The Introductory Computer Programming Course is First and Foremost a *LANGUAGE* Course

By Scott R. Portnoff, *Downtown Magnets High School, Los Angeles, CA*

An fMRI (functional Magnetic Resonance Imaging) study published in 2014 established that comprehension of computer programs occurs in the same regions of the brain that process natural languages—not logic, not math. The unexpectedness of this result was primed in part by the widespread belief that the language aspects of learning how to program are trivial when compared to learning to use programming languages for engineering tasks. In fact, though, the fMRI data is compelling cognitive evidence for the argument that the reason students have been failing introductory programming courses in large numbers—for decades—is because CS educators have underestimated the importance of teaching programming languages as languages per se. Despite the availability of this non-invasive technology for well over two decades, educators have neither researched the cognitive complexities of how programming languages might be acquired, nor tried to seriously understand this process in any degree of depth. Consequently, they have failed to consider what this evidence now implies: (a) 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; and (b) that this new information about the cognitive aspects of programming languages has profound pedagogic implications.

The title of this article—*The Introductory Computer Programming Course is First and Foremost a LANGUAGE Course*—may seem provocative, but the truth is that language instruction is already a component of the introductory programming course. The problem, however, has been the *specific pedagogic* approach for teaching the language aspects of the course—a barebones and long-abandoned teaching paradigm for natural languages modeled after a *Prescriptive Linguistics*[1] approach (explicit rule-based grammar instruction), rather than one that incorporates principles of *Second Language Acquisition (SLA)*[2] Theory (implicit repetitive exposure to language data in meaningful contexts). In the 21st century, there are virtually no natural language classrooms that utilize the Prescriptive Linguistics approach—including those that teach Esperanto and Latin—yet this continues to be the universal instructional model for programming language instruction. Such a model could only make sense if applying syntactic rules was in fact an effective way to learn the grammars of programming languages, a belief that has never been corroborated. Indeed, studies in the literature going back decades overwhelmingly point to the opposite, documenting droves of novice programming students unable to stop making even the most basic of syntax errors [17].

[1] A ***Prescriptive*** grammar is a set of explicit rules for using a language, in most cases taught to enforce uniform usage.

[2] ***Second Language Acquisition*** is the study of how people learn a second language, i.e., how they acquire the capacity to *verbally* comprehend and communicate fluently in a second language.

Even those talented enough to somehow manage to get their programs to run and function properly still compose awkwardly structured programs well into their first year. … Ironically, CS educators are the last holdouts of a language learning strategy for our students that we no longer even employ for the machines at the center of our discipline.

Originally borrowed from foreign language pedagogies used prior to the 1960s, the Prescriptive Linguistics model became the default, unquestioned, and sole methodology used in classes from the earliest days of programming education, even as foreign language instruction itself shifted to other much more effective implicit methods. The fallout from this history has left our students with a conceptual pedagogic gap to bridge on their own, with instructors nevertheless expecting them to magically solve problems using logic mediated by a language in which most struggle to express basic fluency. Even those talented enough to somehow manage to get their programs to run and function properly still compose awkwardly structured programs well into their first year.[3]

It might seem counterintuitive that the small syntactic footprints of programming languages, with their relatively simple and compact grammars, would translate into a lengthy and involved learning process. The specific difficulties in learning a programming language, though, simply differ from those of learning a natural second language. The complications stem in large part from the very small number of control structures that programming languages employ—although infinitely adaptable, they are at the same time semantically broad, diffuse, and ambiguous.

The pedagogic model I have been developing in my high school classroom over the last few years has attempted to remedy this situation by utilizing *implicit* strategies for programming language instruction. A key observation that has consistently come out of these efforts is the effectiveness of *memorization*, an implicit language learning strategy. Specifically, memorization of short programs seemingly overnight facilitates the acquisition of the basic *syntactic* features necessary to avoid compiler errors—an early and stubbornly persistent obstacle for many programming students [17], particularly those in high school. Acquisition of the much larger *semantics* piece—which can be assessed by the competent and efficient application of basic programming concepts—can also be facilitated by the addition of implicit repetitive strategies, but the process is lengthier and gradual, with a timeline mirroring that of natural languages.

Although three pedagogic strategies informed by SLA principles and described and published previously are referenced, this article focuses on how one might go about teaching an arbitrary language feature within a teaching paradigm that adapts principles of a whole-language communicative approach, using *conditional execution* as an example. The article concludes with a description of a secondary curricular model whose unit/chapter content structure is informed by *principles* and *objectives* that underlie second language instructional strategies—adapting, but not using, the instructional strategies themselves, owing to the fundamental and substantial differences between natural and programming languages.

On a personal note, over the ten or so years of teaching my introductory (pre-APCS-A) programming course, I have employed a variety of curricula and tools: Scratch, Alice, cross-curricular problem-solving, Codingbat, Exploring Computer Science, PLTW CSE (CS Principles) and so on. I would be hard-pressed to say that I had much success in reaching most of my students, let alone the traditional demographic of "high-fliers." Despite the innovative and engaging nature of many of these curricula (specifically, the first four), what they all recapitulated was the default explicit language model. Over the last several years, though, it has become progressively evident that implicit language strategies are pivotal, allowing many more of my students access to the curriculum, including clearer conceptual learning for the higher achieving ones as well.

This is not to claim that the explicit language model is patently ineffective—obviously there are talented students who manage to become successful programmers (and instructors); however, I believe they manage to do so despite this type of language instruction. My experience has been that an implicit language teaching model simply reaches more learners. Both the fMRI study and the anecdotal observations of the positive effects of implicit SLA-based instructional strategies reported here are evidence arguing for an alternative instructional model that could potentially have far-reaching outcomes for programming curricula and instruction, particularly at the secondary level.

PART I: FMRI EVIDENCE FOR THE CENTRAL COGNITIVE ROLE OF LANGUAGE IN PROGRAMMING

A research study with mammoth implications for Computer Programming education, published in 2014, was by and large ignored by the Computer Science education community. Not

[3] Even computer scientists working on language-related problems (most notably *Google Translate*) have traded explicit rule-based approaches for implicit data/machine learning algorithms because of the former's inferior outcomes [19]. Ironically, CS educators are the last holdouts of a language learning strategy for our students that we no longer even employ for the *machines* at the center of our discipline.

dismissed, but ignored, as if it were so peripheral to the thinking of CS educators that they could conceive of no place within their conceptual framework from where they might even begin a discussion about it. "Understanding Understanding Source Code with Functional Magnetic Resonance Imaging" reported the results of a study led by Prof. Janet Siegmund of the University of Passau (Germany), in collaboration with researchers at Carnegie-Mellon University (CMU), Georgia Institute of Technology (Georgia Tech) and three research institutions in Magdeburg (Germany) [30]. It describes a meticulously crafted and executed series of experiments using fMRI[4] to measure the brain activity of undergraduate programming students as they tried to understand[5] what small Java programs did. The researchers found strong activations in brain regions that are specifically involved in ***language*** processing tasks—Brodmann Areas 6, 21, 40, 44 and 47 (Table 1).

At the same time, brain regions associated with processing math and logic tasks were minimally activated, as Georgia Tech Prof. Chris Parnin, the study's fourth author, summarized.

> The team found a clear, distinct activation pattern of five brain regions, which are related to language processing, working memory, and attention. The programmers in the study recruited parts of the brain typically associated with language processing and verbal oriented processing (ventral lateral prefrontal cortex). At least for the simple code snippets presented, programmers could use existing language regions of the brain to understand code without requiring more complex mental models to be constructed and manipulated.
>
> Interestingly, even though there was code that involved mathematical operations, conditionals, and loop iteration, for these particular tasks, programming had less in common with mathematics and more in common with language. Mathematical calculations typically take place in the intraparietal sulcus, mathematical reasoning in the right frontal pole, and logical reasoning in the left frontal pole. These areas were not strongly activated in comprehending source code. [27]

What is initially remarkable about these findings is that the brain appears to process both programming languages and natural languages in a similar manner, despite their significant differences. Per the Chomskyan-Schützenberger hierarchy of formal grammars, programming languages are generally classified as Type-2 grammars, which generate what are known as *context-free* languages. Natural human languages—termed *regular* languages in this hierarchy—are classified as Type-3 grammars, a subset of Type-2 grammars. This overlap may be the reason

[4] fMRI (Functional Magnetic Resonance Imaging) is a noninvasive technique that measures precise differences in blood-oxygen levels in the brain—the BOLD (Blood Oxygen Level Dependent) contrast effect. The BOLD effect results from the shifting magnetic properties of the iron atoms present in hemoglobin, depending upon whether the molecule is binding oxygen or not. Neuronal activation is accompanied by an increase in both blood flow and arterial oxygenated (non-magnetic) hemoglobin to the affected brain regions, displacing venous blood containing deoxygenated (highly magnetic) hemoglobin. The non-magnetic oxygenated blood interferes less with the magnetic resonance signal; hence a measurable difference can be detected. There is a lag of 1 to 2 seconds after the triggering event, with the effect peaking at 5 seconds and dissipating at 12 seconds. When neurons continue to be activated due to an ongoing stimulus, the peak broadens to a plateau. After dissipation, the BOLD signal falls below pre-activation levels (the undershoot effect), but eventually rises back to pre-activation levels.

Experimental protocols need to consider various complicating factors. To accurately associate brain regions with cognitive activity requires longer tasks (1-2 minutes) to generate plateaus. Tasks need to be repeated and measurements averaged out to ensure that a threshold statistical confidence level is attained. Activation artifacts from motion must also be minimized, e.g., the head must be kept still, and subjects must refrain from moving their jaws. A critical piece is designing control tasks that filter out baseline cognitive processes—such as visual processing activations in the study cited above—that have minimal, and ideally no, overlap into the cognitive process(es) of interest.

[5] The researchers distinguished between (a) top-down comprehension, where readers are familiar with a program's domain and can pull in knowledge about that domain, and (b) bottom-up comprehension, where programmers encounter a program cold for the first time and must rely on decoding the program line-by-line. The focus of the study was on the latter.

Brodmann Area	BA 21	BA 44	BA 47
Location	Middle temporal gyrus	Inferior frontal gyrus	Inferior frontal gyrus
Cognitive Tasks	Semantic memory retrieval Categorization Linguistic inference Word form processing Recognizing words with similar meanings	Syntactic processing Grammatical processing Sentence comprehension	Semantic processing Coding and retrieval Phonological processing Reading single words Lexical inflection Intonation aspects of prosody
Brodmann Area	BA40	BA 6	
Location	Inferior parietal lobule	Middle frontal gyrus	
Cognitive Tasks	Semantic processing Attention to phonological relations Working memory	Word retrieval Reading novel words Phonological processing Lexical decision on words/pseudo-words	Language processing Working memory

Table 1. 5 Brodmann Areas activated in the 2014 Siegmund study, and known mapped cognitive tasks.

for the brain recognizing the programming language features evaluated in the Siegmund study, including loop iteration, as language. Because the programs studied were no longer than 20 lines, not all programming language features were tested (e.g., method calls and recursion), and it may be the case that certain programming language features will require other mental models besides language. Nevertheless, it appears safe to assume that the basic components of programming language grammars are processed cognitively as language.

Long before fMRI was routinely used to map cognitive tasks to specific brain regions, the major language processing regions of the brain (including the ones implicated in the Siegmund study) had already been mapped—identified anatomically in postmortem patients with aphasia disorders—injuries to the brain that affect expressive/productive and/or receptive/comprehension-related aspects of language processing. The confirmation and additional detail of these mapped language functions provided later by advances in brain surgery (1950s) and fMRI technology (1990s) make these among the most reliably known areas of the brain.

Two methodologies used in the Siegmund study vouch for the validity of its results. First, the selection of the control *task* used to reduce background activations—in this study, asking subjects to identify syntax errors in virtually identical programs—was tested in pilot experiments along with other candidate tasks, and found to be a good compromise between its resemblance to, and dissimilarity from, the experimental *task* of trying to understand—make sense of—the program. If activations of language regions occurred due to incidental or pedestrian aspects of the task, these would factor out when activations from the control task were subtracted. In the words of the research team:

> The most difficult issue in fMRI studies and most other studies that evaluate cognitive processes is to select suitable material and tasks (that is, source code in our case) and devise control tasks that control for brain activation elicited by processes that are needed for programming, but are not specific for it, such as reading itself. It is imperative that source code and tasks without a doubt lead participants to use the cognitive process that is the target of the evaluation, because otherwise, we cannot be sure what we measure (i.e., ensure construct validity). [30]

In a subtraction protocol, the control task acts like a mask to filter out activations not specific for the cognitive task one wants to isolate. Interestingly, other fMRI protocols have been constructed for answering questions beyond the sole mapping of cognitive tasks. In one case, to investigate the subtler distinction of whether syntax is "autonomous" from other features of grammar, Moro devised a protocol that could distinguish between syntactic errors and errors unrelated to syntax. He found that specific nets/networks of interacting regions—not mutually exclusively regions themselves ("pure locationism")—were associated with the different kinds of errors [25].

Support for the assertion that programming language features are not just language-like, but actual language came from a recent fMRI study which found that even operations appearing to resemble those of language will not activate language-processing regions. This study sought to answer the question "*whether natural language provides combinatorial operations that are essential to diverse domains of thought.*"

> We addressed this issue by examining the role of linguistic mechanisms in forging the hierarchical structures of algebra. In a 3-T functional MRI experiment, we showed that processing of the syntax-like operations of algebra does not rely on the neural mechanisms of natural language. Our findings indicate that processing the syntax of language elicits the known substrate of linguistic competence, whereas algebraic operations recruit bilateral parietal brain regions previously implicated in the representation of magnitude. *This double dissociation argues against the view that language provides the structure of thought across all cognitive domains.* [23]

The second methodology used by the Siegmund study that buttresses its reliability is related to a major statistical error affecting the validity of fMRI data analysis reported by Eklund et al. two years later [12]. The error resulted from a fallacy in the correction for *multiple comparisons* in the fMRI process. Studies that had used any of three common software packages for fMRI analysis yielded false-positive rates up to 70%, raising concerns about the reliability of the over 40,000 already published fMRI studies. The Siegmund researchers were unaffected by this error for two reasons: (a) they had used analysis software that avoided these problems; and (b) they performed a further statistical correction to guard against any false discovery rate. Details can be found in the *Data Preparation* and *Analysis Procedure* sections of their paper [30].

Regarding implications, the results of the Siegmund study have a direct bearing on what is known as the *novice programmer* failure problem. For four decades, the literature has documented large numbers of novice programmers failing post-secondary introductory programming courses [3,18,33], so much so that the problem has been called "*one of the seven grand challenges of computing education.*" [22] Indeed, it might be said that the phenomenon is not so much a problem as a feature of these courses. It is no exaggeration to say that the situation is many times worse at the secondary level, where the majority of students steer clear of such classes—2017 enrollments in the AP Computer Science A course were not even one-fifth the numbers of those who took AP Calculus [7], itself an elite course enrolling less than 10% of all high school seniors. Virtually all instructors of introductory programming courses have firsthand experience with this "grand challenge," finding themselves both unable to explain how the "high-fliers" in their classes seem to understand programming from the get-go, while simultaneously at a loss to provide effective help for those who seem to struggle perpetually. CS educators have

long blamed student failure on a lack of talent or work-ethic, despite the decades-long inability of researchers to pinpoint the involvement of such traits, or the inconsistent and inconvenient fact that their failing students may excel in other subjects. Only rarely have instructors considered the possibility that the fault may lie with their own instruction.

Siegmund's team, however, recognized the pertinence of their findings to this long-standing problem.

> …our research will have a broad impact on education, so that training beginning programmers can be improved considerably. Despite intensive research (e.g., Technical Symposium on Computer Science Education, Innovation and Technology in Computer Science Education), it is still rather unclear how and why students struggle with learning programming. With a detailed understanding of the cognitive processes that underlie a developers' every-day task, we might find the right recipe to teach any student to become an excellent software developer (e.g., by including training language skills, since our study showed a close relationship to language processing). [31]

PART II: IMPLICIT LANGUAGE LEARNING STRATEGIES

Beginning in 2013, I began to alter instruction in my introductory (pre-APCS-A) programming course by requiring my inner-city high school freshman to memorize short programs and program fragments. What I observed is that they stopped making syntax errors seemingly overnight. This also resulted in an upswing in overall motivation in all students, but particularly in the ones who typically would have failed to learn much of anything at all. This phenomenon is anecdotal, but dramatic and unmistakable, and has been consistent year after year. This instructional strategy can be understood within a theoretical context of implicit language acquisition (discussed below).

I therefore offer the conjecture that the cause of the persistent novice programmer problem is not some mysterious defect in our students, but rather a gaping defect in the pedagogic model, one that has discounted the importance of language acquisition issues. No one would expect algebra students who lacked foundational mathematics skills to be able to solve simultaneous equations. It is equally as ludicrous to expect students who have trouble acquiring basic literacy in a programming language to be able to compose programs whose logic is mediated by that same *language*. The corollary is that a pedagogy that can devise effective language acquisition strategies while accounting for the peculiarities of an artificial language should allow the clear majority of students the possibility of becoming proficient programmers.

CMU Professor Christian Kästner, the Siegmund study's second author, stated in an interview:

> There is no clear evidence that learning a programming language is like learning a foreign language, but our results show that there are clearly similarities in brain activations that show that the hypothesis is plausible. [1]

Kästner was, of course, both cautious and modest in his claims. CS educators, however, currently operate with no evidence-based cognitive model for how students learn to program. When partial models have been invoked, they have generally *presupposed* the involvement of psychological constructs—such as that "cognitive loads" are lowered with drag-and-drop programming interfaces like *Scratch* or *Alice—without* having done research (i.e., taking experimental measurements) to corroborate such assumptions. Likewise, it has been assumed—but never demonstrated—that programming concepts and skills learned using tools like *Scratch* or *Alice* transfer to text-based languages.[6]

Given the results of the Siegmund study, however, a cognitive model for how students learn to program can now be built on how the language regions of the brain acquire programming languages. For instructors, knowing that fluent programmers process programming languages like natural languages, pedagogies can be crafted to attempt to facilitate this physiological outcome. A natural starting point would be an exploration of both *Second Language Acquisition* theory and the implicit instructional strategies currently used by foreign language instructors. A note of caution—although second language educators have decades of experience and knowledge about how their students learn and the obstacles they encounter, there are major differences between natural languages and programming languages that make instructional strategies used in the foreign language classroom impossible to use (these differences are discussed below). That said, many of these strategies can be adapted; that is, there is no reason why the cognitive language *goals* and *principles* underlying those strategies cannot inform introductory programming language instruction in valuable ways.

In addition, fMRI technologies may theoretically provide a way to cognitively measure whether new (and old) instructional strategies facilitate the acquisition of programming language proficiency. Several recent studies support a model for learning second languages in which brain organization changes/evolves over time. Moderately proficient language learners at first recruit anomalous regions of the brain not ordinarily used to routinely process language. In contrast, highly proficient second language learners utilize the same regions for both native languages and second languages [8,15,35]. A very recent study that investigated the distinct neuronal activations differentiating code review, prose review, and code comprehension tasks also found a similar pattern—that the brains of participants with greater programming expertise treated programming languages more like natural languages [13]. Should further research confirm such patterns for programming language learners, we should be able to not just demonstrate that

[6] "It [computing] won't make you better at something unless that something is explicitly taught, said Mark Guzdial, a professor in the School of Interactive Computing at Georgia Tech who studies computing in education. 'You can't prove a negative,' he said, but in decades of research no one has found that skills automatically transfer." [26]

novice programmers increasingly utilize brain regions classically associated with language as they become more proficient, but also be able to use that model to assess the effectiveness of instructional strategies.

A cognitive model that emphasizes language acquisition would thus have great explanatory potential, allowing researchers to make and test predictions. Such a model does, however, come with several significant challenges. The least difficult of these are scheduling changes, specifically lengthening the amount of time allotted to students to become proficient in the use of a programming language. The reason is simple enough—it is unrealistic to expect significant competence in any language after a single college semester or year-long high school course. An obvious corollary would be to teach one language and one language only until proficiency is acquired (2-3 years in high school, 1-2 years in college). The not uncommon practice of switching programming languages every semester makes about as much sense as a program of instruction that mandates one semester of French, followed by a semester of German, then a semester of Russian. Indeed, the designer of such a curriculum would be sacked for the obvious—that it is impossible for students to acquire even bare competence in any of these languages in such a short period. The fact is, subsequent languages are easier to learn having first gone through the process of comprehensively learning a single second language. The case is even stronger for programming languages— because they all implement the same set of control and data mechanisms in very similar ways, the task of learning a second programming language for those with in-depth knowledge of a first programming language is more like learning a dialect than an entirely new language.

A much more challenging matter is unearthing the kinds of learning conditions and implicit instructional practices that might facilitate the process by which programming languages are better acquired. In fact, a working definition of such pedagogies would be those that facilitate programming languages taking up physical residence in the language processing regions of a novice programmer's brain. In addition to this physiological metric, such strategies should also facilitate performance metrics like fluency/automaticity.

The current Prescriptive Linguistics model of programming language instruction generally proceeds as follows: (a) the instructor explains the syntactic/grammatical rules associated with data and control structures; (b) she works through examples that demonstrate language usage; and (c) she provides students practice via problem-solving exercises. However, this *explicit* instructional model is not one that has been found to help most learners acquire and become proficient users of a second language.

Learning one's primary/native language—whether spoken or signed—occurs *implicitly* (passively, subconsciously, automatically) through repetitive exposure to language data and meaningful interaction with other speakers. It does not occur through the *explicit* (conscious, active, intentional) inculcation of and practice in applying linguistic rules. There is widespread agreement among SLA theorists that *second languages are learned implicitly as well*, the age of the learner notwithstanding.[7] The role of explicit grammar instruction in foreign language instruction is supplemental: (a) as reinforcement for implicit learning; (b) to help second language learners correct language features that were imperfectly learned; and (c) as an aid to improve literacy. Exclusively explicit foreign language teaching models, indistinguishable from those CS educators currently use, were abandoned over three decades ago, with implicit communicative and immersive learning taking their place. A considerable amount of evidence has also accumulated showing that implicit language instruction results in conscious linguistic knowledge in addition to automaticity [14].

An obvious corollary would be to teach one language and one language only until proficiency is acquired ... The not uncommon practice of switching programming languages every semester makes about as much sense as a program of instruction that mandates one semester of French, followed by a semester of German, then a semester of Russian.

Human language is an innate, highly specific cognitive ability quite distinct from general intelligence. Evidence supporting this view includes (as already mentioned) brain localization of language processing; pathologies and genetic disorders that specifically target language without affecting general intelligence; and the opposite—severe retardation that leaves linguistic abilities intact; and an optimal age-related developmental window after which language learning becomes more effortful and difficult. As such, languages are said to be acquired, not learned [16].

The difficulty with trying to adapt communicative instructional approaches for programming languages is that, because they exist solely in written form—i.e., they are unspoken—there are no communities of speakers to interact with. As such, mechanisms comparable to those for acquiring natural languages simply do not exist. How then do those students who become proficient programmers manage to acquire programming languages,

[7] For second languages, "early word and grammar learning relies on declarative memory (and more explicit processes), but that grammar later relies on procedural memory (and more implicit processes)." [32]

seemingly without great difficulty? However it occurs, an SLA-based cognitive model would predict—because repetitive exposure and meaningful interaction appear to be the sole route by which natural languages are acquired—that these principles must somehow operate in how these students learn as well.

One example that corroborates the effectiveness of implicit language instruction is the memorization teaching strategy mentioned earlier. Note that in employing this strategy, the only direction given students is that they memorize the material perfectly—they are given no overt instruction for how to form syntactically correct statements. Nevertheless, the result is that students become able to compose programs without the kind of *syntax errors* that lead to compiler errors, the earliest obstacle reported for novice programmers [17]. The following is a possible mechanism. To memorize a program or program fragment perfectly, students must employ numerous cycles of (a) reading the material and (b) writing it out without looking. This process bombards their brains repeatedly with idealized language data. As with natural languages, the brain subconsciously generalizes from the patterns in the data and *implicitly* constructs the syntactic rules (the grammar) for the programming language [28]. Those who would dispute this kind of implicit mechanism would need to explain how memorization subsequently allows students to (a) compose new programs with syntactically correct sequences of programming statements, as well as (b) identify syntax errors in novel programs.

It is important to note that it is not at all being claimed that other pedagogies used in programming education should be abandoned—although the literature for the past four decades has not revealed *any* instructional strategies to have made a dent in the novice programmer failure problem. Rather I hope to encourage others to modify the programming language course to initially use implicit pedagogies focusing on language in order to lay a linguistic foundation on top of which traditional pedagogies that focus on logic and problem-solving can subsequently take root. I myself limit the use of *most* SLA-based instructional strategies to the first year—the introductory (pre-APCS-A) programming course. Students who continue in the subsequent *APCS-A* course, although far from being fluent programmers, have acquired enough of a comfort level with Java that traditional strategies can be used—with the caveat that my problem-solving philosophy is informed by an SLA-based teaching framework, as described in Part V.

The Siegmund study unambiguously documented the first neurocognitive evidence pertinent to CS programming education. Although the results should have precipitated a frenzy of ongoing discussion and inquiry, they were instead met with a blip of interest that quickly dissipated. A few follow-up studies by other researchers are only now beginning to trickle in, but crucially, there has been no effect on CS education. CS educational pedagogy has from the start been modeled after mathematics instruction, with a primary focus on problem-solving. What has been ignored, however, is that learners must first acquire basic proficiency in the programming language that mediates the logic required for problem-solving, a process that takes

The Siegmund study unambiguously documented the first neurocognitive evidence pertinent to CS programming education.

about one year, currently more time than the CS introductory instructional model provides. Implicit learning strategies have for three decades been the most effective and efficient ways to facilitate the acquisition of both receptive and expressive fluency and automaticity in second languages, and there is much that programming instructors can learn and adapt from foreign language teaching goals and principles. Indeed, models of instruction that fail to give sufficient import to the central cognitive role that language plays for students of programming languages should be considered intellectually suspect. Although a pedagogic model that adds early implicit language instruction would be a partial shift in instructional paradigms, requiring a bit of rethinking as to how we teach the subject, it is not particularly complicated to implement. Such a model would, however, defy CS educators to jettison long-held, but ultimately unfounded assumptions about how their students may learn—and how they themselves might have learned—to become proficient users of a programming language.

PART III: CS INSTRUCTION AT THE SECONDARY LEVEL

Although adding implicit language instruction may increase and widen access to the introductory postsecondary programming curriculum, the greatest potential for progress lies at the secondary level, where, aside from elite school settings, CS education has been an ongoing and futile cycle of implementing—and then discarding—one ineffective curriculum after another. The failure has been due to the near exclusive focus of innovation on the *content* side of curriculum, with little thought given to how *methodologies* can effectively deliver that content. While the potential of a curriculum's content to generate interest is vital, if that content cannot be delivered in a way that will impart *self-efficacy*—a student's confidence in her ability to compose novel working programs on her own—no amount of interest will suffice to convince her to continue study in the field.

Since Sputnik, secondary education has offered an increasing number of college-level courses in math and science as a way to jump-start study of those fields before college. In recent years, and with goals similar in spirit to the social efficiency education movement of the first half of the 20th century, the CISE Directorate in the National Science Foundation (NSF) has promoted the idea that high school CS education is both a bridge to the study of CS in college and a critical part of a pipeline for ultimately filling a huge number of domestic computing job vacancies.

The APCS-A course in fact fits these goals—longitudinal studies completed by the College Board in the past decade have confirmed the course's effectiveness as a bridge to CS study in college [21,24]. There have, however, been long-standing problems that continue to limit the course's influence. APCS-A may be the equivalent of a one-semester introductory college level class, but at the secondary level it plays out in practice as a highly accelerated programming curriculum, even over the course of an entire year. To complicate matters, the course has no programmming prerequisite, attributable in part to the lack of standardization of secondary programming curricula—aside from the APCS-A course itself. Both factors have made it a poor entry point for the vast majority of students, inequitably skewing its demographics towards a subset of the most academically talented, and ensuring low overall enrollment numbers.[8] Notwithstanding, it is, however, an appropriate second year programming language course for those who have taken and passed a rigorous introductory pre-AP *programming* class.

As such, the need has always been for an introductory programming course that aligns vertically with APCS-A, and that utilizes instructional strategies that can credibly prepare students for that next course. Although this may seem unremarkably obvious, an alternate narrative that has emerged during the past decade at the secondary level actively downplays the importance of learning to program. Deemed as a corrective to the "programming-centric" focus of the APCS-A course, the Computer Science Teachers Association (CSTA) codified a framework of five co-equal strands. This new scheme, however, has its problems. Programming (subsumed in a strand called Computer Practice and Programming) was unconvincingly separated from Computational Thinking (algorithms, abstraction, and the like) [10], with the result that some secondary curricula (e.g., Exploring Computer Science) attempt to teach algorithms, such as sorting, *before* students have learned anything about data structures or programming, leading one to question not only the superficial level of rigor of such instruction, but the point of presenting such material at all when it is devoid of meaningful contexts. Moreover, a strand called Collaboration implausibly occupies a space just as important to the study of CS as each of the other four strands. In a mathematics context, this would be the equivalent of arguing that group work—a teaching strategy—is a learning objective equally as vital as the content of any major topic in the Algebra 1 curriculum.

This scheme was promoted despite the broad postsecondary consensus that *programming be the first topic of study in a college CS curriculum* because of its role as the core skill fundamental to the entire discipline, crucial for understanding and plumbing topics—particularly algorithms—in virtually all subsequent coursework on anything beyond a trivial level. In hindsight, CSTA's five-strand framework, an abrupt break from the young organization's past positions, can be seen as an apologetic—a rationale/justification for giving up on the goal of rigorous programming instruction at the secondary level because of the massive failure of educators to effectively teach programming to the vast majority of high school students.[9]

As a replacement, CSTA has instead promoted two secondary *survey* courses, Exploring Computer Science (ECS) and AP Computer Science Principles (AP CSP), whose contents conform to CSTA's new, but questionable, framework. From the start, the rationale for both courses has been to "broaden participation in Computer Science." The two courses, however, have been designed to merely *expose* students to their content and stimulate *interest*, as opposed to delivering substantial and measurable skills that will vertically prepare students for subsequent programming study, the way an Algebra 1 course lays a conceptual foundation that prepares students for the Algebra 2 course. This is a direct consequence of the nature of survey courses, which place emphasis on *coverage* at the expense of *depth*. However, it's unclear that the contents of these courses have much value, in practice resembling not so much a carefully designed sequence of fundamental concepts (as they claim) as a smorgasbord of unrelated topics that can be replaced in the event they prove too difficult for students to learn. In contrast, a survey course might better reflect the field through an overview of the most important contemporary developments in the subfields of Computer Science—Artificial Intelligence, Robotics, Machine Learning, Big Data—emphasizing in particular their real-world applications. Such a course might also include important engineering and cross-disciplinary applications, in areas like DNA Sequencing and Analysis (Bioinformatics), Molecular Modeling, Routing, Astronomy, Linguistics, Journalism and Art. Although in my view this would give students a more realistic and engaging look at the field, it's impossible to really cover these topics in anything other than a very simplistic way if students have no appreciable programming proficiency.[10]

There is also a substantial pedagogic problem with both ECS and AP CSP, in that no level of excitement or interest will convince students to continue study in *any* subject if they don't also have an authentic sense of self-efficacy, a confidence in their ability to correctly apply concepts and skills studied to similar, but novel, situations. Because the courses treat programming as only one of many topics, they unfortunately do as little to prepare students for a subsequent programming course as an introductory course on French culture and literature in translation would for students following up with a second year French

[8] Interestingly, it has been assumed that the demographic inequities are skewed only against females and traditionally underrepresented minorities in favor of white and Asian males. However, when one considers the very low enrollment numbers, a more nuanced—and probably more accurate—view is that the white and Asian males in CS courses are a subpopulation, and that the majority of white and Asian males similarly fail to participate.

[9] For a fuller and more detailed critique of the CSTA framework, see Appendices B and C of reference [28].

[10] In practical terms, what these courses do is eerily reminiscent of the mathematics track system of previous decades in which college-bound students took a math sequence culminating in calculus and pre-calculus, while students not bound for college took a general mathematics sequence terminating with pre-algebra. In this parallel incarnation for secondary CS instruction, one tier (APCS-A) teaches programming while the other (AP CSP and ECS) doesn't, with obvious implications for students' college readiness should they undertake subsequent CS study. Ironically, these survey courses simply recapitulate the very inequities they were intended to eliminate [28].

language class, thereby postponing the many difficulties that they will invariably encounter as novice programmers. The two courses have certainly demonstrated that they can broaden participation, but that participation comes at the cost of academic rigor and preparedness, exchanging these for a watered-down version of CS of questionable utility and relevance.[11]

An introductory pre-APCS-A *programming* course, the first of a two-year programming sequence that culminates in an advanced version of APCS-A, has in fact been taught for years at the highly selective magnet Thomas Jefferson High School of Science and Technology (TJHSST), where freshmen with no programming background take the Foundations of Computer Science pre-APCS A programming course ("classes & objects, loops, if, arrays, files, graphics") and in their sophomore year take APCS plus Data Structures. Note that although these freshmen are highly gifted students conforming to the traditionally inequitable demographic, their instructors have made the judgment that they still require the benefit of a pre-AP programming course to maximize success in APCS-A. The problem remains that, were such a pre-APCS-A programming course to be implemented in less elite educational settings, instructors and students would—again—run up against the novice programmer failure problem because of inadequate pedagogic strategies.

There are also political and structural considerations. Although a host of groups and organizations, such as ACM and Code.org Advocacy Coalition, have argued that CS should be included in the K–12 core curriculum, state boards of education have not been convinced. Part of the problem stems from the boards themselves, which have failed to create CS credentials,[12] resulting in an absence of credentialing programs and a dearth of 9–12 subject-matter competent instructors (i.e., those possessing a B.S. in the field or the equivalent, as has been the norm in countries such as Israel[13]). As such, most public schools still rely on instructors with minimal knowledge of the subject. There are also curricular challenges. The content of secondary CS courses varies enormously, a problem exacerbated by the new survey courses, which have also fractured the long-held consensus about the central place of programming in the secondary CS curriculum. Note that the confusion introduced by these courses could only take root within a secondary teacher workforce whose subject-matter competency is rarely higher than the coursework found in a first-year CS college-level program—and most often not even that. Even were consensus on the primacy of programming to be reestablished, proven pedagogies for *effective* programming language instruction—ones that can ensure that grade-level students can make progress in acquiring programming skills and concepts—remain non-existent. This pedagogic deficiency completes the vicious circle, because whatever teaching techniques and strategies CS credentialing programs might train prospective teachers in would be purely speculative.

In my experience, implicit language instruction appears to be a—if not the—crucial missing piece of programming language pedagogy, and I hope to encourage researchers to investigate this approach.

In my experience, implicit language instruction appears to be a—if not *the*—crucial missing piece of programming language pedagogy, and I hope to encourage researchers to investigate this approach. At the same time, should this prove to be the "magic bullet," I would remind instructors that acquiring proficiency in the use of a second language is a lengthy and gradual process. Utilizing such instruction should produce noticeable and ongoing improvements, but it will not make all students programming wizards overnight. What I witness in my classes, though, is that implicit language instruction, contrary to the traditional Prescriptive Linguistics model, provides the scaffolding necessary for grade-level novice programmers—particularly those who are prone to struggle—to progress and learn.

PART IV: CHOOSING WHICH LANGUAGE FEATURES TO TEACH

Because the syntactic footprints of programming languages are small, it may seem that learning to use them should be uncomplicated and obvious. In my experience, though, only a programming language's basic *syntactic* features, the ones whose errors obstruct compilation, are easily imparted—using the memorization strategy discussed earlier. It takes a much longer time to acquire the *semantics* of a programming language—the knowledge that enables one to write efficient, concise, and clear programs. Generally, such proficiency takes *a minimum of* two years for the most talented students. The difficulties can be attributed to a handful of characteristics peculiar to program-

[11] As survey courses that contain some very simplistic programming units, ECS and AP CSP can only claim to give its participants a superficial exposure to programming concepts. Although, to their credit, some versions of AP CSP (e.g., Beauty and Joy of Computing) do focus almost exclusively on programming competence, these curricula make no substantial inroads into solving the novice programmer failure problem, carrying on the long secondary educational tradition of ineffective programming instruction.

[12] In 2016, California's CTC (Commission on Teacher Credentialing) created a CS supplemental authorization, but not a full CS credential that would have weight comparable to, say, a Math or Science credential. The authorization requires separate courses in five "content areas": Computer Programming; Data Structures and Algorithms; Digital Devices, Systems and Networks; Software Design; and Impacts of Computing (which can be counted if covered in courses from the first four areas). Although this is a significant improvement, it still falls far short of the range of topics that an undergraduate would study for a CS major [5].

[13] The importance of setting the minimum content knowledge for a secondary CS instructor to the equivalent of an undergraduate major in the field cannot be emphasized enough. Those who simply know the basics of programming are missing not only experience in designing mid-size to large software applications, but crucial knowledge about the ways that CS can be applied to a host of problems in subfields within CS proper, as well as in disciplines across the academic spectrum. Both impact not only a teacher's ability to make course content relevant to her students, but her ability to imagine and design instructional material that can illustrate the vast cross-curricular reach of CS.

ming languages. First, the control structures of programming languages—primarily *conditional execution* and *iteration*—are semantically broad and diffuse. One narrows meaning—i.e., performs tasks with specific accuracy—by arranging control structures in specific sequences, and most often in combinations with specific variables and/or data structures. Second, structured programming languages employ user-defined methods (with and without parameters), which add a third dimension of hierarchy overlaying the main body of sequential statements. Hierarchy such as this has no counterpart in natural languages, which are strictly sequential. Third, programming languages make use of nested syntactic blocks, a form of *complex recursion* (in this case *center-embedding*) that is theoretically possible in natural languages, but which in practice is all but impossible for people to process and understand. Fourth, as previously mentioned, the semantics of natural languages are acquired by interacting with native speakers. Programming languages are unspoken—hence there are no communities of "speakers" with which to interact. This last point is probably the most significant obstacle for educators who would like to adapt contemporary foreign language teaching techniques for programming language instruction.

How, therefore, would one go about constructing effective instructional strategies within the context of an implicit language-teaching approach, and what would they look like? This section will, as an example, discuss factors for deciding how to modify the teaching of an early introductory programming topic—*conditional execution*. For those interested in strategies for teaching the semantics of other language features, three that have been described previously [28] are listed here.

1. ***Setting components in relief:*** focusing on specific features and using them in different contexts or environments to highlight distinctions in meaning. Although like *Variation Theory* [20], the key difference is that learning in a linguistic model is *implicit*. This strategy also includes the use of counterexamples.
2. ***Transformations:*** underlying or expanded intermediate syntactic forms posited or invented to explain *abbreviated* syntactic features of programming languages whose mechanisms of operation are implied. For example, students are often confused about the mechanism by which the values of arguments in a method call are passed to the parameters of a method *definition*. An intermediate or underlying form of the method heading (not existing in the grammar) that shows the explicit assignment of argument values to parameters (analogous to transformations in the original Chomskyan Transformational Grammar model) better explains this mechanism, making it visible and concrete.[14] Similarly, the *for-loop* is an abbreviated notation for repeating (however many times) the statements in its body. Explicitly writing out these repetitions helps to elucidate how the three parts of a *for-loop* operate when students first begin to work with this abbreviated control structure.
3. ***Ongoing exposure:*** optimally solving a paradigmatic problem repeatedly to better cement it in memory as an archetypal solution.

Many curricula will begin the topic of *conditional execution* by presenting three patterns of *if/else* statements that (a) have varying degrees of mutual exclusivity and (b) a different number of branches that can be theoretically executed. These two properties are logically implicit in the syntactic structure of each pattern (Tables 2A, 2B, 2C).[15]

Typically, a few examples and counterexamples to illustrate these patterns may be given (Tables 3A, 3B, 3C). Note the logical error in the counterexample of Table 3A—the implication is that the error is structural, attributable to the cascading *if*-statements. However, the error can be corrected by simply reversing the order of the first four if statements (Table 4A). Interestingly, the error will be reintroduced should one supplement this fix with *if-else* statements (Table 4B). When the cascading if-statement pattern is placed inside a method—an environmental change—the Table 3A error can also be fixed if each branch *returns* the

[14] Consider a method definition ***Polygon getPolygon (int nSides)*** and its corresponding method call ***getPolygon (6)***. One can posit an intermediate/underlying form linking the two statements:
Polygon getPolygon (int nSides = 6). To turn this into an exercise that students can compile and run, the assignment statement within the parentheses of this invented underlying method heading can be moved down (or demoted) to the method body, appearing as the more familiar declaration of a local initialized variable. Note that this strategy also clarifies how parameters behave like local variables [28].

[15] The examples in Tables 2 and 3 are taken from Chapter 4 of *Building Java Programs* [29].

Cascading if-statements (NOT mutually exclusive)	if-else statements (mutually exclusive)	if-else statements ending with a default else case (mutually exclusive)
Executes zero or more branches/conditions	Executes zero branches or exactly one branch	Executes exactly one branch

`if (<test1>) {`
`  <statement1>;`
`}`
`if (<test2>) {`
`  <statement2>;`
`}`
`if (<test3>) {`
`  <statement3>;`
`}`
	`if (<test1>) {`
	`  <statement1>;`
	`}`
	`else if (<test2>) {`
	`  <statement2>;`
	`}`
	`else if (<test3>) {`
	`  <statement3>;`
	`}`
		`if (<test1>) {`
		`  <statement1>;`
		`}`
		`else if (<test2>) {`
		`  <statement2>;`
		`}`
		`else (<test3>) {`
		`  <statement3>;`
		`}`

Table 2A	**Table 2B**	**Table 2C**

```
String grade = "";[1]
if (score >= 90) {
  grade = "A";
}
if (score >= 80) {
  grade = "B";
}
if (score >= 70) {
  grade = "C";
}
if (score >= 60) {
  grade = "D";
}
if (score < 60) {
  grade = "F";
}
```

Counterexample.
Cascading *if* statements.

[1]Compilation error if *grade* is not initialized.

Logical error:
grade is always either "D" or "F".

Table 3A

```
String grade = "";[2]
if (score >= 90) {
  grade = "A";
}
else if (score >= 80) {
  grade = "B";
}
else if (score >= 70) {
  grade = "C";
}
else if (score >= 60) {
  grade = "D";
}
else if (score < 60) {
  grade = "F";
}
```

No errors.
[2]Compilation error if *grade* is not initialized.
Mutual exclusion is enforced syntactically by *if-else* statements. The logic inherent in the syntax structure may result in zero statements being executed. The comprehensive range of values, however, ensures that exactly one statement will be executed.

Table 3B

```
String grade;[3]
if (score >= 90) {
  grade = "A";
}
else if (score >= 80) {
  grade = "B";
}
else if (score >= 70) {
  grade = "C";
}
else if (score >= 60) {
  grade = "D";
}
else {
  grade = "F"; // default
}
```

No errors.

Mutually exclusive *if-else* statements with **default** *else*.

[3]Because exactly one statement will always be executed, *grade* will always be assigned a value – therefore it need not be initialized.

Table 3C

letter grade (Table 4C); note that this is an alternate mechanism for enforcing mutual exclusivity. The error, however, will not be corrected in a method that maintains the *grade* variable, with a single return statement at the method's end (Table 4D).

Unfortunately, the fixes in Tables 4A and 4C are themselves counterexamples, because from a maintenance perspective, a design *independent* of statement order is preferable to one where a change in the order will introduce logical errors. Note that the examples in Tables 5A and 5B provide this exact fix—they enforce mutual exclusivity through Boolean logic alone by using non-overlapping conditions that cover all possible cases, rendering the order of the *if* statements inconsequential. Programmed in this manner, it doesn't matter which of the three patterns from Table 2 is used—any mutual exclusivity contributed by the syntax layer will be redundant/superfluous.

Although the logical implications regarding the examples in Tables 2–5 may appear obvious to experienced programmers, they are described here in detail so that one might better appreciate how complex and daunting the possibilities might appear to be students who must juggle this information when encountering such ideas for the first time. Consequently, one might begin to wonder what is gained by including if-else statements in an initial discussion of *conditional execution*.

To exacerbate this conceptual overload, consider the complexity were one to now add a discussion of nesting and compound Boolean expressions comprised of independent conditions. As an example, Tables 6A-6D show possible solutions to the very *first* problem in the Warmup section of the codingbat.com website. A method containing a nested *if-else* within an outer else statement (Table 6A) can be rewritten using an equivalent non-nested (i.e., flat) three-part *if—else-if—else* structure (Table 6B). These can be collapsed to a single *if*-statement, a compound Boolean expression, and two *return* statements (Table 6C). The most compact version uses a single *return* statement and this same compound Boolean expression (Table 6D).

One can go on and on. The point is that the complexities of *if-else* structures arise because their correct functioning is dependent upon the interaction of logic, syntax and environment (e.g., inside a method). More specifically, program logic is distributed among two layers—the syntax layer that governs structural aspects of mutual exclusivity and flow of control, and the *content* layer that uses Boolean expressions to mediate the program's specific logic. What should be clear to any instructor who thinks about pedagogic issues is that this kind of discussion cries out for simplification and scaffolding. The alternative is to cause unnecessary confusion for students.

As it happens, all *if-else* statements—except for those that modify the value of the determining condition variable itself[16] — can be rewritten as cascading *if*-statements. As an example, Ta-

[16] When toggling a boolean condition variable:

`if (pass) { pass = false; }`
`else { pass = true; },`
the *cascading-if* "equivalent" would be :

`if (pass) { pass = false; }`
`if (!pass) { pass = true; }`
which always sets **pass** to **true**.

In such cases, one can opt for a toggle statement: **pass = !pass**. Note, however, that such examples can demonstrate the structural necessity of **if/else** mutual exclusivity, as well as **switch** statements for condition variables having more than two values, when they are introduced later.

```
String grade = "";
if (score >= 60) {
  grade = "D";
}
if (score >= 70) {
  grade = "C";
}
if (score >= 80) {
  grade = "B";
}
if (score >= 90) {
  grade = "A";
}
if (score < 60) {
  grade = "F";
}
```

No error. Reversed the order of the first 4 branches.

Like a *switch* statement without *break* statements, the code falls through all branches, eventually assigning the correct value of *grade*.

Table 4A

```
String grade = "";
if (score >= 60) {
  grade = "D";
}
else if (score >= 70) {
  grade = "C";
}
else if (score >= 80) {
  grade = "B";
}
else if (score >= 90) {
  grade = "A";
}
else if (score < 60) {
  grade = "F";
}
```

Counterexample to Table 4A.

The same error as Table 3A has been reintroduced by enforcing mutual exclusivity syntactically via the *if-else* pattern of Table 2B.

Table 4B

```
String getGrade(int score) {
  if (score >= 90) {
    return "A";
  }
  if (score >= 80) {
    return "B";
  }
  if (score >= 70) {
    return "C";
  }
  if (score >= 60) {
    return "A";
  }
  return "F"; // default
}
```

No error.
return statements within each branch enforce mutual exclusivity.
One condition serves as the default case and has been eliminated.

Table 4C

```
String getGrade(int score) {
  String grade = "";
  if (score >= 90) {
    grade = "A";
  }
  if (score >= 80) {
    grade = "B";
  }
  if (score >= 70) {
    grade = "C";
  }
  if (score >= 60) {
    grade = "D";
  }
  if (score < 60) {
    grade = "F";
  }
  return grade;
}
```

Logical error uncorrected.
grade is always either "D" or "F".

Table 4D

```
String grade = "";
if (90 <= score) {
  grade = "A";
}
if (80 <= score && score < 90) {
  grade = "B";
}
if (70 <= score && score < 80) {
  grade = "C";
}
if (60 <= score && score < 70) {
  grade = "D";
}
if (score < 60) {
  grade = "F";
}
return grade // if in method
```

Table 5A

```
String grade = "F"; // default
if (90 <= score) {
  grade = "A";
}
if (80 <= score && score < 90) {
  grade = "B";
}
if (70 <= score && score < 80) {
  grade = "C";
}
if (60 <= score && score < 70) {
  grade = "D";
}
return grade // if in method
```

Table 5B

```
boolean sleepIn(
  boolean weekday, boolean vacation) {
  if (vacation) {
    return true;
  }
  else {
    if (weekday) }
      return false;
    }
    else {
      return true;
    }
  }
}
```

Table 6A

```
boolean sleepIn(
  boolean weekday, boolean vacation) {
  if (vacation) {
    return true;
  }
  else if (weekday) }
    return false;
  }
  else {
    return true;
  }
}
```

Table 6B

```
boolean sleepIn(
  boolean weekday, boolean vacation) {
  if (vacation || !weekday) {
    return true;
  }
  return false;
}
```

Table 6C

```
boolean sleepIn(
  boolean weekday, boolean vacation) {
  return vacation || !weekday;
}
```

Table 6D

```
int x;
if (bool A) { x = 1; }
else ( x = 2; }
```

Table 7A

```
int x = 0;
if (bool A) { x = 1; }
if (!bool A) ( x = 2; }
// use the variable below
```

Table 7B

```
int x = 2; // default
if (bool A) { x = 1; }
// use the variable below
```

Table 7C

ble 7A shows a simple *if-else* statement, and Tables 7B and 7C show its logical equivalents (though the flows of control differ). Note that the patterns in Tables 7B and 7C require that variables be initialized (with a *dummy* value in 7B and the default value in 7C) to avoid compiler error—a software engineering practice worth encouraging anyway. An introduction to *conditional execution* can thus focus on one syntactic form—cascading *if*-statements. The concept of if-else can now become a refinement that can be postponed to a more advanced treatment of the subject, much like switch statements and *ternary* expressions.

There are several pedagogic advantages to scaffolding the topic in this way. An introductory treatment of the topic will focus on the most important aspect of *conditional execution*—the use of Boolean logic to enforce mutual exclusivity. Second, confusion due to the variety of ways that programming languages can express the same decision-making logic will be lessened. Finally, students will only have to learn a single pattern, the form in Table 7C,[17] which can be used in all environments, including those where methods return values.

Aside from scaffolding, there are cognitive reasons specific to language learning that argue for this simplification as well. When children are learning their first/native language, the *order* in which syntax features are acquired is related to their stage of development [6]. There is also a predictable order, related to difficulty, for features acquired by children learning second languages [11], and, it turns out, for adult second-language learners as well [2].[18]

In summary, this section has demonstrated that an introduction of the topic using the simplified case of cascading *if* statements—which still retains a substantial, but now much reduced, amount of complexity—will provide students a basic, but usable, syntactic foundation for *conditional execution*, which they can later supplement with more nuanced features that the language provides.

PART V: CURRICULAR STRUCTURE USING IMPLICIT LANGUAGE TEACHING METHODS

Having found a way to simplify and scaffold instruction for conditional execution, how would one structure such a unit using an SLA-based teaching model?

A unit in a whole language curriculum taught using a communicative pedagogic approach opens with a brief dialogue contextualized in a specific social aspect of life, such as eating out, family relationships, shopping, sports, vacationing, and the like. Each unit dialogue introduces pertinent *vocabulary*, phrases, idioms and verbal exchanges typical for what one might experience outside the classroom, as well as the *grammar structures* one might use, increasing in difficulty as the course progresses. In the classroom proper, there is an abundance of talking and listening, based on the verbal exchanges modeled in the unit dialogue. A host of exercises provide for extensive *practice* of the material in all four communicative areas—listening, speaking, reading, and writing. Later units may contain two or three related model dialogues, each with its own practice exercises. Each unit concludes with assessments that measure how well the material has been learned in the four communicative areas.

One much used exercise in a foreign language curriculum is the substitution drill. To practice listening and speaking skills, the instructor might pair off students and have them repeat question-rejoinder patterns while substituting different vocabulary items. The drill bolsters both vocabulary and syntactic patterns.

Q1: Do you prefer *milk* or orange juice?
A1: I prefer *orange juice* [*milk*].
Q2: Do you prefer *bread* or *croissants*?
A2: I prefer *croissants* [*bread*]. etc.

This may be followed by open-ended questions, e.g.,

Construct a four-sentence dialogue between you and a family member as you shop for groceries.

Substitution drills are also used to practice purely grammatical features, e.g., noun-article agreement.

Nous pouvons acheter *des* oeufs.
Tu veux *du* jambon?
Avez-vous manger *de la* soupe?

The components in these units are crafted with two key principles in mind—*repetitive exposure in varying contexts* and *meaningful communication*. As mentioned in Part I, the learner's exposure to language features (data) that are used repeatedly, but in varied contexts, is the mechanism by which the brain implicitly discerns patterns that it then inductively generalizes into the syntax rules of an ever-evolving grammar. Meaningful communication is what propels this language acquisition process. That is, the learner's *motivation* to actively communicate is what both drives repeated attempts at communication until her needs and wants have been successfully conveyed, thus honing correct language usage; and keeps her in a state of active listening and ongoing exposure to language data, spurring more cycles of the language acquisition process. In a programming language learning context, though, it makes no sense to say one can "communicate" with a computer. On the other hand, a computer does provide immediate program output—feedback as to how well the program is written. This ongoing cycle of *intentional interaction* on the part of the learner seems to be an effective substitute for *meaningful communication*, allowing for

[17] Although the code fragment in Table 7C is preferable, the stilted, but more explicit, intermediate form in Table 7B, may have considerable explanatory value, particularly in a side-by-side comparison to *if-else*-statements when they are eventually taught. Those who would like their students to perform *actions* inside the *if-blocks* can instead set parameter values in the blocks and follow with a single action statement that uses those parameters.

[18] An interesting aside is that the authors of these last two studies concluded that the results were evidence for a second language acquisition process involving "creative construction," not "habit formation." Creative construction, a process involving *hypothesis testing about the target language*, is what is generally agreed to account for the primary mechanism underlying implicit acquisition of first/native languages. Interestingly, evidence of hypothesis testing in the learning of programming languages surfaced when I observed my students making certain novel syntax errors having to do with the *direction* of assignment of values to variables [28].

the increasingly sophisticated understanding and acquisition of programming language skills. The probable reason for the effectiveness of this substitution is that the key feature shared by communication and interaction is the feedback to learners on how well formed their utterances/programs are.

In a programming language curriculum informed by foreign language pedagogies, a Model Program or Simulation (MPS) will serve as the central component around which each unit is organized, like the role of a unit dialogue in a foreign language course. Much has been written over the past decade about contextualizing programming instruction. Indeed, this author has written and published a detailed 50-page outline for a ten-unit cross-curricular introductory programming course, approved in 2013 as a University of California Office of the President (UCOP) "g" math elective, and called Computer Programming as if the Rest of the World Existed (CPRWE) [28]. Eight of its units is centered around a small to mid-size graphics-based MPS contextualized within one of a diverse range of subjects, including Dynamic Art, Geography, Political Science, Astronomy, and Molecular Modeling.[19]

The challenges for a curriculum writer in drafting the MPSs for each unit are to ensure: (a) that the code for the MPS reflects the particular language patterns one wants the unit to focus on,[20] so that students can see the practical utility of what they are learning; (b) that the MPSs are generally sequenced in order of increasing programming complexity; (c) that new programming concepts/skills are introduced while reinforcing ones previously learned; (d) that each MPS is highly engaging, both visually and intellectually; and (e) that the MPSs are contextualized within academic fields that students already value, and utilize basic concepts the students have previously encountered and already understand.

How is the MPS used? Consider how the foreign language teacher initially introduces the unit dialogue to students, sets up a structure for students to practice speaking and listening skills using as raw material the verbal interactions between the dialogue's characters, and often asks students to memorize the dialogues. In a parallel manner, the programming language instructor, using whole class instruction, initially guides students to individually construct an MPS, a process which can take days or weeks depending upon the size of the program. At various intervals, the instructor will require students to memorize the program (if short), or some newly-taught portion of it. The rationale is two-fold—to facilitate syntax acquisition; and to intimately familiarize students with the structure and vocabulary of the program, so that they can focus on the usage and meaning of the program's statements and methods in subsequent instruction.

There is no exact counterpart, however, to the foreign language classroom's speaking and listening activities that reuse the material from the unit dialogue. Instead, using *guided discovery*, the instructor directs students to incrementally reconstruct the entire MPS (or a small portion of one of the larger MPSs) from scratch in a sequence different from how they were originally guided to build it, specifying only how the partial program will function at each juncture, but not which parts of the MPS to use. The objective is that students will learn the function of the MPS's constituent working *blocks* and get a better sense of how they fit together to achieve the program's logic. At several points during the reconstruction, students are also asked to modify the *output* of the MPS, forcing them to tinker with the code and discover how key syntactic features work.[21] Students go through several cycles of this exercise, reconstructing the same *MPS*, but each time in a different sequence and with different changes to the output. This strategy, *setting components in relief*, is a specific implementation of the principle of repetitive exposure in varied contexts.

Instruction next focuses on giving students repetitive practice with a specific language feature. An example that uses transformational exercises asks students to convert indexed for-loops (that process members of an array) into equivalent statements that use hard-coded indices (Table 8).

Depending upon the unit, students might also construct individual variations of the MPS to let them explore their own creativity (with ongoing feedback from the instructor)—such exercises are extremely easy to implement when contextualized within the field of dynamic art. At unit's end, the instructor formalizes the learning in a traditional way, whether through direct instruction, Socratic seminar, or the like. Having been steeped in the content of the unit for several weeks by this time, students will be better primed to appreciate both basic and subtle details of what they have studied. A formal unit assessment of the knowledge and skills taught over the previous weeks will tell the instructor how successful instruction has been and what details might need to be retaught.

The counterpart to substitution drills for a programming language unit would be a series of exercises designed to provide practice in using one specific language pattern. Four patterns illustrating the use of conditional execution used in the unit *MPS*s of the introductory *CPRWE* course appear in Tables 9A-9D.

What criteria determine exactly which patterns to teach? Patterns taught in traditional programming curricula are often designed to showcase pure logic, are frequently devoid of meaningful application, and use pedestrian problems to illustrate flow of control. In contrast, like a second language communicative model where the patterns taught are those used in the unit dialogues, the patterns in a similarly structured programming language course are drawn from the MPSs. The rationale for the content of the dialogues in a whole-language communicative

[19] Demos of many of these simulations can be viewed at *www.downtownmagnets.org* on the Computer Science program page. A 2nd-year course, *Generative Design/Art*, was approved (2017) by UCOP as an integrated "f" art elective

[20] As will be discussed, the choice of syntax features may not be completely independent from the MPSs used for the curriculum's content.

[21] For example, students may need to modify the parameters of a *for-loop* by changing the *initialization* value of a *counter* variable, its *increment/update* amount, and/or the *termination condition*.

1. for (int b=2; b<5; b=b+1) {
  if (b%3==0) { bookList[b].read(); }
}
	3. if (false) { bookList[2].read(); }
	if (true) { bookList[3].read(); }
	if (false) { bookList[4].read(); }

2. if (2%3==0) { bookList[2].read(); }
if (3%3==0) { bookList[3].read(); }
if (4%3==0) { bookList[4].read(); }
	4. bookList[3].read();

Table 8

Ensuring a moving ball stays within bounds.
If the ball goes beyond the 600 foot mark, set *direction* to move left (-1)
If the ball moves beyond the 0 foot mark, set *direction* to move right (+1)

```
int LEFT = -1;
int RIGHT = 1;
int getBallDirection(int position, int
   directionCurrent) {
  int direction = directionCurrent;
  if (position < 0) {
    direction = RIGHT;
  }
  if (position >= 600) {
    direction = LEFT;
  }
  return direction;
}
```

Table 9A

Given a random number, calculate at which of the 4 edges of a window a word in a Dynamic Word Cloud will be randomly positioned at the start of the program.

```
int getEdge() {
  float n = random(0,100); // 0 <= n < 100
  int edge = LEFT; // 75 <= n && n < 100
  if (0 <= n && n < 25) {
    edge = TOP;
  }
  if (25 <= n && n < 50) {
    edge = RIGHT;
  }
  if (50 <= n && n < 75) {
    edge =  BOTTOM;
  }
  return edge;
}
```

Table 9B

Keep the index value of a 0-based array used for animation in bounds.
Valid array indices are: 0 through (NUM_IMAGES-1)
Animation can play forwards AND backwards.

```
int getNextIndex(int index, int direction) {
  int increment = 1;
  if (direction == BACKWARDS) {
    increment = -1;
  }

  index = index + increment;
  if (index >= NUM_IMAGES) {
    index = 0;
  }
  if (index < 0) {
    index = NUM_IMAGES-1;
  }
  return index;
}
```

Table 9C

Handling a circular queue's discontinuous region.
Given an object traveling east, its position expressed as a valid longitude (-180º < n ≤ 180º, with 180º = International Date Line), determine whether it is within the minor arc defined by two close longitudinal lines.

```
boolean isInRange(float n, float long1, float
   long2) {
  float min = Math.min(long1,long2);
  float max = Math.max(long1,long2);
  if (max-min > 180) {
    float temp = min;
    min = max;
    max = temp;
  }

  boolean inRange = false;
  if (min <= max) {
    if (min <= n && n <= max) {
      inRange = true;
    }
  }
  if (max < min) { // straddles discontinuity
    if (min <= n || n <= max) {
      inRange = true;
    }
  }
  return inRange;
}
```

Table 9D

Circular Queue over continuous range.
Using a 24-hour clock, return true if *hour* is between 8 am and 2 pm inclusive.
Variations: 4pm-8pm, etc.

```
boolean inTimeRange(int hour) {
  boolean inRange = false;
  if (8 <= hour && hour <= 14) {
    inRange = true;
  }
  return inRange;
}
```

Table 10A

Circular Queue over discontinuous range.
Using a 24-hour clock, return true if *hour* is between 10 pm and 2 am inclusive.
Variations: 8pm-3am, etc.

```
boolean inTimeRange(int hour) {
  boolean inRange = false;
  if (22 <= hour || hour <= 2) {
    inRange = true;
  }
  return inRange;
}
```

Table 10B

Circular Queue over two ranges.
Using a 24-hour clock, return true if *hour* is between 10 am and 4 pm or between 9 pm and 3 am.
Variations: Mixtures of ≥1 continuous ranges and/or 1 discontinuous range
11am-2pm OR 6pm-8pm,
9am-1pm OR 4pm-7pm OR 10pm-2am

```
boolean inTimeRange(int hour) {
  boolean inRange = false;
  if (10 <= hour && hour <= 16) {
    inRange = true;
  }
  if (21 <= hour || hour <= 3) {
    inRange = true;
  }
  return inRange;
}
```

Table 10C

Circular Queue over any range, general case.
Using a 24-hour clock, return true if *hour* is between the parameters *start* time and *end* time.
Variations: Implement other circular queue systems
Longitude (-180 ° < degrees ≤ 180°)
Circle ($0 \leq$ degrees < 360, $0 \leq$ radians $< 2\pi$)
Ferris wheel (seats) , Compass (headings), Theater or highway marquee (feet), Airport baggage carousel (feet)

```
boolean inTimeRange(int hour, int start, int end) {
  boolean inRange = false;
  if (start <= end) {
    if (start <= hour && hour <= end) {
      inRange = true;
    }
  }
  if (end < start) { // discontinuous range
    if (start <= hour || hour <= end) {
      inRange = true;
    }
  }
  return inRange;
}
```

Table 10D

curriculum is to provide students the vocabulary, grammar, and verbal exchanges for navigating a social situation that they typify and model. In like manner, the patterns in an MPS are ones that students would employ to solve similar real-life problems that they might encounter.

A series of exercises for the *Circular Queue* pattern in Table 9D would include the sub-patterns shown in Tables 10A-10D. Note that the sample exercises employ scaffolding—the sequence begins with the simplest case and progressively introduces the more complex cases that the method for the general case (Table 10D) will need to handle. It also employs a pedagogy used much in the Algebra 1 classroom—moving from the concrete to the abstract. Note that each sub-pattern should itself provide several exercises (a minimum of 5) to give students sufficient practice.

An SLA-based instructional model dictates that all of the patterns that students are expected to learn need to be taught, i.e., students should not be expected to solve problems whose patterns they have not explicitly practiced, and they should not be expected to magically extend logic—for the simple reason that they are not yet experienced programmers. The assessment following each pattern's (or sub-pattern's) battery of exercises will be the correct application of that single pattern to several novel word problems. The comprehensive assessment for the unit will host problems reflecting the full range of patterns studied. Students will evaluate each problem, decide which among the many patterns is applicable, and craft an optimal solution.

Note that the prevailing instructional model expects students to apply general concepts from lecture or the textbook to solve a full variety of novel problems at a unit's end—problems whose solutions have not been explicitly demonstrated during instruction. Consequently, students are somehow expected to devise the pattern for each problem's optimal solution de novo. The alternative SLA-based instructional model proposes teaching paradigmatic solutions to *each* kind of problem students are expected to solve. With a variety of paradigms to choose from, the key skill students are expected to develop is choosing the paradigm that is applicable to the problem at hand—and then, of course, using it to write an optimal solution. This is comparable to the organization of units in a foreign language curriculum, each dealing with the vocabulary and conversations typical of specific societal domains, e.g., sports, restaurants, travel, and so on. This is not merely a difference in teaching philosophy, but one grounded in educational psychology as well. Any

credentialed teacher knows that the ability to solve problems is—and algorithms themselves are—highly domain-specific; and that good problem-solvers draw upon prior experience and knowledge of specific domains [4,34]. Even so, problem-solving is a process that is still poorly understood.[22]

Finally, note that the approach described in this section will of necessity require that instructors allocate considerably more time to students practicing a unit's concepts and sub-concepts than the current instructional model provides. The instructional tension between breadth (coverage) and depth (detail) is nothing new. Introductory programming language courses have traditionally opted for breadth given the very limited amount of time they have allotted—or more precisely, that they have self-imposed upon their programs of study. Such a limited time frame could never work in a foreign language curriculum, where four semesters are typically allocated for students to acquire proficiency in the fundamental workings of a language. Foreign language curricula probe every topic in depth, because the *breadth* of the curriculum can be adequately covered over the two years budgeted for the program's foundational sequence of courses. Unfortunately, there is no way to reconcile the existing programming language model and course structure with both breadth and depth learning. Something will have to give, and the only resource available is time.

SUMMARY

In his seminal paper on the *errors* made by second language learners, S.P. Corder asserted that errors provided evidence of—and insights into—the process by which learners construct and refine hypotheses about the underlying grammar of the language data they hear. He wrote.

> We have been reminded recently of Von Humboldt's statement that we cannot really teach language, we can only create conditions in which it will develop spontaneously in the mind in its own way. We shall never improve our ability to create such favourable conditions until we learn more about the way a learner learns and what his built-in syllabus is. [9]

Yet, it is exactly the way novice programmers learn that has continued to remain a mystery, obscuring how improvements to teaching might be achieved.

No one denies that the current introductory programming pedagogic model leads to less than favorable learning outcomes—particularly at the secondary level. The model undoubtedly contributes to ongoing low secondary enrollments and the worst demographic inequities of any subject area. Four decades of characterization of the novice programmer failure phenomenon have not produced any improvements in learning, nor pinpointed any credible cause. Predictably then, attempts at curricular innovation, rooted in hunches and overwhelmingly on the content side, have been ineffectual.

In contrast, the 2014 fMRI cognitive study of programmers established that the brain makes sense of computer programs in the regions of the brain long known to be associated with language processing functions, not logic and not math. Although more research will be required to prove the case definitively, this physiological evidence puts a spotlight on an aspect of programming instruction long taken for granted, the Prescriptive Linguistics language model. Both the fMRI study and the consistent anecdotal observations reported here about the positive effects of memorization strategies on syntax acquisition constitute a compelling argument for investigating whether an alternate and frankly more promising approach—implicit language pedagogies informed by both SLA theory and foreign language instructional principles—can enhance our instructional outcomes if scaled. On the flip side, those wanting to devise new content or pedagogic approaches to the introductory programming curriculum, but who ignore the central cognitive role of language in programming, now risk irrelevance. As Corder cautioned some fifty years ago, our teaching will only succeed when it conforms to how the brains of our students actually learn. ❖

[22] From a psychological vantage point, problem-solving is a complex phenomenon, described by Gestalt theorists with notions like "restructuring," "insight," and "entrenchment;" and by cognitivism with a reliance on domain knowledge and heuristics. In all of these, although conditions that facilitate the crucial moments of insight can be listed, there are no satisfactory explanations for how such insights arise. The incubation phenomenon—setting aside a problem after being unable to find a solution, with a solution later popping into one's mind (like a forgotten detail that one remembers after the fact)—argues that problem-solving may be a largely subconscious process.

References

1. Amirtha, T. This Is Your Brain On Code, According To Functional MRI Imaging. *Fast Company*, April 21, 2014; http://www.fastcompany.com/3029364/this-is-your-brain-on-code-according-to-functional-mri-imaging. Accessed 2017 November 22.
2. Bailey, N., Madden, C. & Krashen, S.D. Is there a 'natural sequence' in adult second language learning? *Language Learning*, 24, 2 (1974), 235–243. doi: 10.1111/j.1467-1770.1974.tb00505.x.
3. Bennedsen, J. & Caspersen, M. E. Failure Rates in Introductory Programming. *Inroads - The SIGCSE Bulletin*, 39, 2 (June 2007), 32–36. doi:10.1145/1272848.1272879.
4. Bransford, J., Sherwood, R., Vye, N. & Rieser, J. Teaching Thinking and Problem Solving: Research Foundations. *American Psychologist*, 41, 10 (1986), 1078–1089. doi:10.1037/0003-066X.41.10.1078.
5. California CTC (Commission on Teacher Credentialing). *Supplementary Authorization Guideline Book* (CTC, July 2016); https://www.ctc.ca.gov/docs/default-source/credentials/manuals-handbooks/supplement-auth.pdf. Accessed 2018 March 8.
6. Chomsky, C. *The Acquisition of Syntax in Children from 5 to 10*. (Cambridge, MA: MIT Press, 1969).
7. College Board. AP Data: *Program Summary Report 2017*. (College Board, 2017); https://secure-media.collegeboard.org/digitalServices/pdf/research/2017/Program-Summary-Report-2017.pdf. Accessed 2017 November 22.
8. Corina, D. P., Lawyer, L. A., Hauser, P. & Hirshorn, E.. Lexical Processing in Deaf Readers: An fMRI Investigation of Reading Proficiency. *PLOS ONE*, 8, 1 (2013), 1–10.
9. Corder, S. P. The Significance of Learner's Errors. *International Review of Applied Linguistics in Language Teaching*, 5, 4 (1967), 161–170.
10. CSTA Standards Task Force. *CSTA K-12 Computer Science Standards*. (New York: CSTA/ACM, 2011); http://c.ymcdn.com/sites/www.csteachers.org/resource/resmgr/Docs/Standards/CSTA_K-12_CSS.pdf. Accessed 2017 November 22.
11. Dulay, H. C. & Burt, M. K. Should We Teach Children Syntax? *Language Learning*, 23, 2 (1973), 245–258. doi:10.1111/j.1467-1770.1973.tb00659.x.
12. Eklund, A., Nichols, T. & Knutsson, H. Cluster failure: Why fMRI inferences for spatial extent have inflated false-positive rates. *Proceedings of the National Academy of Sciences*, 113, 28 (2016), 7900–7905.
13. Floyd, B., Santander, T. & 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. doi: 10.1109/ICSE.2017.24.
14. Hamrick, Phillip. Ph.D. Dissertation, Georgetown University, Washington D.C. Development of Conscious Knowledge During Early Incidental Learning Of L2 Syntax. (ProQuest, LLC, 2013), 3642262.

15. Hesling, I., Dilharreguy, B., Bordessoules, M. & Allard, M. The Neural Processing of Second Language Comprehension Modulated by the Degree of Proficiency: A Listening Connected Speech fMRI Study. *The Open Neuroimaging Journal*, 6 (2012), 44–54.
16. Kidwai, A. Knowledge of Language: Noam Chomsky's Innateness Thesis. *Contemporary Education Dialogue*, 5,2 (2008), 245–265. doi:10.1177/0973184913411168.
17. Kummerfeld, S. K. & Kay, J. The neglected battle fields of Syntax Errors. In *Proceedings of the Fifth Australasian Conference on Computing Education*, 20, 105–111. (Adelaide, Australia: Australian Computer Society, Inc., 2003).
18. Lahtinen, E., AlaMutka, K. & Järvinen, H.M. A Study of the Difficulties of Novice Programmers. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (June 27–29, 2005), 14–18. doi:10.1145/1151954.1067453.
19. Lewis-Kraus, G. The Great AI Awakening. *New York Times* (Magazine). December 14, 2016; http://www.nytimes.com/2016/12/14/magazine/the-great-ai-awakening.html?_r=0. Accessed 2017 November 22.
20. Lo, Mun & Marton, Ference. Towards a science of the art of teaching: Using variation theory as a guiding principle of pedagogical design. *International Journal for Lesson and Learning Studies*, 1 (2011), 7–22. doi: 10.1108/20468251211179678.
21. Mattern, K., Shaw, E. & Ewing, M. *Advance Placement Exam Participation: Is AP Exam Participation and Performance Related to Choice of College Major?* (New York, NY: College Board, 2011).
22. McGettrick, A., Boyle, R., Ibbett, R., Lloyd, J., Lovegrove, G. & Mander, K. Grand Challenges in Computing: Education—A Summary. *The Computer Journal*, 48,1 (2005), 42–48. doi:10.1093/comjnl/bxh064.
23. Monti, M. M., Parsons, L. M. & Osherson, D. N. Thought Beyond Language: Neural Dissociation of Algebra and Natural Language. *Association for Psychological Science*, 23, 8 (2012), 914–922. doi:10.1177/0956797612437427.
24. Morgan, R. & Klaric, J. AP Students in College: *An Analysis of Five-Year Academic Careers*. (New York, NY: College Board, 2007).
25. Moro, Andrea. *The Boundaries of Babel: The Brain and the Enigma of Impossible Languages*. (Cambridge, MA: MIT Press, 2008).
26. Pappano, L. Learning to Think Like a Computer. *New York Times*, April 4, 2017. https://www.nytimes.com/2017/04/04/education/edlife/teaching-students-computer-code.html?_r=0. Accessed 2017 November 22.
27. Parnin, C. *Scientists Begin Looking at Programmers' Brains: The Neuroscience of Programming*. Huffington Post, April 23, 2014; https://www.huffingtonpost.com/chris-parnin/scientists-begin-looking-_b_4829981.html. Accessed 2017 November 22.
28. Portnoff, S. M.S. Thesis. California State University, Los Angeles. (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. http://pqdtopen.proquest.com/pubnum/10132126.html?FMT=AI. Accessed 2017 November 22.
29. Reges, S. & Stepp, M. *Building Java Programs*. 3rd ed. (Boston: Pearson, 2014).
30. Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., Saake, G. & Brechmann, A. Understanding Understanding Source Code with Functional Magnetic Resonance Imaging. In *Proceedings of the 36th International Conference on Software Engineering* (2014), 378–389. doi:10.1145/2568225.2568252.
31. Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A. & Brechmann, A. Understanding Programmers' Brains with fMRI. In *Frontiers in Neuroinformatics Conference Abstract: Neuroinformatics* (2014). doi: 10.3389/conf.fninf.2014.18.00040.
32. Tagarelli, K.M. Ph.D. Dissertation, Georgetown University, Washington D.C. The Neurocognition of Adult Second Language Learning: An fMRI Study. (ProQuest, LLC, 2014); 3642262.
33. Watson, C., & Li, F. W. Failure Rates in Introductory Programming Revisited. In *Proceedings of the 2014 Conference On Innovation & Technology In Computer Science Education*. (Uppsala, Sweden: Association for Computing Machinery, June 2014), 39–44. doi:10.1145/2591708.2591749.
34. Woolfolk, A. *Educational Psychology*. 9th ed. (Boston: Pearson, Allyn and Bacon, 2004).
35. Yokoyama, S., Kim, J., Uchida, S., Okamoto, H., Bai, C., Yusa, N. et al. A longitudinal fMRI study of neural plasticity in the second language lexical processing. *Neuroscience Research Abstracts* 58S (2007), S174.

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

DOI: 10.1145/3152433