Towards FORTRAN VI?

By M. J. R. Healy*

Examination of a number of FORTRAN compilers compatible with the ASA standard shows that they have many extra features, and that several of these are common to many compilers. It is suggested on these grounds that a 'super-standard' FORTRAN, standing to ASA FORTRAN as this does to ASA Basic FORTRAN, might be defined.

(First received February 1968)

Scientific computing in Great Britain has changed quite markedly since the Flowers' report appeared in December 1965, and one aspect of this change is the increased acceptance of FORTRAN. It is now taken for granted that any computer for scientific use will possess a FORTRAN compiler, and that this will be compatible with one or other of the ASA specifications (American Standards Association, 1964, 1965). As well as this, most FORTRAN IV compilers now offer a range of features over and above the minimum needed for ASA compatibility. If these extra features are examined a certain measure of agreement between the different compilers is apparent, and this leads to the idea that the time is ripe for a third level of 'standard FORTRAN' standing to FORTRAN IV much as this does to FORTRAN II or ASA Basic FORTRAN. This note aims at setting out some ideas for such a language.

It is in many ways remarkable that a language as old as FORTRAN is still in use at all, and any 'extended FORTRAN' must be able to stand the competition of newer languages such as ALGOL and PL/I. One reason for survival seems to be the fact that its very limitations, while not too serious from the user's point of view, make it relatively easy to compile. Fast compiling is important for a language much used in teaching since students' exercises spend most of their time in the compiling phase, and indeed the same consideration is important in any research-based computer centre where the ratio of new programs to old will always remain high. It follows from this that an extended FORTRAN, while aiming to remove some of the limitations of FORTRAN IV, should remain simple in structure, and that it may be necessary to exclude features which, while occasionally useful, would complicate the compiler structure to an unacceptable extent.

Another historical feature of FORTRAN is the ease with which it has been extended to cover special applications—rather diverse examples can be found in syntax description (Leavenworth, 1964), in list processing (Weizenbaum, 1963) and in simulation (Belkin and Rao, n.d.). This implies that an extended FORTRAN should remain a general purpose language—no attempt should be made to build special purpose features into it. It is a positive advantage of special purpose subroutine packages that their users have access to a general purpose language (see, for example, Healy and Bogert, 1963), and there will be sufficient pay-off if this language is made as easy to use as possible.

FORTRAN has always been a language willing to profit by knowledge of its own implementation—an obvious example is the EQUIVALENCE statement. In this it differs in outlook from ALGOL which is inclined to go to extremes to remain pure from machine-dependence. There seems no reason to blur this distinction; there is much to be said for both points of view, and it is as well to have each of them embodied in a widely available language.

The following, then, is a list of suggested extensions that might lead to a definition of an extended standard FORTRAN. Obviously, they do not constitute such a definition; I have not considered details (such as the permitted length of identifiers) which I regard as of secondary importance.

1. Assignment statements

The obvious extension needed here is the use of mixed-mode expressions. For sheer inconvenience, the rule which forbids X = Y/N ranks high among FORTRAN IV's minor defects. The exact rules for the insertion of transfer functions of course need to be specified, but no difficulty of principle is involved.

Another handy construction which would be simple to implement is the ALGOL-like A = B = C+ D. A convention is needed when the left-hand side identifiers differ in mode; it is best if the value of the right-hand side expression is assigned to each independently, so that I = X = A and X = I = A mean the same thing.

Additions to the list of built-in functions may be considered here. A useful pair would be DOUBLE (X,N) and HALVE (X,N), giving multiplication and division by powers of 2, and integer equivalents of these giving rise to logical right and left shifts. All these should operate correctly with N negative or zero. It should also be possible to extract any particular bit or combination of bits from an integer quantity, and also to count the number of one-bits. Clearly, the results of these latter operations will vary with word length and will thus be machine-dependent, but this seems a poor reason for rejecting them—word-length-inde-

**Medical Research Council, 172 Tottenham Court Road, London, W.1.*

pendent arithmetic is a myth, and a program which compiles into 24K of store is machine dependent in that it will not run on a 16K machine. The operations correspond each to one or a few machine instructions; their continued existence in machine order-codes argues that they are generally considered useful; and it seems unreasonable that they should be barred from a higher-level language on legalistic grounds.

It would be useful to have a version of the DATA statement which could be invoked dynamically during program execution. The main use for this would probably be the setting to zero of a set of locations, the purpose served by the ERASE and CLEAR statements of BTL and Atlas FORTRAN V.

2. Control statements

Integer expressions should be allowed as the parameters of a DO statement, and it should be possible for a DO statement to count backwards, so that

DO 10 I = N — 1, 0, —1

becomes legitimate. These extensions are both simple and useful enough to be uncontroversial; other possibilities are perhaps more a matter for discussion.

(*a*) Real expressions as DO loop parameters. This would allow parameters such as T = 0, PI, PI/180. The difficulty is, of course, the effect of rounding errors on the test terminating the loop. On balance, it seems best to keep this difficulty out in the open by making the programmer cope with it explicitly.

(*b*) ALGOL-like constructions, in the style

DO 10 FOR I = integer list
DO 10 WHILE logical expression.

If the first is implemented, it should allow for the inclusion of ordinary DO-controlling triples in the list.

The arithmetic IF statement could readily be provided with a 'next statement' facility as in HARTRAN—almost always one or two of the three paths are to the next following statement, and this should not need an explicit label.

The logical IF needs a rather more far-reaching extension to remove the limitation that only a single statement is executed when the condition is satisfied. To achieve this demands a way of bracketing together a number of statements. One technique involves statement brackets BEGIN and END, as in ALGOL (this is used by the Atlas FORTRAN V, where it is associated with the provision of block structure). Somewhat simpler is the SDS technique of allowing a satisfied IF to cause the bringing in of the whole line of text on which it stands, and then to allow more than one statement per line—this is easily done by introducing a statement separator such as a semi-colon (which also is useful in other contexts). Combining this with the use of continuation cards enables a substantial piece of text to follow the IF bracket, and makes up to some extent for the lack of an 'else' construction.

These two proposals would remove some of the stiffness which sometimes prevents a FORTRAN programmer from writing his logical tests the natural way round. They also cut down on the number of labels in a program, which helps to speed up compilation.

3. Input/output

This is the place where the largest changes seem called for. FORTRAN input/output is in many ways very powerful, but it has its weaknesses, and its complexity makes it a stumbling-block for beginners. The FORMAT idea was originally designed to describe the layout of a punched card, and it does this very well. It is less effective for a line of printed output, and often quite inappropriate for paper tape. A whole range of extensions such as 'widthless' and adjustable formats have been tried to overcome these difficulties, and in addition, formatless output in some standard layout has been provided. It would be far better, while retaining formatted input and output, to provide as well an independent single-number input/output system such as that originating in Mercury Autocode and developed in its successor languages, EMA and CHLF. Input from paper tape or equivalent media is by a real function which reads in characters up to a terminating character (such as space or end-of-line) and returns a single number. The corresponding output routine has two arguments giving the number of places before and after the decimal point; if the first or second of these is zero, output is in floating or integer form, respectively. Other routines are needed to output a number of spaces, to move up a number of lines, to move to a new page and to input or output a string of characters. This set of routines would share a good deal of the input-output machinery already present in the compiler.

Formatted input–output must be retained, and not only for compatibility reasons; there are probably as many cases in which the use of explicit terminating characters is positively inconvenient as there are of the opposite state of affairs. A few modifications would make the system handier to use. On the output side, an obvious requirement is a means of specifying textual (Hollerith) output without counting characters. The usual limits on BCD record length are irksome for tapes written off-line, and means should be provided for extending them when necessary.

A rather general point is involved in repairing a marked weakness of many orthodox FORTRAN compilers, namely the action taken when the input routines encounter a faulty character or an end-of-file symbol. Not only are these commonly treated as catastrophic failures leading to program abortion, but the amount of information provided concerning the fault is often scanty. This weakness extends to the built-in functions like logarithm and square root. A partial solution is provided in 360 FORTRAN, in which failure destinations can be specified in input statements. A second possibility could be based on another Mercury Autocode feature. This would permit any run-time fault to cause

a jump to a particular label with a code number identifying the particular fault placed in an integer variable. The actual label and variable should be specifiable by a system subroutine, so that they can be changed dynamically. If not specified, default action as at present would be taken.

There is sometimes a need to examine an input record before deciding what format to use in converting it. This leads to the idea of store-to-store conversion—quasi-input/output operations in which a named area of store takes the place of the usual peripheral (some further advantages of this system have been pointed out by Pyle, 1962). Somewhat related to this is the need for column-image input from punched cards; this would be the more useful in the light of the bit-handling functions suggested above. This type of input would also be appropriate in a paper-tape context, both to combat the lack of code standardisation in this medium and also to accommodate tapes punched in 'fancy' codes by automatic data-acquisition equipment.

More far-reaching is the question of character handling. In standard FORTRAN characters can be input and output using A-conversion, but operating on them internally in any way is difficult. A number of means of coping with this have been adopted by different compiler writers, often involving new data types such as TEXT (Atlas FORTRAN V) or CHARACTER (CDC). The awkward matter of machine dependence arises again, with particular force when the contrast between word-organised and byte-organised machines is borne in mind. As throughout, a simple solution is to be preferred over a complex one. One such could involve the storage of character strings in packed form in integer or real arrays, a built-in integer function to extract the Nth character of a given string, and a standardised code of integer equivalents for the permitted character set. The last proposal may sound fatuously optimistic, but the existence of the ISO and ASCII codes gives some hope that agreement on a standard could be reached. These facilities would allow individual characters to be extracted and compared for sorting purposes; blocks of characters occupying whole words (or equivalent units) would also give predictable results when handled by integer arithmetic.

4. Subprograms and program segmentation

The main need here seems to be for multiple entries and exits for subroutines, and again a simple solution is probably the best. This consists in permitting labels as subroutine arguments. It may be necessary in the interests of straightforward compiling to identify a label in a CALL statement by some special symbol which distinguishes it from an integer; indeed two symbols may be necessary to distinguish labels internal to the subroutine (multiple entries) from those external to it (multiple exits). The multiple exit feature is commonly needed to provide error returns, and it should be possible to relate this to the standard fault procedure mentioned above.

There are a number of further extensions connected with subprograms whose usefulness must be balanced against their ease of implementation. One of these is block structure, or subroutines defined within subroutines. Simpler than this is the SDS facility which cancels the definitions of all current labels except those explicitly insulated from its effects. In the same area (basically that of communication between program segments) is the PUBLIC or GLOBAL facility for identifying variables by name rather than by location as in COMMON statements. A programmer often wishes to define a subprogram with a variable number of arguments, a facility usually restricted to assembly languages. To allow complete freedom for recursion would almost certainly be impermissible under the assumed terms of reference, but this is not in any way essential (see Strachey and Wilkes, 1961); it may well be possible to allow for recursively usable subprograms which are explicitly labelled as such.

Conclusion

The above discussion is unlikely to satisfy any particular FORTRAN compiler writer, who will almost surely find that some of his brain children have been omitted. The purpose of a standard language must, however, be borne in mind. It is simply to ensure that a user can run a stranger's program on his own machine with the assurance that it will at least compile—the effects of word length and rounding techniques make it rather much to expect that it should give identical answers. It is therefore not unreasonable to include only those extensions which have proved widely useful, or which overcome known weaknesses in the language. There is no reason why each centre should not possess its own super-extension for internal use provided it sticks to the standard for communication and publication purposes. It should also be remembered that the virtues of a compiler are not limited to the range of statements it will recognise; diagnostic capabilities and the ways in which it interlocks with the surrounding operating system are just as important.

References

Descriptions of most of the FORTRAN dialects referred to can be found in the manufacturers' literature. Exceptions are Atlas FORTRAN V (for which, see Schofield, 1967), HARTRAN (York, 1964) and Bell Telephone Laboratories (BTL) FORTRAN, of which no published description seems readily available.

American Standards Association (1964). FORTRAN *vs* Basic FORTRAN—a programming language for information processing in automatic data processing systems, *Comm. Assoc. Comp. Mach.*, Vol. 7, pp. 591–625.

American Standards Association (1965). Appendixes to ASA FORTRANS, *Comm. Assoc. Comp. Mach.*, Vol. 8, pp. 287–288.
BELKIN, J., and RAO, M. R. (n.d.). *GASP users' manual*, United States Steel Corpn., Applied Research Laboratory.
HEALY, M. J. R., and BOGERT, B. P. (1963). FORTRAN subroutines for time-series analysis, *Comm. Assoc. Comp. Mach.*, Vol. 6, pp. 32–34.
LEAVENWORTH, B. M. (1964). FORTRAN IV as a syntax language, *Comm. Assoc. Comp. Mach.*, Vol. 7, pp. 72–80.
PYLE, I. C. (1962). Character manipulation in FORTRAN, *Comm. Assoc. Comp. Mach.*, Vol. 5, pp. 432–433.
SCHOFIELD, C. F. (1967). *A manual of the Atlas* FORTRAN V *language*, University of London Atlas Computing Service.
STRACHEY, C., and WILKES, M. V. (1961). Some proposals for improving the efficiency of ALGOL 60, *Comm. Assoc. Comp. Mach.*, Vol. 4, pp. 488–491.
WEIZENBAUM, J. (1963). Symmetric list processor, *Comm. Assoc. Comp. Mach.*, Vol. 6, pp. 524–544.
YORK, E. J. (1964). *Atlas* FORTRAN *Manual*, Atomic Energy Research Establishment, Report R-4599.

Correspondence

To the Editor
The Computer Journal

Generation of time delays on analogue computers

Sir,

I have one or two comments on the paper by Riley and Walker (1968) dealing with rational approximations to $\exp(-sT_D)$ and their use in generation of time delays for analogue computers (this *Journal*, Vol. 11, p. 72).

First, a matter of terminology: the paper uses the term 'Padé approximation' to mean any suitable rational approximation. This term should be confined to a rational function for which the first N terms of its Maclaurin expansion agree with those of the function being approximated; N is the number of independent parameters. Thus in the paper only Set 1 is a Padé approximation; the expansion agrees with that of $\exp(-sT_D)$ up to the term in s^8.

Since the Padé approximation devotes all its parameters to securing desired behaviour at the origin, it is not surprising that it is not the best if one wants the delay of the corresponding system to be approximately constant over a range of frequencies. If 'approximately constant' is given some precise meaning, then a well-defined problem exists.

Much work has been done in this area by circuit theorists (Kiyasu, 1943; Thomson, 1949; Abele, 1960; Ulbrich and Piloty, 1960). The first two deal with the Padé approximation (although not in fact using this term) and include analytical formulae for the delay. The others give numerical solutions for systems with an equal-ripple approximation to constant delay. Two results, derived from Abele's tables, may be of interest; a_1 to a_4 and T_D have the same meaning as in Riley and Walker, and the group delay lies within the range $T_D(1 \pm c)$ for $0 \leqslant f \leqslant f_0$.

Another approach is that of Hausner and Furlani (1966), who give design tables for equal-ripple approximations both for phase and for phase delay.

c	T_D/a_1	T_D^2/a_2	T_D^3/a_3	T_D^4/a_4	f_0T_D.
0·01	2·020	9·052	79·69	1126	1·040
0·02	2·041	8·975	80·63	1054	1·133

The coefficients of Sets 2 and 3 of the paper are in the same neighbourhood as these.

Yours faithfully,

W. E. THOMSON

P.O. Research Department,
London, N.W.2
14 May 1968

References

ABELE, T. A. (1960). Übertragungsfaktoren mit Tschebyscheffer Approximation konstanter Gruppenlaufzeit, *A.E.Ü.* Vol. 16, p. 9.
HAUSNER, A., and FURLANI, C. M. (1966). Chebyshev all-pass approximants for time-delay simulation, *IEEE Transactions on Electronic Computers*, Vol. EC-15, p. 314.
KIYASU, Z. (1943). On a design method for delay networks, *J. Inst. Elect. Commun. Engrs of Japan*, Vol. 26, p. 598.
THOMSON, W. E. (1949). Delay networks having maximally-flat frequency characteristics, *Proc. IEE*, Vol. 96, Pt. III, p. 487.
ULBRICH, E., and PILOTY, H. (1960). Über den Entwurf von Allpässen, Tiefpässen und Bandpässen mit einer im Tschebyscheffschen Sinne approximierten konstanten Gruppenlaufzeit, *A.E.Ü.*, Vol. 14, p. 451.

(*Further correspondence appears on pp.* 194 *and* 240)