CERN — DATA HANDLING DIVISION
DD/82-18
8 November 1982

ASPECTS OF FORTRAN IN LARGE-SCALE PROGRAMMING

M. Metcalf

Talk given at the
CERN School of Computing
Zinal, Switzerland, 1982

ASPECTS OF FORTRAN IN LARGE-SCALE PROGRAMMING

M. Metcalf
CERN, Geneva, Switzerland

1. INTRODUCTION

In these two lectures I shall try to examine the following three questions:

i) Why did high-energy physicists begin to use FORTRAN?
ii) Why do high-energy physicists continue to use FORTRAN?
iii) Will high-energy physicists always use FORTRAN?

In order to find answers to these questions, it is necessary to look at the history of the language, its present position, and its likely future, and also to consider its manner of use, the topic of portability, and the competition from other languages. Here we think especially of early competition from ALGOL, the more recent spread in the use of PASCAL, and the appearance of a completely new and ambitious language, ADA.

2. THE EARLY HISTORY OF FORTRAN

FORTRAN was the first high-level language to be developed, and its appearance must be considered to be both innovative and revolutionary. It was innovative not only because of its originality, but also because the use of a high-level language implies the use of a compiler, and compiling techniques had to be developed by John Backus and his team[1] in order to be able to compile the source code and to generate efficient object code, even at that time a primary objective.

Up until the mid-1950's, most programming had been performed in some sort of assembly- or auto-code. The revolution introduced by FORTRAN was to relieve the scientific programmer of the burden of having to program in a way which required knowledge of the details of the underlying hardware, and especially of actual or symbolic registers. This represented a large gain in programmer productivity. At the same time, the simple mathematical-style notation of the language allowed, for the first time, ordinary scientists to approach a computer directly, without needing a specialist as an intermediary, thus opening the field of scientific computing to anyone who was willing to learn a few simple rules.

The first version of the language, released to the public on the IBM 704, was known as FORTRAN II, the original design never having got beyond the prototype stage. By 1964 there were 43 different compilers on 16 different systems. High-energy physicists, with their large computer demands, were among those seduced by the attractions of using a high-level language, and a common dialect, known as CERN FORTRAN, was adopted for use by physicists at CERN. This decision was not taken without much discussion, in which the question of efficiency played a leading part. We note, however, that one of Backus' original design objectives had been efficiency of execution, and if we compare Böck's estimate [2] of a loss of 20% in object code efficiency compared with assembly code, against Backus' estimate of an increase by a factor of five in programming efficiency, we can only marvel that there should have been any opposition at all.

The large number of different implementations began to be a source of great inconvenience, especially with respect to program portability[3], and between 1962 and 1966 ASA (later ANSI, the American National Standards Institute) worked on the first FORTRAN standard. This was a doubly difficult task, as until that time no programming language had ever been standardized, nor had ASA ever produced a standard longer than a very few pages — all that is necessary to describe a screw or a plug. Once this standard was adopted, there was no further need for a CERN standard, and the new standard-conforming compilers were used instead, although some skill and knowledge were required to write programs which used widely implemented extensions to the standard, without inadvertently using extensions which were to be found in only one particular compiler.

It is interesting to consider the advances which were introduced by FORTRAN through the eyes of someone writing in 1969. Jean Sammet[4] listed these as:

i) used available hardware;
ii) the EQUIVALENCE statement, which allowed programmers to control storage allocation;
iii) non-dependence of blanks in syntax;
iv) ease of learning;
v) stress on optimization.

It is certainly true that available hardware was used, as the language was designed specifically to run on the IBM 704, and it contains none of the concepts such as stacks or pointers which would map more easily onto a different type of architecture. On the other hand, this wedding of all high-level languagues to the von Neumann architecture is now considered by Backus to have been a false start in computing technology, making more difficult the introduction of languages which lend themselves to algebraic manipulation and rigorous proof[5], especially because of the basically serial or sequential nature of the operations they describe.

It is difficult nowadays to imagine that the EQUIVALENCE statement should be regarded as an advance, but indeed the tiny core memories of early computers caused enormous problems for programmers, and this simple means of memory management was a great boon, even if present ideas on the undesirability of storage association will lead to its eventual removal from the language.

Compared to many early assembly languages, the FORTRAN source form and syntax were a step forward in liberating programmers from rigid input formats, and allowing, within limits, some degree of free form. Once again, we shall see how the wheel has turned full circle, and that blanks will once again become significant, but in the framework of a yet freer source form.

The advances in ease of learning and the continuing stress on optimization are surely the two hallmarks of FORTRAN, and have greatly contributed to its continued popularity. If future standards remain true to these twin pillars of its strength, it will surely exist for many years to come for a significant class of applications. This we shall return to later.

3. PORTABILITY OF FORTRAN 66

This problem has been treated extensively in the literature[3], and here we touch only on some of the more important aspects, but before doing so, a definition of portability should be given, and I use "the ability to move computer software from one computer system to another and to obtain essentially identical results (or to have the job cancelled)". By "essentially identical results" I mean that they should not differ within the range of significance required for a correct solution of the problem, but may differ in insignificant digits.

A first difficulty encountered by those concerned with the need to transport large programs from one computer to another was the fact that many compilers did not adhere to the standard. A comparison on the CDC 6000, IBM 360, UNIVAC 1100, and three other compilers, carried out in 1970, revealed that the *only* statement implemented with neither extensions nor contradictions was the unconditional branch — GO TO sl.

A second difficulty was the diversification of the language. Many features which were essential for useful, large-scale programming were not defined in the standard which, being permissive, allowed anything which was not specifically prohibited. This meant that for compiler writers it became a lower limit, whereas for compiler users it had to remain an upper limit, clearly a well-nigh impossible situation, except for those with sufficient dedication to learn and apply the standard.

We can now examine a few of the particular problem areas for program portability, under the old standard (although some still remain under the new one).

3.1 Character handling

The standard defined no mapping of Hollerith data onto a computer word. This means that a strict ANSI DATA statement on an IBM system would be, for example,

```
DIMENSION STRING(4)
DATA STRING(1)/4HTHIS/, STRING(2)/4HTHAT/, ...
```

and such statements would have to be rewritten when moving to a CDC computer if the character string were to be in contiguous storage. Problems such as this were intensified by the lack of a standard internal character representation, the lack of operators, and the lack of functions. In practice, standard machine-dependent library routines were used in high-energy physics (HEP) programs, and in some other cases characters were even stored inefficiently one per word in order to remain strictly standard-conforming and portable.

3.2 Input/output

In the area of I/O there were a number of problems, particularly that of number representation for tape interchange of computed quantities, the lack of a standard interface to error-handling facilities, and the poor support of the RECFM=U blocking format on IBM systems. In general, HEP programs have usually used standard machine-dependent packages, such as XREAD (CDC) and IOPACK[6] (IBM) for I/O, and EPIO[7] for data transfer in a machine-independent fashion. Here we recognize that *data* portability is just as important as *program* portability in HEP experiments, where data reduction and analysis for one experiment are performed in a number of different laboratories.

3.3 Precision

The standard did not make any statement about the precision or range of computer arithmetic. This topic has been thoroughly examined by Erskine[8] at a previous School, and here we cite only an instance given by Knoble[9] of what can go wrong. The expression

$$((X + Y)**2 - X**2 - 2*X*Y)/Y**2$$

should always evaluate to 1.000..., but if X is set to 100. and Y to 0.01 on an IBM system, the result will be −39.0625. Even in double-precision the exact result is not quite obtained, as the value 0.01 is a recurring binary fraction, whose truncated value loses further significance when shifted for the addition to X. There is, as yet, no solution to these problems provided by the language; this will come in future standards, and the only advice is to be aware of the difficulties, and alert to their appearance. Detailed advice is given in the references already cited.

3.4 Error handling

During the execution of a program, various errors — or events or exceptions as they are often known — may occur. These may be classified as:

i) I/O errors such as an end-of-file condition or parity check;
ii) arithmetic errors such as division by zero or an attempt to take the square root of a negative number;
iii) errors associated with the operating system or hardware, such as use of an undefined variable or an attempt to exceed an allowed time-limit.

In all these cases, the standard made no statements about any possible recovery action, and HEP programs have made heavy use of standard interface packages in the CERN Program Library.

3.5 Lack of data structures and dynamic space allocation

FORTRAN contains, for the moment, only primitive data structures such as arrays, but not the records and pointers of, for example, PASCAL. This deficiency, combined with the need to reduce program size, which is less pressing on modern virtual memory systems, led to the design and implementation of memory management packages such as BOS[10], HYDRA[11], and ZBOOK[12]. These are in widespread use in HEP programs.

3.6 Lack of bit-handling facilities

In HEP experiments, the raw data is transferred from the on-line computer to the off-line analysis program in units of usually 16 bits. These units do not map well onto CDC computer words, but even on byte machines there is the further problem that the 16-bit unit is subdivided in a manner which requires bit manipulation to extract, convert, and store the data it contains. Here we encounter a conflict between portability and efficiency, as typically we have to choose between using efficient (in-line expanded) manufacturer-supplied bit functions, which are non-portable, or portable library functions, which are inefficient owing to the overhead of the call and the degradation in loop optimization which an external reference engenders.

The solution is a compromise: in those places where efficiency is demonstrably the overriding consideration, the manufacturer-supplied functions are used, and elsewhere the standard library is preferred.

3.7 Lack of environmental enquiry facilities

A large, portable program needs to interact with the environment formed by the hardware and operating system under which it runs. Thus it needs to know about the parameters of the computer arithmetic supported by the hardware (largest number, maximum range), and about such matters as the time used or time left. These interfaces are not yet standardized, and are partially provided for HEP programs in a standard way by library functions which are written for each of the various systems used.

4. PRESENT STATUS OF FORTRAN

After 1966 there came once again a proliferation of dialects and extensions, some for the reasons indicated above. This situation, and the flaws in the language, which were leading to the introduction of large numbers of private pre-processor based extensions, led finally to the preparation and introduction of a new standard in 1978, but nevertheless known as FORTRAN 77 [13]. Substantial difficulties have been encountered with its introduction, due both to the lateness and quality of some compilers, as well as to some significant user inertia, but the new standard is now adopted as the CERN standard for all new programs, and we can hope that its use will spread rapidly.

The fact that the standard is itself fairly backwards-compatible, and that most compilers are even more accommodating, allows a reasonably straightforward transition from the old to the new standard. The two main incompatibilities in the language are the removal of the extended-range DO-loop, which is no loss in these days of structured programming, and the replacement of Hollerith data and constants by the new CHARACTER data type. This latter change has, in fact, led to some major problems of conversion of HEP programs, as Hollerith data were used extensively in argument lists — for instance, as histogram titles — and Hollerith data were freely mixed with numeric data — for instance, in I/O buffers.

In the long-term, it is not the problem of conversion, but any difficulties of using pure FORTRAN 77 which are of greater significance. Since new compilers are now universally available but not universally installed, the most important consideration is to complete the installation of these compilers, and to use them as quickly and as much as possible. In this way, the inevitable conversion period will be shortened, and the problems of programming in an environment with a dual standard will be most quickly eliminated.

5. THE MAIN FEATURES OF FORTRAN 77

FORTRAN 77 is described in many textbooks, and I have attempted to summarize the full language elsewhere[14,15]; the interested reader is referred to these publications. I list here only a few points which I consider particularly relevant for HEP programs:

i) the extension of array declarations and references;
ii) the introduction of the block-IF;
iii) the extension of the DO control statement;
iv) the introduction of the PARAMETER statement;
v) the introduction of the implied-DO in DATA statements;
vi) the introduction of alternate RETURNs;
vii) the extension of the means of defining variably dimensioned arrays;
viii) the introduction of CHARACTER data type;
ix) the very complete (45 page) definition of I/O, including:
   - direct access files
   - internal files
   - execution-time format specification
   - list-directed I/O
   - file control and enquiry
   - new edit descriptors.

6. A DIVERSION: PROGRAM MAINTENANCE TOOLS

An aspect of program portability which is of great importance is the ability to transport source code in a convenient manner between different, incompatible computers, and to have a means whereby more than one person can work on the development of a common program at different sites, without mutual interference, but with the possibility of incorporating new code in an organized way. The only tool now used in HEP programming for this purpose is PATCHY[16], the use of the CDC product UPDATE now being very limited.

PATCHY is a unique and valuable tool for program maintenance and interchange, but it suffers from a number of drawbacks as it ages while the rest of computing advances:
- it is a one-pass processor (sequential file processing);
- it is too complicated for many applications;
- its documentation is consequently difficult to understand;
- it is punched-card oriented (not terminal);
- it has no interface to editors;
- it recognizes only a restricted character set.

Moves are now afoot to design a successor product which in particular will:
- have editor interfaces;
- be simple to understand and use;
- recognize the full ASCII character set;
- use internally direct-access files.

No time-scale for this important project has been set, but it is a product which is clearly of some importance as we enter the LEP era with its huge collaborations of hundreds of physicists in dozens of institutes.

7. PORTABILITY OF FORTRAN 77

This topic is dealt with at length in Ref. 15, and no summary will be attempted in this written version of the lectures.

8. WHY FORTRAN?

FORTRAN, as we have seen, was the first high-level language. Since its introduction it has faced "competition" from a variety of sources, the principal ones being ALGOL and PL/1 in the 1960's, PASCAL in the 1970's, and presumably ADA in the coming decade.

If we examine the reasons why ALGOL failed to oust FORTRAN, they are probably:

- lack of separate compilation,
- non-existent I/O features,
- lateness of compilers,
- difficult (but formal) definition,
- inertia.

ALGOL as such contains better constructs than FORTRAN 66, for instance *if* ... *then* ... *else*, *for* loops and recursion, and has the block structure which has made it the forerunner of PASCAL and ADA. In addition, its formal definition, in which Backus was also involved, made it a powerful publication language. However, when FORTRAN, too, was finally accepted by the relevant journals as a publication language, ALGOL's use declined to its present low level, even in Europe where it originated. But its spirit lives on.

PL/1 was designed by IBM to be the language to end all languages, but its huge size and unwieldy syntax meant that it never caught on in the way IBM intended, despite its enormous resources, and its use is now mainly in high-level systems programming on IBM systems.

The failure of PL/1 contrasts markedly with the phenomenal success of PASCAL[17], designed by one man to be a vehicle for teaching good programming practice and design. It has experienced an extraordinary growth, especially for the smaller-scale applications to which it is best suited. In spite of its success in many smaller and newer areas of computing, particularly among the new generation of programmers, it has not been able to displace FORTRAN, but nor was it ever intended to do so. A decision to use PASCAL on a trial basis in HEP programs, taken in 1976[18], has come to nothing. Here, the reasons must be some combination of:

- "smallness" of the language, e,g, no exponentiation;
- primitive I/O capability;
- lack of mechanism to override strong typing;
- lack of extended precision;
- lack of means to initialize variables (cf. BLOCK DATA);
- lack of complex arithmetic;
- lack of a mechanism to pass variably dimensioned arrays through argument lists (except as an option in the ISO standard);
- lack of an interface to libraries (except in some extensions);
- lack of separate compilation facilities (except in some extensions).

If we consider the future of FORTRAN, then we need to discuss the relative merits of the next standard FORTRAN, known as 8x and described in Ref. 15, and of ADA. This latter is a completely new language designed under the auspices of the US Department of Defense for use in embedded systems. However, it has become a general-purpose language with a strong numerical capability, and specification-conforming compilers are expected to be available by the end of 1984; and by the end of the decade it is possible that its use will be widespread just at the time that FORTRAN 8x compilers become available.

The main new features of FORTRAN 8x are the following:

i) Free form source
ii) Entity-oriented declarations
iii) New form of DO-construct
iv) Array processing
v) Data structures
vi) Precision specification
vii) Enhanced form of CALL
viii) BIT data type
ix) Environmental enquiry
x) Recursion
xi) Dynamic storage
xii) Compile-time facilities

The main features of ADA, according to Barnes[19], are its introduction of so-called data abstraction, plus

- readability,
- strong typing,
- programming in the large,
- exception handling,
- tasking (its raison d'être),
- generic units.

FORTRAN was the first language to introduce 'expression abstraction' by allowing programmers to write statements such as X = A + C(J) without having to worry about registers and other matters. ALGOL introduced 'control abstraction' with its constructs such as *if* X = Y *then* A: = B *else* P: = Q, which relieved the programmer of the need to worry about explicit control with GOTOs. Data abstraction is achieved in ADA by using the enumeration types already found in PASCAL, where the actual representation of the variable green of type colour is of no concern to the programmer, and by adding the possibility to hide private data types in 'packages' (modules).

It is certainly true that the universality of FORTRAN meant that it has, in the past, been used for many purposes for which it was not always appropriate. Anything was better than assembly language. As other languages have been developed, often with a specific purpose in mind, they have displaced FORTRAN in those areas. Looking to the future, the array processing features of FORTRAN 8x will be the jewel in its crown which will, I believe, cause FORTRAN to be in continued use for large-scale numerical calculation for many years to come. This contrast with other languages is shown in the table, which uses the eight criteria of Feuer and Gehani[20] by which they judge languages for scientific applications. FORTRAN stands out as a clear leader.

Table 1

	Pascal	C	FORTRAN 77	FORTRAN 8x	ADA
Extended precision arithmetic		√	√	√	√
Overflow and underflow detection				(√)	(√)
Array operations		?		√	?
Complex arithmetic	?	?	√	√	?
Large no. of maths functions	?	?	√	√	?
Binary I/O	√	√	√	√	?
Routine names as parameters	√	√	√	√	√
Lower bounds for arrays	√		√	√	√

Key: √ = completely defined
? = partial definition, or possible but not defined, or unclear
( ) = possible hardware dependence

REFERENCES

1) J. Backus et al., *in* Programming systems and languages (ed. S. Rosen) (McGraw-Hill, New York, 1967), p. 29.
2) R. Böck, Comput. Phys. Commun. **9,** 221 (1975).
3) D. Muxworthy, A review of program portability and FORTRAN conventions (European Program Institute, Ispra, 1976).
4) J. Sammet, Programming Languages: History and Fundamentals (Prentice-Hall, Englewood Cliffs, NJ, 1969).
5) J. Backus, Can programming be liberated from the von Neuman style? Commun. ACM **21** (8), 613 (1978).
6) R. Matthews, IOPACK User Guide (Z300, CERN Computer Library, Geneva, 1982).
7) H. Grote and I. McLaren, EPIO manual (CERN Computer Library, Geneva, 1982).
8) G.A. Erskine, Proc. CERN School of Computing, La Grande Motte, France, CERN 76–24 (1976), p. 191.
9) H. Knoble, A practical look at computer arithmetic (Pennsylvania State University, Pa., 1979).
10) V. Blobel, BOS manual (DESY, Hamburg, 1979).
11) V. Frammery, U. Herbst-Berthon and J. Zoll, HYDRA topical manual (CERN, Geneva, 1982).
12) R. Brun, M. Hansroul and J.C. Lassalle, ZBOOK user guide (CERN, Geneva, 1980).
13) ANSI (1978). Programming language FORTRAN, X3.9-1978, (ANSI, New York).
14) M. Metcalf, An introduction to FORTRAN 77, CERN report DD/US/11 (1982).
15) M. Metcalf, FORTRAN optimization (Academic Press, London and New York, 1982).
16) H. Klein and J. Zoll, PATCHY reference manual, (CERN, Geneva, 1977).
17) K. Jensen and N. Wirth, PASCAL user manual and report (Springer Verlag, Berlin, 1975).
18) I. Willers (ed.), High-level languages at CERN, CERN report DD/MCM/76/60 (1976).
19) J. Barnes, Programming in ADA (Addison-Wesley, London, 1982).
20) A. Feuer and N. Gehani, Comput. Surv. (ACM) **14,** 1 (1982).