2. A stack or other dynamic storage allocation mechanism is used for efficient data space management.
3. Individual procedures in programs are small making it possible to partition the program into fixed size blocks.
4. In addition, the use of an explicit intermediate interfacing language (STAB-12 in this case) makes it possible to change the entire running context without altering the high-level language part of the compiler or the source text of the programs.

The behaviour of the improved file store under one particular set of conditions (alternate reading and writing of records) is shown in Fig. 2. The actual improvement over the interpretive version varies with the type of load, and can be as high as 4.

ACKNOWLEDGEMENTS

The work described in this paper was supported by a grant from the Computer Board for Universities and Research Councils.

The authors express their thanks to Mr. John T. Hannah, who was responsible for implementing a large part of the file store and who helped with the design of the paging mechanism.

REFERENCES

1. 'The STAB-1 manual' Internal report, University of Strathclyde.
2. 'A design for a Modular File Store, FEP 8i/72', Internal Report, University of Strathclyde.
3. A. J. T. Colin, K. Shorey and W. Teasdale, 'The translation and interpretation of STAB-12', *Software—Practice and Experience*, **5**, 123–138 (1975).

Items 1 and 2 are available on request.

Received 6 *January* 1975

LETTERS TO THE EDITOR

I found the paper 'Random search on the 8-queens problem' by T. W. S. Plum, *Software—Practice and Experience*, **4**, 251–253 (1974), most appalling.

Let me try to be more explicit:

In spite of the impressive list of references, the author has failed to grasp the essence of the problem and its method of solution as brought forward so clearly in Dijkstra's original article 'Structured Programming', pages 72–82. Apparently the power of a small example to illustrate a general principle eludes him. And worse, his article might prevent others from drawing the suggested conclusions. The general principle involved is that when one searches for a solution of a certain problem one may do well by not searching over the entire space in which the solution is embedded, but by confining oneself to a (hopefully) small subset of that space which can be easily generated and is known to contain that solution.

Apart from doing a disservice to programming education the article is misleading in two ways. The author copied, from an earlier solution, an attractive data structure (the vectors DIAG1 and DIAG2), and the observation that the solution could be considered as a permutation, bypassing the amount of programmer's time involved in finding this structure. Secondly, the density of solutions in this problem ($192/8! \simeq 0{\cdot}005$) happened to enable the author to find a solution within an acceptable (?) amount of CPU time (although larger than for Dijkstra's algorithm to generate all solutions).

The absurdity of the authors approach is illustrated by the following two examples:

A sorting algorithm based on random permutations of the original sequence until all elements are 'in order' (this problem is *known* to have a solution, and will probably perform 'satisfactorily' for very short vectors!).

An algorithm to generate integertriples (a, b, c) satisfying ($a \leqslant b \leqslant 100$ **and** $b^2 - a^2 = c^2$) in the order of increasing difference $b - a$ ($= d$):

```
for d := 1 step 1 until 100 do
  for a := 1 step 1 until 100 do
    for b := 1 step 1 until 100 do
      for c := 1 step 1 until 100 do
        if b ↑ 2 − a ↑ 2 = c ↑ 2 and b − a = d
          then . . . . . . . . . .
```

In the sense of Mr. Plum an acceptable first try since very little brainpower is wasted. (This version was indeed produced by a student in an introductory programming course!)

A discussion of the article would be of academic interest only if the solving technique under attack had no practical value. However, even in commercial data processing there is an ample supply of problems, the solution to which may grow in size in a combinatorial fashion. One might think of: network planning, integer programming and many others. Programmers should be well trained in techniques to tackle such problems in an acceptable fashion.

I am not trying to deny the point that one should not bypass a simple solution if it works, but I am afraid much more brainpower and computing time is wasted by programmers trying in vain to tackle complex problems with simple methods, than the other way around. So, if this is a point at all, there is no point in raising it.

Finally, let me remark that the article shows that the so called 'richness of operators' present in APL may well have a blinding effect on programmers using such a language. They may be led rather to the application of inadequate operators (lines [8] and [9], merely because they are available, than to devise operators that fit their task. The disadvantages may not be evident for 'once only programs' but should be so for programs that are to be longer lived.

C. Bron
Twente University of Technology
The Netherlands

APL versus operator precedence

I feel I must respond to the letter from Mr. Dunn [*Software—Practice and Experience*, **4**, No. 3, 299–300 (1974)] where he comments on some views I had expressed on APL. I am afraid that his remarks are typical of the blinkered outlook of the APL fanatics, who seem to regard the language as a scriptural text and computing with APL as some kind of religious ceremony.

First, let me repeat my earlier argument. There is a commonly accepted convention about precedence of arithmetical operators. APL ignores it and I know that other programming languages don't all implement it. I must point out to Mr. Dunn that programming languages are very recent inventions, whereas mathemetical notation and the practice of calculating go back a very long time, certainly before the introduction of System/360. So, when I write of APL making 'nonsense of formulae . . . used in engineering design' I am *not* referring to FORTRAN statements. Mr. Dunn asks about the task of assigning precedences to APL operators, giving as an example left rotation and set membership. I answer that one need only deal with operators that have the same kind of operand, i.e. that have the same domains. For example, $A+B>C$ *must* imply comparing $A+B$ with C, because $A+B$ gives a numerical result whereas $B>C$ gives a logical result and one cannot add a number and a logical value, in spite of APL and PL/1 pretending that one can. There is, therefore, no need to establish relative precedences for + and >. Similarly, left rotation and set membership don't compete, so Mr. Dunn's problem does not arise in this instance. Again, the nonsense about $2<a<4$ being *false* in APL when a is 3 has nothing to do with left-to-right or right-to-left evaluation, merely with ignoring the point I have just made about the domains of operators. There are two ways out of this particular hole, either declare that the construction is not permitted (as most languages do) or translate it to mean $(2<a)$ & $(a<4)$ which is what the writer presumably intended.

Isn't this what grammar is all about, giving rules for parsing statements, and isn't structure the thing that APL completely ignores ?

Mr. Dunn mentions, in support of his arguments, that programming languages rarely allow blanks to replace the multiplication operator the way common habit does. Mostly, of course, this is because of the idiocy of the designers of programming languages, in particular FORTRAN, who decided that a blank was not significant and so did not use it as a separator, the way written language does. However, common habit often goes even further than Mr. Dunn suggests, by eliminating the blank itself, so that *ab* will mean *a* times *b*. But what should the translator of a programming language do when faced with the symbol string *ab* if there are identifiers *a*, *b* and *ab* in use ?

I agree that a programming language should represent the fact that it is used for programming. However, whereas I am sure it is a good thing for a programmer to be aware that his high-level language statements get transformed by some magical process into machine instructions and that he comes to no harm if he also knows the limitations of machine codes, I feel it is unnecessary for him to have to worry unduly about the details of the implementation. All language should be used to express ideas with clarity. To write

$$A0+X\times A1+X\times A2+X\times A3$$

when one means (in mathematical notation, without multiplication signs)

$$a_0+a_1x+a_2x^2+a_3x^3$$

is to confuse the meaning and the method of achieving it.

P. A. Samet
University College London