ON THE DIFFICULTY OF LEARNING TO PROGRAM

Tony Jenkins
School of Computing
University of Leeds
Leeds, UK.
tony@comp.leeds.ac.uk
http://www.comp.leeds.ac.uk/tony/

ABSTRACT

Few students find learning to program easy. This paper considers why this is so by examining what makes this most basic of skills so difficult to acquire.

There are many factors at work. Some are simply inherent in the subject while others have more to do with the modus operandi of teaching departments. Others are deeply interlinked with the expectations, attitudes, and previous experiences of the teaching staff and their students.

If computing educators are ever to truly develop a learning environment where all the students learn to program quickly and well, it is vital that an understanding of the difficulties and complexities faced by the students is developed.

At the moment the way in which programming is taught and learned is fundamentally broken.

Keywords

Programming, Aptitude, Learning Styles

1. INTRODUCTION

Few computing educators of any experience would argue that students find learning to program easy. Most teachers will be accustomed to the struggles of their first year students as they battle in vain to come to grips with this most basic of skills and many will have seen students in later years carefully choosing options so as to minimise the risk of being asked to undertake any programming.

This is a sad and depressing state of affairs. After all, computers are quite useless without programs and programmers to develop them. What is the point of teaching anything about computers to a student who is incapable of producing even the simplest of programs?

3rd Annual LTSN-ICS Conference, Loughborough University

Much of the existing research, in the computing education literature at least, focuses on new and interesting ways to *teach* programming. Papers describe how visual hooks and props can be used to engage an audience, or how programming can be taught almost by subterfuge through the medium of a game. These are all fine ideas, but there is seldom any sign of any concrete evidence that these ways of *teaching* have any impact on *learning*. It is true that students will enjoy them and that they will give good course feedback, but there is precious little evidence that they have learned anything that they would not have from a more traditional lecture.

Less attention has been paid to the subject of programming itself, and what has been written tends to lurk in rarely explored corners of the literatures of psychology and cognitive science. If students struggle to learn something, it follows that this thing is for some reason *difficult* to learn. If educators hope to teach it effectively, they must understand precisely what it is that makes learning to program so very difficult for so very many students.

Programming seems to occupy a particular place of prominence in the first year of degree-level computing courses. It seems to dominate the students' experience; those teaching other courses often complain that the students are spending all their time on programming. There is clearly something wrong (or there is some fundamental misunderstanding) if one part of the curriculum can dominate a whole course in this way.

2. THE QUESTION OF APTITUDE

It is sometimes argued that the students who find programming difficult are simply and solely those for whom programming *is* difficult. There is nothing inherently difficult in the subject; it is simply that some students have no *aptitude*. The skills often cited are problem solving and mathematical ability.

Evidence for the existence of aptitude in this sense is inconclusive at best. It is possible to find studies ([2], for example) that hint at some relationship between programming skills and experience in mathematics and it is possible to find studies that

conclude that there is no connection. An experiment at the University of Leeds designed to stream a programming class based on the results of an aptitude test [3], for example, showed no relationship at all between the final result in programming and measured "aptitude". Other studies [5] have shown that no demographic factor is a strong predictor of success in programming.

It certainly helps to have some experience of programming before starting a programming course [6], but this is not the same thing as aptitude. There exist various programming aptitude tests, but the evidence for their effectiveness is inconclusive at best [11].

If it is not possible to measure aptitude for programming in some convenient way, and if it is possible that "aptitude" for programming does not even exist, the focus for understanding the difficulty of learning to program must turn to a more cognitive view of the learning process.

3. COGNITIVE FACTORS

Two cognitive factors present themselves as possibilities that might make learning to program difficult – learning style and motivation. It is possible that there might be some particular learning style that will allow a student to acquire programming skill quickly and easily, or it might be that students require a particular form of motivation. If a student tends to adopt the wrong styles or has the wrong motivation, they will find learning to program difficult.

3.1 Learning Styles

Students prefer to learn in different ways. Some may regard learning as a solitary process and may learn best that way. Others may prefer a more dynamic learning environment and may learn best by discussions with their peers. Some subjects may demand a particular learning approach but, without guidance, students will tend to adopt the style they prefer or which has served them best in the past. It is a crucial responsibility of the teacher to ensure that the students adopt the most appropriate learning approach for the subject at hand.

The best-known classification of learning styles divides learning into "deep" and "surface" approaches [10]. Deep learners concentrate on gaining an understanding of a topic, while surface learners will concentrate on little more than memorising. It is easy to see how a particular method of teaching (and especially assessment) might drive a student to one of these approaches.

This classification seems to work well for a subject that is essentially a body of knowledge. History, for example, requires surface learning for lists of dates but deep learning for analysis and understanding. It is clear that the surface approach comes first and the knowledge acquired is then developed in deep learning. Programming is not like this. It is not a body of knowledge, it is a *skill*.

It might appear at first sight that deep learning is vital for programming, providing understanding that can be applied in new problem areas. However, it could equally be argued that programming can be learned as essentially a process that amounts to simple "pattern matching" where common problems are spotted and known working solutions applied. This approach sounds very much more like a form of surface learning.

It seems that the best strategy lies between these two extremes. Surface learning can be useful for remembering the details of syntax, or issues such as operator precedence, but elements of deep learning (and hence understanding) are required if a true competence is to be developed.

This is the key to the difference. The two learning styles must be applied *at the same time*. It is not sensible to memorise the rules of the syntax of some programming language and then to move on to apply it. This puts programming beyond the educational experience of most students; it requires a mixture of learning styles that most, if not all, of them have not had to apply before.

3.2 Motivation

Students approach computing degrees with a variety of motivations [7]. Some may have a genuine interest in the subject (intrinsic motivation); some may see their degree as little more than a means towards a lucrative career (extrinsic motivation); others may simply be trying to please their parents or family (social motivation).

The form of motivation does appear to be a factor. It has been shown that, perhaps not surprisingly, students who struggle in programming are more likely to have a primarily extrinsic motivation than their colleagues who excel [13]. Then again it has also been shown [8] that students generally maintain some form of motivation throughout their programming course, even if that comes to hinge on a negative factor such as fear of failure.

Programming students are motivated to succeed. They do not fail on purpose [8]. They will have to learn in a new way but that should not be a problem if their teacher appreciates and addresses this need. Perhaps the root of the problem lies in the subject. Is there something inherent in programming that makes it especially difficult to learn?

4. THE DIFFICULTY OF PROGRAMMING

"Programming" is a complicated business. An experienced programmer draws on many skills and much experience. Some of the skills required bear little obvious relevance to the process of producing program code.

Some of the required skills are obvious; problem solving ability and some idea of the mathematics underlying the process are essential. But there are more. A programmer must be able to use the computer effectively, must be able to create the program in a file, compile it, and find the output. The program produced must be tested, and bugs found and corrected. These are easy skills to identify, and presumably they are addressed in most programming courses.

There are less obvious skills. These might be classed as "life skills". Programming is normally taught as a fundamental subject at the start of a degree course. This is a difficult time for many students – a time of *transition* as they adapt to life and study at university. They may well be living away from home for the first time, they may struggle to make new friends and find their feet in a new environment, and they may struggle to come to terms with managing their own finances and their own private and study time.

In the midst of this they will be encountering some of the most basic material in their programming course. This is potentially quite challenging material that is going to form the basis of the rest of their learning. They will be lost if they do not understand this. This is difficult enough material to master when a student is well settled, but departments' insistence on teaching this during a period of transition can only increase the difficulty.

4.1 Multiple Skills

Programming, then, is not a single skill. It is also not a simple set of skills; the skills form a hierarchy [14], and a programmer will be using many of them at any point in time. A student faced with learning a hierarchy of skills will generally learn the lower level skills first, and will then progress upwards [1]. In the case of coding (one small part of the skill of programming) this implies that students will learn the basics of syntax first and then gradually move on to semantics, structure, and finally style.

Teachers will be all too familiar with the student who produces programs with no indentation, intending to "indent it all later", or without any comments, content to add these later (and only then because there are marks for the comments in the assessment). No experienced programmer would work in this way, and these are bad habits to fall into, but this is an inevitable side effect of the order in which programming skills are learned.

This approach to learning is often reinforced by lectures that concentrate on the minutiae of syntax, and by textbooks that adopt much the same approach. This leads to the student who hopes to gain an understanding of programming and plans to achieve this by reading a textbook. Programming is learned by programming, not from books.

4.2 Multiple Processes

Programming is not only more than a single skill; it also involves more than one distinct process. At the simplest level the specification must be translated into an algorithm, which is then translated into program code. In experienced programmers it is also possible to identify an intermediate process whereby the algorithm is mapped to something resembling a "recipe" for the programme, based on previous experience [9].

The most difficult part of the process is the first, translating the specification into the algorithm. This is also the most important, as it is crucial that a correct and efficient algorithm is used as the basis of any coding. Given a correct algorithm the other processes are essentially mechanical.

Therefore, a student must master three distinct processes. Teaching (and learning), however, can concentrate on the low level issues of syntax at the expense of the higher level, more complex, process of designing an algorithm. Worse, any consideration of algorithm design and efficiency can be relegated to another, apparently unrelated, part of the course! In any case there is surely little point in lecturing students on syntax when they have no idea of where and how to apply it.

Teachers will be familiar with students who can follow the lectures in the programming course, who can dissect and understand programs, but who are totally incapable of writing their own program. They have not mastered all the processes; they can code, but they cannot produce an algorithm.

4.3 The Language

Much has been written about the best language to use for teaching (or learning), and some might argue that too much has been written. There is scant solid evidence that any language is any better or any worse than any other, and the choice continues to be driven largely by the "flavour of the month" in industry.

Most teachers would agree that the purpose of an introductory programming course is to teach the students to *program*; the intention is not to, for example, "teach them Java". The language used is

no more than a vehicle whereby the main objective is to be achieved.

It is hard for students to make this separation. While they grapple with the idiosyncrasies of whatever language they are being taught, it is very difficult to think about higher level abstract concepts. It is only when a programmer has had experience of more than one language that these concepts actually become apparent. And most introductory programming courses teach only one language.

The languages used for teaching were not designed for teaching. There exist languages designed for teaching (Pascal, LOGO), but any department using one of these today would be an object of ridicule. Many would worry about the effect on recruitment of not teaching a language that was used in industry.

Languages designed for serious use by serious programmers are hardly suitable for the novice. As a trivial example, consider the everyday notion of "or", which also has a meaning in programming [15]. A novice programmer faced with the task of programming a conditional statement "*if the answer is y or n*" will be familiar with the everyday linguistic meaning of "or", and may well code (in C):

```
if (answer == 'y' || 'n')
```

This makes perfect sense, especially when read aloud, but is clearly semantic nonsense. Worse, this will compile and appear to work in some cases!

The language used for teaching should be designed for that purpose.

4.4 Educational Novelty

Programming is a new subject for many of the students who take programming courses. In his classic article on teaching programming (which should be required reading for all who teach programming) Dijkstra argues that learning is a slow and gradual process of transforming the "novel into the familiar" [4]. He goes on to suggest that programming is what he terms a "radical novelty" in which this comfortable tried and tested learning system no longer works. The crux of the problem is, according to Dijkstra, that radical novelties are so "disturbing" that "they tend to be suppressed or ignored to the extent that even the possibility of their existence … is more often denied than admitted".

A particular feature of programming (and one that reinforces Dijkstra's message) is that it is "problem-solving intensive" [12] – it requires a significant amount of effort in several skill areas for often a very modest return. At the same time it is "precision intensive" [12] – the modest success that can be achieved by a novice programmer requires a very high level of precision, and certainly a much higher level than most other academic subjects. Dijkstra also notes that the "smallest possible perturbation" in a program of one single bit can render a program totally worthless. This is precision indeed.

Students who start a programming course in higher education have come from a familiar academic setting. There they were studying topics with which they were, on the whole, comfortable and familiar, and which they had been studying for some years. They were probably used to performing well academically and had developed a set of tried, tested and trusted learning and study skills. To arrive in a setting where they are confronted with a totally new topic that does not respond to their habitual study approaches, and where a single semi-colon is the difference between glorious success and ignominious failure, must surely represent a "radical novelty" in Dijkstra's terms. It is perhaps not study skills that are needed, but coping skills. These are found in few programming syllabuses.

4.5 Interest

Learning (or perhaps here "being taught") programming can be very dull. Lectures covering the details of syntax are never going to be especially inspiring, and exercises that involve simple mathematical manipulations of collections of student marks, stock levels, baseball statistics, or bank account details are never going to set the pulse racing. Yet a glance in many programming texts will yield many turgid examples of each of these.

At its best programming can be an enjoyable, creative activity, and many students derive great enjoyment from their programming. They enjoy it even more (and learn more) when they are allowed to work on assignments that inspire them. It is a shame that so few assignments do indeed inspire.

4.6 Reputation and Image

Programming courses acquire the reputation of being difficult. This view is passed to the new students by their predecessors, and is exaggerated in the telling. This perhaps makes it acceptable, even expected, that a student will struggle.

At the same time, there is the public image of a "programmer". This is of a socially inadequate "nerd", spending all hours producing arcane and unintelligible code, fuelled by pizza and caffeine. It is hard to imagine students aspiring to this image.

If students approach a course with an expectation that it will be difficult, and with a negative image of those who excel in the subject, it is very hard to imagine their being especially motivated. And students who are not motivated will not succeed.

4.7 Pace

In a university programming is taught, and therefore learned, to a set timescale. It maters not whether this is one or two semesters or even a number of years. The fact remains that at some point the programming course will end and the students who pass will be "able to program". This means that the pace of the instruction is not under the students' control (and it is more than likely that different students in the class will learn at different paces).

This will lead to the student who has missed a basic concept and who then cannot follow the next lecture. There is no going back; the course is behaving very much as a high-speed train with no brakes. Such a student will quickly come to the view that "they just can't do programming" (the educationalists call this *learned helplessness*), and will attribute this to the perceived difficulty of the subject.

The pace of the course is often driven mostly, of course, by the needs of assessment. This scheme may be reasonable for many subjects, but it is quite ridiculous for learning a skill such as programming.

5. CONCLUSIONS

Programming is certainly a complicated skill to master, and learning to program is correspondingly complex. There are many features of the skill that contribute to this complexity, and this paper has described some of the more important. Some of these issues centre on the nature of the programming skill, while others have more to do with the ways in which the participants teach and learn. It must be possible to overcome these obstacles. It must be possible to learn to program; if it were not there would be no programmers. But how many programmers learned to program solely from a course in higher education? Few indeed.

I talk to many students about their programming course. The most common comment is that programming is "boring and difficult". When I ask whether it is boring because it is difficult or difficult because it is boring, they are seldom sure. But they remain adamant that it is both. Teaching a subject that is boring and difficult is a tricky task indeed. The essential problem is that programming represents an "educational novelty". It represents this for those who teach as much as for those who learn. It is clear that the students' tried and tested learning styles do not work when applied to programming. It is equally obvious that the teachers' tried and tested teaching fair little better.

So, why teach programming in this way? Presumably because that is the way it's always been done. If this no longer works, something will have to change. Learning to program is indeed difficult, but it should not be as difficult as it currently appears.

What should change?

- Programming should never be taught before the second year of any course;
- The language used should be chosen for pedagogic suitability and not because it is popular in industry;
- Programming should be taught by those who can teach programming and not those who can program.
- Programming courses should be designed to be flexible to allow different students to learn in different ways;
- There should be no summative (continuous) assessment to ease pressure on students.
- Departments should acknowledge that programming is difficult and supply adequate support to students.

That would be a decent start.

6. REFERENCES

[1] C. Bereiter and E. Ng. *Three Levels of Goal Orientation in Learning.* Journal of the Learning Sciences, Vol. 1, pp 243-271, 1991.

[2] Pat Byrne and Gerry Lyons. *The Effect of Student Attributes on Success in Programming.* Proceedings of ITiCSE 2001, pp 49-52, 2001.

[3] John Davy and Tony Jenkins. *Research-Led Innovation in Teaching and Learning Programming.* Proceedings of ITiCSE '99, pp 5-8, 1999.

[4] Edsger W. Dijkstra. *On the Cruelty of Really Teaching Computing Science.* Comm. ACM, Vol.32, pp 1398-1404, 1989.

[5] Gerald E. Evans and Mark G. Simkin. *What Best Predicts Computer Proficiency?* Comm. ACM, Vol. 32, pp 1322-1327, 1989.

[6] Dianne Hagan and Selby Markham. *Does it help to have some programming experience before beginning a computing degree programme?* Proceedings of ITiCSE 2000, pp 25-28, 2000.

[7] Tony Jenkins. *The Motivation of Students of Programming.* Proceedings of ITiCSE 2001, pp 53-56, 2001.

[8] Tony Jenkins. *The Motivation of Students of Programming.* MSc Thesis, University of Kent at Canterbury, 2001.

[9] Katherine McKeithen, Judith S. Reitman, Henry H. Reuter and Stephen C. Hirtle. *Knowledge Organisation and Skill Differences in Computer*

Programmers. Cognitive Psychology, Vol. 13, pp 307-325, 1981.

[10] F. Marton and R. Säljö. *On Qualitative Differences in Learning I: Outcome and Process.* British Journal of Educational Psychology, Vol. 46, pp 4-11, 1976.

[11] Lawrence J. Mazlack. *Identifying Potential to Acquire Programming Skill.* Comm. ACM, Vol. 23, pp 14-17, 1980.

[12] D. N. Perkins, Steve Schwartz and Rebecca Simmons. *Instructional Strategies for the Problems of Novice Programmers.* In R. E. Mayer (ed), "Teaching and Learning Computer Programming", pp 153-178, Lawrence Erlbaum Associates, 1988.

[13] Julie Sheard and Dianne Hagan. *Our Failing Students: A Study of a Repeat Group.* In Proceedings of ITiCSE 98, pp 223-227, 1998.

[14] Kathryn D. Sloane and Marcia C. Linn. *Instructional Conditions in Pascal Programming Classes.* In R. E. Mayer (ed), "Teaching and Learning Computer Programming", pp 207-235, Lawrence Erlbaum Associates, 1988.

[15] James C. Spohrer and Elliot Soloway. *Novice Mistakes: Are the Folk Wisdoms Correct?* In E. Soloway and J. C. Spohrer (eds), "Studying the Novice Programmer", pp 401-416, Lawrence Erlbaum Associates, 1989.