Chapter 3

Students Worldwide Do Not Learn to Program

The previous chapter made the theoretical observation that programming courses have demanding goals. This is borne out by concrete evidence: poor results in introductory programming education have been widely reported. Section 3.1 below reviews research on learning outcomes, which indicates that many students are not learning to write programs in CS1. Section 3.2 considers the relationships between code-reading skill and code-writing skill, but sadly, it turns out in Section 3.3 that many students do not even learn to read program code reliably in CS1. Finally in Section 3.4, I review research on the various specific problems that novice programmers have with understanding fundamental programming concepts.

3.1 Many students do not learn to write working programs

> *Few teachers of programming in higher education would claim that all their students reach a reasonable standard of competence by graduation. Indeed, most would confess that an alarmingly large proportion of graduates are unable to 'program' in any meaningful sense.* (Carter and Jenkins, 1999)

This is not recent news. According to the review of CER literature from the 1980s by Robins et al. (2003), "an observation that recurs with depressing regularity, both anecdotally and in the literature, is that the average student does not make much progress in an introductory programming course". Linn and Dalbey (1985) report that most students struggled to get past learning language features and never got to learning about higher-order skills of program planning and general problem-solving strategies for programming. Kurland et al. (1986) concluded that after two years of programming instruction, many high-school students had only a rudimentary understanding of programming. Guzdial reviews the work of Soloway and others:

> *One of the first efforts to measure performance in CS1 was in a series of studies by Elliot Soloway and his colleagues at Yale University. They regularly used the same problem, called "The Rainfall Problem": Write a program that repeatedly reads in positive integers, until it reads the integer 99999. After seeing 99999, it should print out the average. In one study, only 14% of students in Yale's CS1 could solve this problem correctly. The Rainfall Problem has been used under test conditions and as a take-home programming assignment, and is typically graded so that syntax errors don't count, though adding a negative value or 99999 into the total is an automatic zero. Every study that I've seen (the latest in 2009) that has used the Rainfall Problem has found similar dismal performance, on a problem that seems amazingly simple.* (Guzdial, 2011, see also Soloway et al., 1982; Venables et al., 2009)

A common topic of conversation among computing education researchers is the 'Bactrian' grade distribution of many introductory programming courses: students either fail miserably or pass with flying colors, with few 'just doing okay' (Dehnadi and Bornat, 2006). Many explanations have been offered and many studies have been conducted, but the phenomenon remains unexplained (see, e.g., Bornat et al., 2008; Robins, 2010).

Multi-institutional studies

In the past decade, several international working groups have looked into the skill levels of students at the end of CS1 courses, or, in some cases, at the end of a degree program (McCracken et al., 2001; Lister et al., 2004; Eckerdal et al., 2006b). These oft-cited studies have been influential as they produced concrete evidence of the mismatch between the goals of programming education and the actual skills gained.

In 2001, a multi-institutional, multi-national working group chaired by Michael McCracken gave a set of program-writing problems of varying difficulty to students completing CS1 or CS2 in several countries. The students only got an average score of approximately 23 out of 110 points. The authors state that their "first and most significant result was that the students did much more poorly than we expected" and that "the disappointing results suggest that many students do not know how to program at the conclusion of their introductory courses" (McCracken et al., 2001).

Another international working group study explored program design skills. When Eckerdal et al. (2006b) analyzed the designs produced by students, they found "poor performance from students who are near graduation: over 20% produced nothing, and over 60% communicated no significant progress toward a design". They conclude that "the majority of graduating students cannot design a software system". A recent follow-up study by Loftus et al. (2011) produced similar results.

It is clear by now that learning to write programs is a challenging goal. Teachers aware of this have tried to find ways of easing the burden of students as they gradually develop code-writing ability. The question then becomes: what are the relationships between the various goals of programming education? Or more specifically: what other skills does the skill of writing code build on? There is some limited evidence of dependencies between programming skills, so that learning certain skills builds on learning others first.

3.2 Code-writing skill is (loosely?) related to code-reading skill

> *The ability to write program code is what we aim to teach, so anything else that we can discover about students' acquisition of skills must ultimately be considered in the light of their ability to write code.* (Lister et al., 2009a)

Many programming teachers find it intuitive, even self-evident, that one must learn to read code before one can write code, just as one learns to read their first natural language before learning to write in it. Similarly, the ability to trace a program's execution steps would seem to be a prerequisite both for explaining what given code accomplishes and for writing code. However, students of programming tend to give examples and reading tasks little attention compared to writing; some also say right out that writing code is easier than reading (Simon et al., 2009).

Does learning to explain what a piece of code does precede learning to write code? Can people learn to write code successfully without being able to trace its execution (and code of what kind)? What comes before the ability to explain code? More generally, is there evidence of a general learning path of programming skills?

From taxonomy to learning path?

Bloom's taxonomy (Section 2.1) is not of much assistance in the search for a general learning path. Even if we assume that the cognitive categories form a hierarchy of increasingly complex learning objectives and assessments, there is no evidence in the general case that learning a higher-ranking skill requires the lower-ranking skills to be learned first. It has been argued that Bloom's taxonomy does not match the path(s) of skill progression that learners take, either in general or for programming in particular (e.g., Fuller et al., 2007; Anderson et al., 2001; Eckerdal et al., 2007; Biggs and Tang, 2007).

Other frameworks may match the learning process better. A substantial body of empirical research that seeks to discover how programmers' skills develop has recently been produced by the BRACElet project (for an overview, see Clear et al., 2011). This empirical work builds on SOLO (Section 2.2), a taxonomy that was designed to reflect stages of learning.

BRACElet: some evidence of skill dependencies

In terms of one interpretation of SOLO (Section 2.2), code-tracing skills (that is, the ability to step through a program's execution) rank below code-explaining skills (that is, the ability to determine and state the overall purpose of a piece of code), as the former requires multistructural learning outcomes while the latter requires relational ones. If SOLO levels represent a learning path, students would need to learn to trace code of a particular kind and complexity before they learn to explain code of a similar kind and complexity. The BRACElet members hypothesized that learners generally first learn to trace programs, then to explain them, and finally to write them.

A number of BRACElet publications have produced empirical evidence that links the skills of tracing, explaining, and writing code. Students who write programs successfully tend to be able to produce correct overall explanations of what a given piece of code does (Whalley et al., 2006). Students' abilities to explain and write correlate positively (Sheard et al., 2008). Students who cannot trace code are usually also unable to explain what a given piece of code does (Philpott et al., 2007). Summarizing what given code does appears to be an intermediate-level skill, which programming experts use naturally in lieu of line-by-line traces of programs, but which many novices struggle with (Lister et al., 2006b).

Lopez et al. (2008) report that performance level in a code-tracing task accounts for some of the variation in code-explaining performance. Further, they found that code-explaining ability alone, or code-tracing ability alone, appears to account for only a little of the variation in code-writing skills. However, explaining and tracing abilities together account for a substantial amount of the variation in writing ability. They conclude that if one posits that a causal model exists between the skills, then their findings support a hierarchy where tracing is lower than explaining, which is again lower than writing. The later results of Lister et al. (2009b) and Venables et al. (2009) are consistent with the analysis of Lopez et al. In sum, the BRACElet research points "to the possibility of a hierarchy of programming related tasks where knowledge of programming constructs forms the bottom of the hierarchy, with 'explain in English', Parson's problems, and the tracing of iterative code forming one or more intermediate levels in the hierarchy" (Clear et al., 2011).

Simon et al. (2009) followed up on these studies. They found that – contrary to expectations – no meaningful relationships could be established when comparing the marks of a writing task and a comparable reading task (a 'Parson's problem', which requires the sorting of lines of code, and has been shown to assess similar skills to traditional code-writing questions; see Denny et al., 2008). This result may be explained by: 1) the lack of established criteria for comparing the complexity of two comparable programs from a learning point of view; 2) the differences between marking code-reading problems on the one hand and code-writing problems on the other (Simon et al., 2009), and 3) the ambiguities of code-explaining questions in general (Simon, 2009).

Other researchers have found some evidence to support the claim that novices can produce code based on familiar templates even without being good at tracing it (Anderson et al., 1989; Thomas et al., 2004). It can be questioned, however, whether such novices can reliably produce bug-free code and whether they can fix all the bugs in the code they produce without successfully tracing the programs.

As the BRACElet authors themselves have noted, the results from their line of research are not straightforward to interpret, and the matter of a learning hierarchy of programming skills remains unsolved. Furthermore, research has so far has focused on simple tracing, code-explaining, and writing tasks, and little has been shown about the relationship of these skills to other programming skills such as debugging, program design, or dealing with larger amounts of code. Nevertheless, it is easy to agree at least with the cautious conclusion of Venables et al.:

> *In arguing for a hierarchy of programming skills, we merely argue that some minimal competence at tracing and explaining precedes some minimal competence at systematically writing code. Any novice who cannot trace and/or explain code can only thrash around, making desperate and ill-considered changes to their code – a student behavior many computing educators have reported observing.* (2009, p. 128)

3.3 But many students do not learn to read code, either

In the light of the previous chapter, it is not very surprising that learners fail to learn to write and design programs. After all, these are cognitively complex skills that require relational understandings of structurally complex content. What about presumably less complex programming skills, such as tracing and explaining? Even though a strict learning hierarchy may not exist, the skill of program writing is partially dependent on these less complex skills. Are introductory programming courses succeeding in teaching students to read code?

Following up on the McCracken investigation described in Section 3.1, Lister et al. (2004) measured students' ability to trace through a given program's execution. They gave a multiple-choice questionnaire to students in a number of educational institutions around the world. The questions required the students, who were near the end of CS1, to predict the values of variables at given points of execution and to complete short programs by inserting a line of code chosen from several given alternatives. A quantitative analysis of the multiple-choice questions was complemented by student interviews and an analysis of the 'doodles' students made while tracing. Lister et al. found that many students were unable to trace. While there was obviously some variation in students' ability between institutions, the results were disappointing across the board.

Other studies have produced similar results. For instance, an earlier multi-institutional study by Sleeman et al. (1986) analyzed code-driven interviews to conclude that "at least half of the students could *not* trace through programs systematically" upon request and instead "often decided what the program would do on the basis of a few key statements". Adelson and Soloway (1985) found that novices were unable to mentally trace interactions within the system they were themselves designing. Kaczmarczyk et al. (2010) report an inability to "trace code linearly" as a major theme of novice difficulties. The analyses of quiz questions by Corney et al. (2011), Simon (2011), Teague et al. (2012), and Murphy et al. (2012) indicate that many students fail to understand statement sequencing to the extent that they cannot grasp a simple three-line swap of variable values. In one study, the problem existed even among students taking a third programming course (Simon, 2011).

It is clear from these results that students' foundational programming skills commonly do not develop as well as teachers expect. These results are discouraging, but cannot be ignored; of particular significance is the fact that many of the findings have been produced by international, multi-institutional studies which demonstrate that the problem is not local to a particular university, country, or type of institution.

3.4 What is more, many students understand fundamental programming concepts poorly

So far in this chapter, we have focused on skills. What about conceptual understanding? We gain another perspective on the challenge of introductory programming education by looking at what is known about novices' conceptions of programming concepts.

In some disciplines, concepts and phenomena are largely negotiable and up for interpretation. Students may be encouraged to interpret things in a personal way and to develop alternative conceptual frameworks. Certainly, many computing concepts are like this, too. However, computing also features many concepts that are precisely defined and implemented within technical systems. Students are expected to reach particular ways of understanding what the assignment statement in Java does, of what an object is, and of how a given C program executes. Sometimes a novice programmer 'doesn't get' a concept, or 'gets it wrong' in a way that is not a harmless (or desirable) alternative interpretation. Incorrect and incomplete understandings of programming concepts result in unproductive programming behavior and dysfunctional programs. Unfortunately, misconceptions of even the most fundamental programming concepts, which are trivial to experts, are commonplace among novices and challenging to overcome. Recent studies measuring students' conceptual knowledge suggest that introductory programming courses are not particularly successful in teaching students about fundamental concepts, and that the problems are not limited to a single institution nor caused by the use of a particular programming language (Elliott Tew, 2010; Kunkle, 2010).

Over the past few decades, many researchers have catalogued ways in which programming students

struggle with fundamental concepts, and the kinds of incomplete and incorrect understandings that students have exhibited. Variables, assignment, references and pointers, classes, objects, constructors and recursion are among the CS1 concepts most commonly reported as problematic. Many students appear to have problematic understandings about the capabilities of computers and programs in general.

What follows is a review of research on the misconceptions and limited understandings that novice programmers have been found to have about fundamental programming concepts. Some examples of misconceptions appear below; there is a more comprehensive list in Appendix A. The reader can find a longer review of the earlier work on misconceptions within CER in a book chapter by Clancy (2004).

Research on programming misconceptions

Bayman and Mayer (1983) studied beginners' interpretations of statements in the BASIC language by asking students to write plain English explanations of programs. They list a number of misconceptions about BASIC semantics, e.g., `LET` statements are understood as storing equations instead of assigning to a variable. Around the same time, Soloway, Bonar, and their colleagues also explored novice misconceptions and bugs, and discussed how they may be caused by knowledge from outside of programming, particularly by analogies with the everyday semantics of natural language (e.g., Soloway et al., 1982; Bonar and Soloway, 1985; Soloway et al., 1983).

Samurçay (1989) studied the answers that programming beginners gave to three program completion tasks, and reported that variable initialization in particular was difficult for students to grasp. Putnam et al. (1986) and Sleeman et al. (1986) analyzed students' answers to code comprehension tests and subsequent interviews. They listed numerous errors – surface and deep – that students make with variables, assignment, print statements, and control flow. Du Boulay (1986) likewise listed a number of novice misconceptions with variables, assignment, and other fundamental programming concepts.

Pea (1986) listed a number of common novice bugs and suggested that they are rooted in a 'superbug', the assumption that there is a hidden, intelligent mind within the computer that helps the programmer to achieve their goals.

Fleury (1991) and Madison and Gifford (1997) interviewed and observed students to discover various conceptions of parameter passing. Their results suggest that even students who are sometimes capable of producing working code with parameters may misunderstand the concepts involved in different ways.

Recursion has been the focus of several studies. Kahney (1983) discovered that students have various flawed models of recursion, such as the "looping model", in which recursion is understood to be much like iteration. Kahney's work has since been elaborated on by Bhuiyan et al. (1990), George (2000a,c), and Götschi et al. (2003). Recursion is also one of the phenomena investigated by Booth (1992) in her phenomenographic work on learning to program. Booth identified three different ways of experiencing recursion: as a programming construct, as a means for repetition, and as a self-reference; students are not always able to grasp all of these aspects.

Recent themes: OOP and Java

Since the '90s, interest in CER has shifted from procedural programming towards object-oriented programming. Several studies have reported ways in which students misunderstand object-oriented concepts and features of OO languages. Holland et al. (1997) noted several misconceptions students have about objects. For instance, students sometimes conflate the concepts of object and class, and may confuse an instance variable called `name` with object identity. More novice misconceptions about OOP were reported by Détienne (1997) as part of her review of the cognitive consequences of the object-oriented approach to teaching programming.

Fleury (2000) reported that students form their own, unnecessarily strict rules of what happens in programs and what works in Java programming. For instance, some of the students she studied thought that the dot operator could only be applied to methods and that the only purpose of a constructor was to initialize instance variables.

Hristova et al. (2003) listed a number of common errors students make when programming in Java. Many of these are on a superficial syntactic level (e.g., confusing one operator with another) but some suggest deeper-lying misconceptions. Ragonis and Ben-Ari (2005a) reported the results of a wide-scope,

long-term, action research study of high-school students learning object-oriented programming. Their study uncovered an impressive array of misconceptions and other difficulties students have with object-oriented concepts, the Java language, and the BlueJ programming environment.

Teif and Hazzan (2006) observed students of introductory programming in two high-school courses and discuss students' conceptual confusion regarding classes and objects. For instance, students may incorrectly think that the relationship between a class and its instances is partonomic, i.e., that objects are parts of a class. Eckerdal and Thuné (2005) also studied the conceptions that students have of these fundamental object-oriented concepts. Their results highlight the fact that not all students learn to appreciate objects and classes as dynamic execution-time entities or as modeling tools that represent aspects of a problem domain.

Vainio (2006) used interviews to elicit students' mental models of programming concepts. Among his results are a number of misconceptions about fundamental concepts, e.g., the idea that the type of a value can change on the fly (in Java).

Several recent, complementary studies affirmed the existence of a number of incorrect understandings of assignment, variables, and the relationships between objects and variables. Ma (2007) gave a large number of volunteer CS1 students a test with open-ended and multiple-choice questions about assignment in Java. He analyzed the results both qualitatively and quantitatively to understand students' mental models of assignment and reference semantics. I myself explored students' understandings of storing objects and of Java variables through phenomenographic analyses of interviews (Sorva, 2007, 2008). Doukakis et al. (2007) combined findings from the literature and anecdotal evidence to list introductory programming misconceptions, which, the authors argue, arise as a result of students' prior knowledge of mathematics.

Sajaniemi et al. (2008) elicited the mental models that novice programmers have of program state by having CS1 students draw and write about how they perceived given Java programs' state at a specific stage of execution. They discovered numerous misconceptions relating to parameter passing and object-oriented concepts.

As part of a project to develop a concept inventory for CS1, Kaczmarczyk et al. (2010) used interviews to identify student misconceptions about concepts and grouped the misconceptions thematically. Four themes were identified: the relationship between language elements and memory, while loops, the object concept, and code-tracing ability.

Viability

An understanding – even a misconception – is usually not universally useless. It may be viable for a particular purpose but non-viable in the general case. People may be entirely satisfied with their misconceived notions, even for a considerable amount of time. Nevertheless, the kinds of understandings of fundamentals reviewed here and in Appendix A are worrisome because they will cause problems for the novice programmer usually sooner rather than later. Those non-viable understandings need addressing; failure to do so is a failure of programming education.

The literature is effectively unanimous. Introductory programming education is struggling to cope with the challenging task of teaching beginners to create programs. Even teaching students to read programs is a challenge, as is helping students form viable understandings of fundamental programming concepts.