The Failure of Education: Computer Science Left Behind

Edward Li

Abstract

This paper reviews the struggle for first-year college computer science programs to effectively teach students how to solve problems through programming. This paper emphasizes taking a closer look at the material that is being taught in these courses, as well as the individual work done within these courses, and suggests that limiting complex syntax and introducing pair-based work allows for students to gain higher-level problem solving skills required to tackle coding together solutions to problems, while also giving them access to a partner which will help with problem-solving blockers.

I. Introduction

As long as computer science has existed as a field, computer science education has been a touchy subject. For some reason, first-year university programs are unable to successfully educate students on understanding the fundamentals of breaking down problems into code. Though teachers agree that there is a fundamental problem, they can't seem to pinpoint the specific causes of the difficulties, or solutions to them. The goal of the paper is to assimilate common student roadblocks and predictors of success and determine if there is some universal material and methods that educators should adopt for education. This paper specializes in first-year education, since the foundations laid down by fundamentals classes translate to creating not only good habits and better scholars, but better programmers as well. I argue that the best way to move forward with educating first years is by providing them a pair-programming, student-oriented environment where students can focus more on systematic design and composition than worrying about complex syntax on their own.

This paper takes on four main sections. The first two are mainly focused on finding struggles and factors that lead into programmer struggles, through studies done on students. The third section takes a look at some approaches already published when tackling the subject of first year education, and different attempts to take a stab at the main problem. The fourth section is a quick diversion into the world of pair programming, and whether it could be effective for students.

II. Studies on Programmer Struggles

Common anecdote indicates that CS is a uniquely difficult field with a higher dropout rate than most other majors. Reilly and Bergin go so far to state "It is well known in the Computer Science Education (CSE) community that students have difficulty with programming courses and this can result in high drop-out and failure rates."[1] The figure below enumerates some of the more notable attempts to categorize student issues through large-scale assessments.

Source	Objective(s)	Methodology	Results
A multi-national, multi-institutional	To initiate a dialog in the Computer Science	Students participated in charettes (labs) in	Students did more poorly than expected,

study of assessment of programming skills of first-year CS students (McCracken, 2001)[2]	community on how to develop assessments determining whether students can program.	which they would solve some or all of a set of related problems developed by the programming group. Four universities administered various subsets of the questions, with only one administering all three.	scoring on average 22.89/110 points (as graded by the instructor administering the exam). Most students struggled with even discerning the problem from the description. Conclusion was students did not understand how to problem solve, and so could not even begin to solve problems.
A Multi-National Study of Reading and Tracing Skills in Novice Programmers (Lister, 2004)[3]	To find different sources of difficulty than the McCracken group. Namely, investigate the inability of students to trace through code and understand the code written. Maintains claims by McCracken group about an easy 5-step problem-solving method aren't trivial and require an ability to understand code.	12 multiple choice questions (MCQs) asking students to either trace code and determine what it would do manually or select the correct block of code for a nearly-complete snippet.	Students scored 60% on average. Students struggle with reading code, especially in non-idiomatic, "non-standard" snippets of code. The assessment only studied reading comprehension, with the group admitting inability to determine if test takers could write novel pieces of code
Failure Rates in Introductory Programming (Bennedsen, Caspersen, 2007)[5]	To determine if Computer Science courses are legitimately more difficult than courses in other majors.	A questionnaire was given out to computer science educators asking what percent of students passed, and what percent of students failed based on failing or dropping the class.	About 33% of students failed/dropped out of the course. The paper argues that this number isn't necessarily high enough to warrant concern when compared to other fields.

Figure 1: Attempts to Identify Failures of students

The only consensus between the studies is that while CS is inherently difficult to teach and to study, none of the articles seem to be able to agree on what the root cause was. McCracken's[1] core research focuses on whether students are able to discern a problem from a description of said problem, and systematically create discrete programming tasks from them. Lister's group[2] then instead argues that McCracken's group, while on a reasonable path, has ignored the fact that more students simply struggle with an inability to read simple snippets of code and understand them. While that holds merit, the paper itself admits that the assessment created doesn't effectively address the inability for students to write their own novel pieces of code. Meanwhile, it seems that most students and teachers are also more concerned about larger ideas of programming than individual bits[6], further supporting McCracken's thesis that the problem isn't creating short snippets of code themselves, but rather, composing them top-down into an effective, readable solution. Again, McCracken's claims are backed up by one of the biggest correlations between pre-course knowledge and success being math, something covered more in the next section. With this in mind, I find McCracken group's core focus on problem solving is much more important than Lister's focus on syntax. As seen in section IV of the paper, many attempts to educate first-year students strive to take the burden of learning complex syntax away from students, allowing them to focus on core design principles.

III. Attempted Predictions of Success for 1st year Students

In much the same way research has been attempting to drill down on difficulties for students, so too has there been attempts to predict success for freshmen in computer science.

Source	**Objectives**	**Methodology**	**Results**
Predicting The Success of Freshmen in a Computer Science Major (Campbell, McCabe, 1984)[8]	To determine which applicants to computer science programs are more likely to succeed. This theoretically helps admission officers effectively advise students as to whether computer science is an achievable goal.	256-first semester freshmen in a large midwestern university in fall 1979 were studied, where their background scores and performance in two semesters of classes were mapped.	SAT math and verbal scores, high-school ranking, high school mathematics, and science all contributed positively to performance.
Predicting Student Success in an Introductory Programming Course (Hostetler, 1983)[9]	To measure a student's aptitude in computer programming through cognitive skills, personality traits, and past academic achievements.	Reasoning and Diagramming tests from the Computer Programmer Aptitude Battery were administered to students, and compared to the students' GPAs.	Students' GPA was highly correlated with scores on the tests. Math was highly correlated with success. A model was created that could predict success/failure at a 77% rate.
Predictors of Success and Failure in a CS1 Course (Rountree, Rountree, Robins, 2002)[10]	To determine what to teach: which paradigm, what language, what features, and what components are essential.	An optional online survey on status, background, and expectations of the class was given out. Scores in classes were	Students should be told at the beginning of course they're good at self-assessing, and ensure students understand the class is

		matched to replies on the survey.	not a good filler class. The anecdote "successful writers must want to write. Too many people have written," applies.
Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors (Wilson, Shrock, 2001)[4]	To determine factors that promote success in an introductory college computer course.	Students were given a questionnaire about their personal backgrounds, as well as measured using the "Computer Programming Self-Efficacy Scale"	Of factors examined, higher comfort level and previous math experience correlated with doing well, while attribution of personal success to luck and game playing were negative factors. Previous programming classes had almost no effect.
A Study of the Difficulties of Novice Programmers (Lahtinen, Ala-Mutka, Matti Jarvinen, 2005)[6]	To assimilate student and teacher opinions on difficulties experienced and perceived when learning and teaching programming.	A web-based questionnaire on personal background, course contents, and learning aspects was administered to both students and teachers.	Most difficult programming concepts were more abstract and required students to be able to understand larger entities of programming, instead of individual parts. Teachers generally perceived things to be more difficult than students. Students enjoyed practical activities more than lecture/theory.
Programming: Factors that Influence Success (Bergin, Reilly, 2004)[1]	To determine factors for poor student retention on post high school CS education, as well as ways to determine how well students are doing prior to a first assessment being handed out in class.	A questionnaire about grades in tertiary courses and comfort level was administered to students, in tandem with a cognitive test testing sequencing ability, arithmetic reasoning, problem translation skills, and logical ability.	Comfort level with CS, math, and science scores have strong correlation with performance. Gender also played a part. The strongest correlation with success seemed to be individual student confidence in their abilities.

Figure 2: Factors and predictors of success in CSE

Throughout all the studies, one commonality seems clear, that strength in math has a strong correlation with success in computer science [4][8][9]7]. And this makes sense, understanding important parts of specifications and computer science problems require the same skills as word problems[11], such as identifying key information in a problem, and formulating a plan to turn the *information* into *data*[13].

Of note also are the confidence levels of students playing a major role in how they do in a course. Imposter syndrome is rampant in computer science[12], and self-confidence could have multiple factors into it, including gender. Notably, some papers like Bergin and Reilly's *Programming: Factors that Influence Success*[1] suggest that gender has a large role in predicting the success of a first year student, while Wilson and Shrock's *Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors*[4] directly contradict this finding, saying gender has absolutely no effect on the student. This could be due to different cultures in the institutions researched, anecdotal evidence of my own within Northeastern has pointed to rampant imposter syndrome leading to less self-confidence in fields, possibly leading to students being unable to favourably rate themselves, and thus possibly lead to lower scores because of the extra burden.

It could be that social skills also play a factor in both self-confidence and performance in classes[4]. Students who are tech-savvy but aren't as social may have less people they can discuss problems with and gain new ideas from or may be more distracted when sat down in front of technology. Mitigating this social gap is the goal of Peer Instruction and Pair Programming, something elaborated on in part V of this paper.

IV. Paradigms of programming and material to teach

Obviously, each first-year computer science has to choose where its "entry point" into the field is. This involves a careful cocktail languages, constructs, algorithms, methodologies, and paradigms. The following table details some of the more common or notable approaches to this problem, both new and old.

Source	**Objectives**	**Methodology**	**Results**
Teaching Objects-first In Introductory Computer Science (Cooper, Dann, Pausch, 2003)[14]	To determine if there's a simpler way of teaching an objects-first approach, by removing the syntactical challenges that come with using an industry programming language	A new programming environment "Alice" was introduced to students. In Alice, 3D animations can be created by manipulating 3d models, or "objects," using a graphical programming environment, where students can drag and drop expressions.	Students seemed to be able to understand top-down design more. Students were also able to contextualize object-oriented terms and concepts better, since they didn't have to struggle with "has a, is a" in other, more abstract terms.
What Should We Teach in an Introductory	As a preface, the paper is quite old and outdated, but brings	N/A	Language agnostic concepts should be taught in first-year CSE

Programming Course? (Gries, 1974)[15]	points that are discussed time and time again. The paper argues that what should be focused on is a method of problem solving first, and talking about algorithms.		classes; things like how to translate short imperative English statements into imperative code snippets, and algorithms. The only requirements for a language is that it should be modern enough to feature control-flow structures that allow for easy implementation of algorithms taught.
The TeachScheme! Project: Computing and Programming for Every Student (Felleisen, Findler, Flatt, Krishnamurthi, 2003)[16][17]	To discuss the TeachScheme! Project, and how its methodology has been tailor-suited to problems students face — namely that students make mistakes, and that students don't know how to start tackling a problem by programming.	To help students abstract away difficulties with the scope of programming languages, and the mistakes they'll make, small student programming languages that build incrementally on each other are introduced to students using a custom programming development environment. Students are also taught a "Design Recipe," or a method of breaking down a complex problem into data, and then building a program based on data.	Students are able to cope with the syntax of the student languages used, and teachers believe that the program gives their students stronger fundamentals. Female students preferred the course (with pair programming involved) over a traditional computer science course.
To Scratch or not to Scratch? (Hermans, Aivaloglou, 2017)[18]	To observe if beginners in computer science should start with *plugged* instruction, or *unplugged* instruction.	35 elementary kids were split into two groups. Half were given instruction without Scratch for four weeks, then in Scratch[19] for four weeks, while the other half used Scratch the whole time. At the	Students who were taught unplugged were more confident in their ability to understand concepts, and more self-sufficient. Other than that, their scores were largely the same.

		end of the class, the students were assessed on games they created and concepts.	

All these papers seem to agree, the *principles* of computing matter more than the language. That is, each approach agrees that by simplifying programming constructs – whether it be by creating a custom programming environment, or custom languages for students that provide better feedback – allows for students to focus more on the meat of solving problems with the language of computer code. Of note is how similar Alice[14] is to the Scratch Programming Language[19], in that both are graphical languages where blocks of code can only be made through the tailor made programming environment. Traditional first year programs, such as Harvard's CS50[20], tend to take other approaches, where they survey many programming languages and take a peak into what each offer, showing different practical skills. While this may be useful, Felleisen's[21] group argues that learning to mimic instructors is not an effective way to teach students how to design, create, and systematically problem solve effectively.

Of course, there can be concern of transitioning from a "toy" language like Scratch or the Student Languages offered by some of the solutions above into a real language. Or even if there's a benefit to using a visual-based block language instead of a text-based language at all[22]. I argue that for college students, using a text-based language is important, since students are more likely to be cynical of a course if it's taught with images and drag-n-drops, since they'd associate that with children. What the papers here demonstrate is that no matter what, students need to have an environment that makes it easier for them to make mistakes, and track down what those mistakes were. K-12 education may benefit more from a Scratch-based approach, since the programs made are more engaging, but colleges seem better off adopting Felleisen's approach with the TeachScheme! Project, and similar invariants.

V. To Pair or Not to Pair

Recently, the idea of peer instruction[23] and pair programming[24] have taken off in both academic and industrial settings. Peer instruction involves students becoming active participants by teaching each other. Pair programming involves students working as a pair, on one machine at a time, bouncing ideas off each other to produce code. Both involve students becoming active participants, however, one is in lecture. Peer instruction doesn't, however, have any sort of "hard" definition. In a sense, pair programming could be a form of peer instruction, since students are interacting with each other. Since students will run into some companies that peer program in the future, my intuition is to focus more on pair programming.

Pair programming already has a pretty good track record in industry[24]. Though there are those who ridicule it, saying it's "redundant," quantitative studies show that the practice assists engineers in producing higher quality code in shorter periods of time. In a way, it holds both partners accountable, where one could normally get away with "deviating from standard practice," pair programming adds another set of eyes to catch such shortcuts. In academic settings, it's shown to produce a more positive attitude towards collaboration and help retain more students.[25] Because of pair programming, students who are stuck now have someone they can turn to – their partner – before turning to a tutor or Teaching Assistant, greatly reducing the burden the course staff has, and allows course staff to be able to cater to more people. Though final exam scores didn't seem to be affected by pair programming[26], it is still impressive to see that pair programming has an effect on the main body of the class, and no negative effect on any singular person's performance on solo exams.

In that, pair programming's effectiveness is clear. All papers that have looked into it support Pair programming as an effective way to not only prevent mistakes, but also for the students to be able to have at least one other person they know in their field.

VI. Conclusion

After a thorough survey of the state of first-year Computer Science Education, it seems we should approach the field in a way that builds off of a student's background in math. As seen in Section II, students seemed to struggle the most with teasing out relevant information needed for their problem at hand, and finding a way to systematically compose programming blocks into a solution. Section III then enforced that strong math skills would help with this — where there's a direct translation of word problem skills to computer science problem solving skills. IV then showed that a large amount of new-age computer science programs aim to reduce the burden of syntax from students, and again focus on problem-solving skills. Finally, a side-quest was taken to look at the effectiveness of pair programming, and how applying it to education could produce fantastic results. Therefore, without a doubt, a math-problem based curriculum with a student-oriented language/programming environment that involves students pair-programming with each other is the best way to move forward with educating the future of computing.

Acknowledgements

I could not have done any of this without the helping hand of Professor Benjamin Lerner in starting this paper.

References

[1] Susan Bergin, Ronan Reilly, The Influence of Motivation and Comfort-Level on Learning to Program. Proceedings of the 17th Workshop on Psychology of Programming. PPIG'05. 2005.

[2] Michael McCracken , Vicki Almstrum , Danny Diaz , Mark Guzdial , Dianne Hagan , Yifat Ben-David Kolikant , Cary Laxer , Lynda Thomas , Ian Utting , Tadeusz Wilusz, A multi-national, multi-institutional study of assessment of programming skills of first-year CS students, ACM SIGCSE Bulletin, v.33 n.4, December 2001

[3] Raymond Lister , Elizabeth S. Adams , Sue Fitzgerald , William Fone , John Hamer , Morten Lindholm , Robert McCartney , Jan Erik Moström , Kate Sanders , Otto Seppälä , Beth Simon , Lynda Thomas, A multi-national study of reading and tracing skills in novice programmers, ACM SIGCSE Bulletin, v.36 n.4, December 2004

[4] Brenda Cantwell Wilson , Sharon Shrock, Contributing to success in an introductory computer science course: a study of twelve factors, Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education, p.184-188, February 2001, Charlotte, North Carolina, USA

[5] Jens Bennedsen , Michael E. Caspersen, Failure rates in introductory programming, ACM SIGCSE Bulletin, v.39 n.2, June 2007

[6] Essi Lahtinen , Kirsti Ala-Mutka , Hannu-Matti Järvinen, A study of the difficulties of novice programmers, Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education, June 27-29, 2005, Caparica, Portugal

[8] Patricia F. Campbell , George P. McCabe, Predicting the success of freshmen in a computer science major, Communications of the ACM, v.27 n.11, p.1108-1113, Nov. 1984

[9] Terry R. Hostetler, Predicting student success in an introductory programming course, ACM SIGCSE Bulletin, v.15 n.3, p.40-43, Sept. 1983

[10] Nathan Rountree , Janet Rountree , Anthony Robins, Predictors of success and failure in a CS1 course, ACM SIGCSE Bulletin, v.34 n.4, December 2002

[11] Leon E. Winslow. 1996. Programming pedagogy—a psychological overview. SIGCSE Bull. 28, 3 (September 1996), 17-22. DOI=http://dx.doi.org.ezproxy.neu.edu/10.1145/234867.234872

[12] Elizabeth F. Churchill. 2018. Is there a fix for impostor syndrome?. Interactions 25, 3 April 2018

[13] Emmanuel Schanzer, Kathi Fisler, Shriram Krishnamurthi, and Matthias Felleisen. 2015. Transferring Skills at Solving Word Problems from Computing to Algebra Through Bootstrap. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)

[14] Stephen Cooper, Wanda Dann, and Randy Pausch, Randy Pausch. 2003. Teaching objects-first in introductory computer science. SIGCSE Bull. 35, 1 January 2003

[15] David Gries. 1974. What should we teach in an introductory programming course?. SIGCSE Bull. 6, 1 January 1974

[16] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The TeachScheme! Project: Computing and Programming for Every Student. cse2003-fffk, 2003

[17] Matthias Felleisen. TeachScheme!---A Checkpoint. ICFP 2010, 2010

[18] Felienne Hermans and Efthimia Aivaloglou. 2017. To Scratch or not to Scratch?: A controlled experiment comparing plugged first and unplugged first programming lessons. In Proceedings of the 12th Workshop on Primary and Secondary Computing Education (WiPSCE '17), Erik Barendsen and Peter Hubwieser (Eds.).

[19] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. Trans. Comput. Educ. 10, 4, Article 16 November 2010

[20] https://cs50.harvard.edu/college/

[21] Matthias Felleisen. Developing Developers.
https://felleisen.org/matthias/Thoughts/Developing_Developers.html

[22] David Weintrop, Uri Wilensky. 2019. Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. PERGAMON-ELSEVIER SCIENCE LTD, THE BOULEVARD, LANGFORD LANE, KIDLINGTON, OXFORD OX5 1GB, ENGLAND

[23] Leo Porter, Dennis Bouvier, Quintin Cutts, Scott Grissom, Cynthia Lee, Robert McCartney, Daniel Zingaro, and Beth Simon. 2016. A Multi-institutional Study of Peer Instruction in Introductory Computing. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16).

[24] Laurie Williams Robert R. Kessler Ward Cunningham Ron Jeffries. Strengthening the Case for Pair-Programming. IEEE Software, vol. 17, pp. 19-25, 2000.

[25] Nachiappan Nagappan, Laurie Williams, Laurie Williams, Miriam Ferzli, Eric Wiebe, Kai Yang, and Carol Miller, Suzanne Balik. 2003. Improving the CS1 experience with pair programming. In Proceedings of the 34th SIGCSE technical symposium on Computer science education (SIGCSE '03).