A New Perspective for Entry-level Computer Courses

Roberto Ierusalimschy
Departamento de Informática, PUC-Rio
22453-900, Rio de Janeiro, Brazil
roberto@inf.puc-rio.br

Abstract – *Most entry-level computer courses today have a strong* informative *(or instructive) nature, training the student in building small programs or in using some applications. In this text we argue for a more* formative *approach that, while keeping the distinctive hands-on character of this kind of course, emphasizes the basic concepts underlying programming:* abstraction, recursion, *and* structuring.

The Past

The introduction of the first computer courses in the Engineering curriculum, more than thirty years ago, had a clear goal: To enable the future engineer to write her own programs for numerical solution of mathematical models.

At that time, computers where clearly divided between scientific machines, with strong arithmetic performance, programmed mainly in FORTRAN, and commercial machines, with emphasis on string manipulation, programmed mainly in COBOL. FORTRAN was the only language an engineer could possibly use, and the solution of mathematical models was the only useful task to expect from a scientific computer. Programs at that time were small for today's standards: A program with a thousand cards (that is, one thousand lines), which filled a whole card box, was considered quite large. In that context, the stated goal of a computer course could be achieved in one or two terms. Typical students finished the courses with a working knowledge of FORTRAN, and were able to write useful programs for the specific domain of numerical computations.

We do not need to emphasize how much computer science and computer technology have changed since that time. Currently, few engineers need to write their own programs to solve numerical problems; instead, they can use off-the-shelf software, such as MatLab [6] (which may need another kind of programming). Moreover, due to the growing complexity of programs, user interfaces, and programming environments, few engineers have the necessary skills to write fair programs in C. A program with a thousand lines of code is currently considered small. Moreover, if the program uses a graphical user interface, the programmer will need to work with object-oriented programming, call-backs, and other concepts not included in the basic instructional kit.

On the other hand, most engineers use computers for many other activities, using many different tools, such as spreadsheets, text editors, and CAD programs. Computers permeate many office tasks today, and certainly this spectrum will grow further in the future.

The Present

Although some universities still follow the old format for their computer courses, most of them have adopted other syllabuses for their entry-level courses in computer science.

One trend has emphasized an overview of Computer Science, to give the student a broad knowledge of the field. An example of this trend is the course *SEM117: Introduction to Computer Science* [13], from the University of Birmingham.

Another line has focused in "computer literacy": Because engineers have to use different tools, they should be trained in the use of those tools. Civil Engineering at Purdue University adopts this kind of course (*CE 293: Computers and Computer Programming for Civil Engineers* [9]) with topics such as WordPerfect, QuattroPro and AutoCAD.

Finally, many curricula have kept an emphasis in programming skills, usually adopting a more modern language, such as C. For instance, Civil Engineering at MIT adopts a programming course, in C (*1.00: Introduction to Computers and Engineering Problem Solving* [8]), while Cornell is using Java (*COM S 100: Introduction to Computer Programming* [4]).

These trends are often mixed. For instance, Rice University and Yale have one course based both in "hard programming" (in C or FORTRAN) and a numerical tool, Matlab (*CAAM 210: Introduction to Engineering Computation* [10], at Rice U., and *130b: Introduction To Computing for Engineers and Scientists* [14], at Yale). The University of Birmingham also adopts a mixed approach, with a programming course in C++ parallel to the overview course. Civil Engineering at Purdue University also has two courses: Before the computer literacy course, there is an "Introduction to programming in FORTRAN" (*CS 150: Programming I for Engineers and Scientists*).

Overview Courses

Although overview courses are more common in Computer Science curricula, this kind of course sometimes impacts the Engineering curricula, too. A main book in this line is Brookshear's *Computer Science – An Overview*: "For computer science majors and minors in the early stages of their college careers, many of whom mistakenly equate programming and computer science, and for students of other disciplines who want computer literacy beyond the ability to manipulate a particular program or do a little elementary programming." [2]

This kind of course can be very interesting. It presents major areas of computer science, giving the student a great latitude of knowledge in the field. It can also be useful in breaking myths, so common when the subject is computers. However, a major pitfall of this approach is that we lose the hands-on practice. Without application programs or a real language, there is little to do with a computer. Moreover, such courses are mainly informative and necessarily superficial.

An intriguing point is that few other areas adopt similar courses. For instance, few Engineering curricula have an overview course on Civil Engineering, or "Physics in a Glance". Maybe they should. Nevertheless, independently of the adoption of such a course for computing, an overview should not replace a more technical course, but at most complement it.

Computer Literacy

Clearly an engineer must be "computer literate" nowadays. This is not easy: Not only there is an increasing number of tools to master, but also each tool is becoming more and more complex. However, an explicit course on computer literacy is not a good approach. First, a basic skill in most tools can be acquired in a few hours, *as long as* the student is interested in the final results. An example is supplied by secretaries and clerks, who can learn the basics of a text-editor or a spreadsheet program in a short time, assuming the learning will give them a concrete profit; another example is how fast our students master a new video-game.

Second, more advanced skills on specific software tools are better taught when needed. Most programs use a metaphor in their user interfaces. If the user does not understand what the metaphor is about, there is little hope she will understand the software. For instance, one of the problems in teaching a freshman how to use an electronic spreadsheet is that the student does not know what a spreadsheet is for, in the first place. Such a program would be better presented in a management course, for instance. Moreover, only a professional user of a tool can motivate its more advanced features.

Finally, generic computer literacy courses veil a recurrent problem, that currently many lecturers and colleges are computer illiterate, or at least not as fluent as they wish their graduates to be. Computers are pervasive, and the training on their use must be, too. It is of little use to teach a student generic tools, if she has no incentive or opportunities afterwards to apply these tools. On the other hand, if all lecturers use, and motivate their students to use, software tools, and the institution facilitates that use with proper support, students should not need specific courses on computer literacy.

Programming

Many engineering curricula have kept the emphasis on programming in their entry-level computer courses, changing the language from FORTRAN to Pascal or C, and more recently, C++ or Java.

At a first look, such courses have kept the main goal of the past: To enable the future engineer to write her own programs. However, at a closer look, the goals have changed a lot, frequently without a clear recognition. This change is due to the change in the concept of program. As we have discussed, in the past, from an engineer's point of view, programming focused on a very narrow domain, with a fixed (and reasonably simple) language, FORTRAN; the main data structures were vectors and matrices, which are built-in in the language; there were neither fancy user interfaces nor dynamic structures with pointers. All that has changed, and now it is very difficult to form a useful programmer with a one (or two) semester course.

Currently, most students finish their computer courses with a shallow understanding of main programming concepts, such as recursion and abstraction. Because there is little time, the first superficial aspects of a language (its syntax) are overemphasized in detriment of its semantics. Usually, when a student claims to "know" a language, that means she masters its syntax: brackets, commas, etc.

A good illustration of the current overemphasis in syntax is a widespread opinion about Java: "If you know C, it is easy to learn Java." In fact, C and Java have similar syntaxes, but the similarities end here. One of the main concepts in C, pointers, is absent in Java. Other omissions include header files, global variables, and static arrays. By the same token, the main concept of Java, objects, does not exist in C. Classes, interfaces, inheritance, packages, exception handling, and other important parts of Java are also absent in C.

Some courses try to avoid this weight in syntax and language issues. For instance, Cornell's *COM S 100* explicitly states: "The subject of the course is programming, not a particular programming language." Nevertheless, course descriptions seldom include the main concepts involved in programming; generally, they are a list of language features (*if-then-else*, *while*, *array*, *record*, etc).

What Programming is About

Programming is much more than the knowledge of syntactic details of a particular language, or the skill to write a small program in that language. Programming is much

more than a tool: Programming is an intellectual discipline.

Learning how to program can be an excellent introduction to formal reasoning, since the student must use a formal language (the programming language), and has an exacting verification system (the computer). Programming also demands a kind of "meta-knowledge". To program the solution of a problem, the student not only must know the solution, but she must be able to *explain* her solution in precise and simple terms. Computers are stubborn pupils, doing exactly what we ask them to do. To teach them, we must be rather good expositors.

Programming explicitly explores some basic mathematical concepts, particularly *abstraction*, *structuring*, and *induction*. Because programs are easily implemented and executed, programming gives an unique opportunity for a student to have lots of hands-on experience with those concepts.

Abstraction is certainly the single most important concept in computer science. Building and understanding abstractions are key activities for a programmer, as well as for users of any application, above the basic levels. Abstraction is also a key concept to understand computers, because a computer is a huge pile of abstractions: transistors abstracted in logic gates, abstracted in logic circuits, abstracted in computer components, abstracted in memory, abstracted in binary instructions, abstracted in assembly language, etc. A typical difficulty students have with computers is that they try to grasp too many abstraction levels at the same time.

Unlike other disciplines, programming allows a student to manipulate abstractions; when a programmer encapsulates a piece of code in an abstraction (*function*, *procedure*, *subroutine*, etc) and gives it a name, that abstraction is now part of her "reality". She can use that abstraction in other parts of her program without concerns about the details of that code, or how it was implemented. The same can be done with data representations, resulting in data abstraction. The conscious creation and manipulation of abstractions give a student not only a deep understanding of the concept, but a feeling that only hands-on experience can give.

A programming environment is also an ideal laboratory for structuring. Any reasonable programming language allows a student to create several abstractions, and then to join them to build even more complex structures. Again, this hands-on experience of managing complexity is usually difficult to achieve in other areas. Particularly, the manipulation of data structures leads to a better comprehension of *data representation*, the concept that all kinds of information —images, texts, mathematical expressions, and even computer instructions— can be represented (or coded) in terms of a small repertoire of basic data types, such as numbers and arrays.

Finally, recursion puts inductive reasoning to work. Most courses introduce *iteration* (loops) before —or instead of— recursion, on the grounds that the former is simpler. In fact this is true, from a superficial point of view: A loop can be explained as a *go back to step 1*. However, such understanding is useful only for an informal analysis of a program; for instance, you cannot prove anything about a program with it. Worse, when doing synthesis, instead of analysis, this superficial understanding of loops gives few clues about how to use them to solve a particular problem. Recursive definitions, on the other hand, allow easy formal proofs —by induction— and a firm approach for programming synthesis, called *divide-and-conquer*. Finally, for many non trivial algorithms, such as quicksort and tree search, a recursive solution is simpler than an iterative one, even for an informal analysis.

The Language Paradox

> "A most important, but also most elusive, aspect of any tool is its influence on the habits of those who train themselves in its use. If the tool is a programming language, this influence is — whether we like it or not — an influence on our thinking habits." [Dijkstra]

More often than not, an engineer uses a particular language because it is the only one she knows, not because it is the best language for the task (by the same token, a teacher biases her students to use only the language she knows).

An engineer comes in contact with many different languages when using a computer, frequently unaware that she is dealing with a language at all. Each application or domain has its own language: formulae languages in spreadsheets, macro-languages in text-editors, SQL in databases, HTML in the Web, etc. The mastery of an application's language can make the difference between the expert and the mediocre user. Besides that, some programming languages may become obsolete before the student's graduation. Currently, many languages rise and fall in less than ten years. For instance, Pascal, although created in 1970, only gained momentum in the late seventies with the micro-computers (because it was simple enough to fit in a memory of 4K), had its glory with Turbo-Pascal 3.0, circa 1982, and by the early nineties was almost defunct. C++ as it is known today appeared in 1989, had a maximum circa 1993, and is already declining, due to Java. FORTRAN is the exception, not the rule.

Having that in mind, the result seems clear: We should de-emphasize language. After all, "the subject of the course is programming, not a particular programming language." Some people go as far as to suggest to use no language. Instead, such courses should use *pseudo-language*: Natural language (e.g. English) with a programming "flavor". However, without the use of a real programming language, such courses lose a lot: Despite being cumbersome, syntax is an integral part of the game. Without a real language, there is no hands-on. Without a real language, the student cannot grasp the real

meaning of *formal*: "Relating to or involving the outward form, structure, relationships, or arrangement of elements rather than content" [7].

The problem of choosing a language for an entry-level computer course poses therefore an apparent paradox: On one hand, since the emphasis should be on programming concepts, the adopted language seems to be almost irrelevant. On the other hand, the language plays an essential role in the course. One solution is that we choose the language not by its intrinsic usefulness, but for its usefulness in the course.

An excellent option is Lisp/Scheme [3] (Lisp denotes a family of languages, since it has no standard. Scheme is a particular member of that family). Like FORTRAN and COBOL, Lisp is a survivor; it was developed in 1960, and it is in widespread use today. Scheme is a small language, with a tiny syntax and a clear semantics. The main pro of Scheme is its simplicity; most students acquire a working knowledge of the language in a few classes. But there are other pros.

Scheme is interactive. Unlike languages such as C, Pascal or FORTRAN, where a student must use (and learn) a text-editor to create a program, and then feed the program to a compiler, in Scheme you can simply type in expressions, which are immediately executed. Therefore, students can start using the language in the first class.

Scheme has complete data description facilities. Any value in Scheme (even complex ones, such as arrays of arrays or lists of arrays) can be directly written into a program, or printed by the system. Therefore, when you write a sorting function, for instance, you do not need to write two other functions only to create the array and print the result; you just call the function with appropriate data and see the result. Moreover, when we discuss more complex data structures, such as trees, we have a concrete way to represent them that a computer also understands.

Despite its simplicity, Scheme supports most relevant programming concepts, including recursion, iteration, functional abstraction, side-effects, arrays, and dynamic structures. Some important absences are static typing and modules, although the latter is not really important for this level of programming.

Finally, there are several good introductory books about programming that use Scheme. Among them, "Structure and Interpretation of Computer Programs" [1] is already a classic. It is adopted in many Computer Science curricula, but may be too heavy for a typical engineering curriculum. Other interesting books are "The Little Schemer" [12] and "Scheme and the Art of Programming" [11]. (You can find more information about Scheme as a tool in education at http://www.cs.rice.edu/~shriram/Scheme/Education/.)

Back to the Present

Many (but far from a majority) Computer Science and Computer Engineering curricula have already adopted the ideas presented here (e.g. MIT, Cornell, Princeton, UCB, UCLA and Yale [5]). However, with the exception of Electrical Engineering courses, the curricula of other engineering areas (Civil, Mechanical, etc) is mostly unaware of this trend.

Here at PUC-Rio (the Pontifical Catholic University at Rio de Janeiro) we have introduced these ideas in ICC, the entry-level computer course for all engineering courses. This experience is still recent, so unfortunately we do not have much data. Nevertheless, some points are clear. First, the whole course is hands-on, since the first class. The students start using the computer in the first class, and after the second week are already writing small programs. Second, the students learn the language quite fast. After the first weeks, there is seldom a question about syntax, and the course is completely focused on programming. Third, we are able to cover many more programming concepts than before.

Some questions still remain. The main one is how fast they can adapt to a conventional language, when needed. Another question concerns the adoption of Scheme in other courses, such as numerical analysis. On one hand, the expressiveness of the language could bring benefits to those areas, too. On the other hand, other languages may offer better support for specific domains, for instance through better libraries or tools. Moreover, as we said earlier, many lecturers still have a surprising resistance to learn new things, such as a new language.

Acknowledgments

I would like to thank Noemi Rodriguez and Carlos Tomei, for many useful discussions about this subject and suggestions about this paper. This work has been partially supported by CNPq (the Brazilian Research Council).

References

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.

[2] Glenn Brookshear. *Computer Science: An Overview*. Addison-Wesley, 1996.

[3] William Clinger and Jonathan Rees (editors). Revised(4) report on the algorithmic language scheme. *ACM Lisp Pointers*, 4, July-September 1991.

[4] Cornell University. *COM S 100: Introduction to Computer Programming*. http://www.cornell.edu/Academic/Courses97/csen/en248.html

[5] Terry Kaufman. *Schools Using Scheme*. Schemers Inc. http://www.schemers.com/schools.html

[6] MathWorks Inc. *MATLAB: The Language of Technical Computing*, March 1998. http://www.mathworks.com/products/matlab/.

[7] Merriam-Webster. *WWWebster Dictionary*, 1997. http://www.m-w.com/dictionary.htm

[8] MIT. *1.00: Introduction to Computers and Engineering Problem Solving.* http://monett.mit.edu/100/home.nsf

[9] Purdue University. *CE 293: Computers and Computer Programming for Civil Engineers.* http://www.ecn.purdue.edu/courses/

[10] Rice University. *CAAM 210: Introduction to Engineering Computation.* http://www.owlnet.rice.edu/~caam210/

[11] Scheme and the Art of Programming. *G. Springer and D. Friedman.* McGraw Hill, 1994.

[12] The Little Schemer. *D. Friedman and M. Felleisen.* The MIT Press, 1996.

[13] University of Birmingham. *SEM117: Introduction to Computer Science.* http://www.cs.bham.ac.uk/modules/current/sem117.html

[14] Yale University. *130b: Introduction To Computing for Engineers and Scientists.* http://www.yale.edu/ycpo/ycps/E-L/engas.html