Programming LANGUAGE Patterns:
Instructional Strategies for Novice Programmers

Three preliminary units of exercises to better prepare beginning students to solve Level-1 Codingbat problems.

This article introduces three preparatory units of exercises for the Logic-1 and String-1 Java sections of Codingbat. Beginning programming students—in this case grade-level non-gifted high school freshmen—who work through these short units are then able to more independently solve the problems in those sections, thus illustrating one way to mitigate the effects of the decades-old Novice Programmer Failure Problem (NPFP). The NPFP has all but devastated programming instruction throughout America’s urban public high schools, causing teachers to give up on such instruction in favour of non-programming courses (Exploring Computer Science, Computer Science Principles) that instead emphasize sociological aspects of CS and offer a survey of CS-related topics that, without a minimum proficiency in programming, can only be covered superficially. And who can blame them, particularly when the vast majority of grade-level secondary students would fail a rigorous programming class?

Capitulation, however, is not a solution. Programming is the gatekeeper skill for economic opportunity, both for CS study in college and high-paying computing jobs. Sixty-five percent of urban public school students in the U.S. attend high-poverty (40%) or mid-high poverty (25%) schools. To further handicap them by not providing access to rigorous and effective programming instruction is to acquiesce in and perpetuate historical inequities.

Longitudinal studies have yielded only two statistical predictors for choosing a CS college major, both of which involve rigorous programming instruction:

(a) participation in the Girls Who Code program, and
(b) a score of 2-5 on the APCS-A exam.

There is broad agreement that a key objective of any secondary CS class should be that it authentically teaches students skills and concepts needed for subsequent coursework should they pursue study. That these programs boost the rates of CS majors is a strong metric for validating their worth.

That said, the APCS-A course is too advanced for most beginners. A desperately needed prerequisite is a pre-APCS-A programming course that identifies and addresses head-on the difficulties novice programming students encounter. To assert that non-programming courses like ECS and CSP are pedagogically effective in this regard would be implausible: in fact, proponents of these courses consistently downplay and negate the importance and centrality of programming—the consensus view of both postsecondary instructors and the business/corporate community. To contend that exposure to simplified CS topics—even were they to include limited coverage of short programs—will later enable students to surmount the NPFP and magically acquire programming competence is like arguing that watching French
film, eating Parisian cuisine, and learning a few French phrases would adequately prepare students for a second year French language course.

Although the three preliminary Codingbat units utilize well-known instructional strategies (ZPD, scaffolding, Bruner's spiral curriculum), the form in which they have been constructed is rooted in the principles of Second Language Acquisition. This approach is buttressed by three recent (2014-2017) fMRI studies confirming that comprehension of computer programs occurs in the same regions of the brain that process natural languages—not math, not logic. More importantly, they work. The few such strategies I now utilize consistently allow all of my grade-level high school freshmen to make and sustain gains in programming proficiency, suggesting that the root cause of the NPFP is pedagogic, and can therefore be remedied by a change in instruction.

The broader purpose of this article, then, is to advocate for a renewed secondary programming curriculum whose pedagogic framework is informed by Language Acquisition, one that will authentically broaden participation and reduce demographic inequities.

**Strategy 1: Memorization—Acquiring Basic Syntactic Features**

Teachers of the introductory programming course universally observe sizeable groups of frustrated students unable to compile and run their programs because they continue to make the same syntax mistakes over and over. For five years, though, I have observed that this problem disappears after students memorize small programs perfectly. The reason is that language syntax cannot be taught explicitly, e.g. through grammar rules. Rather it is implicitly acquired through repetitive exposure to language data/input. Memorizing a program perfectly mimics this mechanism because students must undertake numerous cycles of reading a program, writing it out without looking, and comparing. In so doing, they bombard their brains repeatedly with idealized language data. As with natural languages, the brain subconsciously constructs the syntax rules implicitly by induction from the patterns in the data. Although the effectiveness of memorization on the rapid acquisition of most syntactic elements is striking, I have found that its usefulness is limited to this single purpose.

**Strategy 2: Writing Methods Using a Single Paradigm**

Programming languages, like natural languages, are infinitely generative. As such, there are countless ways to write a method/function. To narrow the possibilities, the CS teacher—like the foreign language instructor—serves as a kind of “native speaker” for her students, modelling the clearest and most optimal ways to use a programming language in different circumstances. Her curriculum also starts off with simple language constructions, building proficiency with more complex syntactic patterns over time as material is revisited.
int getAtomAngle(int aNumber) {
    int angle = 0;
    if (aNumber == 1) {
        angle = 0;
    }
    if (aNumber == 2) {
        angle = 60;
    }
    if (aNumber == 3) {
        angle = 120;
    }
    if (aNumber == 4) {
        angle = 180;
    }
    if (aNumber == 5) {
        angle = 240;
    }
    if (aNumber == 6) {
        angle = 300;
    }
    return angle;
} // getAtomAngle()

Figure 1

A custom Codingbat Java web page called Logic-1 Basics (https://codingbat.com/home/srp4379@lausd.net/1-logicbasics) is a short unit preceding the website's regular Logic-1 section. This preparatory unit teaches students a single, simple syntactic paradigm for writing methods. The first line of the method body declares a return variable of the same type as the method's return type, and initializes it with a default value (e.g. false, 0, ""). The variable is returned by a SINGLE return statement on the last line. In between are one or more cascading IF statements that test the input parameter(s) and change the value of the return variable (Figure 1). Mutual exclusion is accomplished entirely through Boolean logic. Teaching ELSE or ELSE IF statements, whose logical complexities and side-effects can confuse beginners, is deferred.¹

Every grade-level student was able to complete the Logic-1 Basics unit on her own. Students were then able to focus on how to use the method body model with the more advanced programming logic found in the regular Logic-1 section.

Strategy 3: Problem-Solving in a Particular Domain

There is no general problem-solving ability.² Problem-solving is domain-specific, i.e. if you practice solving problems of a certain type, you are better able to solve novel problems of that same type. Therefore, rather than ask students to independently solve the problems in Logic-1, I guided them through optimal solutions using the method model they just practiced—employing direct instruction, guided discovery and counterexamples; investigating boundary/edge conditions; and pointing out trade-offs between clarity and efficiency.

Proficiency, however, is the result of practice, not just instruction, and students need opportunities to independently practice solving the different categories and difficulty levels of

¹ See Part IV The Introductory Computer Programming Course is First and Foremost a LANGUAGE Course, ACM Inroads 9, 2, Portnoff S (2018) for a discussion of these considerations.

the problems found in the regular section of Logic-1. Although in the interim, my instruction has been as described above, I am currently crafting additional Codingbat units to address this need. Structuring these units using repetition and an exercise sequence that incrementally increases in difficulty, students will learn clear and optimal language patterns for solving the more sophisticated problems in each Logic-1 category.

At the end of the section, students were given a summative assessment using a custom Codingbat page consisting of ten problems that are novel variations on those in the regular section. Seven of these are straightforward, two are of medium difficulty, and one is difficult. The test is open-book, and students were encouraged to analogize from the solutions in the regular section whenever possible. Virtually all students were able to complete the first seven problems, my minimum standard for proficiency. It is expected that these scores will improve with the additional units currently under construction.

### Strategy4: Practicing Semantic Patterns

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
<th>7th</th>
<th>8th</th>
<th>9th</th>
<th>String s</th>
<th>Index / Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.substring(0, 3) → &quot;abc&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.substring(1, 3) → &quot;bc&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int len = s.length()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.substring(len-2) → &quot;yz&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.substring(len-4, len-1) → &quot;wxy&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2**

For the String section, students first complete two preparatory units, String-Basics A (https://codingbat.com/home/srp4379@lausd.net/1-stringbasics), and String-Basics B (https://codingbat.com/home/srp4379@lausd.net/2-stringbasics). These exercises provide practice using the String methods length() and substring(), respectively. Students also refer to a diagram of a String's structure (Figure 2) for expressions that examine the characters at a String's first, middle and final positions. Again, virtually all students were able to complete these two units on their own. As with Logic-1, students then received direct instruction for constructing optimal solutions for the problems in the regular section, using the additional String methods equals(), startsWith() and endsWith(), finishing the section with the open-book unit assessment. As with Logic-1, though, I am authoring additional Codingbat units that will
(a) teach students the semantic uses of these additional String methods; and (b) allow them to learn direct, clear language patterns for optimally solving the various categories of problems in the regular String-1 section on their own.

**Summary**

The three preparatory units described in this article utilize familiar educational theories. However, the focus on Language Acquisition aspects of programming language instruction—helping students acquire the patterns of language through incrementally more difficult repetitive practice—is what appears to boost learning for all grade-level students.