In Praise of APL:
A Language for Lyrical Programming

Professor Alan J. Perlis
Yale University

Many reasons can be given for teaching one or more aspects of computer science (defined as the study of the set of phenomena arising around and because of the computer) to all university students. Probably every reader of this note supports some of these reasons. Let me list the few I find most important: (1) to understand and to be able to compose algorithms; (2) to understand how computers are organized and constructed; (3) to develop fluency in (at least) one programming language; (4) to appreciate the inevitability of controlling complexity through the design of systems; (5) to appreciate the devotion of computer scientists to their subject and the exterior consequences (to the student as citizen) of the science’ s development.

Even though computer science deals with symbolic objects whose nature we study mathematically, it cannot be taught as an orderly development arising from a few fundamental ideas whose existence the student has already observed intuitively during his maturation, such as gravitation and electricity.

It is during this first computer course that the student awakes to the possibilities and consequences of computation. They arise most usefully and in greatest profusion during his writing of programs. He must program and program and program! He must learn how to state precisely in a programming language what he perceives about the nature of symbolic systems. I know of no better way to expedite this awakening than by programming.

But what should the student program? and in what language? I do not place much emphasis on heavy use of other people’ s programs, packages if you will, that perform real services such as statistical packages, data management systems, linear equations solvers, etc. While it is wise to use standard programs when they match one’ s needs, it is more important to master self-expression during this initial contact.

Available time is a limiting factor; a semester provides about 16 weeks of contact. During that interval the student must negotiate a set of tasks that sharpens his abilities and explodes his perceptions of the computer’ s capabilities.

He must be on the computer early and often during the semester and his approach to it must be smooth and easy. The computer system he uses should be time-sharing and interactive, if you will.

Learning to program involves a sequence of acts of discovery punctuated by recovery from errors. As the semester progresses the causes and nature of errors will change. Certain kinds will diminish and even disappear, only to be replaced by errors of deeper significance, harder to isolate and more resistant to satisfactory removal — syntactic gaffes give way to semantic errors — incorrect perceptions of purpose, improper use of means, the use of hammers to swat flies and swatters to level mountains.

To write correct and balanced programs a student may be forced to move between programs that are not related to each other by a few simple textual rearrangements. He must learn to write and test complicated programs quite rapidly. As he moves through the sequence of assigned tasks his ability to express himself fluently should not founder too soon because of language shortcomings. For all of the above reasons as well as a few others, I have come to believe that APL is the most rational first language for a first course in computer science.

It is true that BASIC and FORTRAN are easier to learn than APL, for example, a week versus a month. However, once mastered, APL fits the above requirements much better than either BASIC or FORTRAN or their successors ALGOL 60, PL/I and Pascal. The syntax of APL is not a significant difficulty for the students. The large number of primitive functions, at first mind-numbing in their capabilities, quickly turn out to be easily mastered, soon almost all are used naturally in every program — the primitive functions form a harmonious and useful set. As a data organization, arrays turn out to be extraordinarily useful (though prolonged contact with APL makes one wish for the added presence of more heterogeneous structures).

Style and Idiom

The virtues of APL that strike the programmer most sharply are its terseness — complicated acts can be described briefly, its flexibility — there are a large number of ways to state even moderately complicated tasks (the language provides choices that match divergent views of algorithm construction), and its composability — there is the possibility to construct sentences — one-liners as they are commonly called — that approach in the flow of phrase

organization, sequencing and imbedding, the artistic possibilities achievable in natural language prose.

The sweep of the eye across a single sentence can expose an intricate, ingenious and beautiful interplay of operation and control that in other programming languages is observable only in several pages of text. One begins to appreciate the emergence and significance of style and to observe that reading and writing facility is tied to the development of an arsenal of idioms which soon become engraved in one’ s skull as units.

The combination of these three factors makes it possible to develop an excellent set of exercises in the first course. These exercises can be large in number, cover a wide range of topics and vary widely in complexity, and still be done during the 16 week period. The later exercises can be tied to the design and development of a system — a collection of procedures that, in varying combinations, perform a set of tasks.

In Teaching Computer Organization

To appreciate computer science one requires an understanding of the computer. Once the student understands the computer — its macroscopic components monitored by the fetch-execute cycle and its apparent complexity being controlled by gigantic replication of a few simple logical elements — he can become aware of the important equilibrium between hardware and software — the shifting of function between the two as determined by economic factors — and between algorithm and system as determined by traffic and variation. Using APL it is straightforward to model a computer and to illustrate any of its macroscopic or microscopic components at any level of detail. The programs to perform these functions at every level of description remain small and manageable — about 40 lines or so.

The development of software, e.g., a machine language assembler, is a task of similar difficulty (about 40 lines) and hence possible within the confines of a first course.

Word processing and graphics, increasingly important application areas of computers, can be explored with exercises of no great size, e.g., to do permuted-index of title lists (~12 lines), display, rotation and scaling of composites of polygons (~20 lines), graphing of functions (~5 lines), etc.

With (or even without) the use of 2 or 3 pre-built functions, file processing problems such as payroll, personnel search, etc. can be written in a relatively few lines.

An important consequence of the attainable brevity of these compositions cannot be ignored: the student becomes aware that he need not be forced to depend upon external, pre-packaged and elaborate systems to do what are really simple programming tasks. Instead of learning a new coding etiquette to negotiate a complex external system, he writes his own programs, develops his own systems tailor-made to his own needs and understood at all levels of detail by him. If anything is meant by man-machine symbiosis, it is the existence of such abilities on the man side of the "membrane", for there is no partnership here between man and machine, merely the existence of a growing, but never perfectly organized, inventory of tools that the competent can pick among, adapt and use to multiply his effective use of the computer.

I cannot overemphasize the importance of terseness, flexibility and phrase growth to a beginning student. His horizons of performance are being set in this first course. If he sees a task as a mountain then its reduction to molehill proportions is itself a considerable algorithmic task. While this is true of very large tasks, even when using APL this conscious chaining of organized reductions can be postponed until the student has already collected a large number of useful data-processing functions, engraved in his skull, with which to level mountains.

It is important to recognize that no matter how complicated the task, the APL functions will usually be anywhere from 1/5 to 1/10 the number of statements or lines, or what have you, of a FBAPP (FORTRAN or BASIC or ALGOL or PL/I or Pascal) program. Since APL distributes, through its primitive functions, control that the other languages explicate via sequences of statements dominated by explicit control statements, errors in APL programs tend to be far fewer in number than in their correspondents in FBAPP.

I can come now to the topics of structured programming and program verification. Both are important, but their content and importance depend strongly on the language in which programs are couched. A program is well-structured if it has a high degree of lexical continuity: small changes in program capability are acquired by making changes within lexically close text (modularization).

Since APL has a greater density of function within a given lexical scope than FBAPP, one would expect that APL

programs will support considerably more structure than equivalent size FBAPP programs. Put another way, since the APL programs are 1/5 to 1/10 the size of FBAPP programs, the consequences to APL programs of weak structuring are less disastrous. Recovery from design mistakes is more rapid. Since we can only structure what we already understand, the delay in arriving at stable program organization should be considerably less with FBAPP!

Please note that the emphasis here is on the control of propagation of relationships, not the nonsense of restricting **goto** or bathing programs in cascades of **while** loops.

The verification, formal or informal, of programs is a natural and important activity. It is linked to specification: what we can’ t specify we can’ t verify. By specification we mean stating what is to be output for a given input. We immediately observe that, since specification in FBAPP is extremely tedious and unnatural, we must use some other language. APL turns out to be quite good and has often been suggested as a specification language. Assertions and verification conditions can be much more easily expressed as APL predicates than as FBAPP predicates. Because of the widespread distribution of control into the semantics of primitive functions, for which no proof steps need then be given, APL verifications tend to be, just as their counterpart APL programs, shorter and more analytic than equivalent FBAPP program verifications.

APL and Architecture

The form of the FBAPP languages follows closely the structure of the computers that prevailed during their inception. They have the nice property that one may often optimize machine performance of their compiled programs by transforming FBAPP programs to other FBAPP programs. Control of the computer is more easily exercised with programs in these languages than with APL, since the latter is more independent of current machines. For many programs this control over the target machine performance is quite vital, and APL couples more weakly to the standard computer than does FBAPP.

However, new array processing computers are beginning to appear and, had they been standard 20 years ago, APL and not FORTRAN would have been the prototype of language development. I often wonder at what descriptive levels we would be programming today had that been the case! Since it was not the case, we should not throw out or limit APL. We must seek ways to match it to the common computer.

We must design compilers as well as computers that fit APL better.

More Cost-Effective than BASIC

Cost is an important issue in the instructional process. An APL computer system currently costs about $10K per terminal, about twice the cost of a BASIC system. As APL system designs stabilize and integrated circuitry costs drop, the two figures will coincide at or near the cost of a contemporary terminal. However, even now the APL system is cheaper than BASIC systems for equivalent work loads because one can do more than twice as much with APL in a given period of time than with BASIC!

Let me mention in closing two additional issues regarding the use of APL in an introductory computer science course. First, most university computer scientists don't really know APL. They haven't appreciated what it means to think in APL — to think about parallel operations in arrays and to distribute and submerge explicit looping among its primitive functions. I am reminded of the difficulties many math departments experience when they try to replace calculus by a fine math and combinatorics course as the first meat and potatoes offering by the department to the university. However at Yale we have found that faculty outside the software milieu — in theory, for example — pick up APL quite fast and prefer it to FBAPP. I am sure the same is true elsewhere.

The second issue is of a different kind. I am firmly convinced that APL and LISP are related to each other along an important axis of language design and that acquiring simultaneous expertise in both languages is possible and desirable for the beginning student. Were they unified, the set of tasks that succumb to terse, flexible and expressive descriptions will enlarge enormously without overly increasing the intellectual burden on the student over his initial 16 week contact period.

Above all, remember what we must provide is a pou sto to last the student for 40 years, not a handbook for tomorrow's employment.

First appeared in SIAM News, 1977-06.

created: 2012-10-11 03:00
updated: 2012-10-17 16:25