Separation of syntax and problem solving in Introductory Computer Programming

John M. Edwards*, Erika K. Fulton†, Jonathan D. Holmes*, Joseph L. Valentin*, David V. Beard* and Kevin R. Parker*
*Department of Informatics and Computer Science †Department of Psychology
Idaho State University, Pocatello, Idaho 83209
Email: {edwajohn, fulterik, holmjona, valejose, beard, parkerkr}@isu.edu

***Abstract*—In this research work in progress paper, we discuss the possible benefits of separating syntax practice from problem-solving learning in an Introductory Computer Programming course. We propose a curriculum and associated development tool called *Phanon* that teach the rudiments of programming language through exercises done online outside of class. Having students complete exercises before class frees up classroom time and instructor face time for the higher-order learning tasks of problem decomposition and solving. We report results from a pilot study that are consistent with our hypothesis that these techniques result in improved student outcomes and attitudes and we discuss a future follow-up study.**

***Index Terms*—computer science, programming, education**

I. Introduction

Introductory Computer Programming (CS1), a course that teaches students to solve problems using a computer programming language, generally has low pass-rates (typically around 68% [2], [34]) and a large amount of research has been done to improve student outcomes. A major, identified barrier to student success in CS1 courses is language syntax [30]. Students tend to become frustrated by the mechanics of computer programming limiting their ability to master the problem-solving techniques that CS1 courses generally identify as the primary targets. Block-based languages such as Scratch [25] and Alice [6] have become popular in recent years as they remove syntax entirely from the language, enabling direct solution design. A study showed improved outcomes and attitudes in using a block-based approach [35], however, attitudes and confidence returned to levels of the control group after all students transitioned to a text-based language [36].

In our work, we explore pedagogy that promotes mastery of programming language syntax before attempting to solve problems using programming. We make the claim that language syntax and mechanics learning lies at the lower levels of Bloom's Taxonomy (knowledge, comprehension, and application) while problem solving using language mechanics has place in higher levels (analysis and synthesis). We investigate whether repetition to gain fluency in language syntax leads to more effective learning in problem solving as well as improved student attitudes. We also investigate the effectiveness of student-regulated active learning in the classroom to learn problem solving. We have developed a software tool called *Phanon* that supports our pedagogy and curriculum and have used it in a pilot study involving two sections of CS1.

We discuss related work in separating syntax from problem solving as well as pair programming and exercise-based learning in section II. Section III discusses our curriculum design and study design. We report our findings in section IV. We anticipate that the success of our investigations could lead to a radically new course structure with implications in both higher education and high school computer science instruction, which we discuss in section V.

II. Related work

For over 40 years computer scientists and psychologists have studied how novice programmers taking a CS1 class have learned how to code [16], [17], [27]. CS1 is a difficult class with a typical pass rate of only 68% [2], [34] and even students who achieve a good grade often leave with inadequate understanding of the course content [18]. This is due to the fact that very often good grades are achieved through memorization without sufficiently analyzing a problem before attempting to solve it [9], [38]. Inadequate metacognition is another reason Computer Science students underachieve [24], [33].

A. Syntax before problem solving

Syntax has been identified as an "extraneous cognitive load" [14], [15] and "significant barrier" [12], [30] to the more important objective of creative problem solving. The ideal of a CS1 education would be to achieve the mechanics learning quickly and focus primarily on problem solving [4], [7], [13], [28]. As Oliver and Dobele [22] argued, students begin the learning process at lower levels of Bloom's Taxonomy [3] before reaching later ones that require more abstract thinking [37]. As Mayer puts it, syntax is "learning to program" and problem solving is "learning to think" [17].

B. Pair programming

Despite their popularity [34], traditional, large-class lectures have been shown to be inferior to other methods [10], [26], [29] such as consistent assessment [11], [32], collaborative learning [1], [39], and problem-based learning [8]. Collaborative problem solving has been shown to be more effective than traditional lectures, especially in motivating students [26]. Pair programming is one implementation of collaborative learning that is specific to computer programming education and has been shown to have beneficial effects on student learning [5], [19]–[21].

978-1-5386-1174-6/18/$31.00 ©2018 IEEE

C. Exercise-based learning

A number of different curricula and auxiliary websites have been appearing in recent years that offer programming exercises. CodingBat [23] provides online exercises in Python and Java and is often used as a supplement in CS0 and CS1 courses. CodeKata [31] is similar, but the exercises are designed more for computational thinking and take 30-60 minutes each. Various textbook publishers include online exercises to accompany their textbooks. These exercises are usually similar to CodeKata in that they exercise computational thinking. We know of no curriculum or pedagogy in computer programming that uses exercises in the manner we propose, that is, large numbers of short exercises with the express purpose of giving students familiarity with and mastery of programming language fundamentals and not necessarily problem-solving strategies.

III. Methods

A. Curriculum design

The central piece of our curriculum is mechanics exercises. The exercises teach the student language syntax, or mechanics, through example. Two other complementary pedagogical devices we are exploring are smaller, higher-frequency problem-solving projects, approximately one per day of classroom instruction, and active learning through pair programming.

Exercises are designed to be a tool to teach concepts that are at low levels of Bloom's taxonomy, primarily mechanics and syntax for a specific language. This is inspired by Linn's suggestion that syntax learning be a separate objective from solving programming problems [13]. Exercises are administered via a web-based tool and are progressive in that exercise e_i must be completed before attempting e_{i+1}. Little instruction is given, relying instead on repeated code reproduction and synthesis with progressively reduced helps per exercise. At the beginning of the course almost no reasoning power is required for the exercises. As the course progresses the required mental effort increases, but never to the level of program design. An example exercise early in learning Python iteration is:

Instructions: Print 0 through 3 by replacing xxx *with a range call.*

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

A later exercise in the same set is:

Instructions: Write a for loop that prints all even numbers from 0 to 20.

```
# add code here
```

The student's solution is graded automatically using a set of hidden test procedures. Each exercise is designed to take 5-60 seconds with no more than 60 exercises in a given session. The student is presented with a congratulatory green banner after completing each exercise. The exercise code window has copy/paste disabled, so the student is forced to key in all solution code.

While in traditional CS1 offerings a student may write 20 loops over the course of a semester, students using our approach may write 40 in a single day, and many hundreds over the course of the semester. The reasoning for exercises is drawn from forms of art that require repetition to master fundamentals. For example, a pianist must learn and practice certain exercises and scales in order to achieve technical proficiency. [1] We claim that computer programming is no different. While we agree that problem solving is, and should be, the primary objective of the course, and that program design is best practiced through projects, spending time to master language mechanics can benefit the student in two specific ways. The first is that problem solving can be done without the hobbling experience of syntax errors. We propose that mastering language elements through exercises will result in reduced time and frustration in completing problem-solving projects. The second benefit of exercises is that students are given a large number of successes, one per exercise, and so the student comes to the problem-solving phase of learning with high confidence.

Programming projects are assigned to the students each class day. A student may not start a project unless they have completed the mechanics exercises for the day. We expect that students will be able to focus more effectively on the problem-solving skills having already obtained some mastery of the language mechanics. At the beginning of class each student who has completed the exercises is assigned a random partner with whom to pair program the assigned project. No verbal instruction is given to the class on how to complete the project, but the written instructions often include some guidance. While the students work together the classroom instructor answers questions individually as needed. If the project is not completed by the end of class students can either finish on their own or meet again as pairs outside of class to finish together. A student may not work with anyone other than their assigned partner for a given project. One project per week is done individually. Projects are an even mix of turtle graphics-based problems and text-only problems. Typical projects include drawing geometric shapes and computing statistical measures on numbers input by the user.

B. Study design

We conducted a small-scale pilot study that makes no causative claims, but rather tests for statistically detectable effects to justify a larger scale study the following term. We used two sections of CS1 at our university as our testbed. Students were given the choice to participate in the study: 17 of 18 students in the test section chose to participate and 18 of 25 in the control section participated. All programming

[1] Indeed, the name of our software tool *Phanon* is based on the piano metaphor. Virtually all accomplished pianists are familiar with a book of exercises entitled "The Virtuoso Pianist" by Charles-Louis Hanon. The name *Phanon* is a portmanteau of the words *Programming Hanon*. We have also given the *Phanon* software the subtitle "The Virtuoso Programmer".

was done using our *Phanon* web-based software in Python. Common elements between the two courses included a weekly programming project and exams. Tutoring hours were identical for both sections.

The test section was conducted as follows: before each class period a student logs onto the *Phanon* website and is presented a series of 20-60 mechanics exercises on a language concept such as iteration or conditionals. The student steps through the exercises, advancing to the next only after completing the current one. The test section class was held MWF. On Monday, classtime was used for individual work on the weekly project. Students could finish the project outside of class if they were unable to complete it during class. The project was due Wednesday before class. At the beginning of class each Wednesday and Friday, students were randomly paired and then each pair worked together on a single laptop on an additional programming project. So students in the test section completed, on average, three programming projects per week in addition to the exercises. No classroom instruction was given beyond individual help. The instructor for the test section was a Junior-level undergraduate student majoring in Computer Science with no previous teaching experience.

The control section was taught in a more traditional manner. It was also taught MWF and classtime was used for lecture. The weekly project was identical to the one given in the test section and was also due on Wednesdays. Projects were done individually outside of classtime. (Note that the weekly project for the test section was started in class with the instructor available to answer questions and completed out of class if necessary.) The instructor for the control section was an experienced faculty instructor.

Exams were administered through *Phanon* and were identical between the two sections.

After each set of exercises the student was given a survey asking perceived value and metacognitive questions. Similar questions were also asked pre- and post-project. Project attitudinal survey results are reported in section IV and metacognitive results will be reported in a separate paper.

We fitted the *Phanon* software with a measurement instrument that tracks the usage time of students. We measured how long each student took in completing exercises and projects. Measuring exercises was straightforward, but we discovered early in the term that some students were developing some of their project code outside of the *Phanon* development environment and then pasting the code into *Phanon*. (Note that while exercises had copy/paste disabled, projects did not so that students would be free to move their code around during refactoring.) For this reason we use the median project time for students in our results reporting.

C. Assessment

We assess in two broad categories: competence and perceived value. Competence and perceived value assist in determining whether our efforts will lead to a more proficient and larger workforce, respectively.

Competence is the measure of computer programming proficiency and is measured by exam scores, project scores, and time taken to complete tasks.

Perceived value measures help us to understand and refine comfort level with programming and increased interest in the subject matter. These measures are in three forms: the first is in the form of an attitudinal survey, where at the conclusion of a project the software prompts the student to answer four questions regarding their experience. Two other perceived value measures are pre- and post-course attitudinal surveys and our institution's post-course student evaluations. Post-course surveys and evaluations were not available by the deadline for this paper.

IV. Study results

We measured students' perceived value, project time, project scores, total time spent on the course, and exam results. All results of the study were consistent with the hypothesis that the test section would yield improved student outcomes, with the exception of project and exam scores, which had no detectable difference between the sections. All results reported here include all students who elected to participate in the study with no control for student background. We analyzed results after grouping students by math background and found no detectable difference from results reported here, so, for simplicity of exposition, we omit student ability interactions, which we expect will be an important consideration in our follow-up study. We report only results up to the midterm exam; as explained later in this section, exercises were made available to the control group for optional use after the midterm.

Figure 1 shows box plots of project scores and median time spent on projects per user. Only the weekly projects, which were individual projects in both sections, were compared. Project scores appear comparable. The interesting statistic, however, is time spent on projects. In order to get similar scores, the median test student spent 43% less time than the median control student, a statistically detectable effect ($p = 0.028$). We note that while the control students are spending more time on the projects, that doesn't necessarily mean that they are spending more time on *problem solving* as the control students may be spending more time fixing syntax bugs.

Figure 2 shows student responses to post-project questions related to their experience doing the project. Both questions yield statistically detectable differences between the two sections (challenge $p = 0.0013$ and frustration $p = 0.0181$). This result, coupled with the reduced amount of time spent on projects, indicate that test students had a more pleasing experience doing the projects than those in the control section.

Figure 3a shows the midterm exam score results, where the t-test yields no statistically detectable difference. The midterm exam is structured like the exercises, with self-grading questions, except that exam questions require higher-order problem-solving skills and program design than exercises. An example question is to ask the student to find

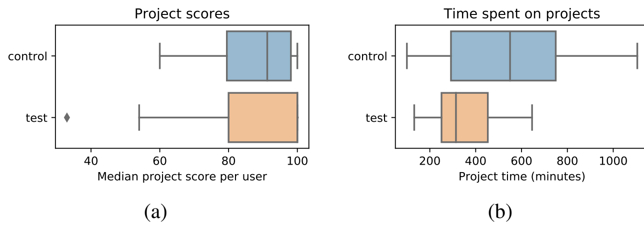

Fig. 1: (a) Median project score for each user in the two sections. (a) Amount of time spent on the project with the median score.

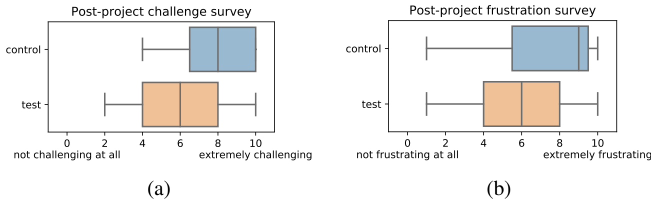

Fig. 2: Student responses to post-project questions about their experience. (a) Response to the question "how challenging was this project for you?" (b) Response to the question "how frustrating was this project for you?"

the average age among a list of student records. So it was interesting and unexpected to see no detectable difference in scores when not only do test students have the benefit of exercises, but also that their project development time is so much lower. We plan to investigate this result further in our follow-up study.

Figure 3b reports the total amount of time students spent on the course. For control students it included time spent completing projects and in-class lecture time. For test students this included time spent completing projects and exercises. The median test student spent 18% more time on the course than the median control student. For perspective, consider that the test student does daily exercises and three projects per week, compared to the control student's lecture time and single weekly project. If project time is considered a primary outcome, then 18% more time spent in the class in order to save 43% of the time on projects may be an attractive option. Further, control students are spending only 23 minutes outside of class per 50 minutes in class while test students spend only 37 minutes, which are both well below our target of two hours outside of class per hour in class.

While final course evaluations were not available by the deadline for this paper, we have various pieces of informal feedback from students. After the midterm exam multiple students in the control section asked for the option to do exercises after hearing about them from the students in the test section. This request was granted and exercises were provided as an optional study help to control students (all results in this paper were collected before this change was made). A student

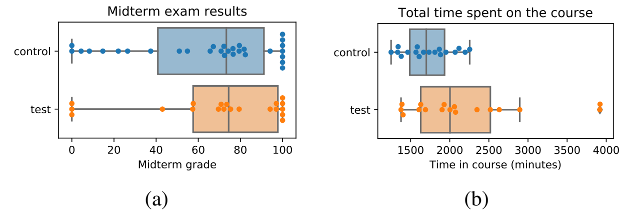

Fig. 3: (a) Midterm exam scores. (b) Total amount of time spent on the course.

in the control section later said in an email to the instructor, "I could have used the exercises from the beginning....I really want to learn and realize if I had these exercises sooner with more usable information that I could take the 'puzzle pieces' and put it together I might be feeling exhilarated rather than discouraged." This anecdotal evidence suggests that exercises may play an explanatory role in the attitudinal and learning outcome improvements measured in our pilot study.

V. Conclusions and discussion

While no causative conclusions can be drawn from our quantitative results because of potential confounding factors (e.g. instructor, time of day, student background, etc), we hypothesize that the improved outcomes measured in this study are positively impacted by the presence of exercises and pair programming. As such, studies we are now pursuing focus on exploring exercises vs. lecture and various combinations that include pair programming. We also expect that the greater statistical power in our upcoming studies will elucidate the effect of exercises on test scores.

Our proposed educational framework carries with it various challenges, especially in the context of higher education. A recognized issue of pair programming and other active learning approaches is that active learning is easiest in small classrooms, and with burgeoning enrollments, many institutions rely on large lecture halls to cut costs. An additional issue is that instructors are often wary of approaches that move the instructor into an enabler and mentor role.

If empirical results of our future, larger study are positive then our approach has multiple important implications. The first is a more effective undergraduate CS1 offering with improved student engagement, metacognition and mastery of the material. Reduced student frustration may also lead to increased retention. Because of the exercises and pair-programming, the in-class instructor takes a lesser role and requires less training. Thus, burgeoning CS1 enrollments can more easily be handled by employing teaching assistants for daily classroom instruction. Possibly more importantly, underfunded high schools in rural and urban school districts with insufficient budget for computer science faculty can be serviced with a computer science curriculum and software that can be deployed with little monetary investment compared to expensive professional development initiatives.

References

[1] T. Altintas, A. Gunes, and H. Sayan. A peer-assisted learning experience in computer programming language learning and developing computer programming skills. *Innovations in Education and Teaching International*, 53(3):329–337, 2016.

[2] J. Bennedsen and M. E. Caspersen. Failure rates in introductory programming. *AcM SIGcSE Bulletin*, 39(2):32–36, 2007.

[3] B. S. Bloom, C. of College, and U. Examiners. *Taxonomy of educational objectives*, volume 2. Longmans, Green New York, 1964.

[4] J. Bransford, S. Brophy, and S. Williams. When computer technologies meet the learning sciences: Issues and opportunities. *Journal of Applied Developmental Psychology*, 21(1):59–84, 2000.

[5] G. Braught, T. Wahls, and L. M. Eby. The case for pair programming in the computer science classroom. *ACM Transactions on Computing Education (TOCE)*, 11(1):2, 2011.

[6] S. Cooper, W. Dann, and R. Pausch. Alice: a 3-d tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, volume 15, pages 107–116. Consortium for Computing Sciences in Colleges, 2000.

[7] P. Denny, A. Luxton-Reilly, and E. Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, pages 75–80. ACM, 2012.

[8] J. Gálvez, E. Guzmán, and R. Conejo. A blended e-learning experience in a course of object oriented programming fundamentals. *Knowledge-Based Systems*, 22(4):279–286, 2009.

[9] A. Gomes and A. J. Mendes. Learning to program-difficulties and solutions. In *International Conference on Engineering Education–ICEE*, volume 2007, 2007.

[10] N. Hawi. Causal attributions of success and failure made by undergraduate students in an introductory-level computer programming course. *Computers & Education*, 54(4):1127–1136, 2010.

[11] N. Khamis, S. Idris, R. Ahmad, and N. Idris. Assessing object-oriented programming skills in the core education of computer science and information technology: Introducing new possible approach. *WSEAS Transactions on Computers*, 7(9):1427–1436, 2008.

[12] S. K. Kummerfeld and J. Kay. The neglected battle fields of syntax errors. australian computer society. In *Proceedings of the fifth Australasian conference on Computing education*, pages 105–111, 2002.

[13] M. C. Linn. The cognitive consequences of programming instruction in classrooms. *Educational Researcher*, 14(5):14–29, 1985.

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

[15] R. Lister. Programming, syntax and cognitive load (part 2). *ACM Inroads*, 2(3):16–17, 2011.

[16] R. E. Mayer. The psychology of how novices learn computer programming. *ACM Computing Surveys (CSUR)*, 13(1):121–141, 1981.

[17] R. E. Mayer, J. L. Dyck, and W. Vilberg. Learning to program and learning to think: what's the connection? *Communications of the ACM*, 29(7):605–610, 1986.

[18] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. 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*, pages 125–180. ACM, 2001.

[19] C. McDowell, L. Werner, H. E. Bullock, and J. Fernald. Pair programming improves student retention, confidence, and program quality. *Communications of the ACM*, 49(8):90–95, 2006.

[20] E. Mendes, L. B. Al-Fakhri, and A. Luxton-Reilly. Investigating pair-programming in a 2 nd-year software development and design computer science course. In *ACM SIGCSE Bulletin*, volume 37, pages 296–300. ACM, 2005.

[21] N. Nagappan, L. Williams, M. Ferzli, E. Wiebe, K. Yang, C. Miller, and S. Balik. Improving the cs1 experience with pair programming. *ACM SIGCSE Bulletin*, 35(1):359–362, 2003.

[22] D. Oliver and T. Dobele. First year courses in it: A bloom rating. *Journal of Information Technology Education: Research*, 6:347–360, 2007.

[23] N. Parlante. CodingBat. http://codingbat.com, 2018. [Online; accessed 2018–04–17].

[24] P. Pirolli and M. Recker. Learning strategies and transfer in the domain of programming. *Cognition and instruction*, 12(3):235–275, 1994.

[25] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[26] L. M. Serrano-Cámara, M. Paredes-Velasco, C.-M. Alcover, and J. Á. Velazquez-Iturbide. An evaluation of students motivation in computer-supported collaborative learning of programming concepts. *Computers in Human Behavior*, 31:499–508, 2014.

[27] B. A. Sheil. The psychological study of programming. *ACM Computing Surveys (CSUR)*, 13(1):101–120, 1981.

[28] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.

[29] M. Stains, J. Harshman, M. Barker, S. Chasteen, R. Cole, S. DeChenne-Peters, M. Eagan, J. Esson, J. Knight, F. Laski, et al. Anatomy of STEM teaching in North American universities. *Science*, 359(6383):1468–1470, 2018.

[30] A. Stefik and S. Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)*, 13(4):19, 2013.

[31] D. Thomas. CodeKata. http://codekata.com, 2018. [Online; accessed 2018–04–17].

[32] D. Traynor, S. Bergin, and J. P. Gibson. Automated assessment in cs1. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 223–228. Australian Computer Society, Inc., 2006.

[33] S. E. Volet. Modelling and coaching of relevant metacognitive strategies for enhancing university students' learning. *Learning and Instruction*, 1(4):319–336, 1991.

[34] C. Watson and F. W. Li. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 39–44. ACM, 2014.

[35] D. Weintrop. *Modality Matters: Understanding the Effects of Programming Language Representation in High School Computer Science Classrooms*. PhD thesis, Northwestern University, 2016.

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

[37] J. L. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, P. Kumar, and C. Prasad. An australasian study of reading and comprehension skills in novice programmers, using the bloom and solo taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 243–252. Australian Computer Society, Inc., 2006.

[38] D. Woit and D. Mason. Effectiveness of online assessment. *ACM SIGCSE Bulletin*, 35(1):137–141, 2003.

[39] L. Zhang, S. KaLyuga, C. Lee, and C. Lei. Effectiveness of collaborative learning of computer programming under different learning group formations according to students' prior knowledge: A cognitive load perspective. *Journal of Interactive Learning Research*, 27(2):171–192, 2016.