From Hieroglyphics to APL - The Relentless Course of History

Homer Hartung

ABSTRACT

There are many historical parallels between older developments of number systems and modern developments of computer languages. These indicate that APL should ultimately develop a dominant position as the best language to use for complex computer applications.

Records of ancient kingdoms and their rulers are found in hieroglyphics. Some of these numbers for populations, armies, herds, etc. The schemes used by the ancient scribes were very simple. In ancient Egypt only a small number of symbols were needed to represent all numbers up to very large counts.

Have you every wondered why we don't use the simpler hieroglyphic representations? Probably not, because we are taught elementary mathematics before understanding the source of its symbols and procedures. Thus, we are indoctrinated with the idea that our representations of numbers were somehow decreed by the nature of the universe. Continued use of Roman numerals on cornerstones, monuments and the like, however, testifies to the contrary.

If you stop and think about it, there would be some advantages to going back to wider use of the Roman numerals. We could do away with the of the top row of keys on typewriters and terminals, and it would be a lot easier to teach to young children for simple counting tasks. You might worry that advanced mathematics would be impossible, but that isn't necessarily the case if we are smart enough. Consider old man Euclid, who was pretty swift in geometry even without modern computing techniques - but then he was a real genius!

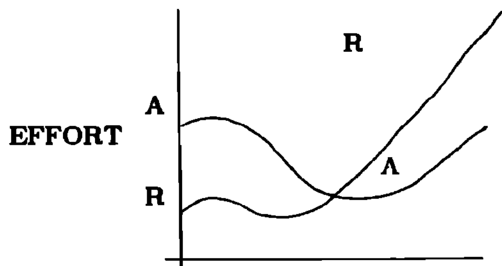

Figure 1
COMPLEXITY OF PROBLEM

The real pros and cons of number representations can be illustrated graphicaly as in Figure 1.

This shows a schematic representation of effort expanded versus complexity of the problem. 'R' stands for Roman numerals and 'A' for Arabic. In both cases, there is an initial hump associated with the difficulty of starting out. The hump for the 'A' curve is larger because the Arabic numbers are based on ten new symbols and rigid rules on evaluation by relative placement. The hump for the 'R' curve is smaller because it involves simple association of counts with familiar letters. The 'R' line is below the 'A' line when problems have a low degree of complexity. Tallies of ballots for example, are certainly much easier wit an 'R' type system than with Arabic numerals. There is a crossover point and a rapid divergence for increasing complexity, however. It is hard to conceive of trying to work logarithms using Roman numerals!

The history of number systems provides a strong parallel with things that are now happening in the area of computer programming languages. The concept of programming languages is new with our present generation. The first one was FORTRAN, invented around 1953, and thousands of similar approaches have been tried since then. A radical departure from the FORTRAN-type of language was proposed in a book entitled *A Programming Language* by Kenneth Iverson in 1962. This introduced a notational scheme which was an extension of matrix algebra. It was given the abbreviation APL and introduced as a computer language in 1969.

Except for APL, all computer languages invented to date have been ad hoc adaptations of mathematical notation such that it can be displayed easily with regular typewriter symbols. The assumption is that the standard typewriter keyboard was also devised by God! All such languages are analogous to the situation with Roman numerals 2000 years ago - only familiar symbols are used.

APL is now in the same situation as Arabic numerals when they were first introduced to Westerners - new and unfamiliar symbols are required. Thus, Figure 1 also represents the pros and cons of APL versus other languages. The 'A' curve represents relative effort in using APL to solve computational problems while 'R' represents what we might call 'regular' programming languages with a standard keyboard.

The parallel extends to the fact that whatever we first learn tends to seem easier and better. Children are not conscious of the hump in the effort curve for using Arabic numerals because they do not know enough to question the rules of the game. Similarly, people who start out learning APL as there first com-

puter language don't have much trouble.

The history of mathematics reveals that the Arabic numerals were around for many centuries before they became accepted widely. Obviously when hieroglyphics are taught to children for counting it is tough to switch them over to a superior but more complex scheme. By the same token, we can expect that the acceptance of APL as a widely used programming language will come about gradually but relentlessly as people make increasingly complex demands on computer systems.

Imagine the fate of an accountant who might try to keep books in Roman numerals because they are easier to learn for hand tallies of inventory. The same fate awaits programmers who insist on using regular languages because they are easier for simple computing tasks.

(These thoughts were picked up at APL83, particularly from Prof. D.B. McIntyre.)

Philip Morris, USA
Research Center
P.O. Box 26583
Richmond, VA 23261
U.S.A.

Processing Multiple Files

J. Rueda

Abstract.

Problems faced by an APL user in dealing with multiple files are explained, and a new approach to solve them based on finite-machine techniques is described.

The Problem.

One reason for short development times for programs in APL is the workspace concept. Many times, in fact, we don't need to write any function; we can review the data available in the workspace by just naming it. Unfortunately, when the amount of data grows we are no longer able to use this facility because the main storage memory is still a limited resource in many of today's computers.

As an example, suppose that we have an application that deals with ***N*** different sets of data. Initially we can keep them in ***N*** matrices in our workspace, but when the size of a set grows sufficiently we have to migrate its matrix to a direct-access file; in this natural way, we arrive at the problem that we deal with; managing multiple random-access files (from an APL environment).

Present Solutions.

The inability of APL to manage files is usually solved with auxiliary processors that perform the I/O operations.

Typically one variable for each file that will be processed is shared, and if direct access is required, it is a common practice to share 2 variables, one to be used as control, where the record number or key to be processed is specified, and another where the real information is transferred.

To solve the general problem of manipulating several files simultaneously we can follow two different approaches:

1. To share and retract two variables each time that a file has to be accessed.
2. To initially share a set of **2×*N*** variables where ***N*** is the number of files to be used.

The first solution is easy to implement but it is inconvenient, since four extra operations have to be done each time a single record is accessed:

1. Share variables
2. Open the file
3. Close the file
4. Retract variables.

The CPU-time required for opening and closing files depends very much on the operating system under which APL is running. For most APL implementations, the time spent establishing a shared variable is long. Thus this solution, although simple to implement, is very slow. Here is a typical function to read a file using this approach;

```
     ∇ Z←N READ1 F;CTL;DAT
[1]     ⍝ F : FILE ID
[2]     ⍝ N : RECORD NUMBER
[3]     ⍝   : KEEP THE A.P. ID
[4]       DAT←F
[5]       Z← ⎕SVO 2 3 ⍴'DATCTL'
[6]       CTL←N
[7]       Z←DAT
     ∇
```

Figure 1