I had the opportunity to spend the latter half of 2003 at Auckland, New Zealand's UNITEC Institute of Technology as a visiting lecturer. As a dyed-in-the-wool computer scientist it was deemed best if I lectured in two of UNITEC's programming courses. The experience was both intellectually rewarding and scenically beautiful.

What struck me most was how much UNITEC's Information Technology (IT) curriculum, as well as all the other IT curricula I have been involved with, focus on the minutiae of programming and the mastery of syntax. True, some IT students are interested in a career in programming (and who might instead be better served by a computer science curriculum), the vast majority of those enrolled in IT programs will not be earning their bread and butter by doing traditional computer programming for a living.

Not to say studying IT is not hard -- it is, especially given its multi- and interdisciplinary nature-- but so is learning computer programming. A good tertiary program in computer science turns out, at best, decent novice programmers who also have a rudimentary grasp of the discipline's vocabulary and theory. If it takes 3-4 years to turn out decent novice programmers from a program whose primary focus is programming, then what are the goals of the programming component in an IT curriculum?

While I believe that discovering how much computer science/programming is needed by today's IT professionals is a great open research question, I nonetheless will put forward my own, highly biased, views on the subject.

There are some who subscribe to the notion that because of the intellectual rigor it fosters, everyone, regardless of discipline, should learn to program. Of course, given the realities of covering disciplinary content in a 3-4 year time frame, this is simply impossible. This is as true for philosophy as it is for IT. Unfortunately, this observation simply begs the question.

Maybe it is instructive to initially focus on the unreasonable boundaries so as to bring the center into focus. As described above it is simply unreasonable for tertiary IT programs to produce good entry-level IT professionals who are also good computer programmers; without doubling the length of the program of study. Similarly, an IT program with no exposure to programming feels like a medical school curriculum that does not cover organic chemistry.

Given the conundrum that programming must be covered, but that decent mastery is beyond their reach, many IT curricula opt for syntax mastery. It is true that syntax mastery is a subset of competent programming. (It is also true that some IT curricula delve deeper into the --uninteresting-- minutiae of syntax than traditional computer science curricula.) Unfortunately, to me, syntax mastery is to competent programming as learning how to focus a telescope is to astronomy. Yes, astronomers need to know
how to focus a telescope, but more importantly they need to ascertain what is interesting/worthwhile to look at and then where to find it in the sky. Yes, computer programmers need to know when to use a "do" loop instead of a "while" loop (in C/C++/Java), but more importantly they need to understand how such a construct at a given location in the program contributes to the successful implementation of an algorithm designed to solve an interesting/worthwhile problem.

A pitfall of the syntax mastery approach is that leaves one with IT professionals who can recite the differences between C-style strings and C++ style strings, but who unknowingly may request a time efficient solution to the Travelling Salesman problem from the programming staff. (A little over twenty years ago I was asked to produce just such a program for a company that produced school bus pickup/delivery routes. I still humorously recall how I, newly entered in industry with my degrees in Business Administration (Finance) and Mathematics/Computer Science in hand, endeavored to educate my completely computer/IT illiterate managers on the impossibility of the task. Insistent on a timely deterministic solution, they were stubbornly unsatisfied with the reasonable approximation I could offer; after all, they believed computers could deterministically solve all "mathematically" posed questions in a reasonable amount of time. My dilemma was worsened by my employers recent purchase of an expensive DEC TOPS-20 computer that was to run their desired program.)

No, I am not advocating that IT curricula embrace the study of Formal Languages and Automata Theory. I am advocating that the programming component of an IT curricula cover the limits of computing. While I do not need our society's IT professionals to be able to do a Turing reduction proof that a given problem is NP-Complete, they should fully understand what an algorithm is, what kinds of problems can be solved algorithmically in a reasonable amount of time, what kinds of problems can be solved algorithmically, but require too much time and therefore need an approximation heuristic, and which kinds of problems cannot be solved algorithmically at all.

Central to the study of the limits of computing is the notion of an algorithm. Central to the study of algorithms is the notion of problem solving. After all a C/C++/Java (or fill in your computer programming language of choice) program is simply the expression of an algorithm, or problem solution, in an unambiguous, mathematically defined formal language. Syntax mastery is useless without algorithmic problem-solving maturity.

One may generously infer that those who advocate the syntax mastery approach in IT curricula are assuming that their students already posses an appropriate level of algorithmic problem-solving maturity. My own observations, made over the years at different institutions and again corroborated at UNITEC contradict this. IT students who had gained significant syntax mastery could not organize a set of program statements to correctly solve simple problems. Nor could they articulate a solution to the same problem when asked to describe their solution in English. These students lacked basic algorithmic problem-solving maturity. Interestingly, traditional computer science curricula are evolving to include more time for building algorithmic problem-solving maturity independent of "programming."

So, if I "ran the zoo" I would focus the baseline programming component of an IT curriculum on three goals:

- The building of basic algorithmic problem-solving skills.
- The understanding of the limits of computing.
- The development of enough syntax mastery to allow for the successful implementation of simple programs, and more importantly for a through understand of software development; i.e. basic programming maturity.
- (Optionally) The development of intermediate programming skills.

This approach may be summed up as a call to substitute current textbook choices such as "C++ in 24 Hours" for Henry Walker's "Limits of Computing" and David Harel's "Algorithmics."

Is syntax mastery difficult? I do not think so, but my former UNITEC students would beg to differ. Is developing programming skills difficult? Unlike with syntax mastery, I would say yes. Is the development of algorithmic problem-solving skills difficult? To my
thinking, this is the hardest of the three. Yet without algorithmic problem-skills one would not know what to program (programming skills) or how to syntactically express it (syntax mastery). Clearly too great a focus on syntax mastery is not enough for the development of basic algorithmic problem-solving skills.