Syntax exercises in CS1

John Edwards
john.edwards@usu.edu
Utah State University
Logan, Utah

Joseph Ditton
jditton.atomic@gmail.com
Utah State University
Logan, Utah

Dragan Trninic
dragan.trninic@gess.ethz.ch
ETH Zurich
Zurich, Switzerland

Hillary Swanson
hillary.swanson@usu.edu
Utah State University
Logan, Utah

Shelsey Sullivan
shelsey.sullivan@usu.edu
Utah State University
Logan, Utah

Chad Mano
chad.mano@usu.edu
Utah State University
Logan, Utah

ABSTRACT

This paper investigates the idea of teaching programming language syntax before problem solving in Introductory Computer Programming (CS1). Theories of procedural skill acquisition imply that syntax should be taught with a pedagogy and curriculum quite different from that used in teaching problem solving. We draw from this literature to propose a practice-based pedagogy and curriculum to teach students syntax before they learn its application, something we call a "syntax-first" pedagogy, which uses skilled performance in syntax to scaffold learning of problem solving. We report results of a controlled study investigating whether learning syntax using pedagogy suitable for procedural skill acquisition (e.g. repetitive practice) prior to learning problem solving influences student performance. A syntax-first pedagogy is complementary to almost any other teaching approach: in our study, simply adding carefully designed syntax exercises to an existing CS1 course resulted in higher exam scores, lower student attrition, and evidence that plagiarism rates may be lower.

CCS CONCEPTS

• **Social and professional topics → CS1**.

KEYWORDS

CS1, syntax, cognitive load, memory

ACM Reference Format:
John Edwards, Joseph Ditton, Dragan Trninic, Hillary Swanson, Shelsey Sullivan, and Chad Mano. 2020. Syntax exercises in CS1. In *Proceedings of ICER '20: International Conference on Computing Education Research (ICER '20).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Introductory Computer Programming (CS1) is a first college course in computer programming that is infamously challenging, with failure rates as high as 35% [33]. Substantial effort has been put into discovering why CS1 is chronically problematic, and many findings target student frustration with programming language syntax [9, 23, 28, 45]. However, syntax requires very little cognitive effort [36], so why should it cause so much student frustration? In a sense, it could, and possibly should, be the easiest part of programming rather than the biggest source of frustration.

Upon identification of syntax as an obstacle, pragmatic researchers and practitioners began developing and using block-based languages such as Alice [7] and Scratch [39]. Block-based languages prevent syntax errors so the student can focus on problem solving. While the syntax-free approach has been successful by many measures, student frustrations return when transitioning to a text-based language [51]. This makes sense: problem solving and the ability to type code are different skills, and proficiency in problem solving does not lead to proficiency in typing error-free code.

In the present work we explore the idea of focusing first on programming language syntax and then teaching problem solving once students have achieved proficiency in the necessary syntactical constructs. The idea of isolating and mastering syntax using established techniques for procedural skill acquisition has, perhaps surprisingly, not been investigated in depth. We further investigate using syntax proficiency as a scaffold for learning problem solving.

The theory behind using separate pedagogy and curriculum for syntax and problem solving is well established. We can look at typing code as a skill, where a skill is defined as "goal-directed, well organized behavior that is acquired through practice and performed with economy of effort" [38] (as cited by Johnson [20]). It is generally agreed upon that skill acquisition is done through practice, while acquisition of more complex knowledge and abilities, such as problem solving, is actively debated.

In addition to exploring the theory behind isolating syntax learning and using it as a scaffold for higher-cognitive learning, this paper reports results from a study that implemented a practice-based curriculum for learning programming language syntax for the test group, and identical curriculum for problem solving for both groups. Analyses of keystroke data and project/exam scores support the notion that practice-based learning of syntax is beneficial to student performance, and are consistent with the hypothesis that using syntax proficiency as a scaffold for learning problem solving results in improved student outcomes. We seek to answer the following research questions in a CS1 context:

RQ1 How does a syntax-first pedagogy affect student attrition?

ICER '20, August 10–12, 2020, Dunedin, New Zealand

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

RQ2 How does a syntax-first pedagogy affect project and exam scores?
RQ3 How does a syntax-first pedagogy affect the time required for students to complete programming projects?

Our overall hypothesis is that syntax exercises result in improved student outcomes and student attitudes. See Section 3.4 for specific hypotheses that drive our study.

In this paper, we show that syntax exercises have a strong effect on student attitudes and outcomes. We suggest that our results not only have important theoretical implications, but also that they have strong practical application. Syntax exercises are complementary with virtually all CS1 pedagogies and curricula and can be added to a course with very little disruption to an existing syllabus. A potential criticism of our work is that improved outcomes from additional student effort is not a novel finding; in response, we show that well-designed syntax exercises can result in disproportionately large improvements in student outcomes.

This paper first discusses the theory behind our approach and other related work (Section 2), followed by our methods (Section 3), results (Section 4), discussion (Section 5) and conclusions (Section 6).

2 THEORY AND RELATED WORK

2.1 Syntax and cognitive load

To drive our discussion of syntax as a barrier to success we make three fundamental observations of CS1 pedagogy:

- Observation 1: CS1 teaches a number of skills [12]. We focus on two distinct skills: the first skill is the syntax of a computer programming language, and the second is problem solving using the language as a tool. (Using du Boulay's taxonomy, we define syntax as a combination of notation and structures, and problem solving as the specification, testing, and debugging portions of pragmatics [12].)
- Observation 2: Learning of language syntax requires lower order thinking than does problem solving [36]. However, instructors generally teach syntax and problem solving simultaneously as a single conflated concept. Indeed, classroom instructors often fail to even emphasize to their students that syntax and problem solving are two different things.
- Observation 3: Language syntax is often a major barrier to student success in CS1 [9, 23, 45], and has been called an "extraneous cognitive load" [28, 47]. One approach to solving the syntax problem is to find a programming language with syntax that is more easily learned for students [45]. Another approach is to remove syntax entirely [7, 39].

One potential reason that syntax is a barrier is that students are expected to simultaneously grasp the syntax of a computer language and problem-solve using the language as a tool. In other words, the syntax of a language (i.e., a tool) may be conflated with problem solving (i.e., tool use). If so, students' ability to pay attention to and learn from problem solving is compromised by their lack of familiarity with using the tool.

The concern is that, when students lack the ability to fluently use the tools at their disposal, their problem-solving processes will require more resource-intensive cognitive engagement. In other words, they will struggle with the tool itself, rather than tool-use. This burdens the working memory capacity, which is limited and required for all conscious processing [34]. In turn, students are unlikely to be able to devote cognitive resources to both wrestling with syntax and engaging in problem-solving (see, e.g., [22, 47]). This may be avoided if students could first develop a baseline syntax familiarity prior to and as preparation for more formal instruction.

There are numerous accounts of skill acquisition (e.g., [4, 11, 18]). A point of agreement among all is that skillful actions become automated. Automation frees up cognitive resources, which may subsequently be leveraged for problem solving. Therefore, one way we can prepare students for problem solving is by automating their production of syntactically-correct code.

The present intervention draws on this fundamental idea of reducing students' unnecessary cognitive burden by automating their production of code syntax, thereby enhancing their learning. We accomplish this by providing students with exercises that develop their fluency with syntax, preparing them for future learning from instruction (cf. [43, 49]).

A syntax-first approach is not new. Linn and Dalbey [27] proposed a chain of three cognitive accomplishments of which knowledge of single language features was the first. Xie et al. [52] put writing correct syntax second in their sequence of four program-writing skills (behind program tracing, similar to Schulte's block model [42] which emphasizes reading comprehension). Multiple works suggest that the ability to write correct syntax would aid in later problem solving [1, 5, 29]. The problem is that, as Xie et al. suggest, "although there have been many theories with implications for CS1 instructional design for programming skills, few have been translated to concrete instruction" [52].

2.2 Practice

Skill acquisition is obtained through practice [20]. In the three phases of skill acquisition suggested by Fitts and Posner [18] (see also [17]), the second (fixation) and third (automation) phases both rely heavily on practice to fix the skill and then achieve nearly error-free performance even in the presence of distractors. Skilled performance reflects past experience [44], requiring many exposures to work together [31]. Even child prodigies show steady progress through practice [15], and Ericsson and Charness [16] argue that practice is more important than genetics. Repetition is a critical component to practice for automaticity [30], with response time in autonomous performance decreasing as a power function of the number of exposures [30, 35]. Drawing on the robust theory that repetitive practice supports development of autonomous skill performance, our design of syntax pedagogy is centered on repetition with the goal of fixation and automation of typing syntax.

2.3 Similar studies

As mentioned, research into the effects of repetitive practice of syntax, or of any specific pedagogy for syntax learning, has been thin, although a few recent studies show increasing interest in the topic. Leinonen et al. [25] conducted a controlled study in which students in the test group were given a version of syntax exercises before working on their regularly-scheduled programming projects. Although none of their results were statistically significant, their

measurements of number of keystrokes to project completion and time to project completion were mixed, with test students requiring more effort on some projects and less effort on others. The authors conclude that syntax exercises may not be meaningful, particularly in their context where programming projects are small and very frequent. However, their concept of syntax exercises differs fundamentally from ours: in our study, syntax exercises are very small, very numerous, and are designed such that errors are highly infrequent. Leinonen et al. [25] expect students to make syntax errors in the exercises and, in fact, designed the tool to automatically highlight syntax errors. Thus, their tool may result in greater cognitive effort, while our exercises are designed to engage autonomous, error-free execution as early as possible.

A recent study by Gaweda et al. [19] used a similar approach to Leinonen et al. [25], where students simply re-typed code shown to them while the tool highlighted lines with errors. In this study, students who completed the exercises had higher exam scores, on average, and a negative correlation of number of exercises completed with build failures in programming projects. However, the group completing the exercises was self-selected, calling causitive claims into question.

Anecdotal evidence from the efforts of Portnoff [37] in an inner-city high school suggest that memorizing code snippets decreases syntax errors and increases student motivation. Importantly, a particular increase in motivation was noted in students who would not have been expected to succeed.

The study most similar to ours is an exploratory study by Edwards et al. [14] that uses high-frequency syntax exercises similar to our own. Their results were consistent with the idea that syntax exercises would improve student outcomes and attitudes, however, potentially confounding variables hobbled their ability to make any definitive claims. Nevertheless, decreases in student frustration and increases in project/exam scores were observed, suggesting the need for a larger, better-controlled study.

3 METHODS

In this section we discuss the design of our study, measurement instruments, and specific hypotheses.

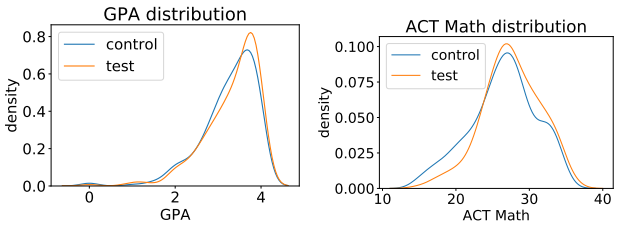
3.1 Study design

Our study was conducted over weeks 3-7 (five weeks in all) in each of two consecutive semesters of a CS1 course at a mid-sized public university in the western United States. The CS1 course is for majors and non-majors, and fulfills a STEM general education requirement. We call the first semester students the control group and the second semester students the test group. Data reported in this paper are drawn only from students that opted into the study according to our IRB protocol, although all students in the course were given identical treatments. See Table 1 for the course layout.

The control and test groups were comparable in terms of GPA and Math ACT score (see Rountree et al. [41] for a discussion of math ability as a predictor for success in programming). See Figure 1. Median GPAs in the control and test groups were 3.40 and 3.54, respectively, with identical interquartile ranges of 0.8; the distributions of the two groups were similar (Mann-Whitney $U = 30132.0$, $n_{\text{control}} = 259$, $n_{\text{test}} = 251$, $p = 0.154$). Median Math ACT scores for

	w3			w4			w5			w6			w7		
	M	W	F	M	W	F	M	W	F	M	W	F	M	W	F
exercises	x	x	x	x	x	x	x	x	x	x	x	x	x		
project			p4			p5			p6			p7			p8
exams									e1						

Table 1: Course layout of weeks 3-7, the weeks of the study. The course was a total of 14 weeks long. Exam2 was at the end of week 9. Exercises were done as marked only in the semester of the test group.



(a) (b)

Figure 1: Distribution between the control and test groups of (a) GPA and (b) Math ACT scores.

both groups were identical at 27, with identical interquartile ranges of 5.0; the distributions of the two groups differed in a rank-sum test ($U = 22522.0$, $p = 0.0134$).

During the five weeks of the study students had in-class lecture three days a week and one programming project due each Friday. Beginning in the third week we introduced "syntax exercises" to the test group, with week seven being the final week exercises were available. Three times per week, before each class lecture, students in the test group were required to log into web-based software called *Phanon* and complete between 20 and 60 syntax exercises. Exercise design is described in detail in the next section.

Each Friday a programming project was due. Projects were labeled p4, p5, p6, p7, and p8 (projects p1, p2, and p3 were non-programming projects on background material and were completed before syntax exercises were introduced). Projects p4-p8 were designed to gradually increase in difficulty. Each project had two parts. One part was based on Turtle graphics and required students to draw specific pictures, such as a snowman or dart board. The other part of the project was a text-based program such as a compound interest calculator; game show simulator; or rock, paper, scissors game. No starter code was given for projects. Projects were graded by course Teaching Assistants. The *Phanon* software, used for syntax exercises, was also used as the IDE for programming projects. Project code must be submitted through *Phanon*, so even if a student wrote their code with a different IDE they must at least paste their result into the *Phanon* editor in order to submit it.

All syntax exercises and programming projects were completed in the Python programming language.

At the end of week 3 of the study (week 5 of the course) a midterm exam, called e1, was given. The exam was composed of four types of questions:

- *Write One* - questions where students were asked to either write or modify a line of code or to provide which operator or function performs a certain function. Example: *To comment a single line, what character(s) are used?*
- *Interpret One* - questions where students were asked if a line of code was correct, to state what results the code would have, or to identify certain types of programming constructs, such as string literals. Example: *What will be the output/result of the following program?* `print(5+2)`.
- *Interpret More* - like *Interpret One* but with more than one line of code. Example: *The identifier* `val` *should be assigned a string that is received from the user. Which of the following lines of code will accomplish this?*
- *General* - questions that asked about languages, types of errors, conventions, software development processes, etc. Examples: *According to standard naming conventions, which of the following would be a proper name for a constant value that designates the maximum value?* and *What type of error prevents a program from running at all?*

Exam e1 had 8 *Write One*, 11 *Interpret One*, 12 *Interpret More*, and 22 *General* questions. The exam was graded using automated tests.

The programming projects and exams were identical for both control and test groups. Project graders were different across semesters. Each semester had three sections, for a total of six sections. All sections but one were taught by the same instructor.

3.2 Syntax exercise design

Each syntax exercise is assessed using automated tests and is designed to take the student between 20 and 40 seconds. Each exercise presents a prompt such as "This code counts from 1 to 10. Change the code so that it counts from 1 to 12." Many exercises include a hint, such as, "*Hint:* change the 10 to a 12." Once the student makes the required changes to the code and successfully runs the program they are briefly presented with a congratulatory banner and are allowed to proceed to the next exercise.

The exercises gradually increase in complexity but always in such a way that the student needn't design code, nor even think very long about what they need to do. For example, after modifying or filling missing portions of multiple `for` loops, a student is instructed to write an entire `for` loop that is identical or almost identical to one they have recently modified. The subsequent exercises have the student write entire `for` loops until, at the conclusion of the session, the student may have typed more `for` loops than they might type over the entire semester of programming projects. Progression of the exercises for other syntax constructs is similar. Only rarely do instructions explain programming or syntax concepts so as to keep cognitive burden low [48] and to allow conceptions to emerge from experiential execution of the exercises [40]. A sample series of exercises are in the appendix.

Each set of exercises is designed to take no more than 10 minutes for any student. The software also allows for "mulligans," where the student can see the solution for an exercise if they get stuck. Mulligans are allowed no more than once every 30 minutes. Copy and paste are disabled in the syntax exercise editor, forcing the student to key in each character required by the solution.

It may be instructive to note what syntax exercises are *not*. They are not assessments of a student's knowledge or ability. Indeed, a student likely will never have seen the syntax construct being practiced. Exercises do not teach how a construct is used. For example, an exercise prompt would not tell the student that `print` statements would be useful for debugging, or that an `if` statement would be useful for conditional execution. Indeed, while students may have no idea what the construct is used for by the end of the session, they are at least confident in their ability to write the construct.

In our design of syntax practice, students read, modify, and complete about 100 code snippets every week. This approach is supported, in part, by the work of Lahtinen et al. [24], who studied survey results regarding difficulties in learning to program. They report that students and teachers felt that far more learning was gained from example programs than any other type of learning material.

The syntax exercises are scheduled such that the class session following exercises on a given construct discuss the construct, what it is used for, and how it is used to solve problems. With this curriculum design, syntax proficiency gained during the syntax exercises scaffolds learning of problem solving in two ways. First, syntax exercises are designed to promote student confidence and interest, encouraging engagement during class. Second, and most importantly, the instructor can feel free to show and discuss code snippets that might ordinarily have left many students lost and confused. Having familiarity with the syntax, students can read code snippets and not only follow along, but also start to understand context and purpose. Outside of the classroom, students working on their programming projects can be more confident in their ability to implement code after they have designed an algorithm to solve the problem. They can focus more on the more challenging task of problem solving, reducing their cognitive load as they are not spending time on frustrating syntax constructions and errors.

3.3 Measures and data

The *Phanon* software used for both syntax exercises and programming projects captures timestamped keystrokes as well as copy and paste events (for projects only, since copy and paste are disabled for exercises) and run events. We measure effort on projects using the number of keystrokes for a student. We measure attrition by comparing the number of students that attempted a project to the number of students that attempted project p4 (the first project in the study). We define attempted as having executed at least one event (e.g. keystroke, paste, run, etc). We also use project scores (graded by humans) and exam scores (graded by automated tests).

In our analysis we perform t-tests and 2-sided proportion z tests. We report p-values of our statistical tests as just one component among others that together contribute to understanding of our results [50]. We do not make threshold-based claims of statistical significance, nor do we use p-values to indicate effect size or importance [6]. In order to interpret p-values appropriately, we report that we conducted 12 statistical tests with accompanying significance measures in this paper. Effect sizes are reported as Cohen's d and h measures.

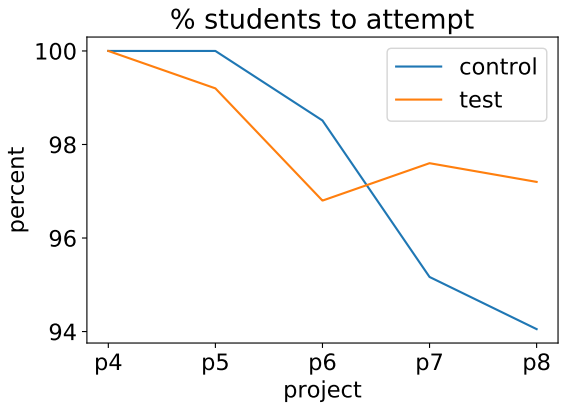


Figure 2: Percentage of students who attempted project p4 that attempted projects p5 through p8. Note that the y-axis starts at 94% to emphasize differences.

3.4 Hypotheses

Based on our research questions and theoretical framework we have the following hypotheses:

- H1 The attrition rate will drop.
- H2 Project and exam scores will increase on average.
- H3 Students will complete their projects more quickly, i.e., with fewer keystrokes.

4 RESULTS

We measure sample size by the number of participating students that attempted project p4, the first project in the period of study. The control group had 271 students and the test group had 254 students. On average, students executed 4401 keystrokes per project, for a total of 11.6 million keystrokes analyzed.

4.1 Attrition

Our first hypothesis was that the attrition rate in the test group (who had syntax exercises) would be lower than the control group (who did not have syntax exercises). See Figure 2. Of all students who attempted project p4, the first programming project during the study, 94.1% of control students attempted project p8, while 97.2% of test students attempted p8, for a reduction in attrition of 3.1% over the five weeks of the study ($z = -1.74$, $p = 0.082$, $h = -0.16$). Results for project p7, which was not as difficult as p8, were less strong (from 95.2% to 97.6% attempt rate; $z = -1.47$, $p = 0.140$, $h = -0.13$), while less difficult projects earlier in the course (p5 and p6) saw little change. If the addition of syntax exercises was the cause of the reduction in attrition for p7 and p8 then it appears that the effect of the exercises may be weaker for easier projects.

It is possible that attrition was reduced not by the procedural skill acquisition elements of syntax exercises but by the gamification and low-fear elements in the syntax exercise experience. Indeed, the gamification elements were introduced to enhance the low-fear nature of syntax exercises, yet may have complicated interpretation of the results.

project	control	test
p4	96.8 ± 11.9	96.5 ± 13.4
p5	92.7 ± 16.9	89.3 ± 21.3
p6	84.8 ± 26.8	84.4 ± 25.3
p7	90.4 ± 25.4	89.5 ± 24.9
p8	78.8 ± 34.1	77.7 ± 31.0
exam1	71.9 ± 15.6	78.9 ± 11.9
exam2	61.6 ± 17.4	63.4 ± 22.4

Table 2: Mean scores with standard deviations of projects and exams.

	m	control mean	test mean	t	p
Write One	8	70.7	86.2	-7.61	$< 1e^{-4}$
Interpret One	11	73.0	83.6	-6.12	$< 1e^{-4}$
Interpret More	12	63.7	65.9	-1.20	0.230
General	22	74.8	84.0	-5.81	$< 1e^{-4}$

Table 3: Means and t-test statistics for differences in scores on exam e1 by question type. m is the number of questions of that type.

4.2 Project and exam scores

Our second hypothesis was that project and exam scores among the test group will be higher than those of the control group on average. The results (see Table 2) held two surprises. The first is that the assignment scores of the test group were actually *lower* than that of the control group. The second surprise was that, while the assignment score differences were opposite our hypothesis, the exam scores were not only consistent with our hypothesis, but the effect size exceeded our expectations: exam e1 was given at the end of the third week of syntax exercises and the scores for the test group were 7 percentage points higher than the control group ($t(264) = -5.87$, $p < 0.001$, $d = -0.51$), a far larger increase than we expected. In Table 3 we see that all question types showed a increase in score for the test group, with the strongest evidence of a statistically observable effect in the *Write One*, *Interpret One*, and *General* question types.

The effect of exam score improvement is diluted in exam e2, with test group scores just less than 2% higher on average than control group scores ($t(264) = -1.04$, $p = 0.299$, $d = -0.09$). This is consistent with our hypothesis because exam e2 took place two weeks after the conclusion of the administration of syntax exercises, so half of the exam was on material not supplemented by syntax exercises.

An alternative explanation for the improvement in exam performance could be that the second semester of the study used an identical exam to the first semester for comparison purposes, whereas the exam was changed each semester prior to the study. This put the students in the test group at an advantage if they chose to cheat, which could explain the improvement in the score for the exam e1. However, this explanation breaks down somewhat when explaining the reduction in improvement for exam e2.

The result that students with syntax exercises got *lower* project scores than the control group is unexpected, but the fact that the effect is reversed when is comes to the exam suggests that some other variable may be coming into play. We discuss the possibility of a reduction in plagiarism as an explanation for project-exam differences in Section 5.1. Of course, the lack of effect in project scores also suggests that perhaps the null hypothesis should not be rejected.

4.3 Number of keystrokes on projects

Our final hypothesis was that students in the test group would complete their projects with fewer keystrokes. Again, a surprise: students in the test group required *more* keystrokes on average. At project p4 (the first project in the study) both groups used roughly the same number of keystrokes but by project p8 they had diverged. In project p8 the median number of keystrokes in the control group was 3896 (IQR: 1284 – 7051) while the test group median was 7207 (IQR: 3963 – 10095). See Figure 3a for the distribution of keystrokes. In investigating this result we discovered that the control group had many more projects with large amounts of code pasted into the editor than the test group. (Recall that only the syntax exercises had paste disabled, so students were able to cut and paste as normal in the programming projects.)

The difference in pasting behavior between groups and plagiarism as a possible cause are discussed in detail in Section 5.1. We control for large pastes by excluding all projects for which the total number of pasted characters exceeded the total number of keystrokes by a factor of four. (Four is a somewhat arbitrary factor, but it is robust: we tested with 1, 2, 8, and 16 with very similar results.) This measure is meant to flag projects, which we call "pasted" projects, where the work was done primarily outside of the *Phanon* IDE. See Figure 3 for the effect of controlling for pasting. Figure 3a shows the normalized distribution, regardless of whether projects were pasted or not, of the number of keystrokes for project p8. (Other projects show similar behavior, as shown in Figure 4.) Both control and test groups have a somewhat normal appearing curve between 5000 and 10000 keystrokes, but they both also have a spike near zero, with the control group spike dwarfing that of the test group. From Figure 3b, which is the same chart but excluding pasted projects, we see that pasted projects strongly skewed the number of keystrokes toward zero, with the larger number of pasted projects in the control group skewing the results further than the test group. We see from Figure 3c that all pasted projects have very few keystrokes, showing that pasted projects were, indeed, developed outside of the students' *Phanon* IDE, i.e., the high paste rates were not caused by students simply cutting and pasting their code over and over.

Returning to our hypothesis that students in the test group would require fewer keystrokes, we controlled for pastes in keystroke distributions (see Figure 5). Even after controlling for pastes our hypothesis was still opposite what actually happened: again, students with syntax exercises required *more* effort to complete projects than those without. We see three possible explanations for the increase in effort. The first is that an uncontrolled variable is the cause. Of the possible variables listed in our threats to validity (Section 5.5) we believe the most likely variable to be the difference in student

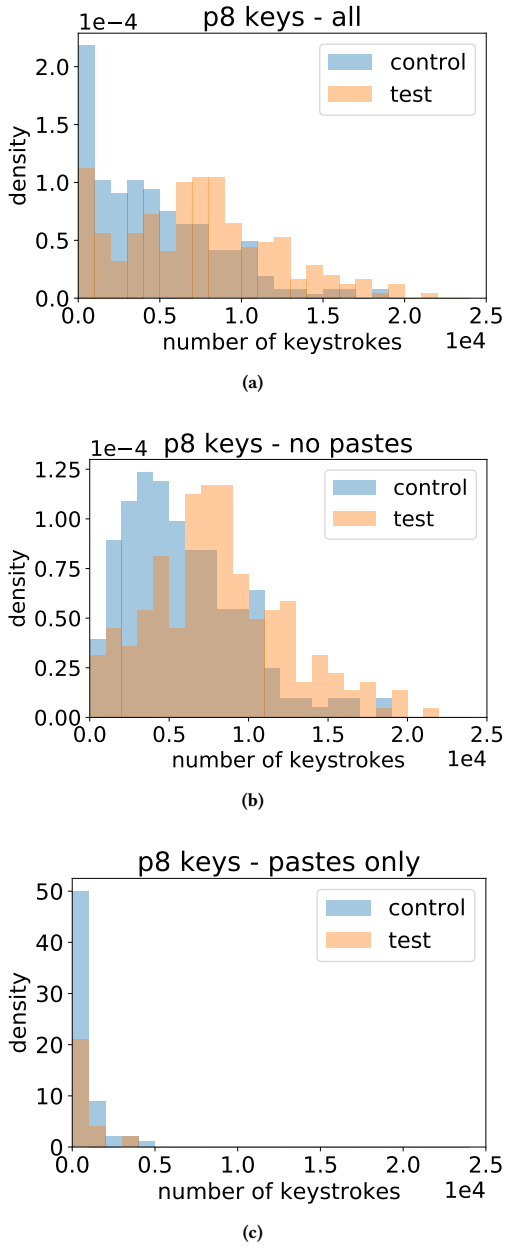


Figure 3: Normalized distribution of number of keystrokes used in completing project p8. Both control and test groups have curves that would appear to be somewhat normal except for the spike near zero, which are primarily students who pasted their code into *Phanon*. (a) Distribution of number of keystrokes for all submissions. (b) Distribution of number of keystrokes for only projects that were not pasted (see Section 4.3). (c) Distribution of number of keystrokes for only projects that were pasted. Not normalized.

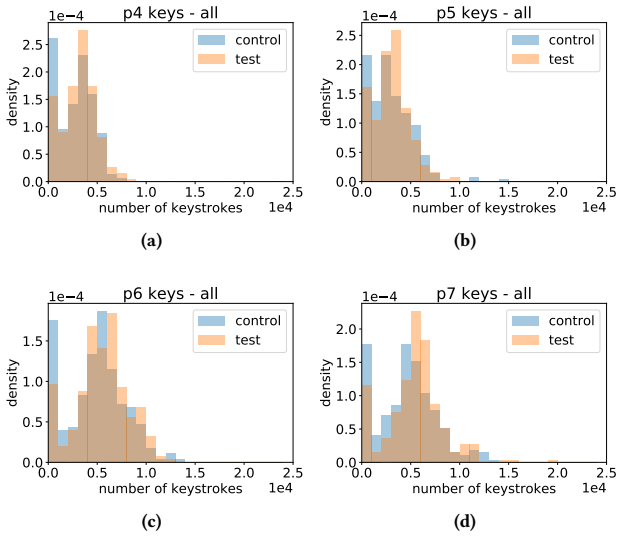


Figure 4: Normalized distribution of number of keystrokes used in completing projects p4-p7.

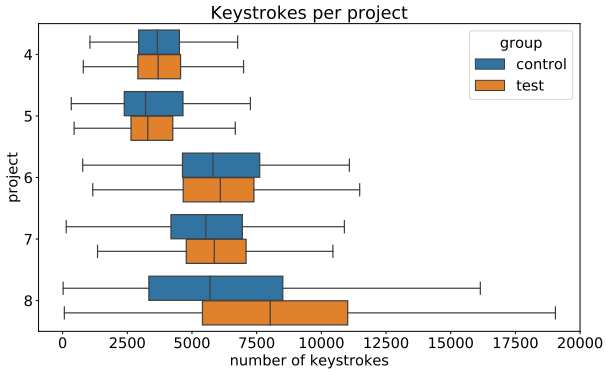


Figure 5: Number of keystrokes per student in projects. Outliers are not shown. Student submissions in which significant amounts of code were pasted are not included.

preparation and ability from spring to fall semester. Given the similar ACT and math GPA distributions shown in Figure 1 we see this as somewhat unlikely.

A second explanation is that syntax exercises cause a direct increase in keystrokes: possibly students are more practiced at typing and thus tinker more with the code or are more likely to type everything when refactoring instead of moving code with cut and paste.

A third explanation is that because of syntax exercises, or some other factor, less-prepared students are attempting and remaining engaged with programming projects. Reduction in attrition could cause this: students who normally would have dropped out or disengaged from the course may have continued on and required more effort to complete the projects than students who were less likely to drop out. Fewer students from the test group disengaged

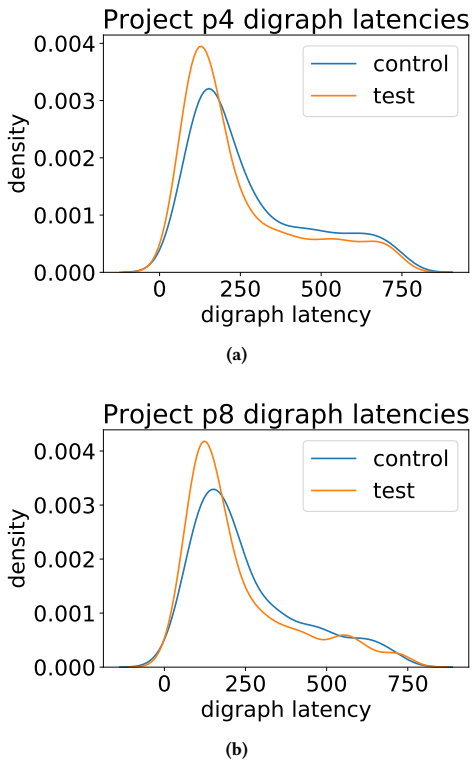


Figure 6: Distribution of digraph latencies for (a) project p4 and (b) project p8. Distributions for the other projects are similar.

and those are the very students that we would expect to require more time to complete their projects. Reduction in plagiarism could also contribute as we discuss in Section 5.1.

Interpretation of the increase in number of keystrokes is difficult to interpret in our study design. We do note, however, that average digraph latencies for project p4 are 35ms lower, on average, for the test group ($t(784) = 3.88$, $p = 0.0001$, $d = 0.18$), and project p8 has similar results (see Figure 6). Digraphs are pairs of keystrokes and are often used to measure and predict student performance [13]. The latency of a digraph is the amount of time elapsed between the two keystrokes of the digraph, and is typically measured in milliseconds. Thus, faster typists have lower digraph latencies. In Figure 6 we consider the distribution of digraphs that are between 30 and 750 ms, as faster than 30 ms indicates an error in measurement and greater than 750 ms indicates disengagement from typing [10, 26]. A marked decrease in the latencies for students in the test group indicates that they are taking less time on projects at least in the typing of the code. We note that while a difference in attrition between the control and test groups doesn't begin to appear until the third and fourth projects, the difference in digraph latency appears immediately, in the very first programming project during the study (Figure 6a). This is because the syntax exercises target programming constructs that are used in the project that follows the exercises, so the effect is immediate, while attrition differences are

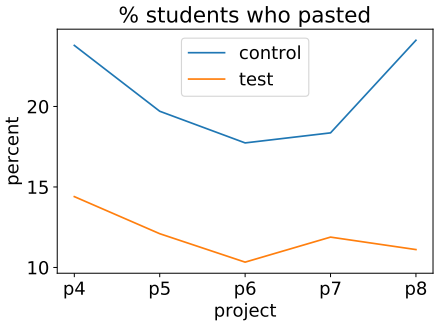


Figure 7: Percentage of students who pasted code across projects. Note that the y-axis starts at 10% to emphasize differences.

latent due to the easiness of early projects followed by increasing project difficulty.

5 DISCUSSION

In this section we discuss plagiarism as a potential hidden variable that could help explain some of our unexpected measurements. We also relate our results back to cognitive load theory and discuss practice as an exploration tool. We then discuss some practical implications of our results.

5.1 Plagiarism

We here explore how plagiarism may have played a role in the increased keystroke counts and the fact that exam scores went up while project scores held steady or decreased.

We begin by noting the difference between the test and control groups in the number of students who pasted their project code. In Figure 7 we see that in project p8, 24% of control students pasted their code, while only 11% of test students pasted ($z = 3.79$, $p = 0.0002$, $h = 0.35$). Recall from Section 4.3 that we consider a project to be "pasted" when the total number of pasted characters exceeds the total number of keystrokes by a factor of four, so the number of keystrokes must be very small relative to the amount of pasted code. We explore two possible causes of the drop in "pasted" projects.

First, we discovered that a small number of students, usually high-performing students who had prior programming experience, wrote their code in a more sophisticated IDE (typically PyCharm) and then pasted their solutions into *Phanon*. It is possible that fewer students in the test group used an IDE other than *Phanon* because of their familiarity with *Phanon* – syntax exercises could only be completed in *Phanon* since paste was disabled for the exercises. Thus, student familiarity with *Phanon* could be an explanation for the drop in pasted projects.

A drop in plagiarism could be a second reason the test group had fewer pasted projects. While we don't know whether to attribute the drop in pasting to a reduction in plagiarism or to a reduction of writing code in another IDE, we can infer with some confidence that a reduction in plagiarism is at least a major contributing factor: recall the decrease in project scores but increase in exam scores in the test group (Table 2). A high rate of plagiarism naturally leads to high assignment scores but lower exam scores due to students not learning the material, explaining Table 2. And similar to the effects of reduced attrition, the test students who would have plagiarized had they been in the control group may very well be the students who would require more keystrokes to complete a project, explaining Figures 3a and 5. Additional evidence of a reduction in plagiarism comes from digraph, or typing speed, analysis (see Figure 6). While students in the test group were using more keystrokes to complete projects, they were typing faster, an effect that has been linked to success in CS1 [13, 26], supporting the claim that, while students in the test group received lower project scores, they were more prepared for success in the course, something that couldn't be said of students who plagiarize. The hypothesis that plagiarism is reduced with syntax exercises is consistent with every expected and unexpected result from our study: syntax exercises resulting in lower project scores, higher test scores, lower digraph latencies, and more keystrokes on projects, on average.

5.2 Cognitive load and deeper learning

One emergent finding from this work is that the intervention appears to have enhanced students' overall learning in the class, as indicated by exam results. At a glance, it may seem surprising that an intervention focused on syntax would have a strong impact on learning more broadly. However, this is anticipated by theory, at least in part. Recall that the purpose of preparatory exercises was not merely to improve syntax, but to reduce the cognitive load of students engaged in problem solving activities. This would, in turn, allow students to devote more of their cognitive resources to problem solving, resulting in greater learning overall.

However, an alternative, or complementary, explanation is also possible for these results. Writing from an embodied cognition perspective, Trninic [48] writes that "repetitive exercises, sometimes derided as drills, can nonetheless be a fountainhead of insight and discovery" (p. 146). Drawing on motor skill learning theory [4] and the neural reuse theory [2], Trninic argues that conceptual understanding actually emerges from engaging in and reflecting on varied procedural actions. The importance of varied practice is crucial, since it provides opportunities for the practitioner to notice the relevant underlying patterns. Teachers support this process by challenging the learner to reflect on and signify their actions "within a discipline's semiotic system" (p. 149). In other words, it is not merely that practice results in fluency and automation that frees up valuable working memory; the very act of repetitive practice, when exercises are well-designed, provides opportunities for reflection and deeper learning.

5.3 Implications

While a cursory evaluation of our hypotheses and measurements overall would indicate mixed results, a careful look suggests that students with the intervention could be more prepared for success in CS1.

We note that the study done by Edwards et al. [14] resulted in an increase of project scores and decrease of time spent on projects, and one could ask why our measurements were mixed while theirs were not. In the 2018 study the test group not only had syntax exercises but they also did 2/3 of their projects in class in a pair programming

setting while the control group did theirs using more traditional methods. The in-class pair programming may have affected both the attrition and plagiarism rates, making comparison between their study and ours tenuous.

That said, our study presents a number of important results. The first is the possible effect of syntax exercises on attrition. The fact that so few students disengaged when they had syntax exercises has important implications for a course with chronically high attrition [3]. A second result is the improvement in exam scores, indicating improvement in learning outcomes. A study where syntax exercises are used through an entire semester would reveal insights into their effectiveness with more complex syntactical structures (e.g. nested loops, procedures, classes) as well as the effect of syntax exercises on final grades.

While conclusions regarding project scores and effort in terms of number of keystrokes can't be made, the unexpected possibility that syntax exercises discourage plagiarism is a strong motivation for further study. First, a study that isolates the plagiarism variable would help shed additional light on our conjecture regarding plagiarism, although we recognize that such studies are difficult to design and execute. An additional, and possibly even more revealing study would be to explore the effects of syntax exercises on attitude and how that might affect students' willingness to plagiarize or drop out. Indeed, while the theories on which we based our hypotheses and study say a lot about performance, they don't necessarily have a lot to say about attitudes.

On a practical note, the results indicate that syntax exercises have worth and, happily, they require only moderate effort for both the instructor and the student. From the perspective of instructor effort, sets of syntax exercises can be created for a given language and reused across semesters and across instructors. With syntax exercises designed to be done with little cognitive effort and at a high rate, and with paste functionality disabled, cheating doesn't make a lot of sense as it is more work for the student than it is worth. So exercises don't need to be modified between terms even if exercise solutions are readily available to students. From the student's perspective, syntax exercises are also relatively lightweight. On average, students spent 8 minutes on each set of exercises, for a total of 24 minutes per week. Indeed, in light of the potential criticism that it is no surprise that additional practice leads to improved outcomes, we posit that not just any type of additional practice would yield this much return on investment by the student.

5.4 Expertise reversal effect

While we have suggested that syntax exercises are designed to be most effective for novice programmers, we have not presented any direct measurements supporting this, nor do we discard the idea that they may be effective for experienced programmers as well. Nevertheless, we must take care to not alienate the experienced programmer: expertise reversal effect theory describes the phenomenon of instructional techniques that are highly effective for novices losing their effectiveness and even becoming counterproductive for students with more aptitude [21]. Our anecdotal evidence from interactions with students suggests that even experienced students didn't mind the exercises and may have found them to be helpful to help cement fluency [46].

5.5 Threats to validity

Our study was conducted across two semesters with the control group in the spring and the test group in the fall, possibly causing sample bias. One instructor taught five of the six sections, with a different instructor teaching one section in the spring. Neither instructor was involved in the study during the study period beyond agreeing to use identical teaching and assessment methods across both semesters and using syntax exercises (written by the first author) in the fall. While both semesters of students had the same access to a tutoring lab, personnel manning the lab, as well as graders, were not the same across semesters. The difference in graders could have had an effect on project scores, but exams were graded using automated tests.

6 CONCLUSIONS

We have presented results of a study looking at the effect of syntax exercises in a CS1 course. Simply adding approximately 25 minutes per week of structured syntax practice to our course resulted in higher exam scores and lower attrition rates after 5 weeks. Remaining results were mixed: project scores did not improve and project completion times actually increased. We have suggested reduced plagiarism rates as a confounding variable which, if true, would have important implications in courses where cheating is an issue.

If replication studies support our findings, we suggest three important implications of this work. First, and most importantly, a very simple modification to a CS1 curriculum with almost no effort by the instructor (if they use syntax exercises that are already available) can result in gains in student outcomes. Furthermore, the added load to the student may be low enough that existing programming projects needn't be modified. This highly practical result can have an immediate effect on CS1 courses.

Second, the results are consistent with the hypothesis that the proposed pedagogy improves student outcomes particularly among students who normally may fail or drop the course, possibly increasing CS graduation numbers.

Third, the theories of cognitive load and embodied cognition are supported by our results and may lead to additional innovations as we identify ways we can reduce cognitive load, particularly utilizing repetitive practice that can support and encourage deeper learning.

Our work leads to a number of questions for future work. The CS1 course used for this study fulfilled STEM general education requirement for the university, and thus had a mix of CS majors and non-majors. It is possible that the effects of syntax could be quite different, especially in courses designed for science and engineering majors (e.g. [8]), a candidate context for a replication study. In our study, in-class lectures were essentially the same, including some content on syntax, unnecessarily so for the test group. We would be interested to see the effect of new lectures that assume a familiarity with syntax. Similarly, our exam design did not consider the master learning approach of syntax exercises – designing our assessments taking the exercises into account could have been beneficial both to student outcomes and interpretation of our results [32]. It would also be interesting to study the effect of implementing practice schedules [20] in syntax exercises. And lastly, what types of students, novices and/or those with experience, are helped most by syntax exercises is an open question.

APPENDIX

Following is a series of syntax exercises teaching `for` loops in Python. This is the first time the student will have seen `for` loops. The instructions are shown in italics and the starter code follows in monospace font.

1. Run the code. Change it so that it outputs

```
0
1
2
3
```

Hint: you will change the 3 to a 4.

```
for i in range(0, 3):
    print(i)
```

2. This code has a `for` *loop. Change the code so that it outputs*

```
0
1
2
```

```
for i in range(0, 4):
    print(i)
```

3. Change the code in the `for` *loop so it outputs*

```
0
1
2
3
4
```

```
for i in range(0, 3):
    print(i)
```

4. Change the code in the `for` *loop so it outputs*

```
1
2
3
```

Hint: you will change the `0` *to a* `1` *in the* `range()` *call.*

```
for i in range(0, 4):
    print(i)
```

5. Change the code so it outputs

```
13
14
15
```

```
for i in range(0, 16):
    print(i)
```

6. Fill in the missing code to output

```
1
2
3
```

```
for i in :
    print(i)
```

7. Fill in the missing code to output

```
0
1
2
3
4
```

```
for range(0, 5):
    print(i)
```

8: Fill in the missing code to output

```
0
1
2
```

Hint: the colon at the end of the first line is missing.

```
for i in range(0, 3)
    print(i)
```

** Exercises 9-11 not included **

12. Change only line 2. Fix the bug so that the following is output:

```
0
1
2
3
```

Hint: `j` *was accidentally put in the* `print()` *call instead of* `i`*.*

```
for i in range(0, 4):
    print(j)
```

13. Write a `for` *loop to output*

```
0
1
2
3
```

** Exercises 14-20 not included **

21. Write a `for` *loop to output*

```
12
13
14
15
16
```

REFERENCES

[1] John Robert Anderson, C Franklin Boyle, Robert Farrell, and Brian J Reiser. 1984. *Cognitive principles in the design of computer tutors.* Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF PSYCHOLOGY.

[2] Michael L Anderson. 2010. Neural reuse: A fundamental organizational principle of the brain. *Behavioral and brain sciences* 33, 4 (2010), 245–266.

[3] Theresa Beaubouef and John Mason. 2005. Why the high attrition rate for computer science students: some thoughts and observations. *ACM SIGCSE Bulletin* 37, 2 (2005), 103–106.

[4] NA Bernstein. 1996. Essay 6: on exercises and motor skill. *Dexterity and its development. Lawrence Erlbaum, New Jersey* (1996), 171–205.

[5] Duane Buck and David J Stucki. 2000. Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. *ACM SIGCSE Bulletin* 32, 1 (2000), 75–79.

[6] Ronald P Carver. 1993. The case against statistical significance testing, revisited. *The Journal of Experimental Education* 61, 4 (1993), 287–292.

[7] Stephen Cooper, Wanda Dann, and Randy Pausch. 2000. Alice: a 3-D tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, Vol. 15. Consortium for Computing Sciences in Colleges, 107–116.

[8] Paul Denny, Andrew Luxton-Reilly, Michelle Craig, and Andrew Petersen. 2018. Improving complex task performance using a sequence of simple practice tasks. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. 4–9.

[9] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. 208–212.

[10] Paul S Dowland and Steven M Furnell. 2004. A long-term trial of keystroke profiling using digraph, trigraph and keyword latencies. In *IFIP International Information Security Conference*. Springer, 275–289.

[11] Hubert L Dreyfus and Stuart E Dreyfus. 1999. The challenge of Merleau-Ponty's phenomenology ofâĂĺembodiment for cognitive science. *Perspectives on embodiment: The intersections of nature and culture* (1999), 103.

[12] Benedict Du Boulay. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73.

[13] John Edwards, Juho Leinonen, and Arto Hellas. 2020. A Study of Keystroke Data in Two Contexts: Written Language and Programming Language Influence Predictability of Learning Outcomes. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 413–419.

[14] John M Edwards, Erika K Fulton, Jonathan D Holmes, Joseph L Valentin, David V Beard, and Kevin R Parker. 2018. Separation of syntax and problem solving in Introductory Computer Programming. In *2018 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–5.

[15] K Anders Ericsson. 2002. Attaining excellence through deliberate practice: Insights from the study of expert performance. In *The Pursuit of Excellence Through Education*, Michel Ferrari (Ed.). Lawrence Erlbaum Associates Publishers, Mahwah, New Jersey, 21–55.

[16] K Anders Ericsson and Neil Charness. 1994. Expert performance: Its structure and acquisition. *American psychologist* 49, 8 (1994), 725.

[17] Paul M Fitts. 1962. Factors in complex skill training. *Training research and education* 1962 (1962), 177–197.

[18] Paul M Fitts and Michael I Posner. 1967. Human performance. (1967).

[19] Adam M Gaweda, Collin F Lynch, Nathan Seamon, Gabriel Silva de Oliveira, and Alay Deliwa. 2020. Typing Exercises as Interactive Worked Examples for Deliberate Practice in CS Courses. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*. 105–113.

[20] Addie Johnson. 2012. Procedural memory and skill acquisition. *Handbook of Psychology, Second Edition* 4 (2012).

[21] Slava Kalyuga. 2009. The expertise reversal effect. In *Managing cognitive load in adaptive multimedia learning*. IGI Global, 58–80.

[22] Paul A Kirschner, John Sweller, and Richard E Clark. 2006. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational psychologist* 41, 2 (2006), 75–86.

[23] Sarah K Kummerfeld and Judy Kay. 2002. The neglected battle fields of Syntax Errors. Australian Computer Society. In *Proceedings of the fifth Australasian conference on Computing education*. 105–111.

[24] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A Study of the Difficulties of Novice Programmers. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '05)*. ACM, New York, NY, USA, 14–18. https://doi.org/10.1145/1067445.1067453

[25] Antti Leinonen, Henrik Nygren, Nea Pirttinen, Arto Hellas, and Juho Leinonen. 2019. Exploring the Applicability of Simple Syntax Writing Practice for Learning Programming. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 84–90.

[26] Juho Leinonen, Krista Longi, Arto Klami, and Arto Vihavainen. 2016. Automatic inference of programming performance and experience from typing patterns. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 132–137.

[27] Marcia C Linn and John Dalbey. 1985. Cognitive consequences of programming instruction: Instruction, access, and ability. *Educational Psychologist* 20, 4 (1985), 191–206.

[28] Raymond Lister. 2011. Programming, syntax and cognitive load (part 1). *ACM Inroads* 2, 2 (2011), 21–22.

[29] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, et al. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin* 36, 4 (2004), 119–150.

[30] Gordon D Logan. 1990. Repetition priming and automaticity: Common underlying mechanisms? *Cognitive Psychology* 22, 1 (1990), 1–35.

[31] Gordon D Logan. 2005. Attention, Automaticity, and Executive Control. In *Experimental cognitive psychology and its applications*, A. F. Healy (Ed.). American Psychological Association, 129–139.

[32] Andrew Luxton-Reilly, Brett A Becker, Yingjun Cao, Roger McDermott, Claudio Mirolo, Andreas Mühling, Andrew Petersen, Kate Sanders, and Jacqueline Whalley. 2018. Developing assessments to determine mastery of programming fundamentals. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*. 47–69.

[33] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*. ACM, 125–180.

[34] George A Miller. 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review* 63, 2 (1956), 81.

[35] A. Newell and P. Rosenbloom. 1981. Mechanisms of skill acquisition and the law of practice. In *Cognitive Skills and Their Acquisition*, J.R. Anderson (Ed.). Erlbaum, Hillsdale, New Jersey.

[36] Dave Oliver and Tony Dobele. 2007. First year courses in IT: A bloom rating. *Journal of Information Technology Education: Research* 6 (2007), 347–360.

[37] Scott R Portnoff. 2018. The introductory computer programming course is first and foremost a language course. *ACM Inroads* 9, 2 (2018), 34–52.

[38] Robert W Proctor and Addie Dutta. 1995. *Skill acquisition and human performance*. Sage Publications, Inc.

[39] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[40] Wolff-Michael Roth and Jennifer S Thom. 2009. Bodily experience and mathematical conceptions: From classical views to a phenomenological reconceptualization. *Educational studies in mathematics* 70, 2 (2009), 175–189.

[41] Nathan Rountree, Janet Rountree, Anthony Robins, and Robert Hannah. 2004. Interacting factors that predict success and failure in a CS1 course. *ACM SIGCSE Bulletin* 36, 4 (2004), 101–104.

[42] Carsten Schulte. 2008. Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceedings of the Fourth international Workshop on Computing Education Research*. 149–160.

[43] Daniel L Schwartz and Taylor Martin. 2004. Inventing to prepare for future learning: The hidden efficiency of encouraging original student production in statistics instruction. *Cognition and Instruction* 22, 2 (2004), 129–184.

[44] Larry R Squire. 1984. Human memory and amnesia. *The neurobiology of learning and memory* (1984).

[45] Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 19.

[46] Shelsey Sullivan. 2020. *An analysis of syntax exercises on the performance of CS1 students.* Master's thesis. Utah State University.

[47] John Sweller, Jeroen JG Van Merrienboer, and Fred GWC Paas. 1998. Cognitive architecture and instructional design. *Educational psychology review* 10, 3 (1998), 251–296.

[48] Dragan Trninic. 2018. Instruction, repetition, discovery: Restoring the historical educational role of practice. *Instructional Science* 46, 1 (2018), 133–153.

[49] Tamara Van Gog and Nikol Rummel. 2010. Example-based learning: Integrating cognitive and social-cognitive research perspectives. *Educational Psychology Review* 22, 2 (2010), 155–174.

[50] Ronald L Wasserstein and Nicole A Lazar. 2016. The ASA statement on p-values: context, process, and purpose.

[51] David Weintrop and Uri Wilensky. 2017. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 3.

[52] Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Andrew J Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253.