Towards FORTRAN VI? Part 2. FORTRAN in the modern world

By D. F. Hendry* and P. A. Samet†

The previous paper (Healy, 1968) made several suggestions for improving FORTRAN from the user's point of view, mainly by removing irksome and unnecessary restrictions as well as by pointing out valuable extensions to the language. We now make further suggestions, based on efficiency considerations as seen by the software writer and the machine manager.

(Received February 1969)

Although FORTRAN has received many additions and improvements since it was first introduced about 12 years ago, basically it still presents the user with the same type of machine environment as in the beginning. Machine hardware, however, has changed very considerably and so have the ways of using this hardware. Multiprogramming (batch) machines, time-sharing (conversational) machines, random access backing stores and similar devices are now in common use, yet FORTRAN has remained blissfully unaware of these. In our view it is necessary for FORTRAN to take note of these developments if it is to remain a machine language of wide application.

1. Multiprogramming and conversational systems

The distinguishing feature of the new generation of machines is their ability to have several programs resident simultaneously and swapping control between them. It does not matter at this stage whether the operating system is a batch-processor running several streams, a conversational system with interactive terminals or some mixture of the two, neither does it matter whether the programs are entirely in the core store or partly in a backing store, possibly with a (paged) virtual memory device. The competition is for *store* and a program that takes more than it needs is a liability. FORTRAN is a major culprit in this sense because of the absolute dimensioning of arrays. This means that space is reserved for a large matrix although any particular run may require only a fraction of this amount. Dynamic arrays, in the ALGOL sense, have much to commend them, are not difficult to implement and are no more inefficient than the present absolute dimensions. Fully dynamic array bounds, where the dimensions—and hence the space required —can vary during the program run are perfectly feasible but could play havoc with the location of COMMON variables. A simpler step towards the economical use of space would be a *parameter facility*, which would allow insertion of dimensions at load time. What is required is a special way of indicating that an array's dimensions will be supplied later. A possible method is by setting dimensions to zero in the source text, thereby specifying the number of subscripts, and then, at load time, giving the name of the array (perhaps also the subprogram in which the array occurs) and its actual dimensions. This would be especially convenient in a conversational system, where the loader could be made to request actual dimensions when 'zero dimensions' are encountered. An alternative method is to give the master program a 'privileged subroutine status' and then using the adjustable dimension facility of subprograms. The first method is probably simpler. Something very like it has been implemented in the FORTRAN compiler written for the Atlas at the Chilton Laboratory.

As well as multiprogramming systems where several independent programs compete for the processor's facilities, it is now becoming possible to have systems where several parts of the same job can be run in parallel. Sometimes this is done by use of a computer with several processing units sharing the same store, sometimes by extensions of the more common multiprogramming facilities. It would seem desirable to have facilities within a programming language to indicate that certain phases of a process may be done simultaneously, if this allows better use of machine resources. The program structure, showing which parts could be obeyed simultaneously, could be made available in a manner akin to the way an overlay structure is usually specified, although it is a graph rather than a tree that is necessary. PL/1 includes multi-tasking facilities. The extension should be such that it is ignored if the program is presented to a computer that does not allow such multi-tasking.

A program that hoards peripherals is as much of a liability as one that acquires more space than it needs. All operating systems will require the user to state which peripheral devices he needs (although often a card reader and printer are allocated without explicit request), in particular which private files will be required. Only the user is in a position to know when a device is no longer needed by his program and he should be able to RELEASE it, i.e., to hand it back to the supervisory program for reallocation to another user program. The *allocation* of peripherals is a function of the operating system, not of the program, and a wise system delays allocation as long as possible.

2. Effect of new hardware

Here we comment on two features: random access stores and character sets.

The normal FORTRAN input/output statements are

* *Institute of Computer Science, 44 Gordon Square, London WC 1*
† *Computer Centre, University College London, 19 Gordon Street, London WC 1*

intended only for sequential media like magnetic tapes.* Newer devices, like disks, drums, magnetic card file units, however, allow access to information in other ways determined more by the inter-relationships between items than by their physical placing in the store. Some methods for non-sequential access to files would seem to be necessary.† We readily admit that it is possible, and also very efficient in the right circumstances, to use disks, etc., as if they were tapes.

Character sets are of rather less significance than some of our other points, but we notice that extended 64-character sets are now becoming more usual. It would appear reasonable to allow a statement of the form:

IF (A > 0.0) . . .,

instead of insisting on

IF (A .GT. 0.0) . . .,

if such characters are available. The advantage for the user is greater legibility of his program as it is in a more natural form, and the compiler writer—and his compiler—have a slightly easier task. In passing we note that if the use of such additional characters became widespread there would be considerable user-pressure on the manufacturers to agree on a standard card code, to the benefit of the whole community.

3. Further suggestions

The remaining points stem from present-day knowledge of how to implement programming languages and observation of user habits.

The first two are direct consequences of the fact that very few users, especially users of large machines, write programs in an assembly code. This implies that very large programs will be written in a high-level language and that debugging information has to be made available in the high-level context.

For programs that are too large for the main store, some form of *overlay* structure is essential. The person who writes in assembly code is able to control such overlays but usually such facilities are denied to the high-level programmer. Several software systems in current use do allow such an extension, but it would be useful to have a unified way of implementation. It might be worthwhile to notice that really two kinds of overlay are required, for program and for data. We mentioned earlier that efficient operation of multi-programming systems demands that programs use as little space as possible. Overlays are important also in this respect, since the amount of core store available to a program may be severely limited, whereas backing store is often more readily available. There are many implementations of overlays, e.g., on IBM 360, ICL 1900, CDC 6600 to name a few. In some cases the working of the overlay structure is explicit in the user's program, other implementations move it 'behind the scenes' and the specification of the overlay structure is given as a series of control statements. A rather simpler device of the same general kind was the CHAIN facility available on the IBM 7090. Intelligent use of segmentation facilities may well increase the total throughput of the system.

Debugging and program tracing may be thought of as belonging to the operating environment, not to the programming language. To a certain extent this is true but not entirely. When testing programs, it is highly desirable to have *trace* information made available only when a particular event occurs. This means that statements to control tracing have to be inserted into the program, and so they become part of the language. (The information that is extracted must refer to the source text, not to the compiled code, a matter we do not pursue here, as this is a compiler facility and has nothing to do with the language.) Although there should be an agreed form for these debugging statements, they could probably be made optional, so that they are ignored by a compiler without trace facilities.

Tracing facilities are available on many machines but often they have been implemented locally and are not part of the standard software, let alone part of the language. In many cases the information given is almost totally oriented towards the assembly language programmer. The FORTRAN G compiler on IBM 360 includes a DEBUG facility, which is absent from the other compilers, the compiler on the London Atlas has some facilities of this kind, there is a simple TRACE in the ICL 1900 compilers and a rudimentary facility exists on the CDC 6600.

A possible way of signalling where such tracing is to take place is the letter T (or the word TRACE) in column 1, rather like a comment, followed by some appropriate syntax. Use of column 1 has two advantages: it is easily seen by the programmer and it is also easier for recognition by a compiler. A system without tracing facilities could then regard this information in the same way as it does a comment, i.e., it is ignored in the compilation.

Increased efficiency of the compiled program can also be achieved by having some form of co-operation between the user and the compiler. There are many things about his program that are known to the programmer but which may be extremely difficult or even impossible to discover during compilation. In particular, it is only the user who knows about the expected flow of control, the relative number of times various parts of the program will be executed, or the use of peripherals (which we have already mentioned). We put forward the idea of *hints*. These would be in the form of comments that could be acted on by the compiler.

Examples of such hints could be the information that a particular subroutine is called with the same parameters as on a previous occasion, that a particular subroutine does not change COMMON variables, and another would be the indication that a loop was the innermost loop of a program so that optimisation here would be worthwhile whereas another part of the program is traversed so seldom that optimisation gains nothing. Incidentally, such hints would also improve the documentation of a program.

As hints are a type of 'super-comment', we suggest that these, too, should start in column 1 with H (or the word HINT). There would have to be a limited vocabulary and appropriate syntax for such hints. A step in this direction has been taken by PL/1 with the introduction of the attribute 'ABNORMAL'. It is

* And even then 'read-backward' facilities are not used.

† Direct access facilities are available in the compilers on the IBM System/360.

worth noticing that FORTRAN I had a FREQUENCY statement, which allowed the programmer to give the probabilities associated with the paths followed after an IF statement. This facility allowed some optimisation of the use of index registers. For some obscure reason, FREQUENCY did not survive into later versions of FORTRAN.

A possible extension concerned entirely with user convenience is more flexibility about numerical constants at input. We suggest that input routines should accept whatever number of digits is offered and arrange the conversion to internal form so as to ignore (truncate or round) insignificant digits, rather than signal a fault. It is annoying and baffling to the average user to discover that the single precision value of π is 3·141593 on one machine and 3·1415926535 on another and that the first machine refuses to accept the accurate version, even if it cannot store it. It is also a totally unnecessary restriction that can easily be removed.

Conclusion

It has always been the claim of FORTRAN supporters, when arguing for the merits of the language against the virtues of its competitors, that it makes efficient use of a computer. It is in this spirit that we have drawn attention to several areas where agreed extensions are likely to be of value in the modern machine environment. Several of our suggestions presuppose particular forms of hardware facilities: compilers on more restricted machines should simply ignore statements intended to make better use of sophisticated equipment, rather than throw out the program as 'containing errors'.

We know that many of our proposals have, in fact, been implemented in some form or another on a variety of machines. Often, however, these extensions have been done in incompatible ways or else they have remained purely local. It is our hope that some action will be taken to ensure that those language extensions that are thought to be generally desirable are made in an agreed manner. Only if this is done will FORTRAN remain a viable language in general use among a wide and varied body of users.

We are grateful to Mr. M. J. R. Healy for permission to use the title of his article and also for his helpful comments.

Reference

HEALY, M. J. R. (1968). Towards FORTRAN VI? *Comp. J.*, Vol. 11, pp. 169–172.

Book Review

Mathematical Theory of Switching Circuits and Automata, by Sze-Tzen Hu, 1968; 253 pages. (University of California Press, $9.)

The development of switching theory over three decades has reached a point at which this book is particularly welcome. As Dr. Hu writes in his preface, most of the major problems have now been solved, and it is time to organise these results as a branch of pure mathematics in a way which will reveal their basic simplicity to both mathematician and engineer.

Rather than attempt to cover the entire field, the author prefers to take a few important topics and demonstrate the possibility and the value of carrying out such formalisation and simplification. The outcome is a structure of ideas which certainly have implications outside switching theory as such. A relationship is clearly visible, for instance, between the theory of prime implicants and that of 'resolvents' in logical calculi. Much of this book may be regarded as a valuable contribution to the study of finite Boolean Algebras.

After an introduction to switching functions and their various representatives, we are given a very thorough discussion of methods for finding the most economical disjunctive or conjunctive canonical forms for arbitrarily given functions. Here the terminology follows the topological approach of Mueller and Roth. This leads us to the more general problem of realising functions with a given set of logical devices. The notion of decomposition of a function in terms of simple decompositions develops the ideas of R. L. Ashenhurst, and this section culminates in a minimisation algorithm, described and illustrated with typical care.

In the last chapter the reader may perhaps consider himself to be taken on an unnecessary detour. Dr. Hu had decided to deduce the properties of finite-state machines as special cases by first setting up a theory of sequential machines with possibly infinite sets of states. If this is the hardest part of the book to assimilate, it is not unnecessarily so, once one accepts the worthwhileness of the approach; the author's explanations are always painstakingly clear. An automaton is defined as a function from the set of all input tapes into [0, 1], an n-input sequential machine as a function from $S \times [0, 1]^n$ into S, where S is a possibly infinite state-set. That every automaton is realisable by some sequential machine is almost trivial. That for each automaton there exists a (finite or infinite) 'minimal' machine which realises it is a result of some depth, and it receives a notably elegant treatment from Dr. Hu.

One might hesitate to recommend this volume to anyone completely unfamiliar with switching functions, but to those with at least slight experience of the practical problems involved and consequent motivations it should prove highly stimulating. A set of well chosen exercises consists in part of straightforward applications of the text, in part of developments of the theory that tie up with references in the deliberately concise bibliography. The book is pleasantly produced, and printing errors have been minimised.

MICHAEL BELL (London)