phenomenon has been documented for at least two decades [13].

> … students using an unfamiliar or new programming language waste considerable time correcting syntax errors. Studies have shown that excessive time spent on correcting syntax problems can be detrimental to long-term success as students become disheartened with programming.

For seven years running, though, I have observed that this problem disappears (1) after students memorize small programs containing newly-introduced syntactic features/patterns, data structures or control structures, and (2) then demonstrate that they have done so sufficiently by writing or typing them out perfectly. What is striking is that this strategy is effective in the absence of any instruction pertaining to the new syntax features.

One possible mechanism for why this works is as follows. As stated, language syntax cannot be taught explicitly, e.g., through grammar rules; rather it is implicitly acquired through repetitive exposure to language data/input. Memorizing a program and reproducing it without error mimics this process insofar as students must undertake numerous cycles of reading a program, writing it out without looking, and comparing. In doing so, they bombard their brains repeatedly with idealized language data. As with natural languages, the brain subconsciously constructs an internal mental representation of the syntax rules implicitly by induction from the patterns in the data.

Although memorization consistently facilitates the rapid acquisition of most syntactic patterns, by itself, it is insufficient for helping students acquire the semantic component of these patterns—how they can be used clearly and efficiently in a multitude of situations. This requires not just repetitive practice, but contextual variation as well, and lots of experience.

For other issues that arise when addressing PLs as languages, please see [23, pp. 72-101].

RESOURCES FOR TEACHING *CODINGBAT* SECTIONS LOGIC-1 AND STRING-1 IMPLICITLY

Codingbat.com is a "live" web site where students can practice solving programming problems and receive instant feedback on the success or failure of specific test run cases. Its **Authoring** feature allows instructors to tailor their instruction by creating custom pages of original problems.

The Java sections of Codingbat consist of broad categories of problems, grouped under headings like Logic 1 and 2; String 1, 2 and 3; Array 1, 2 and 3 and the like. However, even with the Codingbat *Help* pages, novices can still experience difficulties in the Level-1 sections and require intensive instruction and intervention. In the past, I've attempted to mitigate these difficulties via direct instruction. Predictably, the effectiveness of that approach was limited to a minority comprising the most academically talented students.

The reason is that programming languages, like natural languages, are infinitely generative. As such, there are countless ways to solve problems, most of them wildly inefficient. There is also wide latitude for students inexperienced in a new language to make errors and quickly become discouraged. To narrow the possibilities, the CS0 teacher—like the foreign language instructor—can serve as a kind of "native speaker" for her students, modeling a small set of the clearest and simplest, (if not necessarily the most optimal) language constructions and patterns that will reliably address the specific challenges posed by different categories of problems. This minimizes confusion and allows students to develop a basic level of proficiency and automaticity by using a small number of consistent and uncomplicated programming language patterns. Over time, they can add to their programming repertoires ever more complex syntactic patterns as material is revisited, along the lines of Bruner's spiral curriculum hypothesis.

It is within this framework that I am using the Authoring feature to write custom Codingbat web pages of problem sets that students complete prior to tackling the regular Logic-1 and String-1 sections. These custom web pages comprise two broad sections.

The first section concerns itself with students learning to use two tools for writing methods: (a) a single uncomplicated language pattern that can be used as a template for writing a method, applicable to both the regular Logic-1 and String-1 sections; and (b) key API methods, in this case, learning the mechanics, functionality (what information/data they calculate/return), and calling patterns for five String methods. This section is called Understanding the Tools.

The second section of problem sets is intended to familiarize students with simple and clear strategies for using these two tools to solve the categories of problems in the regular Logic-1 and String-1 sections. This section is called Problem-Solving Strategies.

SECTION 1: UNDERSTANDING THE TOOLS

The Logic-1 problem set in the Understanding the Tools section introduces a single syntactic paradigm/method template—one that returns a Boolean, int, double/float or String value. This pedagogic strategy is a way for students to acquire a reliable first syntactic pattern that they can use when writing any method without being overwhelmed and confused by too many choices as they work in a new and unfamiliar language.

The first line of the method body declares a return variable of the same type as the method's return type and initializes it with a default value (e.g., false, 0 or ""). The variable is returned by a *single* return statement on the last line. In between are one or more *cascading* IF statements that test the input parameter(s) and change the value of the return variable (Figure 7). Mutual exclusion is accomplished entirely through Boolean logic—that is, the teaching of ELSE and ELSE IF statements, whose logical complexities and side-effects can confuse beginners, is deferred (See Part IV of [22] for a discussion of these considerations).

The first String-1 problem set in the section introduces length(). The second problem set introduces the one- and two-parameter versions of substring(). The third introduces endsWith() and the one- and two-parameter versions of startsWith(); equals() and equalsIgnoreCase(); and toLower-Case().

The first String-1 problem set is also intended to clarify early syntax misunderstandings commonly held by novices, such as not recognizing that a method's parameters are its input and that its return statement hands back (outputs) the result (Figure 8). Given enough repetition, the problems in this section quickly clarify these basic mechanisms.

```
int getAtomAngle(int aNumber) {
  int angle = 0;
  if (aNumber == 1) {
    angle = 0;
  }
  if (aNumber == 2) {
    angle = 60;
  }
  if (aNumber == 3) {
    angle = 120;
  }
  if (aNumber == 4) {
    angle = 180;
  }
  if (aNumber == 5) {
    angle = 240;
  }
  if (aNumber == 6) {
    angle = 300;
  }
  return angle;
}
```

Figure 7: Simple Method Paradigm.

Paradigm / Example

```
int howManyLetters(String str) {
  int len = str.length();
  return len;
}
```

Problem (student attempt)

```
int howManyLetters2(String word) {
  int len = str.length();
  return len;
}
```

Figure 8: Confusion about input parameters.

Many novices also have trouble with the concept of return *type*. Figure 9 is one question in a formative-quiz that connects the type of the variable in the return statement to the method's return type.

What is the return type for method_F() ?

```
RETURN_TYPE method_F( String name, int hour, float journey ) {
  String word = name;
  int day = hour;
  float distance = journey;
  boolean happy = false;
  int[] list = { hour, 2 * hour, 3 * hour, 4 * hour };
  return distance;
}
```

- a String
- b int
- c Not enough information is given to determine the return type.
- d float
- e boolean
- f int[]

Figure 9: Formative Quiz on the Return Type.

The Logic-1 problem set and the three String-1 problem sets utilize repetition and a problem-sequence that incrementally increases in difficulty. Although they utilize well-established instructional strategies—ZPD, scaffolding, Bruner's spiral curriculum, constructivism—the utmost incrementalism and repetition they use, illustrated in Figure 10 and Figure 11, are rooted in the principles by which second languages are gradually and implicitly acquired.

srp4379@lausd.net 1-logicbasics > isSchoolDay

prev | next | chance

public **boolean** isSchoolDay(int day)

Given a number in the range 1-7 that represents a day of the week (1=Sunday, 2=Monday, etc.), return true if the day is M-F, false if it's the weekend.

Example 1:

```
boolean schoolDay = false; // default: it's the weekend (1 || 7)
if (2 <= day && day <= 6) {
  schoolDay = true;
}
return schoolDay;
```

Example 2:

```
boolean schoolDay = true; // default: it's a weekday (2-6)
if (day == 1 || day == 7) {
  schoolDay = false;
}
return schoolDay;
```

NOTE: When using an IF statement to decide between 1 or more possibilities:

(1) Declare a variable - **of the type returned by the method** - and assign it one of the possible values.

For example, if the method is called:
public **String** isSchoolDay(int day)
you could declare a variable: **String** yesNo = "No";

OR, if the method is called:
public **int** isSchoolDay(int day)
you could declare a variable: **int** sDay = 0;

(2) The last line will be a single **return** statement.
It will return the **value of the variable** (the correct answer).

(3) In between the variable declaration (first line) and the return statement (last line), place one or more **if** statements to test the **PARAMETER(s)** for other possibilities. If an **if** statement is true, its body will **modify** the value of the **return variable**.

Figure 10: Logic-1, Problem 1. Understanding the Tools.

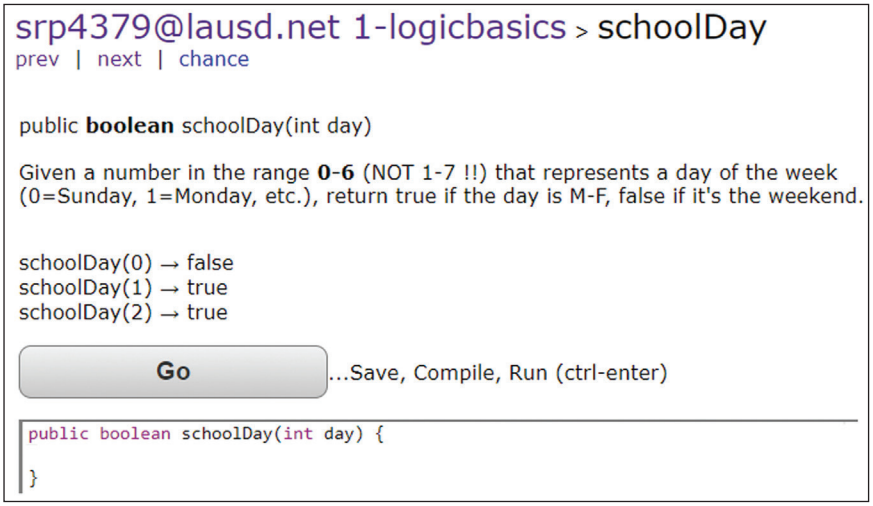

Figure 11: Logic-1, Problem 2. Understanding the Tools.Slight incremental change from Problem 1.

Figure 12 shows the first problem in the second String-1 problem set. The problem introduces the two-parameter version of the substring() method. Students are also encouraged to refer to a diagram that lays out the structure of a String, with expressions for finding its first, middle and final positions (Figure 13) should the arguments passed to the method differ in length.

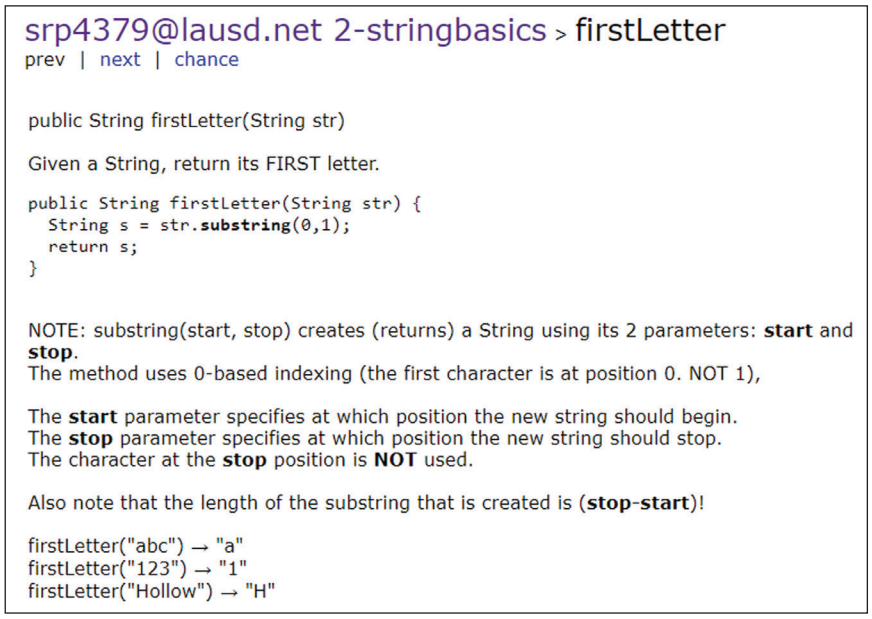

Figure 12: String-1, Problems set 2, substring(). Understanding the Tools.

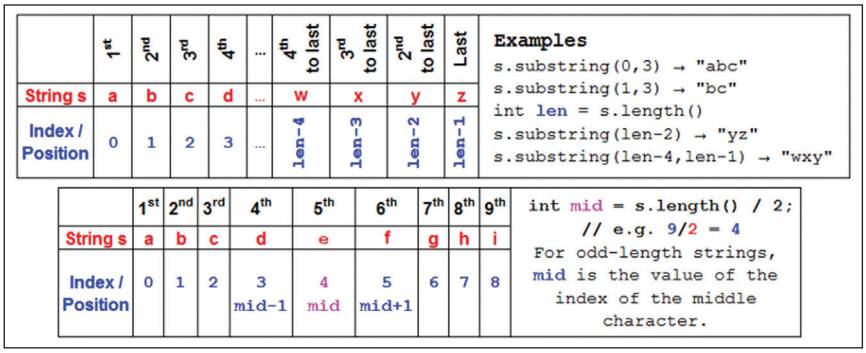

Figure 13: Model of the Structure of a String with expressions for accessing the first, middle and last characters.

SECTION 2: PROBLEM-SOLVING STRATEGIES

The Problem-Solving Strategies section familiarizes students with strategies for solving specific categories of problems, each strategy comprising one or two clear and simple exemplar language patterns.

The need for this section is that, although there is a widespread belief that a general problem-solving ability exists, the reality is that psychologists have yet to discover one—let alone confirm that such an ability can be taught. Problem-solving is domain-specific, i.e., if you practice solving problems of a certain type to the point of automaticity, you are better able to put similar problems into a recognizable context where the same strategies can be used [21]. To simply give students API methods and expect that strategies for their use are self-evident is like handing out foreign language phrases for a particular social setting (restaurant, library, sports event) and expecting students to intuit their use in the many possible contexts that may arise, without first having seen any example scenarios. Nobody would think this would be a rational way to teach second languages, and it would likewise be hard to justify a similar approach in an introductory programming language course.

Soloway's Rainfall problem continues to be cited as a benchmark for how one might successfully assess problem-solving abilities in CS1. Such sentinel problems are typically studied in the APCS-A Java course in a unit on indefinite loops, and textbooks such as *Building Java Programs* explicitly explain the algorithm/solution for this particular problem. As such, when practiced repeatedly, the pattern for solving this and similar problems becomes automatic, and students will pass this benchmark. That instructors would somehow expect novice students to be able to synthesize de novo algorithms to this or any of the problems in the CS1 course—as if the myriad ways one can use control structures and API methods is inherently self-evident—seems a bit like wishful thinking. Moreover, to use such arbitrarily chosen problems as benchmarks seems arbitrary as well, and ultimately self-defeating. If instructors want students to be able to solve certain types of problems, they can simply demonstrate the applicable language patterns and give students repeated practice solving similar problems.

All that said, many instructors still believe—despite the lack of evidence—that a general problem-solving ability does indeed exist. I urge these instructors to imagine themselves in the places of their novice programming students when asked to solve problems, whose patterns they have never practiced, using only a limited set of general principles. In this vein, consider the following task: write a program to determine the percent homology between two Strings by calculating their maximal alignment (note: this is the edit-distance problem and its variants, with applications to text editor autocomplete features and sequence alignment tasks in bioinformatics, among others). Though the solution involves only 2-D arrays and nested for-loops (CS1 material), many CS college professors encountering this task for the first time would be hard-pressed to figure out the optimal—let alone any decent—algorithm to solve it, without referring to an algorithms text, and particularly in a timed situation. The difficulty comes, not from being unfamiliar with the data structures, but from not having used them for an unfamiliar purpose. Again, once studied and practiced, the strategy can become recognizable and automatic.

For a CS0 course, therefore, an approach that makes sense to me is that exams assess whether students can recognize which strategy/language pattern or combinations are applicable for solving a particular problem—and then write the solution, of course. It goes without saying that students should have previously had experience with all strategies tested. I do something similar in CS1 several weeks prior to the APCS-A exam when I prep students for the test. This has involved categorizing the free-response problems that have appeared on the exam over the last several years and practicing the problems in each category as a group. Even as the particulars of each problem may differ considerably, students can come to appreciate the data structures that they have in common, and the specific algorithmic patterns used to process them (e.g., traversal, modification).

The regular Logic-1 section of Codingbat includes these subcategory types:

(a) Boolean combinations of parameter values;
(b) integer parameters tested for inclusion within number ranges;
(c) modulus operations;
(d) increasing/decreasing or largest/smallest series of numeric parameters;
(e) extracting and testing individual digits in a 2-digit number.

Figure 14 shows how the method paradigm practiced in the custom Logic-1 section can be applied in a regular Codingbat section problem (in1To10) that is a hybrid of the (a) and (b) subcategories: a Boolean method with two parameters, one integer, one Boolean.

```
public boolean in1To10(int n, boolean outsideMode) {
  boolean inRange = false;
  if (!outsideMode) {
    if (1 <= n && n <= 10) {
      inRange = true;
    }
  }
  if (outsideMode) {
    if (n <= 1 || n >= 10) {
      inRange = true;
    }
  }
  return inRange;
}
```

Figure 14: Logic-1 Basics pattern used in a more complex Codingbat Logic-1 problem.

Again, the decisions to forego IF/ELSE statements in favor of cascading IF statements that accomplish mutual exclusion exclusively through logic, though perhaps somewhat uncomfortable for CS1 instructors who would like their students to produce more optimal code, nevertheless yields a method that is clear and easy to read. And that's exactly the point: to teach CS0 novices a simplified conditional pattern as a way to minimize confusion and build a linguistic foundation that will allow more of them to later succeed in a subsequent CS1 course when these patterns can be extended and refined.

String-1 subcategories are of two types:
(a) manipulation of one or more input Strings to return a new String; and
(b) analysis of an input String to determine whether it contains a specified sequence or pattern.

The Problem-Solving Strategies problem sets demonstrate how to simply and clearly (and some-times optimally) solve such problems using the five Java String methods alone or in combination. Figure 15, 16, 17 and 18 show the detailed Review/Summary sections preceding the battery of problems in the sets that, in these cases, will demonstrate strategies for using substring() and startsWith().

first3Letters

public String first3Letters(String str) Given a String, return **ONLY** its **First 3** letters.

Review:
You've had lots of practice using **substring()** to extract the first or last letters of a string.

String str = "ABCDEF";

str.substring(**0,1**) creates a substring with just the 1st letter: **"A"**
str.substring(**1,2**) creates a substring with just the 2nd letter: **"B"**
str.substring(**0,2**) creates a substring with the first 2 letters: **"AB"**
str.substring(**1,4**) creates a substring with the second 3 letters: **"BCD"**

str.substring(**1**) creates a substring without the first letter: **"BCDEF"**
str.substring(**2**) creates a substring without the first 2 letters: **"CDEF"**
str.substring(**3**) creates a substring without the first 3 letters: **"DEF"**

Notice that the **number** of characters you extract is the **difference** between the 2 parameters of substring(), e.g.
str.substring(**2,5**) extracts **5-2=3** characters.

If you want to extract the final characters, you need to know the length, e.g.

String str = "TUVWXYZ";
int len = str.length();

str.substring(**len-1**) creates a substring with just the last letter: **"Z"**
str.substring(**len-2**) creates a substring with just the last 2 letters: **"YZ"**
str.substring(**len-4**) creates a substring with just the last 4 letters: **"WXYZ"**

str.substring(**len-2,len-1**) creates a substring with just the 2nd-to-last letter: **"Y"**
str.substring(**len-3,len-2**) creates a substring with just the 3rd-to-last letter: **"X"**
str.substring(**len-5,len-2**) creates the substring: **"VWX"**

str.substring(**0,len-2**) creates a substring without the last 2 letters: **"TUVWX"**
str.substring(**0,len-3**) creates a substring without the last 3 letters: **"TUVW"**

The parameters for substring have to comply with the 2 rules below:
(1) They must be in the range **0** through **len**. That is, they cannot be **negative** and they cannot be **greater than len**.
(2) If you are using 2 parameters, the 2nd must be **greater than or equal to** the 1st - it **cannot be smaller**.

Figure 15: String-1 Problem-Solving Strategies. Introductory Review. substring() Part A.

Now to solve the problem. Normally you would solve it as follows:

```
public String first3Letters(String str) {
  String s = str.substring(0,3);
  return s;
}
```

But suppose the **input** parameter has **FEWER** than 3 letters.
For example: "AB" or "A".
When you run the program with these inputs, it will "blow up" when it tries to execute the line:
String s = str.substring(0,3);
because the 2nd parameter **3** violates the rule that both parameters have to be in the range(0,len).

The way to solve this problem is use an **if** statement that will
check whether the string is long enough to use in the substring statement.

```
public String first3Letters(String str) {
  int len = str.length();
  if (len >= 3) {
    String s = str.substring(0,3);
  }
  return s; // ERROR: Can't find s !
}
```

Notice the ERROR: **s** has now become a local variable in the body of the **if** statement.
You have to declare it **outside the curly brackets** so that the **return** statement can see it:

```
public String first3Letters(String str) {
  String s = "";
  int len = str.length();
  if (len >= 3) {
    s = str.substring(0,3);
  }
  return s;
}
```

Now if the string is shorter than 3 letters, the method will return the empty string.
However, the specs for the method don't tell us what to do in this case.
Here is a new spec:

public String first3Letters(String str) Given a String, return ONLY its First 3 letters.
If the string has fewer than 3 letters, just return the string itself.

This is easy to implement. Simply initialize **s** with **str** rather than the empty string ("").

```
public String first3Letters(String str) {
  String s = str;
  int len = str.length();
  if (len >= 3) {
    s = str.substring(0,3);
  }
  return s;
}
```

Figure 16: String-1 Problem-Solving Strategies. Introductory Review. substring() Part B.

beginsHow

public boolean beginsHow(String str)

Given a String, return **true** if it starts with "How". **false** if it doesn't.
The method is **case-sensitive**. Capitals DON'T match their lowercase counterparts, i.e. "Abc" does not match "aBC".

Explanation:
Begin completing this method as usual:

```
public boolean beginsHow(String str) {
  boolean begins = false;
  if (_____) {
    begins = true;
  }
  return begins;
}
```

Use the String method **startsWith(String s)** inside the parentheses:

```
public boolean beginsHow(String str) {
  boolean begins = false;
  if ( str.startsWith("How") ) {
    begins = true;
  }
  return begins;
}
```

The method below uses the other version of **startsWith**(String s, **int start**), that has a 2nd parameter indicating at which **position** to start looking.

```
public boolean beginsHow(String str) {
  boolean begins = false;
  if ( str.startsWith("How",0) ) {
    begins = true;
  }
  return begins;
}
```

In the code above, the **0** in startsWith() starts the search at the beginning of the string, with the 1st character.
So **startsWith("abc")** and **startsWith("abc",0)** do the same thing.

Figure 17: String-1 Problem-Solving Strategies. Introductory Review. startsWith() Part A.

You can also start the search at other positions:
1 means start at the **2nd** character
2 means start at the **3rd** character

```
public boolean beginsHowAtIndex1(String str) {
  boolean beginsAtPos1 = false;
  if ( str.startsWith("How", 1) ) {
    beginsAtPos1 = true;
  }
  return beginsAtPos1;
}
```

The method above will match strings like
"**x**How are you?" or "**#**How is that possible?"

Note: you can also use an expression for where to start:

int **len** = str.length();
int **mid** = str.length()/2;
len-1 means start at the last character
len-2 means start at the 2nd-to-last character
len-3 means start at the 3rd-to-last character
mid means start at the middle character*
*If the string has an even length, there are two middle characters
and **mid** is the position of the 2nd one.

For example:
str.startsWith("mm", mid) matches "rec**mm**end"
str.startsWith("mm", mid-1) matches "su**mm**er"
str.startsWith("ed", len-2) matches "trust**ed**" [an alternative to **endsWith()**!]

Finally, note that -- unlike with **substring(int start, int stop)**,
which has restrictions on the range of values for its **start** and **stop** parameters --
there are **NO RESTRICTIONS** on the range of values for the **start** parameter of **startsWith()**.
That is, a negative nunber or one larger than the length of the string will **NOT** cause a runtime error.

This **internal** error checking means that your programs can often be much easier to write
using **startsWith()** than those that use combinations of **length()**, **substring()** and/or **equals()**.

beginsHow("How are you?") → true
beginsHow("how do you do?") → false
beginsHow("Who are you?") → false

Figure 18: String-1 Problem-Solving Strategies. Introductory Review. startsWith() Part B.

SUMMATIVE ASSESSMENTS

At the end of each section, students are given a summative assessment using a custom Codingbat page consisting of ten problems that are novel variations on those in the regular section. Generally, seven of these are straightforward, two are of medium difficulty, and one is difficult. The test is open-book/open-computer, and students are encouraged to analogize from the problem-sets they have completed whenever possible. Sample problems include the following.

STRING-1

String conCat2(String a, String b): Returns a concatenation of the two strings. The shorter string will PRECEDE the longer string. If the strings are the same length, return b + a.

String reverse(String str): Given a string with 3 characters, return a string with the letters in reverse order.

String dropMiddleChar(String str): Given a string whose length is odd, return the same string WITHOUT the middle letter.

String goodBad(String str): If the string starts or ends with "good", return a string with the letters "good" replaced by the letters "bad". If "good" does not appear in the string, just return the string unchanged.

boolean nearEnd(String str). Returns true if str *nearly* starts with "123" and *nearly* ends with "789". If you ignore the first character, the string "a123bcd" *nearly* starts with "123". If you ignore the last character, the string "the789z" *nearly* ends with "789".

LOGIC-1

Boolean diceDoublesOrLucky7(int n1, int n2). A dice cube has 6 sides, with the numbers 1-6. Given two integers that represent a pair of dice, return true if you rolled doubles, i.e., if the two dice show the same number. Also return true if the dice numbers add up to 7. Otherwise return false.

Boolean digitsEqualOrTwice(int num). Given a 2-digit integer (between 10-99 inclusive), return true if both digits are the same, or if one digit is twice the value of the other. NOTE: n / 10 will extract the left digit (the digit from the tens-column). n % 10 extracts the right digit (the digit from the ones-column).

Boolean consecutiveOrder(int a, int b, int c). 3 numbers are consecutive if they are in ascending (increasing) numerical order and each successive number differs from the previous number by 1. Examples of consecutive numbers are: 1,2,3 or 9,10,11. Return true if the 3 numbers are in consecutive order.

int largestOf3(int n1, int n2, int n3). Given 3 numbers, return the largest value.

int middleOf3(int n1, int n2, int n3). Given 3 numbers, return the median, which is the middle value. If there is no middle value, as in the case where two numbers are the same, return the value of the number that appears twice.

SUMMARY

The SLA strategies and resources discussed in this article are being integrated, on an ongoing basis, into the UCOP-approved pre-APCS-A Computer Programming course mentioned earlier. A Schoology (Learning Management System) course containing all instructional materials has also been written.

The author would be happy to provide hyperlinks and access to the Schoology course; Codingbat custom sections and assessments; and other support materials/resources, to all interested CS0 and CS1 instructors and researchers. If they were to find these useful and would like to collaborate in conducting statistical studies to confirm their efficacy, that would be very much welcomed. ❖

References

1. Barlow-Jones, G. and Chetty, J. Bridging the Gap: The Role of Mediated Transfer for Computer Programming. *2012 4th International Conference on Education Technology and Computer* (ICETC 2012); http://www.ipcsit.com/vol43/001-ICETC2012-C0019.pdf. Accessed 2020 Feb 26.
2. Bennedsen, J. and Caspersen, M. E. Persistence of elementary programming skills. *Computer Science Education* 22, 2 (June 2012), 81-107; https://doi:10.1080/08993408.2012.692911.
3. Code.org; https://code.org/educate/csp. Accessed 2019 Nov 6.
4. College Board. AP Computer Science Course and Exam Description (Fall 2017); https://apcentral.collegeboard.org/pdf/ap-computer-science-principles-course-and-exam-description.pdf. Accessed 2020 Feb 26.
5. Fink, S. Don't Rely on Cute Apps and Games to Teach Coding. Turn to Your Students Instead. *EdSurge* (Jan 18, 2020); https://www.edsurge.com/news/2020-01-18-don-t-rely-on-cute-apps-and-games-to-teach-coding-turn-to-your-students-instead. Accessed 2020 Feb 26.
6. Floyd, B., Santander, T. and Weimer, W. Decoding the representation of code in the brain: An fMRI study of code review and expertise. In *ICSE '17 Proceedings of the 39th International Conference on Software Engineering*, (2017), 175–86; https:// doi: 10.1109/ ICSE.2017.24.
7. Girls Who Code. Girls Who Code Annual Report 2018; https://girlswhocode.com/2018report/gwc-annualrep2018.pdf. Accessed 2020 Feb 26.
8. Girls Who Code; https://girlswhocode.com/research/. Accessed 2020 Feb 26.
9. Goode, J., Chapman, G., and Margolis, J. Beyond Curriculum: The Exploring Computer Science Program. *ACM Inroads* 3, 2 (2012), 47-53.
10. Gray, J. CS Principles Update: A Statewide Model for Deployment of CS Principles. *CSTA Voice* 9, 2 (May 2013), 6.
11. Institute of Education Sciences. *The Condition of Education 2019* (NCES 2019-44). National Center for Education Statistics; https://nces.ed.gov/pubs2019/2019144.pdf. Accessed 2020 Feb 26.
12. Kliebard, H. In Kliebard, H. *The Struggle for the American Curriculum 1893-1958*. (New York and London, RoutledgeFalmer, 2004).
13. Kummerfeld, S. K. and Kay, J. The neglected battle fields of syntax errors. *ACE '03: Proceedings of the fifth Australasian conference on Computing education*. Vol. 20 (January 2003), 105–111.
14. Margolis, Jane and Fisher, Allan. *Unlocking the Clubhouse: Women in Computing*. (Cambridge, MA, MIT Press, 2002).
15. Margolis, J. *Stuck in the Shallow End: Education, Race, and Computing*. (Cambridge, MA, MIT Press, 2008).
16. Margolis, J., Ryoo, J., Sandoval, C., Lee, C., Goode, J., and Chapman, G. Beyond Access: Broadening Participation in High School Computer Science. *ACM Inroads* 3, 4 (2012), 72-78.
17. Mattern, K., Shaw, E., and Ewing, M. *Advance Placement Exam Participation: Is AP Exam Participation and Performance Related to Choice of College Major?* (New York, NY: College Board, 2011); https://files.eric.ed.gov/fulltext/ED561044.pdf. Accessed 2020 Feb 26.
18. Monge, A.E., Fadjo, C.L., Quinn, B.A., and Barker, L.J. EngageCSEdu: Engaging and Retaining CS1 and CS2 Students. *ACM Inroads* 6, 1 (2015), 47-53.
19. Morgan, R., and Klaric, J. *AP Students in College: An Analysis of Five-Year Academic Careers*. (New York, NY: College Board, 2007); https://files.eric.ed.gov/fulltext/ED561034.pdf. Accessed 2020 Feb 26.
20. National Science Foundation. *Award Abstract #1240944: CS 10K: A Model for Statewide Deployment of CS Principles Courses* (2013); https://www.nsf.gov/awardsearch/showAward?AWD_ID=1240944. Accessed 2020 Feb 26.
21. Passmore, T. Polya's Legacy: Fully Forgotten or Getting a New Perspective in Theory and Practice? *Australian Senior Mathematics Journal* 21, 2 (2007); https://files.eric.ed.gov/fulltext/EJ779108.pdf. Accessed 2020 Feb 26.
22. Portnoff, S.R. The Introductory Computer Programming Course is First and Foremost a LANGUAGE Course. *ACM Inroads* 9, 2 (2018), 34-52. https://doi: 10.1145/3152433: https://dl.acm.org/citation.cfm?id=3152433. Accessed 2020 Feb 26.
23. Portnoff, S.R. M.S. Thesis. CSULA. (1) The Case for Using Foreign Language Pedagogies in Introductory Computer Programming Instruction (2) A Contextualized pre-AP Computer Programming Curriculum: Models and Simulations for Exploring Real-World Cross-Curricular Topics. *ProQuest, LLC*, 2016, 262; 10132126; https://pqdtopen.proquest.com/pubnum/10132126.html?FMT=AI. Accessed 2020 Feb 26.
24. Schnabel, B. Recent Good News on Participation and Opportunities for Young Women Studying Computer Science. *The Advocate* (CSTA, Sept 18, 2019); http://advocate.csteachers.org/2019/09/18/recent-good-news-on-participation-and-opportunities-for-young-women-studying-computer-science/. Accessed 2020 Feb 26.
25. Seehorn, D., Carey, S., Fuschetto, B., Lee, I., Moix, D., O'Grady-Cunniff, D., Boucher Ow-ens, B., Stephenson, C., Verno, A. *CSTA K-12 Computer Science Standards*. Revised 2011. The CSTA Standards Taskforce. (ACM, New York, NY, 2011). http://scratch.ttu.ee/failid/CSTA_K-12_CSS.pdf. Accessed 2020 April 6.

26. Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., and Saake, G. Understanding source code with functional magnetic resonance imaging. *ICSE 2014. Proceedings of the 36th International Conference on Software Engineering*. (June 2014), 378-389. (ACM, New York, NY, 2014); https://doi: 10.1145/2568225.2568252.
27. Siegmund, J., Apel, S., Begel, A., Peitek, N., Hofmeister, J., Bethmann, A., Parnin. C., Kästner, C., and Brechmann. A. Measuring Neural Efficiency of Program Comprehension. *ESEC/FSE 2017 Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. (August 2017): 140-150. (ACM, New York, 2017); https://doi:10.1145/3106237.3106268.
28. Tate, E. How 60 Minutes Oversimplified the Gender Gap and Overlooked Women in Tech. *Edsurge/Diversity and Equity* (March 7, 2019); https://www.edsurge.com/news/2019-03-07-how-60-minutes-oversimplified-the-gender-gap-and-overlooked-women-in-tech. Accessed 2020 Feb 26.

Scott R. Portnoff
Downtown Magnets High School, Computer Science Dept.
1081 W Temple St., Los Angeles, CA 90012
srport@alum.mit.edu

DOI: 10.1145/3381026