Read, Manipulate, and Write: A study of the role of these cumulative skills in learning computer programming

Laura Zavala
Department of Physics and Computer Sciences
Medgar Evers College of the City University of New York

It is widely agreed that in the Computer Science discipline a lot of practice is needed, in particular for (computer) programming courses. Introductory programming courses usually have a significant amount of time dedicated to practice, inside and outside the classroom. Some practitioners have noted that, although practice is good, asking novice students to write programs after only teaching them syntax rules and a few examples might not be the right approach. Based on this notion, this paper proposes three phases that students should sequentially master in the process of learning computer programming: code comprehension, code manipulation, and code writing. This paper presents a small-scale study conducted to corroborate this intuition. Students from two introductory programming courses were evaluated on code comprehension, code manipulation, and code writing skills in two different content areas. The results obtained are in line with our intuition that these are skills that students should sequentially master in the process of learning to write programs.

Corresponding Author: rzgutierrez@mec.cuny.edu

Introduction

High failure rates abide in introductory computer programming courses (CS1) all over the world with a pass rate estimated to be around 63%. This has motivated many researchers to search for causes and solutions to the problem. The literature abounds with research on pedagogies and innovative approaches for CS1 courses, specifically, active and collaborative learning approaches such as pair-programming, peer-lead instruction, flipped classrooms, and live coding [1][2][3][7][8][9].

Regardless of the approach used and the degree to which it is used in the classroom, the need for considerable practice in CS1 courses is indisputable and widely acknowledged. The degree and approach used might not make a big difference as long as practice is guaranteed. It is the type of practice given to students that needs further consideration. Practice in programming courses is most of the time equated with programming and more specifically with writing code. This might be adequate for advanced, upper-level courses. However, for CS1 courses, the fact that code comprehension skills are a precursor to code writing skills should not be ignored; practice of the former should be ensured before asking students to do the latter.

Some practitioners have discussed the need for a *reading before writing* approach in CS1 courses [4][10][11][12][13]. The underlying idea is inspired in the observation that children learn to write after they have spent several years of reading (or rather, being read to) and speaking the language. By the time they are asked to write, they have been exposed to the syntax and semantics of the language, as well as different models of writing. The same underlying idea has been explored in other problem solving oriented disciplines. For example, Sweller and Cooper [5] compared the efficacy of a worked-examples approach versus a problem-solving approach. In their experiments, some students learnt certain algebraic processes by solving problems, while other students learnt the same algebraic processes by studying complete solutions to those problems (i.e. worked examples). They found that students who learned algebra with worked examples performed significantly better than those who were asked to solve problems instead.

The analogous *reading before writing* approach in CS1 courses is more of a *read before you manipulate and manipulate before you write* approach. Students should be able to not only read code before they can write code, but also they should be able to manipulate code that is given to them (i.e. modify or use). Formally, there are three phases that students should sequentially master in the process of learning to write programs: code comprehension, code manipulation, and code writing. This paper presents the results of a study conducted at a small urban college with the goal of corroborating this intuition.

Although it can be argued that code comprehension and manipulation work is covered in CS1 courses through sample solutions, design roadmaps, implementation hints, and starter code, the emphasis is, by far, on code writing. Further, providing students with all these resources does not really constitute practice on

code comprehension and manipulation; it is not the main focus. Finally, students are rarely or never assessed on these skills to ensure that they are ready to start writing code.

Background

Linn and Clancy [12] emphasize the importance of learning patterns of program design. They propose the use of programming case studies to teach students program design skills. They argue that novice students should be exposed to many case studies before they can design and write their own programs. The authors used this approach for several years at Berkeley and they reported significant improvement in students' design skills. In a similar effort, Kirschner, Sweller, and Clark [6] recommend providing students with worked examples and process worksheets, which should provide a description of the steps they should go through when solving the problem as well as hints or rules of thumb that may help to successfully complete each step.

In [4], an ITiCSE conference working group presents an assessment developed to evaluate the programming competency of students that have completed their first one or two courses in computer science. As a result of applying the assessment to a large population of students from different universities, they established that many students do not know how to program at the conclusion of their introductory courses. In the search for a cause to this problem, Lister et al. [13] tested students from seven countries at the conclusion of their CS1 courses on two abilities: (i) their ability to predict the outcome of a program, and (ii) their ability to complete a near-complete program. The authors note that poor performance in one or both abilities might be an explanation for the fact that many students do not know how to program at the conclusion of their introductory courses. This work is closely related to that presented in this paper. An important difference is that they only test students on code comprehension skills and speculate on a relation of those skills to the ability of students to write code. The work presented in this paper examined that assumption given that students were tested on both, code comprehension skills as well as code writing skills of similar complexity.

Problem formulation

The high-level goal for the study presented in this paper was to collect initial data to shed light into the belief that (1) code comprehension, (2) code manipulation, and (3) code writing are phases that students should sequentially master in the process of learning to write programs. The specific goal was to answer the following questions:

a) Do (most or all) students that show proficiency in a given skill also possess proficiency in the skills that precede it in the sequence?
b) Do (most or all) students who show deficiency in, or lack of, a given skill, also lack the skills that follow it in the sequence?

Methodology

Students from two introductory programming courses were evaluated on their code comprehension, code manipulation, and code writing skills in two different content areas.

Code comprehension is the ability of students to understand a code fragment or a program. Some ways to assess this skill include asking students to (i) select the piece of code, from a set of choices, that performs a specific task; (ii) verbally describe what a code fragment or program does, (iii) indicate what is the value returned by calling a given function with specific argument values; (iv) indicate what a program would display on screen when it runs; (v) indicate what the value of one or more variables is after executing a code fragment or program; and (vi) identify the part of a program where a specific action is carried.

Code manipulation involves using or modifying existing code. The skill can be assessed with exercises where students have to be able to manipulate existing code, such as: (i) completing a piece of code (either by writing the missing code, or choosing it out of a set of choices), (ii) writing function calls to provided functions so that a specific result is obtained; (iii) building a program from a set of fragments of code, not all of which might be part of the solution; (iv) reordering a scrambled program.

Code writing is the ability of students to write code for a given task (even if provided with a similar example and its solution). Assessment of this skill is widely carried and the straightforward and widely used method is to redact the specification of a task or problem, for which the student has to write a code fragment or program that carries out such task or constitutes the solution to the problem.

The following two content areas were used for the evaluation: arrays using *for* loops and C++ STL container classes. The first area was evaluated in a first Object Oriented Programming (OOP) course (*group 1*). The second area was evaluated in the next course in the sequence (*group2*), where students learn about data structures and algorithms. Both courses are taught using the C++ programming language. The assessment instrument was a quiz containing four code comprehension questions, two code manipulation exercises, and two code writing exercises, all of similar complexity. The quiz was administered by phases: first,

the code writing exercises, then the code manipulation exercises, and at the end, the code reading questions. Students worked on each phase one at a time and had no access to the questions on the other sections.

Content area: arrays using *for* loops. The questions consisted of a short program containing one or two arrays that are manipulated through one or two *for* loops. For the code comprehension questions, students were asked to indicate what the program would display on screen when it runs. The code manipulation exercise consisted of a scrambled program that students had to order. In the code writing exercise, students were asked to write the *for* loop. Figure 1 shows some of the questions in this content area.

Content area: C++ STL container classes. The questions consisted of short programs where data are inserted into one or two C++ STL container objects. Then, after some operations are performed on the container objects, their content is displayed to screen. For the code comprehension questions, students were asked to indicate what the program would display on screen when it runs. For the code manipulation exercises, students were asked to sort a scrambled program. The code writing exercises required students to write the program from scratch. Figure 2 shows some of the questions in this content area.

What will be the output of the following program? In other words, what is printed to screen when you run it?

```
#include <iostream>
using namespace std;
int main ()
{
  int foo []={"Matt","Jen","Andy","Zoe","Ted",
                        "Zane","Mike","Rob","Alice"};
  for (int index=0; index<9; ++index)
  {
    if (foo[index]> "Matt")
            cout << foo[index] << endl;
  }
  return 0;
}
```

(a)

Assume there is an array of *string* values called *names* where the names of 100 students have been stored.

Assume further that there is an array of *float* values named *grades* where the corresponding grades of those students have been stored.

Write a *for* loop to print the names of the students who have a grade greater than 50.

(b)

Figure 1. Questions given to group 1: (a) code comprehension; (b) code writing

What will be the output of the following program? In other words, what is printed to screen when you run it?

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main ( ){
    map<string, string> actors;
    actors.insert(make_pair("Mark Ruffalo", "The Avengers"));
    actors.insert(make_pair("Mark Ruffalo", "The Avengers: Age of Ultron"));
    actors.insert(make_pair("Hugh Jackman", "X-Men"));
    actors.insert(make_pair("Hugh Jackman", "The Wolverine"));
    actors.insert(make_pair("Scarlett Johansson", "The Avengers"));
    actors.insert(make_pair("Scarlett Johansson", "The Avengers: Age of Ultron"));
    cout << "ACTORS AND THEIR MOVIES:" << endl;
    for (map<string, string>::iterator it= actors.begin(); it!= actors.end(); it++)
         cout << (*it).first << "- " << (*it).second << endl;
}
```

Figure 2. Code comprehension question given to group 2

Results

Code comprehension questions were worth 10 points each, for a total of 40 in that section. Code manipulation questions were worth 15 points each, for a total of 30 in that section. Code writing questions were worth 15 points each, for a total of 30 in that section. Tables 1 and 2 show the results for each student in terms of percentage grade in each section. Students have been anonymized. Figures 3 to 6 show a summary of the results. The stacked line charts in Figures 3 and 4, in which the cumulative scores by level are plotted, show that code reading is the most dominant skill, followed by code manipulation, while code writing is the weakest one. The same pattern is shown in Figures 5 and 6, where the number of students with a score greater than 60 for each skill has been plotted. The pattern is less steep for group 2 because some students scored higher at code manipulation than at code reading. This might be explained by the fact that their mistakes in the reading questions had more to do with the properties of the data structures that were particularly used. For example, the C++ *map* container (Figure 2) does not allow repetition of a *key* value and the elements are always sorted by it. Incorrect student answers would either show repeated values or show the elements unsorted.

Conclusions

Based on the expressed need of a *reading before writing* approach for CS1 courses, an analogous approach has been presented that is more of a *reading before manipulating and manipulating before writing* approach. Students should be able to not only read code before they can write code, but also they should be able to manipulate code that is given to them. To shed light on the validity of this claim, a small-scale study was conducted where students' code comprehension, code manipulation, and code writing skills in two different content areas were evaluated. The results obtained are in line with the original premise that code comprehension, code manipulation, and code writing are phases that students should sequentially master in the process of learning computer programming. However, since the author teaches at a small school, extensive quantitative assessment is not possible. Therefore, this work is continuing in collaboration with colleagues at larger institutions to conduct similar experiments at their colleges. Further, the experiment will be conducted with a larger pool of computer science students at the original institution, not only with those in introductory programming courses.

Student	Scores (%)			Average
	Level 1 (read)	Level 2 (re-order)	Level 3 (write)	
S1	82.50	80	66.66	76.38
S2	100	86.66	86.66	91.10
S3	82.50	66.66	50	66.38
S4	75	50	0	41.66
S5	50	16.66	0	22.22
S6	50	33.33	0	27.77
S7	100	86.66	66.66	84.44
S8	62.50	16.66	16.66	31.94
S9	100	50	33.33	61.11
S10	37.50	0	0	12.50
S11	82.50	66.66	66.66	71.94
Average	**74.77**	**50.30**	**35.15**	

Table 1. Results by learning-to-code skill (level) for each student in group 1

Student	Scores (%)			Average
	Level 1 (read)	Level 2 (re-order)	Level 3 (write)	
S1	87.5	100	73.33	86.94
S2	87.5	50	50	62.50
S3	75	73.33	50	66.11
S4	80	83.33	66.66	76.66
S5	100	100	100	100
S6	62.5	66.66	33.33	54.16
S7	62.5	0	0	20.83
S8	37.5	33.33	33.33	34.72
S9	62.5	66.66	50	59.72
S10	87.5	100	83.33	90.27
S11	87.5	100	83.33	90.27
S12	100	100	100	100
Average	**75.50**	**72.78**	**60.28**	

Table 2. Results by learning-to-code skill (level) for each student in group 2

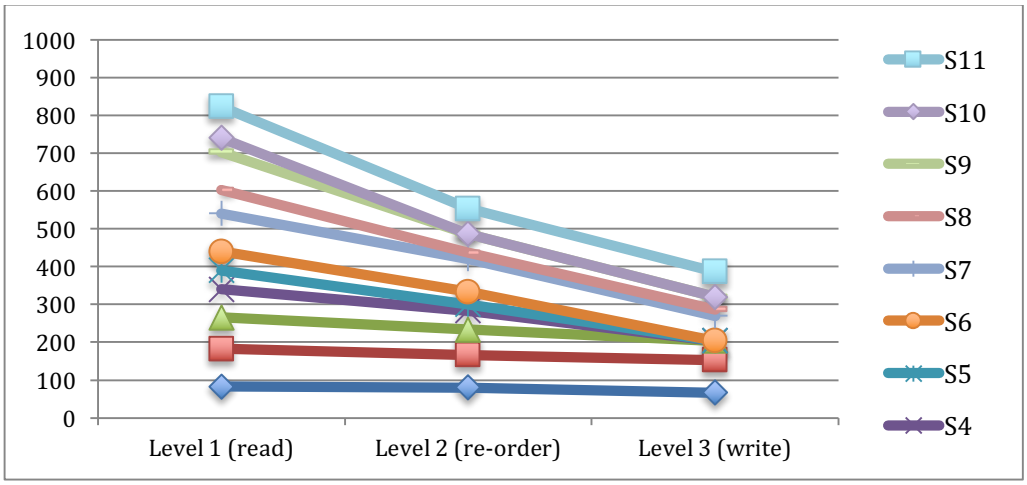

Figure 3. Group 1 cumulative scores by level

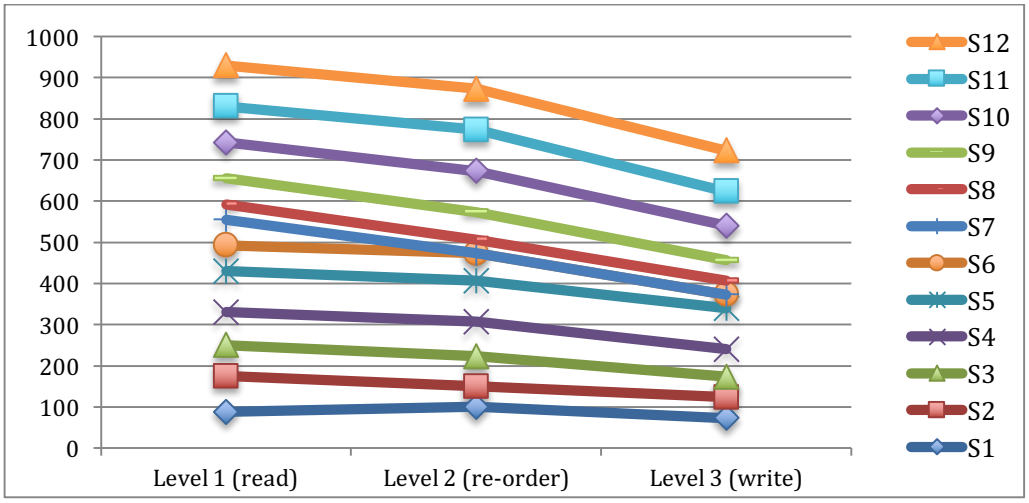

Figure 4. Group 2 cumulative scores by level

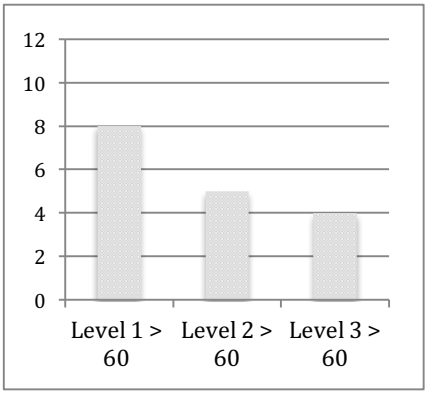

Figure 5. Number of students from group 1 with a score greater than 60

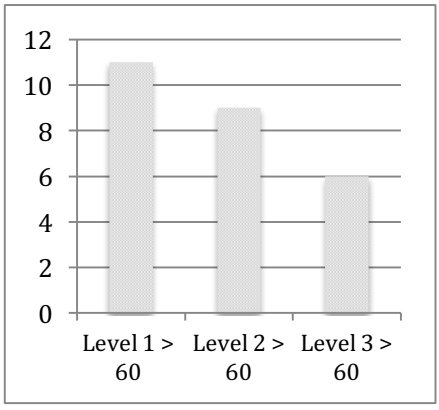

Figure 6. Number of students from group 2 with a score greater than 60

References

1. Leo Porter and Beth Simon, "Retaining nearly one-third more majors with a trio of instructional best practices in CS1," *in Proceedings. ACM Computer Science Education (SIGCSE '13). 44th Annual Technical Symposium* (2013): 165-170, ACM.
2. A. Robins, J. Rountree, and N. Rountree. "Learning and teaching programming: A literature review," *Computer Science Education* 13, no. 2 (2003): 137–172.
3. A. Sanwar, "Effective teaching pedagogies for undergraduate computer science," *Mathematics and Computer Education* 39, no. 3 (2005): 243–257.
4. McCracken, M., V. Almstrum, D. Diaz, M. Guzdial, D. Hagen, Y. Kolikant, C. Laxer, L. Thomas, I. Utting, T. Wilusz, "A Multi-National, "Multi-Institutional Study of Assessment of Programming Skills of First-year CS Students," *SIGCSE Bulletin* 33, no. 4 (2001): 125- 140.
5. Sweller, J., & Cooper, G. A., "The use of worked examples as a substitute for problem solving in learning algebra," *Cognition and Instruction* 2, no. 1 (1985): 59–89.
6. Kirschner, P.A., Sweller, J., and Clark, R.E., "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, no. 2 (2006): 75-86.
7. Judith D. Wilson, Nathan Hoskin, and John T. Nosek, "The benefits of collaboration for student programmers," in Proceedings. *ACM Computer Science Education (SIGCSE '93). 24th Annual Technical Symposium* (1993): 165-170, ACM.
8. Henry M. Walker. 2011, "A lab-based approach for introductory computing that emphasizes collaboration," *in Proceedings. Computer Science Education Research Conference* (2011): 21-31, Open Universiteit, Heerlen.
9. Marc J. Rubin. 2013, "The effectiveness of live-coding to teach introductory programming," *in Proceedings. ACM Computer Science Education (SIGCSE '13). 44th Annual Technical Symposium* (2013): 651-656, ACM.
10. Mark Guzdial and Judy Robertson, "Too much programming too soon?," *Communications of the ACM* 53, no. 3 (March 2010): 10-11, ACM.
11. Mark Guzdial. 2015, "What's the best way to teach computer science to beginners?," *Communications of the ACM* 58, no. 2 (January 2015): 12-13, ACM.
12. Marcia C. Linn and Michael J. Clancy. 1992, "The case for case studies of programming problems," *Communications of the ACM* 35, no. 3 (March 1992): 121-132, ACM.
13. Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas, "A multi-national study of reading and tracing skills in novice programmers," *Working group reports from ITiCSE on Innovation and Technology in Computer Science Education* (2004): 119-150, ACM.