On the teaching of computer programming to adults

John Lenarcic
RMIT University, Melbourne, Australia
John.Lenarcic@rmit.edu.au

***Abstract**: Reflections on the problems faced in teaching novice computer programmers are presented in an informal, stream-of-consciousness manner, based on field experience and folk wisdom acquired with RMIT Technical and Further Education (TAFE) students. Questions of a diverse nature are raised on research strategies to pursue for pedagogic innovations in this area.*

***Keywords**: teaching innovations, novice computer programmers*

An IT academic once remarked to me, in jest, that computer programming is as boring as "bat shit". Programming may appear to be a tedious activity to the spectator, but why is this so? The very act of writing a computer program (or program code) is a task that involves reframing an often ill-defined problem as a system of interlocking text-based components consisting entirely of sequence, selection and repetition statements. (Sequence statements are command-like sentences that initiate one action after another in order. Selection statements are also command-like sentences that are used for making a choice between alternative actions. Finally, repetition statements are - you guessed it - command-like sentences that are used for performing some set of actions over and over, usually until some condition is encountered that forces a stop to it.) Repetition is an essential part of any piece of software in the making. Needless to say that "repetitive" is a synonym for "monotonous" which in turn also means "boring"! Facetious logic aside, perhaps the reason for the dreary reputation of programming as a "nerdy" pastime is due to how it is currently taught to adult students at universities and colleges.

Soloway (1986) states that textbooks used in software development courses for novices focus on the syntax and semantics of constructs in a programming language. After 17 years nothing has really changed, with the syntax and semantics approach still being the way programming is taught at most institutions today. The syntax of a computer language is the set of structural patterns that individual tokens of the language must adopt to form valid statements in a program. Semantics deals with the meaning of these component statements and the program as a whole. (I usually describe the distinction between syntax and semantics to my students by first writing Noam Chomsky's famous sentence "*Colourless green ideas sleep furiously*" on the whiteboard (Chomsky, 1957). This sentence appears to sound right in that we can tell that it's a properly formed sentence. Adjectives are used OK. Nouns and verbs are placed in the right order. This is syntax at play. But the whole sentence makes no logical sense and has no meaning. That's semantics or a lack of it!)

To attain computer literacy, students of programming are shown the meaning of the syntactic components of a computer language and how they are individually used in very simple examples (in a manner similar to that of a phrase book for travellers.) They are then given relatively straightforward practical exercises to undertake so that their newly acquired knowledge of syntax can be put into practice. The very first program that students are taught to write is one that simply displays the sentence “Hello World.” on screen. By convention, this is generally the first program that most novice programmers write, regardless of what the computer language may be. Guzdial and Soloway (2002) maintain that this opening approach is symptomatic of the outdated view of computing and students that many IT educators have. In an age of affordable multimedia computing for the masses, it is no wonder that students find it difficult to be inspired by merely displaying a line of text. Many students today are part of the “Nintendo” and “MTV” generation of audiovisual aficionados and this is a possible contributing factor to IT education being dubbed by some to be “tedious and dull” (AAUW, 2001).

Guzdial and Soloway (2002) advocate a “multimedia-first” approach to the teaching of computer programming. In other words, inspire the students by getting them to play with sounds and simple animations. This of course assumes that students are of sufficient technical sophistication in the first place. Novices may be able to grasp writing a “Hello World” program because of its sheer simplicity but going beyond this level is another story. It is here that most students fall over because they can’t put the previously learned pieces of syntactic theory together into one program whole. Learning to program is like learning to ride a bicycle, I often tell my students. I can show them the mechanics of the theory in class but only the students, on their own, can be in control of how soon they can ride and not fall over.

Lots of practical experience is involved in the path from novice to expert programmer. The prevailing philosophy of most IT educators that I know is that the best way to learn how to write code is to write code. As Thomas Edison said: *“Genius is 1% inspiration and 99% perspiration.”* Programming students are usually required to submit several programming assignments for assessment during the course of a semester. Once again, these are meant to gauge a student’s ability to comprehend the theory and apply it in a practical context. This is the way computer programming is taught at most academic institutions at post-secondary level today and it has probably been carried out in this fashion since the dawn of IT.

The learning of software development mainly occurs in a computer laboratory environment with PCs on benches in fixed positions facing a whiteboard and projection screen at the front of the room. The décor is Spartan and not at all aesthetically pleasing to say the least. The isolation enforced by the individual workstations doesn’t facilitate context-based learning. Situated cognition encompasses the latter approach in that learning is considered to be primarily social in nature (Hansman, 2001). Communities of shared practice facilitate both the incubation and transfer of knowledge. Sheard and Hagan (1999) outline the design of a new learning environment to assist weak introductory programming students at tertiary level. The “environment” discussed is the style of delivery not the actual physical surroundings of learning, which presumably are immutable for technical reasons. Procedures for assessment, assignment work, tutorial classes, group exercises and lectures are summarised. In the latter, role-playing activities are included to sustain interest in the proceedings.

To exploit the benefits of context-based learning, it would perhaps be a better idea to experiment with “pair programming” in a lab environment (Williams and Kessler, 2000). This is a practice in which two programmers work side-by-side at one computer, constantly collaborating on the same piece of work. The technique is primarily aimed at professionals, who claim significant increases in productivity and quality of software products after its acceptance. It could be adopted in an educational context but it might also be seen to raise the risk of plagiarism even more so the technique could prove to be politically unsound at many academic institutions.

In the TAFE sector within the RMIT School of Business Information Technology, an introductory programming course generally consists of a one hour theory lecture per week and four hours of practical work in a computer laboratory, spanning a 15-week academic semester. It must be said that teaching programming using a lecture format isn't the ideal approach. Most students are bored to tears by lectures that dwell on technical minutiae, such as where to place a semicolon in a computer language statement. But to master programming one must have the patience and fortitude to tame the proverbial "devil in the detail". More learning takes place in the labs where students engage in practical activities and the instructor acts as a mentor, almost in a "master-apprentice" relationship. One of the problems with computer programming is that it has almost always been in an identity crisis, much like the discipline of computer science itself (Nwana, 1997). Is it a science or an art or a craft or a skill?

No one has yet provided a definitive answer.

Computer programming is an adult activity, if not by definition then by practice. As Perlis (1982) notes with tongue-in-cheek: *"Perhaps if we wrote programs from childhood on, as adults we'd be able to read them."* The uninitiated may cling to urban myths that children or young teenagers can become adept at the skill but that is primarily due to sensationalist reporting by the media over-inflating the prowess of fledgling hackers, who often perpetrate their acts using a "recipe-based" approach. No, the kind of programming that I am referring is an offshoot of general problem solving from first principles, one that requires the representation of some limited domain of reality with meticulous precision and attention to detail. One has to be able to closely analyse a real-world problem, understand it so as to make explicit that which was implicit, and then translate all of this into a language that a "dumb" computer can comprehend. The computer is merely an external cognitive tool that amplifies the abilities of the person that programs it. So, if you put garbage in, you can only ever expect garbage out. Perlis (1982) states: *"You think you know when you learn, are more sure when you can write, even more when you can teach, but certain when you can program."* This feeling of certainty is a hallmark of the skilled programmer, even though it is generally accepted that error-free software is the mythical exception rather than the norm.

Computing programming is often dubbed a very difficult activity in the literature (e.g., Pane, et al, 2001). To quote Perlis (1982) once again: *"Most people find the concept of programming obvious, but the doing impossible."* Most consumers would appreciate the idea of programming a VCR to tape a TV show but anecdotal evidence would suggest that actually doing it is unachievable by the masses. Otherwise, what else would explain the near ubiquitous "flashing display" on most VCRs in service or the invention of G-code? And the programming of a VCR is vastly simpler than programming in the C++ computer language, say. Some of the difficulty in learning how to program a computer is acknowledged as being inherent to the skill itself. However, part of this complexity could be due either to the poor design of languages or to the fact that it is not taught in the right way (Pane, et al, 2001).

Dijkstra (1989) laments at the use of comfortable metaphors and mundane analogies to teach programming, frowning upon the continued description of the new with yesterday's vocabulary. As Dijkstra (1989) contends: *"Coming to grips with a radical novelty amounts to creating and learning a new foreign language that cannot be translated into one's own* *mother tongue.”* He believes that students should be taught the joy of rigorous thinking by being shown the beauty of mathematics. Formal methods derived from mathematics could serve as a *lingua franca* to facilitate the teaching of programming in an optimal manner. By developing the intellectual stamina to face uncomfortable truths, the novice can then begin to tame the complexity that is computer programming. It's the “castor oil” approach to education: This medicine is good for you; so it tastes bad but given time you might get used to it.

Devlin (2001) is also of the opinion that mathematics is important for budding software engineers. Abstraction is difficult for the human brain to cope with and this is what software development is fundamentally all about. As a species we evolved primarily to interact with the concrete structures of our physical environments not the virtual ones exemplified by computer programs. Mathematical thinking reinforces repetitive learning of abstractions. Many TAFE students, mature-age or otherwise, have little or no training in higher-level mathematics. Indeed, the students with no mathematical background generally exhibit the most difficulty with computer programming. Monroe and Orme (2002) provide some guidance on how to expand the mathematical vocabulary of students; however their advice is for primary school teachers. What should probably be a prerequisite for the novice programmer is some exposure to advanced mathematics beyond basic arithmetic, such as a palatable introduction to discrete mathematics, but this would be a syllabus policy decision outside of the authority of teachers in the trenches.

Soloway (1986) writes that research of the time indicates that computer language constructs do not pose major obstacles for novice programmers. The real problem is that learners don't know how to put the pieces of the jigsaw together in composing and coordinating components of a program. They may understand fragments of program code on their own but have enormous difficulty assembling these parts into a working whole. Amazingly this is the same remark that I often get from adult students today! The focus on instruction of the syntax and semantics of programming language constructs is wrong according to Soloway (1986), as it promotes an undue emphasis on the finished program as the final result of the whole process. A program is a set of instructions that transforms a computer into a mechanism that controls how a real-world problem can be solved. But a human being – the programmer – needs to have an explanation as to why the program solves the particular problem.

According to Soloway (1986), learning to program should be viewed as learning how to put together mechanisms and how to compose explanations. Accentuating the theoretical content of an introductory course and making the underlying abstractions of programming explicit, in addition to covering the rules of programming discourse, can achieve this. In other words, students should be shown what programming has in common with other problem solving tasks. It should be stressed to novices that programming is a design discipline with the output of the process being an artefact that performs some desired function (i.e. a “mechanism”). The trail of information in creating this artefact is an “explanation”. It's a new philosophy for interpreting the act of programming. Of course, I can think of no contemporary introductory course or textbook that currently adopts this pedagogic strategy. Once again, the reasons are probably political.

Eliot Soloway is one of the pioneers of “software psychology”, a neglected field of IT that was partially inspired by the ideas espoused in Gerald Weinberg's landmark 1971 book “The Psychology of Computer Programming” (Weinberg, 1971). This text was one of the first to deal with programming as a human cognitive activity. In fact, it's probably one of the only

existing books still in print that does so, as most texts tend to dwell excessively on the technical aspects of programming. In the early 90's, during a stint as a Lecturer in Software Development at Monash University, I was motivated by Weinberg's book to develop a dedicated postgraduate course in this vein. Except it
was not called "Software Psychology" because that would have raised the ire of the Psychology academics. Rather, it was given the more innocuous title of "Behavioural Issues in Software Development". Arguably the first and last course of its type in Australia, it was too introspective in a psychological sense for the powers-that-be who championed courses that dealt with the latest technical fads of the time, and it died an unceremonious death after only one semester. Without postgraduate courses such as this, university IT departments cannot hope to persuade students to do research in a similar area. And without a critical mass of research students in software psychology one cannot hope to expect findings that could eventually make life easier one day for the rank-and-file teacher of programming.

How can the teaching of programming be improved? I believe that one has to look at computer languages from a fresh, new perspective before anything else can be done. In April 2002, I gave a presentation at the 6th Conference of the Australasian Cognitive Science Society entitled "*Cognitive Dynamics of Programming Languages.*" Are computer languages "tools" akin to the user interface of a machine or are they artificial dialects with all or some of their inherent linguistic properties? My talk addressed the issue of whether the acquisition of computer languages actually changed the way people could think.

"*Programming is the new Latin*" was the slogan that many an early computing teacher espoused, according to diSessa (2001), but such a notion also lead to an "antiprogramming" backlash (e.g., Pea and Kurland, 1984). It was still unclear as to whether learning to program made people more logical and powerful thinkers, as it was once believed that the learning of Latin would do. However, the many arguments that went to and fro ignored the work of amateur linguist Benjamin Lee Whorf…

The concept of linguistic relativity (also know as the Sapir-Whorf hypothesis) suggests that natural languages influence the way their speakers think (Whorf, 1956). It could be argued that programming languages share more than just metaphoric links with natural languages. For example, both are constructs dictated by the frameworks of syntax and semantics, albeit a computer language is devoid of speech and exists only as a form of writing. Could this be a reason for why learning to programming is so often dubbed a difficult task?

Perhaps adults find it so hard to learn their first programming language because it is more like a natural language than most computer scientists would care to admit. It has been taught like it was a physical tool to master when the mode of instruction should have been similar to that required to gain fluency in a second tongue. Gaining competency in a second natural language as an adult learner has always been deemed to be challenging. But at least the subject matter is considered from a linguistic angle for the pedagogic approaches involved in language learning. I propose that we should teach programming languages as if they were a second natural language to be acquired. The first step should be to teach students to read before they can write. Remember that computer languages have no analogue to speech so novices can't learn how to talk first. Their goal is to become fluent in the composition of complex programs, something vaguely similar to writing a novel. Now, one would not aim to write a great novel until one has at least read a few. Same idea here: read good program code first, identify the bad stuff and then go on to do the actual creative writing.

Zeller (2000) advocates the adoption of an automated system to allow students to read, review and assess each other's programs in order to improve quality and style. Of course, this presupposes that students have learned to write code first. To encourage the reading of code, I would like to see the development of computer program "literature", a library-based resource of good and bad examples that exists solely for critical analysis by novice and expert alike. Knuth (1992) outlines the technical details of what the paradigm of "literate programming" would entail. Basically, it would involve the development of a technological infrastructure that would allow one to curl up in a chair by an open fireplace while reading a good computer program. This has yet to be convincingly realised in the practical sense.

In what other way can software be treated as literature? Book groups are a relatively recent phenomenon. In these gatherings interested parties discuss the merits or otherwise of a particular novel. Hagan and Sheard (1998) discuss the value of discussion classes for teaching introductory programming. Preliminary findings indicate that such classes, which are held in rooms without computers, lead to an improvement in student results. The tutor's responsibility in such a class is to incite debate about programming concepts rather than simply spoon-feeding answers. Once again, the clientele in the situation described are tertiary students.

Postgraduate courses in education are far too generic in their subject matter for specialist practitioners such as IT academics. Teachers in different disciplines face different, unique problems. One standard set of pedagogic theories can't possibly fit all situations. Programming teachers would benefit immensely from undertaking a graduate diploma in education that actually focused in part on strategies derived from software psychology meshed with contemporary pedagogic theories. This could be achieved by offering an elective via team teaching in a generic diploma: one member from an education faculty and the other from an IT faculty. The latter individual would have to be well versed in software psychology as well as the nuts-and-bolts of computer programming. Indeed, an interdisciplinary research venture involving academics from IT, education and psychology may be the best approach to demystify the art of computer programming for everyone.

References

AAUW (2001). *Tech-savvy: Educating girls in the new computer age*. New York: American Association of University Women Education Foundation.

Chomsky, N. (1957). *Syntactic structures*. The Hague: Mouton.

Devlin, K. (2001). The real reason why software engineers need math. *Communications of the ACM, 44*(10), 21-22.

Dijkstra, E.(1989). On the cruelty of really teaching computing science. *Communications of the ACM, 32*(12), 1398-1414.

diSessa, A. A. (2001). *Changing minds: Computers, learning and literacy,* Cambridge, MA: MIT Press.

Guzdial, M. and Soloway, E. (2002). Teaching the Nintendo generation to program. *Communications of the ACM, 45*(4), 17-21.

Hagan, D. & Sheard, J. (1998). The value of discussion classes for teaching introductory programming. *Proceedings of the 3rd annual conference on Integrating Technology into Computer Science Education* (pp. 108-111). Dublin City University, Ireland.

Hansman, C. A. (2001). Context-based adult learning. *New Directions for Adult and Continuing Education,* No. 89, 43-51.

Knuth, D. E. (1992). *Literate programming*. CLSI Lecture Notes No. 27, California: Stanford University.

Monroe, E. E. & Orme, M. P. (2002). Developing mathematical vocabulary. *Preventing School Failure, 46*(3), 139-142.

Nwana, H. S. (1997). Is computer science education in crisis? *ACM Computing Surveys, 29*(4), 322-324.

Pane, J.F, Chotirat, A.R. & Myers, B.A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies, 54*, 237-264.

Pea, R. & Kurland, M. (1984). One the cognitive effects of learning computer programming. *New Ideas in Psychology, 2*, 1137-1168.

Perlis, A. J. (1982). Epigrams on programming. *ACM SIGPLAN Notices, 17*(9), 7-13.

Sheard, J. & Hagan, D. (1999). A special learning environment for repeat students. *Proceedings of the 4th annual SIGCSE/SIGCUE conference on Integrating Technology into Computer Science Education* (pp. 56-59). Cracow, Poland.

Soloway, E. (1986). Learning to program = Learning to construct mechanisms and explanations. *Communications of the ACM, 29*(9), 850-858.

Weinberg, G. M. (1971). *The psychology of computer programming*. New York: Van Nostrand Reinhold.

Williams, L. A. & Kessler, R. R. (2000). All I really need to know about pair programming I learned in kindergarten. *Communications of the ACM, 43*(5), 108-114.

Whorf, B. L. (1956). *Language, thought and reality*. (J. B. Carroll, ed.), Cambridge, MA: MIT Press.

Zeller, A. (2000). Making students read and review code. *Proceedings of the 5th annual SIGCSE/SIGCUE conference on Innovation and Technology in Computer Science Education* (pp. 89-92). Helsinki, Finland.