CERN-Data Handling Division
DD/84/7 Rev. 2
March 1985
M. Metcalf

HAS FORTRAN A FUTURE?

(Presented at the Europhysics Conference on Software Engineering, Methods and Tools in Computational Physics, Brussels, 21-24 August 1984)

DD-vf

HAS FORTRAN A FUTURE?

Michael Metcalf
Data Handling Division
CERN, CH-1211 Geneva 23

Abstract

For over 25 years FORTRAN has dominated all other programming languages in the field of scientific and engineering computation. Although much denigrated by computer-science purists, it has consistently shown itself to be attractive to scientific users because its basic simplicity and power of expression appeal to non-specialists. Can this situation continue? Will the introduction of FORTRAN 77 lead to an upsurge in the use of the language, providing it with momentum sufficient to carry it through to the end of the decade? Shall we witness a conflict between FORTRAN 8x and ADA?

This lecture will take stock of the present status of FORTRAN and describe its likely development, before going on to speculate on possible trends until the turn of the century.

1. FORTRAN today

FORTRAN has a long if not always glorious history. As the first high-level programming language [1] it spread rapidly in those non-commercial application areas for which it was well or reasonably well suited, if only for a lack of competition. Although in the meantime other languages suited for scientific use have appeared, they have failed to attract the same following, possibly because they not only had to compete with an already well established product, but also themselves lacked the twin features of FORTRAN which have kept it in the lead so long - ease of use and efficiency of object code (see also [2]).

Just at that point in the late 1970's when FORTRAN seemed, intellectually, to be dying on its feet, a new revision of the standard was published [3] and the new features of FORTRAN 77 breathed life again into the expiring body, although it took some time for compilers to become universally available. Whilst PASCAL has made some inroads into the smaller applications and is in widespread use as a vehicle for teaching programming, a wholly admirable situation, in the real-world of big applications FORTRAN remains dominant, and is often taught as a follow-on language to PASCAL, in order to prepare students for their later careers.

At the present time I have the impression that we are on the steep part of the changeover curve between the old 1966 standard and the new standard. Many vendors offer only FORTRAN 77 compilers, or provide, but no longer support, compilers based on the old standard. A survey of European IBM sites I conducted at the beginning of 1984 [4] showed that about two-thirds of the use of FORTRAN was with the VS compiler, though that may hide some use of FORTRAN 66 as it is contained in that compiler as an option. On the CERN mainframes the use of FORTRAN 77 in what is claimed to be an ultra-conservative community has reached 40%, and all new programs will be written to the new standard. On the smaller CERN machines, FORTRAN 77 is used exclusively. An introductory course and guidelines for its use with respect to portability and optimization have been published [5], [6], [7], [8], and the first three of these references have appeared combined in a more detailed form as [9].

As well as there being strong evidence for a changeover in the use of FORTRAN, there is also some indication that the use of FORTRAN in absolute terms is growing. It is very difficult to measure the use of different programming languages, as very few statistics are available, and even then it is unclear whether the relevant parameter is lines of code written, number of compiler calls, total execution time, or whatever. It seems, nevertheless, that the spread of FORTRAN usage into microcomputers provides an extension of its use into a new area. At the same time there are signs that the big manufacturers are strongly supporting FORTRAN language and compiler developments. This may not necessarily be what they would like to invest in, but is in their commercial interest as a response to perceived user demand.

2. Major Flaws

In spite of the strength of FORTRAN's position, it is not regarded even by its most outspoken advocates as being an ideal tool for scientific programming. This is a problem which besets all programming languages to some extent, and results in part from the slowness of language standards development, and the slowness with which the new standards are made available to users in the form of reliable compilers. Thus one's expectations are always in advance of what is available. Even viewed in that light FORTRAN has some notable drawbacks, resulting mainly from its position as an early high-level language combined with the conservative nature of the last revision.

In the past, FORTRAN's deficiencies were alleviated by the widespread use of language extensions provided in non-standard ways by compiler vendors, and by the development of many pre-processors. These latter were often implemented by large organizations as a means of providing structured programming constructs for in-house use. In 1975 a survey by Reifer and Meissner compared no fewer than 55 such products [10]. The introduction of FORTRAN 77 was the death knell for most of these pre-processors as it contained a sufficient number of worthwhile new features to make their continued use unattractive. The two most significant new features in this respect were the IF...THEN...ELSE construct and the character data type. Since then, only two new FORTRAN pre-processors have appeared in the literature, but neither is based on the new standard, generating FORTRAN 66 not 77 source code [11], [12].

A contemporary list of FORTRAN's deficiences begins with one of its most well-known and most used features :

i) Storage association through the COMMON and EQUIVALENCE statements. The first of these makes it impossible to define any scope of variable other then local or global; it is impossible to restrict the scope of a variable to a group of subprograms only. The second allows potentially dangerous aliasing of locations, even between variables of different data types.

ii) Lack of data structures. FORTRAN allows only two simple data structures, the array and the COMMON block. The former is restricted to elements of a single data type, and the latter cannot be manipulated as an entity. The high-energy physics community has designed or produced no fewer than five packages to overcome these two problems. (We note, that in these important application packages, involving the management of data structures containing elements of several types, the EQUIVALENCE statement is essential).

iii) Rigid source form. The FORTRAN source form is based on the out-moded punched card, and is inappropriate for entering and editing source code from a terminal.

iv) Poor error handling. FORTRAN allows detection and recovery from parity and end-of-file conditions on external files, but the handling of other hardware conditions such as overflow and underflow is not specified in the language.

v) Lack of a bit data type. The bit is the most fundamental data type, and has to be manipulated by many programs, especially in high-energy physics, in a totally non-standard and non-portable way, as this data type does not exist in the language.

vi) Lack of means to control the precision of computer arithmetic. The real and double precision data types do not have guaranteed ranges or degrees of significance, and there is no means to ensure that computer arithmetic is stable across a range of computer architectures.

vii) Lack of environmental enquiry facilities. Linked to the previous point is the inability to pose questions about the run-time environment of a program, for instance the largest available integer or real quantity, or the time of day.

But these points are mere technical details compared with the more fundamental criticism of the Nobel prize-winning physicist K. Wilson of Cornell University [13]. He attacks the whole concept of continuing to program in languages such as FORTRAN (he picks on FORTRAN because of its exclusive use in the physics community), and proposes instead transformational systems which would combine a numerical method with systems of equations, say, to produce what might well be FORTRAN code, but as an intermediate language. He complains that whereas information in an advanced text book is organized in a logical fashion, the same amount of information in a FORTRAN program is distributed throughout every subprogram and DO-loop in an unintelligible way. His criticism has been expressed also in a letter to the FORTRAN standards committee, X3J3, in which he adds that FORTRAN and other existing high-level languages are totally out of tune with modern developments in hardware, especially graphics, and in programming environments and he urges the committee to stop its work! [14]. These scathing comments cannot go ignored, but are based in part on two fallacies. The first is that it is not possible to write clear programs in FORTRAN, the second is that the next FORTRAN standard is just "tinkering" with the old, whereas in fact it is a major advance on the present standard bringing, as we shall see below, important new facilities which are of great benefit to scientific programmers.

The fact that a significant investment has been made in a FORTRAN environment, the Toolpack project [15], shows how much importance is attached by many organizations to providing just that type of programming support which Wilson believes to be so necessary. Although Toolpack has yet to be formally announced, over 700 requests for it have already been received, and this is yet another manifestation of the continuing interest in FORTRAN, typified also by the (unpublished) talk by the computing veteran D. McCracken at a recent SHARE meeting entitled "Why Engineers should learn FORTRAN in the '80's". The potential impact of Toolpack merits further discussion.

3. Toolpack

The traditional method of programming in FORTRAN has been to construct a card deck, or more recently a file of card images, which is presented to a compiler and loader to prepare the program for execution. The compiler and loader were often the only software tools, as we now call them, used by many FORTRAN programmers, although in high-energy physics we are used to maintaining our codes under a source code management system. Such primitive methods of working are now regarded as totally inadequate, but the only progress which has been achieved and made available to users of most mainframes is the replacement of the key-punch and card reader by a file editor and terminal. In the best case the compiler may offer interactive debugging facilities, and only very few users numerically speaking have access to the sophisticated facilities offered on the even more advanced personal workstations.

This lack of progress is in strong contrast to the widespread realisation that the complete environment in which a programmer works is just as significant for productivity as the language and compiler he employs, if not more so. The two most significant developments in environments are in the design of the Ada Programming Support Environment (APSE), for which there currently exists no implementation, and the Toolpack Integrated System of Tools (IST). The Toolpack project is a joint enterprise led by the Argonne National Laboratory and including American laboratories and universities, as well as NAG in the U.K. It sets out to provide a file system which allows various "views" to be taken of, say, a FORTRAN subprogram. For instance, seen by a FORTRAN intelligent editor, the text file will consist of source code, of lexical entities and of a parse-tree. Seen by a transforming tool, the text may be instrumented to check on execution counts and array bounds or to test embedded assertions in order to help demonstrate the correctness of the program. All these possibilities rest on a complex file base containing different representations of the program. The facilities planned for a final version (only some are ready now as prototypes) are :

i) A FORTRAN intelligent editor, which will accept abbreviated FORTRAN keywords, position statements correctly, check input for syntactic and semantic consistency, allow searches for specified variables, and allow editing within specified regions of a program, such as the scope of a DO-loop or a given subroutine.

ii) A formatter, which will lay out the code in a neat and consistent fashion.

iii) A structurer, to extend FORTRAN's structured constructs.

iv) A dynamic testing and validation aid, enabling programs to be instrumented to provide trace, summary and error information, as well as checks on assertions. The large amount of output provided by this tool will require others to extract the required information.

v) A dynamic debugging aid offering such facilities as snapshots, breakpoints and single-stepping, with the ability to examine the current values of variables.

vi) A static error detection/validation aid, based on lexical, syntactic, semantic and data flow analysis, and enabling the detection of many coding errors before any execution is attempted.

vii) A static portability checking aid, checking for non-standard features.

viii) A documentation generation aid, by which it is planned to use the static and dynamic analyses to produce some form of documentation.

ix) A program transformer, providing a means of translating from one dialect of FORTRAN to another, to change the arithmetic precision or to create special-purpose control or data structures. This will be achieved using three specific tools, known as a template processor, a macro processor and a correctness-preserving transformer.

Toolpack is an ambitious project which, like most ambitious software projects, is running behind its original schedule. The current plans are to have early releases this year, suitable for preparing programs of up to about 10,000 lines. In spite of these limitations, it is an important attempt to embed FORTRAN in a powerful framework, and the long-term fate of FORTRAN may well depend as much on its environments as on its own strengths and weaknesses as a language.

4. FORTRAN 8x

We are now in the era of FORTRAN 77, and have speculated that the new standard combined with the huge existing investment in FORTRAN programs will, by themselves, keep the language alive for a long time. In this competitive world, however, it is not sufficient to stand still, and those who regard the FORTRAN style as the one most suited to their own needs and working methods have to consider how the language should evolve, regardless of other developments such as Toolpack, but taking into account the incessant claim of devotees of the PASCAL family of languages, and especially of ADA, to possess the key to the ultimate truth in programming methodology.

The main task of standardizing FORTRAN is entrusted to an American National Standards Institute (ANSI) committee known as X3J3. The rules of ANSI require that a standard be confirmed, withdrawn or revised after a five-year period, and X3J3 decided shortly after the introduction of the new standard in 1978 to aim at a revision of the language, and now aims to publish a new draft standard for public comment in 1985, with a final publication in 1988. The language defined by that new standard, currently known as FORTRAN 8x, will not be a standard in the classical sense, choosing and defining one implementation of a feature from a number of existing versions, but rather a total revision of the language. Nevertheless, the goal is to remain backwards compatible with the current standard and, to the largest extent possible, to introduce features which have at least been demonstrated in the context of other languages, even if not in FORTRAN dialects themselves. Thus, FORTRAN 8x should be modern, reliable and portable.

The new standard will effectively be a superset of FORTRAN 77, but many of the features of FORTRAN 77 will be "deprecated", their functionality either being connected with storage association and available in a more modern form in the new features, or redundant, or otherwise regarded as bad practice (see Table 1). Thus, whilst FORTRAN 77 programs will be guaranteed to work under FORTRAN 8x compilers, and whilst it will be possible to continue to use deprecated features until they are removed in a subsequent revision cycle, it will be possible, by avoiding deprecated features, to write code which has a totally different appearance to the FORTRAN with which we have been familiar for nearly three decades. This code will, however, be guaranteed a very long life.

Table 1

Deprecated features of FORTRAN 8x

Storage association

- EQUIVALENCE statement
- COMMON statement
- BLOCK DATA
- ENTRY statement
- Assumed size arrays
- Passing a scaler entity to a dummy array

Redundant

- FORTRAN 77 source form
- DIMENSION statement
- DATA statement
- DOUBLE PRECISION statement
- Arithmetic IF
- Computed GO TO
- FORTRAN 77 DO statement
- Statement functions
- Specific names for intrinsic functions

Other

- PAUSE statement
- ASSIGN and assigned GO TO
- Alternate RETURN

The list of features in Table 1 clearly implies the addition of many powerful new features to replace and extend them. The rest of this section consists of a rather brief summary of some of the more important of these new features, although to do them justice requires a book rather than a section. The powerful data type and array processing features are described separately in the two following sections.

We note that D. Williams has made a powerful defence of the alternate RETURN [16], and it remains to be seen whether his plea will be heeded by X3J3.

4.1 Source form

A new source form will allow free form source input, without regard for columns. Comments may be in-line, preceded by an exclamation mark (!) and continuation lines are indicated by an ampersand (&)

on the previous line. The character set is extended to include the full ASCII set, including lower case letters (which in FORTRAN syntax will be interpreted as upper case). The underscore character is accepted as part of a symbolic name, allowing one to write variable names such as CROSS_PRODUCT, up to 31 characters in length.

4.2 Significance of blanks

The blank will become a significant character which may be used as a separator and which, therefore, may not be embedded in a name or keyword. Thus

```
END FILENUNIT
```

must be written as

```
ENDFILE NUNIT
```

to be an acceptable statement.

4.3 Attribute oriented declarations

The present standard requires declarations to be made as one attribute followed by a list of variables possessing that attribute. The new standard will require declarations to consist of a list of variables possessing a common set of attributes, following a list of those attributes :

```
REAL, ARRAY(50), SAVE, INITIAL(50*1.)::A, B, C
```

In this example, we note the new way in which variables may be initialised by an INITIAL rather than a DATA statement.

4.4 Precision specification

New attributes which may be declared are the range and precision of real variables, allowing true portability of numerical software, with guaranteed results. The long form of the declaration is illustrated by

```
    REAL, PRECISION10=12, EXP_RANGE=100, ARRAY(10)::X
```

4.5 Environmental Inquiry

The attributes of the environment can be interrogated using a set of new intrinsic functions. These provide information such as the smallest and largest positive numbers (real or integer), the absolute spacing of real numbers near the value of the argument to the function, and the model base for real numbers. These functions will also ease the task of those writing portable numerical software. For instance, as a convergence test in a fit procedure one will be able to write

```
    IF (A-B.LE.EPSILON(A)) THEN
```

rather than having to specify an actual value for "epsilon", which may not be achievable on some machines but which is too coarse for others.

Another set of intrinsic procedures will give access to the local date and time, and to the difference between that time and a reference time (GMT).

4.6 IMPLICIT NONE

The IMPLICIT NONE statement removes the FORTRAN default typing of variables and functions, and thereby makes type declarations obligatory even for real and integer entities.

4.7 BIT data type

A new BIT data type is introduced. Arrays of bits may be declared as

```
     BIT B1(8), B2(32), B3
```

and operations BAND, BOR, BNOT and BXOR performed upon them and bit constants :

```
     B1 = B1.BAND.B2(9:16)
     B3 = B1(1:1).BXOR.B'1'
```

Assignments such as this rely on the array processing features to be described in Section 5 below. Functions to transform bit arrays into integers or logicals and vice versa are available.

4.8 Extension to type character

Some extensions to the facilities associated with the character data type will allow null strings, some new intrinsic functions and the ability to overlap the two sides of an assignment, making

```
     A(:5) = A(3:7)
```

a legal statement.

4.9 CASE construct

The new CASE construct allows the execution of one block of code, selected from several, depending on the value of an integer, logical or character expression. An example is

```
SELECT CASE (ITEMP.EQ.100)
  CASE (.TRUE.)
       BOIL=.TRUE.
       LIQUID=.FALSE.
  CASE (.FALSE.)
       BOIL=.FALSE.
       LIQUID=.TRUE.
END SELECT
```

A default CASE clause is allowed. Although there is some measure of overlap between the functionality of the CASE construct and the existing (and unchanged) IF...THEN...ELSE construct, the CASE allows cleaner code to be written in many circumstances, and also requires that one of the conditions be met (or that the DEFAULT clause be present). It is a replacement of the computed GO TO.

4.10 Loop construct

The introduction of real DO-loop parameters and indices into FORTRAN 77 is now regarded as having been a mistake. The new loop construct has the general form

```
[name] DO [(control)]
    block of statements
REPEAT [name]
```

(where square brackets indicate optional items). The control parameter, if unspecified, implies an endless loop; if present, it may have one of two forms:

```
    i = intexp1, intexp2, intexp3
```

or

```
    intexp4 TIMES
```

The optional name may be used in conjunction with CYCLE and EXIT statements to specify which loop in a set of nested loops is to begin a new iteration or which is to be terminated, respectively.

4.11 Enhanced CALL

The calling procedure in FORTRAN is fairly primitive. For each dummy argument in a called subprogram there must exist a corresponding actual argument of the same type in the call. The new standard will allow arguments to be defined by keywords as well as their position in the sequence, and will allow them to be optional. Thus, a subroutine with an initial line like

```
        SUBROUTINE NAME(A, B, N)
```

might be called as

```
        CALL NAME(N=I, A=X)
```

where the presence or absence of B can be established in NAME by calling the intrinsic function PRESENT, and A and N are defined by their keywords.

4.12 Recursion

Another new feature associated with procedures is the ability to define recursive functions and subroutines, particularly useful in such applications as list processing and multi-dimensional integration.

4.13 Internal procedures

The old statement function, restricted to a single statement, has been generalized to become an internal procedure, either a function or a subroutine. This mechanism is not only more useful than the one it replaces, but includes a facility to overload operations and the assignment operation in a way which is essential for exploiting the full power of the user defined data types which we will meet below. Where a single operator symbol is used for operation on data of different types, internal procedures corresponding to each data type, but having the same name, may be defined within one program unit, and the correct one will be assigned by the compiler.

4.14 Extensions to I/O functions

There are only two new important features proposed to FORTRAN's already very extensive I/O facilities (apart from the obvious extension to handle the new data types to be described below). The first is the ability to position a file using the OPEN statement, and to specify which actions may be performed on it (e.g. read-only). The second feature is a name-directed I/O facility designated by a double asterisk, e.g. :

```
READ (UNIT, **) A, I, X
```

which can read a record like

```
X=4.3, A=1.E20, I=-4
```

containing the named items in any order.

5. Array processing

One of the two major new facilities in FORTRAN 8x is its array processing syntax and associated features. (The second, modules and derived data types, will be dealt with in the next section.) The wealth of new features in this area makes it difficult to summarize them in a short publication, and the description which follows can provide only a glimpse of their power. More details are given in ref [17]. Before beginning, however, it is worth mentioning the two main justifications for developing such facilities at all.

The first stems from a need to simplify and extend the FORTRAN syntax for handling arrays. This is achieved by defining an array as an object which can be treated as a whole, rather than on (mainly) an element-by-element basis, as is the case in existing FORTRAN. Tests using the new notation have shown that reductions in the length of code written to solve a given problem can be up to a factor eight.

The second justification is connected with the increasing use of array and vector processors for tackling large-scale numerical problems in science and engineering. A major difficulty in writing FORTRAN code for these computers has been the limited ability of their compilers to detect vectorizable sections of code. This has meant either accepting a lower level of performance than can potentially be achieved, or delving into messy, non-standard vector extensions or even assembler language. The new extensions make obvious to a compiler the vector nature of the code, and permit a high degree of portability not only between different models of vector processors, but also between vector and scalar processors in general. On each hardware model the compiler can generate that object code which is the most efficient. This efficiency extends to the optimal generation of temporary storage for intermediate arrays, as these no longer have to be declared explicitly by the programmer, with a possibly consequent negative effect on performance when these are additionaly involved in the storage association inherent in the use of EQUIVALENCE and COMMON statements.

5.1 Arrays as objects

An array is defined to have a shape specified by its number of dimensions and the extent of each dimension. Two arrays are conformable if they have the same shape. The operations, assignments and intrinsic functions are extended to apply to whole arrays on an element-by-element basis, provided that where more than one array is involved they are all conformable. Where a scalar value is involved, its value is distributed as necessary. Thus we may write

```
REAL, ARRAY(5, 20)::X, Y
REAL, ARRAY(-2:2,20)::Z
.
.
Z = 4.0*Y*SQRT(X)
```

In this example we may wish to include a protection against an attempt to extract a negative square-root. This facility is provided by the WHERE...ELSEWHERE construct :

```
WHERE (X.GE.0.)
      Z = 4.0*Y*SQRT(X)
ELSEWHERE
      Z = 0.0
END WHERE
```

which tests on an element-by-element basis. An assignment statement inside a WHERE block implies an ordered execution over the individual elements. A FORALL statement allows element-by-element processing over a specified index range, but without regard to order.

5.2 Array sections

It is clearly not always appropriate to address a whole array, and a means is provided to select sections through an array. Such sections are themselves array valued objects, and may be used wherever an array may be used, in particular as an actual argument in a subprogram call. Array sections are selected using a colon (:) notation. For an array

```
REAL, ARRAY(-4:0, 7)::A

A(-3,:)  selects the third row of A
A(:, 3)  selects the third column.
```

A triplet notation similar to that used for DO-loop parameters permits references to non-contiguous array elements:

```
A(0:-4:-2, 1:7:2)
```

selects in reverse order every second element of every second column of A. As sections such as this can be passed as actual arguments, clearly the called subprogram can make no assumptions about storage association in the passed section, but on the contrary requires a means such as a dope-vector to describe the nature of the passed array. For this reason an array is passed as a whole object, and not referenced simply by a single address acting as a pointer to the first passed element, with implied assumptions about the positions of all other array elements with respect to that first one.

5.3 Dynamic arrays

A further facility for handling part of an array, and which goes beyond the array section, is provided by the IDENTIFY statement. This allows a dynamic aliasing of part of an array as in

```
IDENTIFY (DIAG(I) = X(I,I), I=1:100)
```

which dynamically defines a vector DIAG of length 100 which contains the diagonal elements of the array X. From this point on in the program, DIAG may be used just as if it had been declared as an array in a declarative statement. The index I in this example has no scope beyond that of the statement in which it appears.

Other dynamic facilities for arrays are the ability to ALLOCATE and FREE local arrays, the introduction of automatic arrays (local arrays with variable dimensions), useful as local scratch storage, and the possibility to pass as an argument an assumed-shape array, specified in the called subprogram as, for instance,

```
REAL A(:,:,:)
```

The extents of the dimensions can be determined if necessary by the UBOUND intrinsic function. Where required, the shape of an array may be changed using the RESHAPE intrinsic function, and for the particular case of changing a multi-dimensional array to a vector and vice versa the PACK and UNPACK intrinsic functions are foreseen.

5.4 Other features

The list of new intrinsic functions concerned particularly with arrays goes far beyond the few just described. A full list is given in [16] or [8], and a few worthy of mention here are those to form the sum and product of the elements of an array, to find the smallest or largest element, to merge arrays and to shift the elements of an array. These powerful new functions operate, where appropriate, optionally under a mask of logical values.

The last feature to be mentioned is the ability to define array valued constants, which are enclosed in square-brackets:

```
[1,1,2,3,5,8]
```

and which may be manipulated in the same way as any other array.

From this short summary it is not possible to appreciate the full power of these array features, which is best illustrated by extensive examples. The fact that the new company ETA Inc. has announced that it intends to incorporate these facilities into the FORTRAN compiler for its GF-10 supercomputer does mean, however, that we can hope to see actual examples of working code long before FORTRAN 8x becomes generally available.

6. Modules and data types

The last set of new features to be outlined are not only individually very useful, but in combination provide a means whereby a programmer can define his own data types and operations on those types. Here some parallels to ADA features and concepts will become apparent.

6.1 Modules

The first of these features is the MODULE subprogram. This is a program unit which, apart from the header line and final END statement, may contain only specification statements and internal procedures. Modules may be imported into any other program unit by a USE statement specifying the name of the module. The USE statement contains a mechanism for resolving name clashes between imported and local entities.

Entities declared in a module may be restricted in their scope by the use of the PRIVATE attribute (a PUBLIC attribute exists too, but this is the default).

The module is first and foremost a direct replacement for the COMMON statement, and the fact that it may include data initialization statements means that it includes the functionality of the BLOCK DATA subprogram. The important advantage over the COMMON statement is the fact that a module is defined once and for all, and that each occurrence in a program unit will therefore be identical, making impossible all the pitfalls and dirty tricks which can be played with COMMON blocks. The possibility to propagate identical copies of internal procedures throughout a whole program can assist in the construction of procedure libraries.

6.2 Derived data types

FORTRAN possesses only a limited set of pre-defined data types, integer, complex, etc., and has hitherto lacked the possibility to build user defined data types. This will be possible in FORTRAN 8x, using a system illustrated by the example

```
     TYPE STAFF_MEMBER
          CHARACTER(LEN=20)::FIRST_NAME, LAST_NAME
          INTEGER::ID, DEPARTMENT
     END TYPE
```

which defines a structure which may be used to describe an employee in a company. An aggregate can be defined as

```
     TYPE(STAFF_MEMBER), ARRAY(1000)::STAFF
```

defining 1000 such structures to represent the whole staff. Individual staff-members may be referenced as, for example, STAFF(NO), and a given field of a structure as STAFF(NO)%FIRST_NAME, for the first name of a particular staff-member. More elaborate data types may be constructed using the ability to nest definitions as in

```
TYPE COMPANY
     CHARACTER(LEN=20)::NAME
     TYPE(STAFF_MEMBER), ARRAY(1000)::STAFF
END TYPE
.
.
TYPE(COMPANY), ARRAY(20)::COMPANIES
```

to define a structure to define companies.

6.3 Data abstraction

It is possible to define a derived data type, and operations on that data type may be defined in an internal procedure. These two features may be combined into a module which can be propagated through a whole program to provide a new level of data abstraction. As an example we may take an extension to FORTRAN's CHARACTER data type whose definition must be of a fixed and pre-determined length. A user-defined derived data type, on the other hand, may define a set of modules to provide the functionality of a variable length character type, which we shall call STRING, (the example is due to J. Wagener). The module for the type definition might be

```
MODULE String_type
     TYPE String(Maxlen)
          INTEGER::Length
          CHARACTER(LEN=Maxlen)::String_data
     END TYPE String
END MODULE String_type
```

With

```
USE /String_type/
.
.
TYPE(String(60)), ARRAY(10)::CORD
```

we define an array of 10 elements of maximum length 60. An actual element can be set by

```
CORD(3) = 'ABCD'
```

but this implies a re-definition or overloading of the assignment operator to define correctly both fields of the element. This can be achieved by the internal procedure

```
INTERNAL SUBROUTINE C_to_S_assign (S,C)ASSIGNMENT
          TYPE (String)::S
          CHARACTER(LEN=*)::C
     S%String_data = C
     S%Length      = LEN(C)
END INTERNAL SUBROUTINE C_to_S_assign
```

which can be included in the module, together with other valid functions such as concatenation, length extraction etc. to allow the user defined string data type to be imported into any program unit where it may be required, in a uniformly consistent fashion. This powerful new feature allows users to define data structures of arbitrary complexity, for instance for list-processing, interval arithmetic etc. These can be coded in modules which can be used not only in one program, but be placed in libraries for wider use.

7. The crystal ball

Since the title of this paper is "Has FORTRAN a Future?", the concluding section must finally come to grips with the difficulties of prophecy. Viewed in isolation, it might be imagined that FORTRAN, as a demonstrably useful language, could stay with us for a very long time, especially if it is kept "up-to-date" by decennial revisions. It is, however, impossible to think about FORTRAN's future without also considering potential and actual rivals.

For many small-scale applications there will certainly be a continuing trend for PASCAL and sometimes C to be used, where FORTRAN might have been used before. This will happen because of the widespread use of PASCAL as a teaching language, making it the lingua franca of computing, and because of the widespread availability of C compilers as the UNIX operating system continues its onward march. For real-time applications and process control there will surely be a move towards ADA as soon as proper compilers for that language become available. The fundamental question is whether the hard-core FORTRAN applications in large-scale scientific, numerical and engineering fields will be significantly influenced by moves to newer languages. Since PASCAL has many limitations for this type of activity - poor I/O, no extended precision, no complex arithmetic, no exponentiation, etc. - its ability to displace FORTRAN from below is inherently curtailed. ADA, on the other hand, although designed to replace real-time languages in embedded systems, has turned out to have powerful numerical capabilities, as shown by Hammerling and Wichmann [18]. If the problems of building program libraries with a language which has very general precision definitions and of interfacing it to FORTRAN can be overcome, it is conceivable that some FORTRAN users would prefer to move to that language for new applications. Given the inertia of programmers, and their heavy investment in existing code, that is likely to be a move of small proportions. More probable, if ADA ever gets off the ground outside the DoD, is that new programmers who have learnt ADA as a first programming language will slowly cause FORTRAN to become a language used only by an ageing generation.

But what are the straws in the wind ? On the one hand we learn that MIT has chosen to standardize on four languages for its ATHENA project, which is planned to couple several thousand personal workstations over the whole campus. They are LISP for the purists, FORTRAN for the realists, C because it is in the selected operating system and PASCAL for teaching. No-one wanted ADA at all. On the other hand we know that Digital is about to announce an ADA compiler for its VAX machines and clearly if it works well, many potential FORTRAN application programmers might be tempted at least to try it. At the same time, interest in ADA in US universities seems to be tailing off, although it remains strong in Europe, and is certainly much stronger than academic interest in FORTRAN, as may clearly be seen by comparing any issue of ACM FORTEC Forum with any issue of ACM ADA Letters.

What we basically see is a huge investment in ADA creating an irresistible force, which will shortly meet the enormous inertia of FORTRAN, an immovable object. One key to the final outcome will be the acceptance or otherwise of the new FORTRAN standard, and its rapid and successful implementation, enabling it to compete with ADA on an equal footing, as by that time ADA should be fairly well established. The other key will be user reaction. The fact that a useful FORTRAN program can be written by a novice in a day, even in FORTRAN 8x, makes it attractive for non-specialists who use computers as one of many other tools. ADA is a language for experts, and the final division may then well be ADA for big specialist applications, FORTRAN for big applications written by non-specialists, PASCAL for small applications, with non-procedural languages displacing all three in the long-term.

References

[1] Backus J. et al. (1957). In "Programming Systems and Languages" (S. Rosen ed.) pp.29-47. McGraw Hill, New York, 1967.

[2] Metcalf M. (1982) Aspects of FORTRAN in large-scale programming. In Proceedings of the 1982 CERN School of Computing, CERN 83-03, pp. 140-146.

[3] ANSI (1978) - Programming Language FORTRAN, X3.9-1978, ANSI, New York.

[4] Metcalf M. (1984) Survey of user reaction to FORTRAN 8x proposals. Minutes of ISO/TC97/SC5/WG9. April 1984, Geneva.

[5] Metcalf M. (1982) An introduction to FORTRAN 77, CERN DD/US/11.

[6] Metcalf M. (1983) FORTRAN 77 coding conventions, CERN DD/US/3.

[7] Metcalf M. (1983) Design conventions for FORTRAN programs, CERN DELPHI/83/99.

[8] Metcalf M. (1982) "FORTRAN Optimization", Academic Press, London and New York.

[9] Metcalf M. (1985) "Effective FORTRAN 77" Oxford University Press, Oxford.

[10] Reifer D.J. and Meissner L.P. (1975) Structured FORTRAN Preprocessor Survey UCID-3793, LBL, Bekerley.

[11] Sakoda J.M. (1979) ACM Sigplan Notices, 14, 1, 77-90.

[12] Wagner N.R. (1980) ACM Sigplan Notices, 15, 12, 92-103.

[13] Wilson K. (1983) CERN Courrier, 25, 5.

[14] Wilson K. (1983) Minutes of X3J3/157, pp. 188-189.

[15] Osterweil L. and Clemm G. (1984) An extensible toolset and environment for the production of mathmatical software, In Proceedings of the International Conference of Tools, Methods and Languages, North-Holland, Amsterdam and New York.

[16] Williams D.O. (1984) Alternate RETURNs, SIGPLAN Notices, 19, 10.

[17] Crowley T. (1984) Array features in FORTRAN 8x, In Proceedings of the International Conference of Tools, Methods and Languages, North-Holland, Amsterdam and New York.

[18] Hammerling S.J. and Wichmann B.A. (1981) Numerical Packages in Ada. In "The relationship between numerical computation and programming languages" (Reid J.K. Ed.) pp. 225-244, North-Holland, Amsterdam.