FORTRAN

J. W. BACKUS AND W. P. HEISING

Summary—The fundamental concepts of FORTRAN, the most widely used high-level, scientific programming language, are set forth and the significant characteristics are described in historical order from inception of FORTRAN in 1954 to the present time. The basic problem of how to get high quality programming from an-easy-to-write high-level language is emphasized.

INTRODUCTION

MUCH HAS BEEN written about FORTRAN over the years it has been on the computing scene. Its shortcomings and merits have been debated at length. The language and properties of various FORTRAN systems have been described in detail in a number of manuals. The present paper will, therefore, not include a description of the language (a list of references giving brief or complete descriptions of the language appears at the end of the paper), nor will it take up again questions of comparisons or possible improvements.

The current interests of people who write compilers and devise programming languages seem to center on finding orderly and efficient methods of transforming elaborate source programs into legitimate object programs. This paper describes some of the considerations that led the group developing FORTRAN during 1954 to 1957 to put extreme emphasis not on orderly and quick translation techniques, nor on elaborate source language features, but rather on the efficiency of the object programs themselves. There is some discussion of problems raised by such emphasis and their impact on the language.

DESIGN ENVIRONMENT

Most of the FORTRAN language specifications were frozen almost ten years ago. It was therefore designed in a different atmosphere within the computing industry than exists today. At that time, most programmers wrote symbolic machine instructions exclusively (some even used absolute octal or decimal machine instructions). Almost to a man, they firmly believed that any mechanical coding method would fail to apply that versatile ingenuity which each programmer felt he possessed and constantly needed in his work. Therefore, it was agreed, compilers could only turn out code which would be intolerably less efficient than human coding (intolerable, that is, unless that inefficiency could be buried under larger, but desirable, inefficiencies such as the programmed floating point arithmetic usually required then). There were some systems around at the time which tended to confirm this popular view.

This then was the professional climate in which the design of the FORTRAN language was begun and completed. Furthermore, the system was viewed as applying to just one machine, the IBM 704. For, although the FORTRAN group hoped to radically change the economics of scientific computing on that machine (by making programming much cheaper), it never gave very much thought to the implications of machine-independent source languages applied to a variety of machines. Rather, the group's attention was focussed on what seemed then to be the challenge: could an automatic coding system take over a lot of tasks from the programmer and still produce object programs of competitive efficiency? The 704 was the first commercial computer with built-in floating point, therefore most common operations were fast enough that poor coding of loops would greatly reduce efficiency.

Confronted with this scepticism and a machine which would make small lapses in arrangement of coding show up as sizeable inefficiencies, the group had one primary fear. After working long and hard to produce a good translator program, an important application might promptly turn up which would confirm the views of the sceptics: this application would be of the sort FORTRAN was designed to handle and, even though well programmed in FORTRAN, its object program would run at half the speed of a hand-coded version. It was felt that such an occurrence, or several of them, would almost completely block acceptance of the system. (Of course, few would feel that way today.)

MAJOR DESIGN PROBLEMS

The question then arose: what part of a machine-produced code would be most subject to inefficiencies? The coding of the principal arithmetical calculations of the problem clearly offered some opportunities for ineptness. But fairly simple rules could be envisaged to avoid many of them. Rather, the worst inefficiencies, it was believed, would come from address calculations in which one or more multiplications were used to find addresses of elements in an array. If these elements were accessed in an orderly fashion, the human-coded program would obtain the address of each succeeding element by adding an appropriate constant to the address of the previous one. For machine-coded programs to avoid computing k_1+i+k_2j (with its multiplication) to get the address of $A_{i,j}$, the translator program would have to recognize when i and j were changing by con-

Manuscript received February 27, 1964.

J. W. Backus is with IBM Corp., Thomas J. Watson Research Center, Yorktown Heights, N. Y.

W. P. Heising is with IBM Corp., Data Systems Division, New York, N. Y.

IEEE Trans. on E. Comp. EC13 No4 Aug 64

stant amounts and be equipped, for example, to produce coding which would preserve the address of $A_{i,j}$ and add the constant k_2d to it to obtain the address of $A_{i,j+d}$.

Thus, one problem was clearly that of distinguishing between, on the one hand, references to an array element which the translator might treat (in the resulting object program) by incrementing the address used for a previous reference, and those array references, on the other hand, which would require an address calculation starting from scratch with the coordinates of the referenced element. One of the difficulties in determining whether a reference to $A_{i,j}$ could be handled in the efficient way involved tracing back through all preceding computational paths to see which ones contained modifications of i and/or j only by constants.

After some study of the problems posed by the desire to increment old addresses to get new ones, it was found that the number of cases needing different treatment which could be visualized was very large indeed. It became clear that all cases could not be treated optimally, at least not in the first translator. For example, a loop containing both a reference to $A(I, J)$ and the statement "$J = J + F(K)$" might permit the incremental calculation of the addresses of $A(I, J)$ but only if either K remains constant within the loop or F is a linear function of K and K varies linearly within the loop. Just to handle this one simple case, the translator would have to check for changes in K within the loop and if it found one, and all of them were linear, then it would have to look up the definition of F, and if it was linear, the translator would have to compute the appropriate increments for use in the object program. In addition, the translator would first have to ascertain that the case at hand was this one; *e.g.*, that the references to $A(I, J)$ in this loop did not also belong to other intersecting loops with different treatment of I and J.

Influence of Design Problems on Language

In view of the conclusion, on the one hand, that all cases of address calculation could not be treated optimally, and the fear, on the other, that unfavorable efficiency comparisons might result in rejection of the system, it was felt that there should be some way for the user to know which loops would be efficiently treated and which would not be so treated. A loop-forming statement, the DO statement, had already been proposed as a convenience to the user. (A DO statement specifies that a block of statements should be executed repeatedly for successive values of an index variable.) It was decided to use this statement as the boundary between incremented-address loops and computed-address loops, in the following sense: within a loop controlled by a DO statement addresses depending on its index variable would in general be obtained by incrementing techniques, whereas other references to arrays would be effected by calculating the required address from the subscript values. This rule was to hold even though an array reference was deep within a "nest" of DO statements. All of the testing of index quantities, modification of addresses, and initialization of quantities at the beginning of a loop were to be treated as efficiently as possible in such nested loops. This was felt necessary again to avoid embarrassing comparisons in cases in which the inner loops of a nest were to be repeated only a small number of times, whereas the outer loops were repeated a large number of times.

A number of restrictions in the language now began to emerge which were needed if the analytical abilities of the translator were to be kept within bounds and the above efficiency requirements met:

1) The index variable of a DO statement should vary only by a constant increment. Unfortunately, the state of understanding then and the machine for which the system was planned dictated a positive increment.
2) The expressions appearing as subscripts should be linear in the index variables.
3) The number of subscripts should not exceed three, since more subscript positions would have increased the number of cases to be distinguished by a large factor.
4) Control should not pass from outside into a block of statements governed by a DO statement. This type of restriction is found in most programming languages.

In addition to the restrictions listed above, others arose from a desire to simplify and speed up the translation process at what seemed a very small price in convenience to the user (*e.g.*, variables must comprise no more than six characters).

Influence of Design Problems on Translator

The effect of the attitudes in the computing industry and of the FORTRAN group circa 1954–1956, as outlined above, was to make the group expend great effort toward eliminating every inefficiency it could from object programs produced by the translator. Every eye was focused so intently on that particular ball that some wryly amusing circumstances resulted after the translator began operating. For example, one concern was to avoid unnecessary storing and loading of index registers. The analysis to prevent this involved a Monte Carlo calculation of the relative frequency of execution of the parts of the program, followed by a complicated treatment, in order of frequency, which assigned index registers to index quantities within larger and larger "regions" of the program. Much of the considerable time spent in this part of a compilation was devoted to the process of "region formation." Naturally, among the first dozen or so programs compiled, a large complicated one turned up which had no index quantities at all. Nevertheless, of the 20 minutes it took to compile, 10 minutes were devoted to "region formation," despite the fact that this left the program entirely unchanged.

In general, the result of the concentration on efficiency was that compilation time was often a significant portion of the total running time of an object program. This was particularly true for small one-shot programs, since their running time was small and since the 704 compiler required about one minute to translate the program: "STOP"; after the first minute, it would turn out about 100 instructions per minute. On the other hand, the object programs produced were quite efficient; as long as the user formed loops with DO statements and did not specify waste motion in his source program, he usually got a program which compared well with hand-coded ones.

Thus, despite the slowness of compiling, by 1964 standards (and its impediment to debugging in source language), the 1957 FORTRAN compiler did effect considerable programming economies over the practices of that period and made the 704 computer directly accessible to a much larger group of users. The computing community inspected the object programs, found them acceptable, and slowly proceeded to make larger and larger use of the system and its conveniences.

The subsequent development of the FORTRAN language and the many compilers which were produced for a great variety of computers did much to improve the convenience and economies offered by the system. In particular, the problem of slow compilation, with the debugging difficulties which ensued, was much lessened by the introduction of FORTRAN II language (see next section) and by a better balance in later compilers between compilation speed and object program speed.

Growth of the FORTRAN Language—FORTRAN II

In the original FORTRAN system for the 704, an entire application was a single FORTRAN program which was compiled in a single (rather lengthy) run. If a minor change was to be made in any part of the program the entire program had to be recompiled, and since many small errors are normally encountered and corrected during checkout, total compilation time on the computer was considerable. A second weakness also became apparent in that, practically speaking, an application had to be done entirely in FORTRAN, or else not at all.

Accordingly, in 1958 an expanded FORTRAN language, known as FORTRAN II, was introduced to meet the objections above. It made it possible for the user to describe a process with a number of separately compilable programs, one main program and a number of subprograms. New statements were added to the FORTRAN language which described certain data as common to all programs as well as conventions for one program to invoke or be invoked by another. Thus an application might consist of many FORTRAN programs and, if a change were required, normally only a single program need be recompiled.

FORTRAN II permitted several programmers to divide a large job more conveniently, each programmer writing one or more subprograms. Furthermore parts of an application for which FORTRAN was unsatisfactory or inconvenient could be written in symbolic machine code as independent subprograms. Additional provisions were made for handling and editing alphanumeric information for input and output. Although FORTRAN was initially designed as and remains primarily a language for scientific computing, the ability to print table headings and other indicative information adds significantly to its usefulness. Indeed, FORTRAN is used to some extent for conventional data processing by judicious use of hand-coded subroutines. However, the FORTRAN language is not well suited to large scale use for this purpose.

Acceptance of FORTRAN II was immediate, in fact, the extent of FORTRAN usage accelerated greatly and has continued to broaden over the course of time.

FORTRAN IV

By 1961, it had become apparent that the acceptance of FORTRAN II was so great that design of an augmented FORTRAN language together with completely new compilers was instituted. The 7090 FORTRAN II compiler had been based on the identical internal design developed for the 704, and a complete rewrite afforded opportunity to introduce a number of simplifications to the internal structure and also to augment the FORTRAN language in several directions desired by users of FORTRAN.

The enriched language, which was called FORTRAN IV, contained among its principal additions:

1) labelled COMMON areas, as many as a FORTRAN programmer might find convenient. As contrasted with languages such as ALGOL and COBOL, FORTRAN applications need not be compiled in one fell swoop. A complete application may consist of a series of modules (called subprograms) independently compiled and tested. This aspect of FORTRAN, which is now perhaps its most distinctive feature, requires especially careful attention to the layout of data areas that will be directly referred to by more than one subprogram. It is necessary at compilation time to know which data are to be thus shared, *i.e.*, are in "COMMON."

When many programmers are working on a single application, there must be mutual agreement on the layout of "COMMON," somewhat like the necessity for carefully controlled and agreed on master record design in data processing work. Changes in the "COMMON" layout may require recompilation, and changes to the method of allocating COMMON plus the addition of many "COMMON" areas in FORTRAN IV allows for the orderly growth of such shared data areas as a complex application is being developed with minimum impact on already compiled subprograms of the same application,

2) double precision and complex quantities were added to the language and the interpretation of arithmetic operators was extended to deal with them in a natural manner,

3) logical (truth) variables and the admission of relational expressions, the logical connectives "and," "or," and "not" were introduced as well as a new conditional "IF" statement. Thus

"IF(X.GT.3.AND.X.LT.8)$Y = X$"

means

"If $X > 3$ and $X < 8$, set $Y = X$,"

4) a "DATA" statement was added to preset conveniently any data areas to specified values prior to the initiation of execution,

5) the original FORTRAN performed operations on only two "types" of data, integer and real (*i.e.*, floating point). The initial letter of the name of a variable served to distinguish them, thus all variables beginning with *I*, *J*, *K*, *L*, *M*, or *N* denoted integers. This was occasionally awkward, for example, a variable for counting might have to be called "NCOUNT" to indicate it is integer. With the introduction of three new types, double precision, complex and logical, it did not seem desirable to extend the first letter "declension" system to encompass them. Accordingly, five new "type" statements were introduced to declare that variables whose names were listed were of a given type, *e.g.*, "INTEGER COUNT,TOTAL)." In the absence of explicit type declarations, the old rule of implication of real or integer variable still applies,

6) certain statements referring to "TAPE," "PUNCH," etc., were eliminated in favor of a wholly symbolic I/O unit designation in keeping with the capabilities of modern operating system monitor programs. "QUOTIENT OVERFLOW" and similar 704 triggers went quietly to oblivion at the same time.

In 1963, a number of FORTRAN IV compilers appeared on computers produced by various manufacturers. The use of FORTRAN IV has been growing steadily although FORTRAN usage is still probably preponderantly FORTRAN II.

FORTRAN Standards

The existence of many FORTRAN compilers, all handling similar but slightly different FORTRAN languages has become increasingly troublesome as the range and size of application work written in FORTRAN has grown, and users frequently wish to transfer work from one computer to another of a different type. Accordingly in 1962, a FORTRAN standards committee, operating under the procedures of the American Standards Association and under the sponsorship of the Business Equipment Manufacturers Association was formed to work on FORTRAN standardization. Most of the principal U. S. manufacturers and computer users associations concerned with FORTRAN are participating in drafting American Standard FORTRAN. FORTRAN IV is serving as the basis for this work. So many thousands of programmers now use FORTRAN that the standards work is primarily concerned with committing common existing practice to writing rather than attempting to alter or even to add to FORTRAN at this time. This work is nearing completion and it is felt that a useful purpose will be served by simplifying the transferability of FORTRAN programs between computers. While the process of converting from one computer to another will never be painless, the reduction of a number of arbitrary differences between FORTRAN compilers will perhaps further enhance FORTRAN usage and serve as the basis of continued evolution in the future.

Acknowledgment

R. K. Ridgway, the present FORTRAN coordinator in IBM, provided valuable assistance in preparing this paper.

References

[1] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt, "The FORTRAN Automatic Coding System," Proc. of the Western Joint Computer Conference, Los Angeles, Calif.; February, 1957.

[2] "FORTRAN IV Language," IBM Corp. 7090/7094 Programming Systems File No. 7090-25 Form C28-6274.

[3] J. W. Backus, "Automatic Programming: Properties and Performance of FORTRAN Systems I and II," *Proc. Symp. on the Mechanization of Thought Processes*, Teddington, England; 1958.