Discussion and correspondence
The proposed new standard for FORTRAN: A critical examination

A. C. Day\(^1\), P. A. Clarke\(^2\), D. Hill\(^3\), and J. K. Reid\(^4\)

The FORTRAN Specialist Group of the BCS has for some years been in communication with X3J3, the ANSI Committee engaged in producing a new FORTRAN Standard. The FORTRAN Specialist Group set up a working party under the chairmanship of Colin Day to carry out a critical examination of the new standard. The present paper arose out of that examination, but is entirely the work of the authors listed above, and cannot be said to represent the views of the FORTRAN Specialist Group, or indeed of the BCS.

The name used here for the new standard (or proposed standard) is FORTREV, the name used by X3J3 for their working document. Note that at times we use FORTREV as the name of the standard, the language and the document. For our examination we used the latest edition of FORTREV available to us, namely that dated 26 September 1975. We have not considered the subsets, but confined our study to the full language. Readers who wish to examine FORTREV for themselves may order copies from:

ACM
PO Box 12105
Church Street Station
New York
NY 10249
USA.

There are two main viewpoints from which the new FORTRAN standard might be considered. The FORTRAN programmer on the one hand will compare it with the 1966 standard and with the implementations with which he is familiar. He will be pleased to see that new features have been introduced which permit him to do things which were formerly not possible, whilst still allowing his current standard-conforming programs to run. This we might call the ‘user’s’ viewpoint.

On the other hand, the computing community at large will compare it with other contemporary programming languages. The most striking aspect of the new standard may in this case be the perpetuation (one might say intensification) of the ad hoc tradition of FORTRAN, and the almost complete neglect of experience gained concerning the design of good programming languages. We might call this the ‘designer’s’ viewpoint.

We have tried to examine the new standard from both these viewpoints in the current paper. Indeed, as some of us might be described as ‘users’ and others as ‘designers’, whilst all of us exemplify both attitudes to some extent, we could hardly do other than reflect this dichotomy in the survey.

Upward compatibility

Of the sixteen items of incompatibility between the old and the new standards, the majority relate to features which are rarely used. Others prevent the possible misuse of features which were overlooked in the 1966 standard. The two significant incompatibilities which will require users to make program changes are:

(a) The removal of type Hollerith (from DATA and CALL statements).
(b) The INTRINSIC statements must be used when handing over intrinsic functions as arguments to a subprogram, whereas EXTERNAL must still be used when handing over all other subprogram names.

Regularity

The new standard contains many generalisations. Usually these have improved the regularity of the language, but there are still exceptions to the generalised rules, and in some cases the extended rules have themselves introduced further irregularities.

It is satisfying to find that most mixed-mode expressions are now permitted. Exceptions are mixing double precision and complex (because of the unfortunate omission of DOUBLE PRECISION COMPLEX) and taking the real power of a complex number. Exception rules have been introduced to indicate an undefined value where a DOUBLE PRECISION COMPLEX value would have been formed.

In many places where currently only an integer constant or integer dummy argument is permitted, the new standard permits an integer expression. This is allowed within array declarators, for subscripts and substring bounds in the EQUIVALENCE statement, and when declaring the length of character variables. Unfortunately expressions are prohibited in many situations (for no apparent reason) where they would be useful. In these cases only explicit constants may be used. Examples of this occur in the PARAMETER statement, and in the DATA statement, where such a facility would have been useful both for repeat counts and for actual data values. It is especially annoying that a local array may have its size described by a parametric expression, but then may not be initialised in the same way.

Most of the intrinsic functions now have ‘generic’ as well as specific names. For example, the functions MAX0, AMAX1 and DMAX1 all have the generic name MAX. The compiler chooses the specific function according to the type of argument. These generic names will make the writing of programs and their conversion from one length to another substantially easier. However, this attribute only applies to the intrinsic functions, and has not been generalised so that user-defined functions may be declared to be generic.

We strongly welcome the clear statement that a signed zero is to be regarded as equal to an unsigned zero.

The new language includes specifiers which resemble assignment statements embedded within other statements. For instance, the statement

```
OPEN (UNIT = 7, RECL = N)
```
causes I/O unit number 7 to be opened, the record length to be used being specified by the value of the variable N. Unfortunately, these specifiers sometimes act as if the assignment were right-to-left, and sometimes as if it were left-to-right. For instance, the statement

```
INQUIRE (UNIT = 7, RECL = N)
```
makes an enquiry about I/O unit 7, and causes the variable N to be overwritten with the record length. The same syntax is...

\(^1\)Computer Centre, University College London, 19 Gordon Street, London WC1H OAH.
\(^2\)Computer Department, Rothamsted Experimental Station, Harpenden, Herts.
\(^3\)Micro Computer Systems Ltd., Woking, Surrey.
\(^4\)Computer Science and Systems Division, AERE Harwell, Oxfordshire OX11 0RA.
therefore used in the OPEN and INQUIRE statements with identically opposite meanings. The ‘users’ among us welcome this facility. The ‘designers’ among us find it astonishing that such a feature should be designed by a standards committee. It is obviously the ‘keyword parameter’ feature of such macro assembly languages as /360 BAL. Just as a parameter to a subroutine may be used to supply a value or to receive a value, so the keyword parameter may work either way. However, BAL does not already have an assignment statement with the same syntax, as does FORTRAN.

New features raise the problem of how to organise new syntactic devices with which to represent them. PL/I overworked parentheses; FORREV is tending to do the same with asterisks. The old standard used asterisk for multiplication, exponentiation, and as a repetition factor in DATA statements. One might consider that the length attribute for CHARACTER variables has a similar meaning to a repetition count, so justifying

\[
\text{CHARACTER}\ast n \text{ var}
\]

However, asterisk is also used as the format specifier for list-directed I/O, as a dummy argument for variable return, as the first character in a comment line (as an alternative to ‘C’), as an indication that a character string argument may be of variable length, and as a flag for a label passed as an argument in a CALL statement.

Characters
The facility for manipulating characters as well as numbers is long overdue. However, the new type CHARACTER is not a confirmation of any de facto standard, but an innovation by the standards committee, and as such, there is no body of experience with its use.

Along with this new type we have a crop of new restrictions. Zero length character strings (e.g. arguments to subprograms) or zero length substrings are not permitted. Statement functions may not be of character type and there are restrictions on character expressions forming part of the actual arguments to statement functions.

The only character operator is concatenation (//). A substring notation is provided for variables, but this cannot be applied to expressions.

Storage used for holding characters may not at any time be used for holding other types since there is a total ban on such associations taking place through COMMON, EQUIVALENCE or argument association. While we accept that this ban aids portability, we regret its existence because it prevents reuse of inactive storage and prohibits efficient programs which handle groups of characters in integer storage. Some of us think it unfortunate that such a complete separation has been made between character variables and arrays on the one hand and all other values on the other. The relation between the normal storage unit and a character unit obviously depends on the processor, but this fact is already faced within FORREV when unformatted I/O has to cope with mixed values.

The omission of a more fully defined collating sequence for characters reflects the current incompatibilities between the existing processors on the market. It is to be regretted that the ANSI X3J3 committee neither agreed on a national or international standard for a full collating sequence, nor provided a method of mapping characters into small integers efficiently.

Input/Output
Three new types of file have been introduced, the direct access file, the stream file and the internal file. These are welcome additions. Addressing direct access files is extremely convenient since each record is associated with a record number which must be quoted in the READ and WRITE statements. Internal files permit much greater use to be made of the power of FORMAT processing; for instance a record may be transferred into an internal file and read from there with several different FORMATS.

List-directed (free-format) input-output has been introduced for both stream and sequential files, and is a welcome, powerful feature. On input adjacent items are normally separated by one or more blanks or the end of a record and a slash is used to terminate the set of items to be read by a single input statement. Repetitions are permitted as in DATA statements. Null entries, which leave the value of the associated variable unchanged, are also permitted. On output sensible decisions are made, consistent with the possibility that for stream files such output may be required to become subsequently free-format input, and that for sequential files such output will be directed to a printer.

The removal of the facility for reading into a Hollerith string within a format is to be welcomed, as it eliminates a little-used feature and thereby enables a compiled format to be placed in a read-only area of store.

Apostrophes may now be used to delimit literals in FORMAT statements so that counting characters is no longer necessary. Such literals now take exactly the same form as character constants. Embedded apostrophes are represented by two consecutive apostrophes, which can lead to some very clumsy results. For example, the character string 'ISN'T' is represented by the character constant "'ISN'\"T'\".

FORREV now permits (in OPEN and CLOSE statements) the specification of information such as file name, record length, status etc., which has previously had to be given in a command language (or JCL). Our opinions differ as to whether this will aid portability. A special problem arises with the file name. Hitherto FORTRAN has exclusively used machine-independent unit numbers when referring to files. The file name is a character string whose form is processor dependent. FORREV even recognises explicitly that a processor need not permit named files. If the name is read from a data card, a program may still retain its portability, but the onus is on the user to keep it so.

The new standard contains a facility for programs to survive an input/output failure by use of the specifier ERR = label. Processing continues at the label specified and no further use can be made of that I/O unit. This facility can hardly be said to provide effective error recovery from failures due to I/O errors or to incorrectly formatted data.

The end of an input file may now be detected by means of the specifier END = label in a READ statement. When the end of file is reached, control is passed to the specified label. This may seem a trivial expansion to the language, but it removes the anomaly under the old standard that end of file could be written but not read. Those who have had experience of adding an extra record to the end of a file in order to supply a terminator for FORTRAN, and then forgetting to remove this record when the same file was to be read in another way, will certainly strongly welcome this feature.

The INQUIRE statement may be used to retrieve information about a file or a unit. Throughout the revision period, the FORTRAN Specialist Group has complained about INQUIRE (including its spelling) but to no avail. We have suggested alternative syntax to make clear when a specifier is supplying information, and when it is requiring information to be placed in a variable, but this has not been acted upon. There are two variants of the INQUIRE statement, one in which the file is specified (INQUIRE by file) and one in which the unit is specified (INQUIRE by unit). The remaining eleven specifiers are the same for each variant. The possible effects depend heavily on whether the file is direct access, stream, sequential, formatted, unformatted or connected to a unit. It is notable that the descriptions of both the INQUIRE and OPEN statements each necessitate the use of the word 'if' about 30 times in two to three columns of text. The specifiers NAME =
and NAMED are surely too close in form to avoid confusion. The specifier MAXREC= is used in the OPEN statement to give a limit to the number of records a file may have, and apparently is available for use with any kind of file. In the INQUIRE statement it is only available for direct access files, and is used to ascertain the maximum record number. We can only think that this is an error in the FORTREV document, and that MAXREC= should always be limited to direct access files. Another specifier which is used both to give information in the OPEN statement and to retrieve it in an INQUIRE statement is RECL= which refers to the length of records. In this case FORTREV is quite explicit that the specifier may only be used (in either statement) for direct access files, which we find an astonishing limitation. This whole area is one of great confusion. These same facilities would be better provided within the existing syntax by means of a set of library functions. It is perhaps not inappropriate that the INQUIRE statement should have been designed by a subcommittee called the 'I/O Ad Hoc Group'.

Control structures

The DO loop has been extended considerably in power and usefulness. The index may be any integer, real or double precision variable, and the initial, terminal and incrementation parameters may be any integer, real or double precision expressions. Negative increments are permitted. At the beginning of the loop the parameter expressions are evaluated and the interaction count determined. The latter governs how many times the loop is to be executed before normal termination occurs, and may be zero. The value of the index on normal termination is now defined (as one increment more than its last value within the loop). Many people would have liked to see the awkward 'extended range of a DO loop' removed from the standard. Although the term has disappeared, the concept has in effect been strengthened. Jumps into 'active' loops are now permitted, an active loop being one left earlier by a transfer of control, whose index has not become undefined or redefined, and which is not in the range of another DO loop which becomes inactive or whose DO statement is executed. From the user's point of view, this is easier to understand than the extended range rule, but it still encourages bad programming practice and exemplifies what we mean about the ad hoc tradition of FORTRAN being perpetuated.

Extensions to both the assigned and computed GO TO statements have been made. Being permitted to omit the list of statement labels from an assigned GO TO makes it much more convenient as the terminal statement of an open coded subroutine. The computed GO TO may now be controlled by any integer, real or double precision expression. If the result is out of range, the GO TO is treated as a CONTINUE.

Whilst the whole issue of control structures is subject to considerable debate, the X3J3 committee have played safe by failing to introduce any of the numerous proposed alternatives into the standard. Although they have by this means avoided introducing extensions which might be queried later on, their failure to provide the basic control structures for structured programming (such as an IF . . . THEN . . . ELSE facility) may be regarded by some as a damning indictment of the whole standard. (As a result of user pressure, X3J3 are now considering adding such features at the very last minute.)

Moreover, the confirmation of the existing de facto standards for alternative RETURN and the ENTRY statement will mean that the standard countenances the multiple-entry, multiple-exit subroutine which is anathema to the advocates of structured programming.

Other extensions

Traditionally FORTRAN variables local to a subroutine have retained their values between calls unless an overlay intervened. In decreeing that such variables became undefined the old standard was allowing for both the overlayed and non-overlayed cases. The new standard provides a SAVE statement which permits specified (or all) variables, arrays and common blocks to retain their values between calls.

DIMENSION statements for dummy arrays may now contain integer expressions in the subroutine arguments. This will be particularly useful in the subroutine library situation, where currently one may have to ask the user to supply unnecessary arguments such as the values of N and N + 1, simply in order to have the right arguments for variable DIMENSION statements. We welcome, too, the facility to define a lower bound for a subscript, chiefly because this allows the user to define it as zero.

The end line now becomes a statement in its own right, and can be labelled and executed (acting as a STOP in the main program and as a RETURN elsewhere). This is a welcome improvement.

Omissions

For numerical applications, which must constitute a large proportion of FORTRAN programs, the major obstacle to portability has been the need to convert from single to double precision when switching to a machine which provides less accuracy. The introduction of the IMPLICIT statement and generic functions will undoubtedly simplify the process of conversion, but these do not provide a complete solution to the problem. The conversion of constants is facilitated by the PARAMETER statement (which allows a symbolic form for a constant to be declared), but as with the IMPLICIT statement, it only applies to the current subroutine.

The omission of DOUBLE PRECISION COMPLEX makes any program which uses COMPLEX potentially non-portable.

The PARAMETER statement is a poor substitute for thorough-going macro facilities. For one thing, in order to preserve the modularity of FORTRAN, it only applies to one program unit, whereas it will mostly be needed throughout a compilation. Another extension which would make the PARAMETER statement far more worthwhile would be to allow the value to be an expression consisting only of constants and previously defined symbolic parameters.

There are real problems in respect of the provision of workspace for library (and other) subprograms, and it is most disappointing to find that FORTREV does little more than the old standard in this area. A subprogram is likely to need several arrays of varying types and of variable orders, and it may be possible to save significant amounts of space by making some of them share storage. One possible solution would be the provision of a stack of workspace, such as is provided in ALGOL 60, but such a concept runs counter to FORTRAN tradition. Failing this solution, which would operate without the need for handing over arrays as arguments, the user would often like to provide just one work array to be variably partitioned by the subprogram. This can be done under both the old standard and under FORTREV by calling the subprogram indirectly through another subprogram which uses elements in the work array as actual arguments corresponding to the required work arrays in the subprogram required. It is very unsatisfactory that such facilities cannot be obtained directly (for instance, by a dynamic form of EQUIVALENCE) and in any case does not provide for the sharing of storage by work arrays of differing types.

Conclusions

In the face of increasing competition from other programming languages, FORTRAN has held on to its position. Not even its keenest advocates could claim that this is due to the power, beauty and regularity of the language. Rather, FORTRAN has stayed the course (warts and all) because of its portability.
simplicity, efficiency in execution and widespread implementation.

The revision of the FORTRAN standard was long overdue. Now that FORTRAN has emerged, the question to be asked is, will it enjoy the same success as the vintage product? This obviously depends on a number of factors. The new language can no longer be described as simple. The language is certainly more powerful, but so often the extra power is built in by means of ad hoc extensions. A few ad hoc rules are manageable; more than a few become overwhelming. The irregularities not only constitute an extra load for the learner and user; they add to the problems of the implementor. It may be that the new language will fail to secure the widespread implementation which is predecessor had, largely because of its amorphous accretions.

In the days when FORTRAN first became king, there were no competitors. Now there is a plethora of them, and many programmers who have enjoyed well-designed languages with greater power than FORTRAN will hardly want to change unless there are considerable advantages to be gained. Portability could be one such advantage, but it depends on widespread implementation, which may not happen. It seems that the main driving force behind FORTRAN will be the impetus (or momentum, or even inertia) derived from the success of FORTRAN, rather than much merit inherent in the new proposals themselves.

**Postscript**

Individuals who wish to send their own comments on the proposed new standard should write to:

Lloyd W. Campbell  
X3J3 Secretary  
BRL-CSD, Bldg. 328  
Aberdeen Proving Ground  
MD21005  
USA.

---

**A note on estimating hit ratios for direct-access storage devices**

M. T. Pezarro  

Computer Laboratory, University of Cambridge, Corn Exchange Street, Cambridge CB2 3QG

---

A formula is presented for estimating the expected number of higher level record groupings (blocks, tracks or cylinders) hit while accessing a certain number of distinct records in order of their device address. The recent appearance in the literature of two other conflicting formulae, also examined here and which apparently answer the same question, prompted the development of this formulae.

(Received April 1976)

A common question arising in the context of skip-serial processing of disc or drum files is how many higher level storage cells will be hit in the course of accessing a certain number of records. Higher level groupings typically consist of blocks, tracks or cylinders. More precisely, the question is: if $k$ distinct records with known addresses are accessed in serial order from a file of $n$ records, which is grouped in higher level storage cells of $b$ records each, then what is the expected number of cells hit, assuming that the $k$ records are randomly distributed over the file of $n$ records. Performance prediction for inverted file organisations, in which lists linking all records having the same key-value for a particular key are maintained, is one application which gives rise to this question. These lists are usually kept in order of device address so that all records can be accessed in a single serial scan of the device, thereby ensuring that if a cell contains more than one record it need be retrieved only once and if the file is stored on a disc device then seek time is minimal.

In deriving a formula to estimate the expected number of cells hit one must bear in mind the following point. The fact that the $k$ records are distinct and are accessed in serial order by device address implies that one is only concerned with how $k$ records, chosen at random from $n$ records, are distributed among $m$ cells (where $m$ is the smallest integral number of cells sufficient to hold all the objects). It is the distribution of the $k$ records which determines how many cells they occupy. A suitable formula may be developed by considering an analogous problem. Suppose there are $m$ boxes holding a total of $n$ objects and each box has a capacity of $b$ objects. If $k$ objects are chosen at random, what is the expected number of boxes holding the $k$ objects? Here higher level storage cells correspond to boxes and records to objects.

Now, the expected number of boxes holding the $k$ objects is the product of the number of boxes times the probability that a box contains at least one of the $k$ objects (i.e. the mean of a binomial distribution). Consider the probability that a box contains none of the $k$ objects. This is the number of ways of placing $k$ objects among the other $m-b$ boxes divided by the number of ways of placing the $k$ objects among all $m$ boxes.

If we define the number of places among all $m$ boxes as $p$, where $p = m * b$, and let $C$ be the number of boxes (cells) hit, then

$$E[C] = m \left[ 1 - \binom{p-b}{k} \left/ \binom{p}{k} \right. \right]$$

(1)

where $k > p-b$ define $\binom{p-b}{k} = 0$.

Using $p$ instead of $n$ means making the simplifying assumption that the records are uniformly distributed among the cells. This formula gives the correct result for two obvious boundary values for $k$, $k = 1$ and $k = n$. If $k = 1$ then $E[C] = 1$ since only one cell can be hit in accessing one record. When $k = n$ then $E[C] = m$ because all cells must be hit in accessing all records.

Two other formulae that seem to address the same question have appeared in the literature recently. One due to S. J. Waters (1975) is:

$$E[C] = m[1 - (1 - k/m) \uparrow b]$$

(2)