Learning to Program is Easy

Conference Paper · July 2016

DOI: 10.1145/2899415.2899432

1 author:

Andrew Luxton-Reilly
University of Auckland
131 PUBLICATIONS **3,285** CITATIONS

Learning to Program is Easy

Andrew Luxton-Reilly
Department of Computer Science
University of Auckland
Auckland, New Zealand
andrew@cs.auckland.ac.nz

ABSTRACT

The orthodox view that "programming is difficult to learn" leads to uncritical teaching practices and poor student outcomes. It may also impact negatively on diversity and equity within the Computer Science discipline. But learning to program is easy — so easy that children can do it. We make our introductory courses difficult by establishing unrealistic expectations for novice programming students. By revisiting the expected norms for introductory programming we may be able to substantially improve outcomes for novice programmers, address negative impressions of disciplinary practices and create a more equitable environment.

Keywords

novice; programming; computer science education; standards; expectations; learning outcomes; curriculum; CS1

1. INTRODUCTION

Learning to program is easy. Almost everyone can learn how to write their first computer program with minimal effort. Although the syntax of each programming language differs, producing a simple program that can print out a string of letters (such as that presented in Figure 1) is trivial to write, understand, reproduce and modify.

```
print('Hello World')
```

Figure 1: A simple Python program

Children have been successfully programming computers for more than 40 years [23], using a variety of programming languages and environments [16]. Recently, programming has been introduced into national curriculum for primary school children in both the UK [11], and Australia [13]. Other countries, such as New Zealand [4] and Denmark [12], have recently introduced programming at secondary school level. It is clear that children are capable of learning programming, and furthermore, that in several countries, it is expected that **all** children will learn programming as a standard part of their education.

In contrast, the commonly held belief of Computer Science educators and researchers at tertiary level is that programming is difficult to teach, and difficult to learn. In a 2003 review of the literature on teaching and learning programming, Robins summarizes the orthodox view:

> *Learning to program is hard however. Novice programmers suffer from a wide range of difficulties and deficits. Programming courses are generally regarded as difficult, and often have the highest dropout rates. [26]*

This view of computer programming as a difficult skill to learn may have implications for equity and diversity. It may encourage students to engage in activities that are not conducive to learning, it is potentially unfair to students and may also lead to research practices that focus on student shortcomings rather than curriculum deficiencies.

This paper challenges the orthodox view that computer programming is innately difficult to learn. Instead, an alternative explanation is proposed to explain the challenges faced by students as they learn to program.

2. STUDENT PERFORMANCE

The belief that learning to program is difficult appears to be widespread among teachers and researchers involved in Computer Science Education (e.g. see [5, 10, 26, 35]). Discussions about programming frequently begin with the **premise** that it is hard. Mark Guzdial noted in a 2010 blog post:

> *So what makes programming so hard to learn? Here's a possibility: It's inherently hard. Maybe the task of programming is innately one of the most complex cognitive tasks that humans have ever created.[14]*

So why do we think programming is hard? What is the evidence? In the following sections, we consider data from research studies of novice programmer ability, and course achievement data.

2.1 Failure rates

The conventional view of programming expressed in the Computer Science Education community is that:

ITiCSE '16, July 09 - 13, 2016, Arequipa, Peru

ISBN 978-1-4503-4231-5/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2899415.2899432

> *. . . learning to program can be an incredibly difficult task, to the point where the phrases "failure rate" and "programming course" are almost synonymous. [35]*

Although it is often claimed that high failure rates are commonly observed in programming courses [6, 10, 26], there is little conclusive evidence to either support, or dispute, this view.

Bennedsen and Caspersen [5] conducted an empirical study to determine average pass rates among introductory programming courses. Instructors from 61 institutions contributed data, which revealed that an average of 67% of students passed their introductory programming course. Although the number of institutions was a relatively small sample of those teaching introductory programming, they concluded that it was hard to justify the claim that introductory programming courses have a high failure rate based on the data they collected. However, the authors note a significant risk of sample bias since those responding to the survey are active CS education researchers and more likely to be concerned and proactive in improving their teaching.

A subsequent study by Watson and Li [35] analysed data from 161 introductory programming courses across 15 countries. They found that an average of 67.7% of students passed, confirming the previous findings of Bennedsen and Caspersen [5]. An analysis of historical data revealed no significant differences in pass rates since the 1980's, suggesting that a pass rate of around 67% is typical and has remained so for more than 30 years.

Although the existing data seems to suggest that approximately a third of students in CS1 fail to learn how to program, it is not obvious whether this is particularly unusual for introductory courses.

It was not possible to obtain average pass rates for all courses taught internationally, but an analysis of national data from New Zealand revealed that the average pass rate across all degree-level courses was 82% [27]. Compared with this measure, the average introductory course appears to be substantially lower, which may explain why programming courses are perceived to be more difficult than other courses.

2.2 Novice programmer ability

There has been extensive research around novice programming capability, and overwhelming evidence that novice programmers cannot successfully complete the tasks we set. As early as the the 1980s, researchers such as Soloway and his colleagues showed that most novice programmers were unable to develop a working program for relatively simple problems, such the *Averaging Problem / Rainfall Problem* described below [30].

> *Write a program that repeatedly reads in positive integers, until it reads the integer 99999. After seeing 99999, it should print out the average.*

The more recent multi-national, multi-institution studies such as those by McCracken et al. [22], Lister et al. [20], and Whalley et al. [36] provide further evidence that novice student programmers cannot perform at the level expected by researchers.

McCracken et al. [22] formed a ITiCSE Working Group out to answer the question: *Do students in introductory computing courses know how to program at the expected skill level?* The authors explicitly developed a set of learning objectives that described the expected level of skill for students at the end of their first year of study. Students were expected to be able to complete the following steps:

1. Abstract the problem from its description
2. Generate sub-problems
3. Transform sub-problems into sub-solutions
4. Re-compose the sub-solutions into a working program
5. Evaluate and iterate

Students in four universities were given programming tasks to solve. The vast majority of students were unable to develop correct solutions and many were unable to produce a solution that even compiled correctly. McCracken et al. [22] concluded that students in introductory courses did not know how to program at the expected skill level, and that the problem is fairly universal.

However, concerns were expressed that the tasks specified by McCracken et al. [22] were simply too difficult for students. This motivated a subsequent ITiCSE Working Group known as the "Leeds group", which investigated student performance on what was thought to be an easier task — the ability of students to trace code [20].

In the Leeds study, students from twelve universities were asked to answer a set of 12 multiple choice questions that required them to understand short samples of programming code. In general, students performed better than in the McCracken study with approximately half of the students correcting answering at least 2/3 of the questions. Although it is encouraging that some students can read and understand code, the bottom quartile answered less than 1/3 of the questions correctly. The authors express concern that a course with a pass rate of 75% would result in very weak students, who could correctly answer only 5 of 12 questions, potentially advancing to a typical CS2 course even though they demonstrated a very weak understanding of code.

Several multi-national, multi-institutional studies from the BRACElet group [36] investigated novice programmer performance through the analysis of student responses to exam questions. The SOLO taxonomy [8] was used to distinguish between students who could correctly determine the output of a piece of code by performing a code trace (i.e. multi-structural answers), and students who abstracted the meaning and could describe the purpose of a code segment in more general terms (i.e. relational answers). In one study that used the SOLO taxonomy to categorize student solutions to exam questions, only around one third of all students were able to produce an answer in relational terms. Even in the top quartile, less than half of the students responded in relational terms [21]. Lister et al. state that the ability to abstract the meaning of a set of instructions as a single "chunk" of code is a criticially important component of code writing. They claim:

> *. . . students who cannot read a short piece of code and describe it in relational terms are not intellectually well equipped to write similar code. [21]*

In a study exploring the impact of different teaching approaches on novice programming understanding, Tew et al. [34] found that few students at the end of CS1 had a solid understanding of programming fundamentals. An average of only

42.29% of students who had completed CS1 answered introductory topic MCQs correctly. After completing a second course (CS2), an average of 61.53% of the students answered the introductory topic MCQs effectively.

2.3 Summary

The literature on failure rates (see section 2.1) and novice programmer ability (see section 2.2) have provided a substantial body of evidence that students are not able to program *at the level expected by instructors* by the end of an introductory programming course. Evidence suggests that the majority are unable to either read or write code typically covered in introductory programming courses. It is clear that it is difficult to master the content currently covered in typical CS1 courses in the available time. It is less clear that this means it is difficult to learn how to program.

3. EXPECTATIONS

In a post to the SIGCSE mailing list in 2014, Stephen Bloch recalled how he sometimes introduces the subject of programming in a CS course:

> *"Suppose I assigned you to write a fifteen-page paper, due two months from now, on Napoleon's invasion of Russia. [...] Now suppose I told you that the paper must be written in Swedish, using a quill pen. Now what would you need to know? [...] And if I expected you to learn all those things, from scratch, in one semester, you'd think I was nuts. That's what we'll be doing in CS1, trying to learn half a dozen different levels of knowledge at once." [9]*

Other teachers have compared learning programming to learning to write poetry in a foreign language. Anecdotally, such analogies are not uncommon among teachers describing programming — a subject in which students are typically expected to develop a mental model of a notional machine [31], understand a formal language, acquire technical expertise with a variety of software development tools, and to solve previously unseen problems using their knowledge of the machine and language.

The Computer Science Curricula 2013 [1] discusses the various tradeoffs between design decisions in first year courses, but is appropriately non-prescriptive with respect to first year curricula. Although there are exceptions, the expectations we have for a first programming course are reasonably similar across different institutions and across different countries. As a community, we have internalized what we think students should be able to achieve at the end of a novice programming course. Typically, students are expected to be able to write small programs that minimally use conditions, loops and arrays. They are frequently expected to solve problems using functional decomposition and write programs involving multiple methods. Although they are often not explicitly assessed, we also frequently expect students to acquire familiarity, if not expertise, with the systems and tools they use while programming on a given platform. We expect a lot, and the evidence indicates that many, if not most, students are unable to meet our lofty expectations.

3.1 Assessments

Not only do we expect many different kinds of learning within a CS1 course, it seems likely that the assessments we use to evaluate learning are simply too hard. Despite their working group consisting of international experts in the field, McCracken et al. [22] acknowledge that their expectations may have been too high. In other words, prior to the formal research project, a group of expert teachers who were also active Computer Science Education Researchers overestimated the capability of novice programmers. In a later study of exam questions, Whalley et al. [37] conclude that is likely programming educators are *systemically underestimating the cognitive difficulty* of their instruments for assessing programming skills of novice programmers.

The widespread use of assessment instruments that are more cognitively demanding than instructors expect have implications for our pass rates and grade distributions. Lister [19] expresses concern that the grade distributions we see in CS1 are not reflective of student ability, but rather the methods used to grade. He observes:

> *... if the computing education research community is going to have a productive discourse about CS1 grade distributions, then we must consider the validity of current approaches to grading. [19]*

Robins [25] suggests that concepts in programming are highly inter-related and proposes Learning Edge Momentum theory to explain how highly integrated concepts may result in the binomial distribution that is sometimes observed in CS1 courses. It is unclear whether or not programming *concepts* are more inter-related than other subjects that do not typically have binomial grade distributions. However, a study of exam questions by Petersen et al. [24] indicated that the *exam questions* used in CS1 were highly inter-related, and students were required to have knowledge of many different concepts before they were able to answer the questions successfully. Our *assessment practices* are a likely cause of binomial distributions. It is not necessarily the case that programming concepts are densely intertwined, but rather, our assessment instruments consist of questions that are densely intertwined and do not give students the opportunity to show which concepts they understand and which they do not.

4. UNREALISTIC EXPECTATIONS

Ultimately, the difficulty of any subject depends on the standards by which success in that subject is measured. For example, all primary school students are expected to learn basic mathematics skills such as addition. If success in year one mathematics was determined by performance of adding two single digit numbers together, then "learning to add numbers" would be considered to be easy. If instead, students were expected to add two 10 digit numbers together, then "learning to add numbers" would be described as extremely difficult. Learning to add numbers would have high failure rates and would be discouraging, perhaps leading to larger numbers of people avoiding mathematics in the future.

There is nothing intrinsic to the subject that makes it difficult to learn, but rather our subjective assessment of how much a student "should" be able to achieve by the end of the course that determines the difficulty. Learning to program

is not difficult, as we have become accustomed to believing, but rather our expectations of what students "should" be able to do at the end of a first course are unrealistic.

It is time for the Computer Science Education community to undergo a paradigm shift. Rather than viewing the subject of programming as being intrinsically difficult to learn, we should be viewing our learning outcomes for novice programming courses as being unrealistic. The evidence that students are not meeting our expectations is overwhelming. In fact, research on novice programmers for at least 30 years has suggested that students have *never* met our expectations. This is not a problem with the students. This is (probably) not a problem with the way that we are delivering content. This is almost certainly a problem with the assessment of novice programming skills, derived from a mismatch between the progress that a novice programmer can realistically make in a single course and the established norms of expectation for what we *think* novice programmers *should* be able to achieve after a single course.

4.1 Consequences for Instructors

One potential problem that can occur if instructors have internalized an unachievable standard is that we assume the problem is with the students rather than the course, or we start to treat the subject itself as being difficult and one in which we now *expect* students to fail. As Bennedsen and Caspersen note:

> False views on failure and pass rates can have serious implications for the quality of introductory programming courses. A lecturer with a high failure rate might accept that "this is just the way programming courses are since all programming courses have high failure rates" and consequently not take action to improve the course in order to reduce the failure rate. [5]

4.2 Consequences for Students

If a course expects too much from students, then it is likely to result in higher rates of undesirable behavior, such as surface learning strategies and plagiarism [28]. The workload is likely to be too demanding for students and they are more likely to drop the course. Such issues have a greater impact on marginalized groups in the subject, perhaps contributing to the ongoing inequity between genders in Computer Science.

In Computer Science courses, drop rates of 30–40% are reported by several institutions [3]. A number of studies have identified excessive workload as being a significant factor in the decision of a student to drop out of a course, or to change majors. A recent study reported that "most incoming students appear to underestimate the number of hours of work that they are likely to be doing" [29]. In another study, students reported that the programming exercises took too much time and that the workload was higher than their other courses [18]. Students find that the workload in Computer Science courses is too high, the course is too difficult and takes too much time, so students drop out [17].

In an environment with such demanding expectations and excessive workload, students who are already familiar with the subject matter have a significant advantage. A study of factors contributing to the success of novice programmers found that prior programing experience was a significant factor [39]. A more recent study found that students who entered novice programing courses with prior programming experience perform significantly better than those with no experience [15].

Given that more women than men are choosing to study Computer Science without any prior programming experience [29], this places women in novice programming courses at a severe disadvantage. A study of the reasons that women give for changing majors from Computer Science to another subject found that the excessive workload was a major factor, along with the low grades given in Computer Science courses [7]. The low grades awarded in particularly difficult courses appear to discourage women more significantly than men [40]. Students who believe that the pace and workload of novice programming courses is high, especially given their level of experience, are unlikely to pursue a Computer Science major [2].

As a community of educators, we have established unrealistically high expectations for the level of achievement that students can reach at the end of a single course. This has resulted in problematic rates of plagiarism, excessively high workload, high failure rates and high drop-out rates. These practices appear to have an especially high impact on women and may partially explain the gender inequity observed in the discipline of Computer Science.

5. ALTERNATIVES

Research on novice programmer ability has provided substantive evidence that students are not able to meet instructor expectations *within the time frame of a single course* (see section 2.2). However, it seems likely that students are capable of achieving the desired standards over a longer time frame. Tew et al. [34] provided evidence that less than half of the students at the end of a CS1 course could correctly answer questions about introductory programming topics. However, almost two thirds of the students at the end of a CS2 course were able to answer similar questions. This suggests that the introductory topics take most students longer to master than the time in a single course permits.

A longitudinal study by Teague and Lister [32, 33] provides further evidence that students may take longer than expected to acquire programming knowledge. One particular student was observed near the end of a typical introductory programming course attempting to understand code that used a loop to rotate all the elements in an array by one place to the right. The student was unable to clearly articular how the code worked and could not successfully rewrite the code to shift the elements in the array one place to the left. However, the same student was able to perform these tasks successfully one year later. Although the student struggled with basic concepts throughout their introductory course, he progressed to master that material a year later and eventually graduated with a high grade point average [32].

Although it is currently unknown how long it takes typical students to master basic programming skills, it appears to take longer than we expect. Introducing CS1 material at a slower pace has been reported positively, and may be a viable interim solution [38].

To truly establish reasonable and realistic expectations for student outcomes at the end of a novice programming course, we first need to know how long it takes students to master the various programming fundamentals. To this end, a challenge for the CS Education community is to deter-

mine an appropriate time frame for mastery of fundamental programming concepts and establishing appropriate expectations for the end of a introductory programming course — not based on historical norms, but rather on research based evidence.

6. CONCLUSION

Our current approach to teaching programming is to cover too much content too rapidly and expect students to be able to program at a higher level than they are capable of achieving at the end of an introductory programming course. The expectations we set for our students result in programming courses that are notoriously time-consuming and have high drop out and failure rates. These factors appear to have a greater impact on women and may be partially responsible for the gender inequity observed in the Computer Science discipline. This is not because programming is intrinsically difficult (at least not at novice level), but rather because we have collectively adopted disciplinary norms that are, and always have been, unrealistic. Learning to program is easy — all we need to do is collectively shift our view, and teach to achievable outcomes. The paper concludes with a challenge to the Computer Science Education community — collect research-based evidence of what novice programmer *can achieve* at the end of a first programming course and use evidence to derive realistic expectations for achievement.

7. REFERENCES

[1] ACM/IEEE-CS Joint Task Force on Computing Curricula. Computer science curricula 2013. Technical report, ACM Press and IEEE Computer Society Press, December 2013.

[2] L. J. Barker, C. McDowell, and K. Kalahar. Exploring factors that influence computer science introductory course students to persist in the major. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, pages 153–157, New York, NY, USA, 2009. ACM.

[3] T. Beaubouef and J. Mason. Why the high attrition rate for computer science students: Some thoughts and observations. *SIGCSE Bull.*, 37(2):103–106, June 2005.

[4] T. Bell. Establishing a nationwide cs curriculum in new zealand high schools. *Commun. ACM*, 57(2):28–30, Feb. 2014.

[5] J. Bennedsen and M. E. Caspersen. Failure rates in introductory programming. *SIGCSE Bull.*, 39(2):32–36, June 2007.

[6] S. Bergin and R. Reilly. The influence of motivation and comfort-level on learning to program. In P. Romero, J. Good, E. A. Chaparro, and S. Bryant, editors, *Proceedings of 17th Workshop of the Psychology of Programming Interest Group*, pages 293–304, Sussex University, June 2005.

[7] M. Biggers, A. Brauer, and T. Yilmaz. Student perceptions of computer science: A retention study comparing graduating seniors with cs leavers. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 402–406, New York, NY, USA, 2008. ACM.

[8] J. B. Biggs and K. F. Collis. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press, New York, 1982.

[9] S. Bloch. Re: Motivating students: Was survey results. SIGCSE-members@listserv.acm.org, 14th October 2014.

[10] R. Bornat, S. Dehnadi, and Simon. Mental models, consistency and programming aptitude. In *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78*, ACE '08, pages 53–61, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[11] N. C. C. Brown, S. Sentance, T. Crick, and S. Humphreys. Restart: The resurgence of computer science in uk schools. *Trans. Comput. Educ.*, 14(2):9:1–9:22, June 2014.

[12] M. E. Caspersen and P. Nowack. Computational thinking and practice: A generic approach to computing in danish high schools. In *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136*, ACE '13, pages 137–143, Darlinghurst, Australia, Australia, 2013. Australian Computer Society, Inc.

[13] K. Falkner, R. Vivian, and N. Falkner. The australian digital technologies curriculum: Challenge and opportunity. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*, ACE '14, pages 3–12, Darlinghurst, Australia, Australia, 2014. Australian Computer Society, Inc.

[14] M. Guzdial. Is learning to program inherently hard? Retrieved from: https://computinged.wordpress.com/2010/04/14/is-learning-to-program-inherently-hard/, April 2010.

[15] D. Horton and M. Craig. Drop, fail, pass, continue: Persistence in cs1 and beyond in traditional and inverted delivery. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 235–240, New York, NY, USA, 2015. ACM.

[16] C. Kelleher and R. Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137, June 2005.

[17] P. Kinnunen and L. Malmi. Why students drop out cs1 course? In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 97–108, New York, NY, USA, 2006. ACM.

[18] P. Kinnunen and L. Malmi. Cs minors in a cs1 course. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pages 79–90, New York, NY, USA, 2008. ACM.

[19] R. Lister. Computing education research: Geek genes and bimodal grades. *ACM Inroads*, 1(3):16–17, Sept. 2011.

[20] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull.*, 36(4):119–150, June 2004.

[21] R. Lister, B. Simon, E. Thompson, J. L. Whalley, and

C. Prasad. Not seeing the forest for the trees: Novice programmers and the solo taxonomy. *SIGCSE Bull.*, 38(3):118–122, June 2006.

[22] 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. *SIGCSE Bull.*, 33(4):125–180, Dec. 2001.

[23] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA, 1980.

[24] A. Petersen, M. Craig, and D. Zingaro. Reviewing cs1 exam question content. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 631–636, New York, NY, USA, 2011. ACM.

[25] A. Robins. Learning edge momentum: a new account of outcomes in cs1. *Computer Science Education*, 20(1):37–71, 2010.

[26] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13:137–172, 2003.

[27] D. J. Scott. A closer look at completion in higher education in new zealand. *Journal of Higher Education Policy and Management*, 31(2):101–108, 2009.

[28] J. Sheard and M. Dick. Directions and dimensions in managing cheating and plagiarism of it students. In *Proceedings of the Fourteenth Australasian Computing Education Conference - Volume 123*, ACE '12, pages 177–186, Darlinghurst, Australia, Australia, 2012. Australian Computer Society, Inc.

[29] J. Sinclair and S. Kalvala. Exploring societal factors affecting the experience and engagement of first year female computer science undergraduates. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling '15, pages 107–116, New York, NY, USA, 2015. ACM.

[30] E. Soloway, J. Bonar, and K. Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Commun. ACM*, 26(11):853–860, Nov. 1983.

[31] J. Sorva. Notional machines and introductory programming education. *Trans. Comput. Educ.*, 13(2):8:1–8:31, July 2013.

[32] D. Teague and R. Lister. Longitudinal think aloud study of a novice programmer. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*, ACE '14, pages 41–50, Darlinghurst, Australia, Australia, 2014. Australian Computer Society, Inc.

[33] D. Teague and R. Lister. Programming: Reading, writing and reversing. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE '14, pages 285–290, New York, NY, USA, 2014. ACM.

[34] A. E. Tew, W. M. McCracken, and M. Guzdial. Impact of alternative introductory courses on programming concept understanding. In *Proceedings of the First International Workshop on Computing Education Research*, ICER '05, pages 25–35, New York, NY, USA, 2005. ACM.

[35] 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*, ITiCSE '14, pages 39–44, New York, NY, USA, 2014. ACM.

[36] J. L. Whalley and R. Lister. The bracelet 2009.1 (wellington) specification. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*, ACE '09, pages 9–18, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[37] J. L. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, P. K. A. 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*, ACE '06, pages 243–252, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[38] K. J. Whittington, D. P. Bills, and L. W. Hill. Implementation of alternative pacing in an introductory programming sequence. In *Proceedings of the 4th Conference on Information Technology Curriculum*, CITC4 '03, pages 47–53, New York, NY, USA, 2003. ACM.

[39] B. C. Wilson and S. Shrock. Contributing to success in an introductory computer science course: A study of twelve factors. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '01, pages 184–188, New York, NY, USA, 2001. ACM.

[40] J. Wolfe and B. A. Powell. Not all curves are the same: Left-of-center grading and student motivation. In *2015 ASEE Annual Conference and Exposition*, number 10.18260/p.24527, Seattle, Washington, June 2015. ASEE Conferences. https://peer.asee.org/24527.