```
integer k, p; real y, a, b;
KDF9  4, 0, 0, 2;
      ZERO; DUP; DUP; =V1; =V2;
   2; SET 1; +; ="p";
      "p"; "k"; −; J1 > Z;
      V2; V1; "a"; "b"; ×+F; =V1; =V2;
      "p"; J2;
   1; V2; V1; ROUND F; ="y";
ALGOL
```

The formal parameters, which are each enclosed in quotation marks in the KDF9 text, are replaced automatically by sequences of instructions for getting access to the required quantities. The system allows calls by name and by value. In the example, V1 and V2 correspond to the local variable *s* of the Report version. In this version the sum is accumulated double length and then rounded.

The reason for providing so fully for machine-code procedures is to simplify the introduction of features which it would otherwise be inconvenient or impossible to express within ALGOL. For example, one might need a set of procedures for evaluating some frequently-used functions, and the running speed of the ALGOL versions, even when translated by an optimizing compiler, might not be sufficiently close to that of the corresponding machine-code versions. Input and output, and magnetic-tape procedures must either themselves be in machine code or make use of machine-code procedures. Something like the scheme we have proposed is necessary if one wishes to get beyond the stage of "read one number, punch one number."

As was mentioned earlier, machine-code procedures and ALGOL procedures can be included in the library. It follows that the user of library procedures for input and output need know nothing about the KDF9 User Code.

A matrix scheme proposed for use with our ALGOL system (Denison, 1962) makes use of a number of procedures which have already been expressed in ALGOL. It is probable that they will be rewritten in User Code for the sake of speed.

Acknowledgement

Acknowledgement is due to those colleagues whose work is described in these notes. Publication is by permission of The English Electric Company Limited.

References

DENISON, S. J. M. (1962). "A Proposed ALGOL 60 Matrix Scheme." Paper to be presented to the IFIP Congress 62.
DIJKSTRA, E. W. (1961). "ALGOL 60 Translation," *ALGOL Bulletin*, Supplement No. 10, Mathematisch Centrum, Amsterdam.
DUNCAN, F. G., and HAWKINS, E. N. (1959). "Pseudo-Code Translation on Multi-level Storage Machines," *Proceedings of ICIP, Paris*, p. 144.
DUNCAN, F. G., and HUXTABLE, D. H. R. (1960). "The DEUCE Alphacode Translator," *The Computer Journal*, Vol. 3, p. 98.
The English Electric Co. Ltd. (1961). *KDF9 Programming Manual.*
GREEN, J. S. (1961). *Introduction to ALGOL 60 Programming for the KDF9*, The English Electric Co. Ltd.
HAWKINS, E. N., and HUXTABLE, D. H. R. (1962). "A Multi-pass Translation Scheme for ALGOL 60," *Annual Review Automatic Programming*, Vol. 3 (to be published).
NAUR, P. , ed. (1960). *Report on the Algorithmic Language ALGOL 60*, Regnecentralen, Copenhagen
NAUR, P., ed. (1962). "*ALGOL Bulletin*, No. 14," Regnecentralen, Copenhagen.
RANDELL, B., and RUSSELL, L. J. (1961 and 1962). Descriptions of work for DEUCE and KDF9 in internal memoranda of the Atomic Power Division, The English Electric Co. Ltd.

Operating experience with FORTRAN

By A. E. Glennie

My purpose in this talk will be to describe the lessons that I have learned from my own experience, and that of my colleagues, in using FORTRAN during the last three years. Some of the points I shall make are specifically about the FORTRAN language itself; others are about automatic coding in general, and computer operating systems incorporating compilers. I hope that what I have to say about the latter aspects, as revealed in the use of FORTRAN, may be of interest and value to those of you whose interests and preferences may be in other languages—or in fields other than scientific computation.

The History of Our Use of FORTRAN

I think that you will be interested to hear how the use of FORTRAN in the U.K.A.E.A. developed, as our story is quite typical of the evolution of a laboratory's technique and practice. When, in 1958, we started our first experiments with FORTRAN on the IBM 704, we had had a long tradition of machine-language coding, but had also had some experience of automatic coding.

We were not, I think, prejudiced against automatic coding, yet we were somewhat disappointed with our first experience with FORTRAN. This was the

FORTRAN I language, not FORTRAN II, which we now use. In retrospect, it is easy to explain our disappointments: there are some obvious reasons, such as the unreliability of the FORTRAN I compiler and the conservatism of our mathematicians, who had grown accustomed to machine-language coding. Yet I now believe that there were two principal reasons for our early lack of faith in FORTRAN.

Firstly, the coding produced by the FORTRAN compiler seemed to be of much poorer quality than that produced by our better programmers. In those days we had not fully realized how expensive machine-language programming can be because of the expense of debugging. As computers become faster, and cheaper per unit of computation, the quality of the coding of programs (in the sense of compactness and elegance) becomes less important than the ease with which they may be written and corrected. We are now prepared to sacrifice the frills and elegancies in coding; we are also willing to trade a certain amount of program speed for the ease of program preparation, communicability of programs, etc., brought by automatic coding.

The second principal reason against the use of FORTRAN I was that it lacked the ability to create and use subroutines, except by a process so difficult as to be ignored by the average user. This meant that the user of FORTRAN I had to write his whole program as a "main program" and to compile it all together. This was a process very expensive of computer time. since the program could not be made to work in parts. which could then be assembled, but had to be re-compiled to correct every mistake. Since FORTRAN compilers are relatively slow (because they attempt to economize the object program), the use of FORTRAN I was expensive except for very small jobs which would not require much debugging. It used to be argued that automatic coding would be of use only to the casual user with a small problem. It is not sufficiently appreciated that the greatest benefits of automatic coding are to be expected for the most difficult, complex and lengthy programs, where the use of automatic programming may assist the programmer sufficiently to enable him to penetrate the complexities of the problem.

This, then, was the main shortcoming of FORTRAN I, which was remedied in FORTRAN II. It was impracticable to segment a very large program into parts.

Because of the characteristics of FORTRAN I outlined above, we were rather slow to appreciate the properties of FORTRAN II, particularly its use in the very largest of problems. We had continued to use FORTRAN for small jobs, but still used machine language for the largest jobs. Perhaps the most significant factor in popularizing the FORTRAN II language with us was the introduction (in the autumn of 1960) of the "FORTRAN Monitor System." This is a system for automatic operation of the IBM 709 or 7090 computer, which allows a FORTRAN subroutine to be compiled and then incorporated with previously compiled parts of a program, and then the whole program to be run. In spite of the name, the FORTRAN Monitor System is not bound to the FORTRAN language; it is merely a system for combining subprograms, whether written in machine language or FORTRAN. But it was the "Load and Go" ability of the FORTRAN Monitor System which brought FORTRAN to life. Previously a FORTRAN program had to be compiled in one machine run and then it could be obeyed in another run. With the FORTRAN Monitor System, one run sufficed for translating and obeying the program.

It is now our considered policy to write all future programs in an automatic code language, and we have chosen FORTRAN II, not perhaps for its intrinsic virtues, but because it was available to us on the IBM 7090. We find FORTRAN II to be a very satisfactory language, subject to some trivial criticisms which follow.

FORTRAN Syntax

I should now like to consider FORTRAN from four points of view and discuss its syntax, its semantics, the behaviour of the compiler, and the economics of using it.

In talking about the syntax of FORTRAN I am tempted to make an analogy. FORTRAN in its syntax is like the English Common Law, where precept guides and the law is not codified. To be quite blunt, the syntax of FORTRAN is almost impossible to represent by an exact description as is ALGOL. I do not mean to imply that it is impossible to know whether what you write in FORTRAN has proper syntax: it is usually quite easy, but you must refer to a set of rules which do not possess the regularity of the ALGOL rules—and, in critical cases, the only test of the syntax is to try an example on the compiler. In the last resort, FORTRAN is defined by its compilers and will show small differences from one machine to another. Yet one should not exaggerate this point. For inexperienced programmers, the difficulty rarely arises, as they do not attempt to write programs of sufficient elaboration and subtlety to come against any difficulty in syntax, except in the writing of input or output statements. These are the parts of the FORTRAN language that are most recursive in their syntax, and could admit a concise syntax description as in ALGOL. They give the average user a lot of trouble, particularly in the construction of input or output formats, which are the specifications of the relations between quantities within the computer and, for example, the appearance of these quantities on the printed output.

Perhaps the chief reason for the frequency of errors in FORTRAN input or output statements lies in their generality and power, which tempt the programmer to write statements which are unnecessarily complicated, I have heard many programmers wish for a simpler way of getting printed results, without the necessity of specifying the number of digits before and after the decimal point, the space between columns, and so on,

which must always be specified in FORTRAN. What they want is the "?" printing that is provided in Mercury Autocode.

The semantics of FORTRAN

I now wish to say a few words about the semantics of FORTRAN—about the kinds of meanings that FORTRAN statements have, which colour the methods that can be used when a program is written in FORTRAN and not in machine language. My experience has been that it is very easy to underestimate the power of a language, like FORTRAN, which is ostensibly suited only for scientific numerical calculation. With some trivial subroutines, written in machine language, it is possible to write data-processing programs, e.g. compilers (even FORTRAN compilers) in FORTRAN, though perhaps not as economically (in terms of program efficiency) as in machine language. This possibility of extension of FORTRAN II is due to the ability to use (by FORTRAN statements) machine-language subroutines, by which the processing power of the language may be extended at will. This has been one way in which we have used FORTRAN, i.e. as a way of writing only part of a program, completing the rest in machine language, where there is some advantage in using machine language.

Returning to the FORTRAN language itself, there are a few obvious shortcomings which make it awkward to express the programmer's intention on occasion. The most notorious of these is the restriction on the so-called "Do statements" by which the repetitions of loops may be controlled. There is a restriction here that the variable which is changed at each repetition of the loop must always be increased, otherwise the loop is never-ending. This is an annoying restriction but no more than that, since it is simple to define functions of the counting variable that move in the manner desired.

Another annoyance of the same sort concerns statements for conditional branching. These test whether a numerical expression is positive, negative, or zero—a three-way test. Usually the programmer wishes to make a two-way test, yet he has always to specify three outcomes.

For simple programming, the semantics of FORTRAN are very straightforward and have caused little difficulty. After all, FORTRAN is quite a limited language and therefore simple.

FORTRAN Compilers

FORTRAN compilers are known to be complex and slow. This is because they do what few other compilers attempt, the economization of the use of index registers. FORTRAN allows arrays of numbers of up to three dimensions, and the FORTRAN compilers make strenuous attempts to minimize the number of instructions required to provide access to data in such arrays, whenever the data is being scanned in a regular manner. Instead of calculating (by multiplication and addition) the address of an element of a multidimensional array, the program produced by the FORTRAN compiler will build such addresses by additions only, if the addresses are controlled by counted loops. In many problems this leads to a very considerable improvement in the efficiency of the resultant program as compared with a program produced by a compiler which does not perform this type of economization.

The effect of this is to allow the user to make free use of arrays of numbers without any qualms that he will cripple his program by so doing.

There is no doubt that this economization has made the FORTRAN compilers extremely complicated and difficult to perfect; and in fact, FORTRAN compilers are always being corrected as obscure faults are discovered: not that this causes much trouble to the average user as his probability of being affected by one of these obscure faults is very low.

The Economics of FORTRAN

I should like to close by making a few tentative remarks about the economics of FORTRAN, since the principal reason for using any such language is often economic. I shall not attempt any conclusions but give some facts, from which you may draw your own conclusions.

Firstly there is the cost of translation. On the IBM 7090, this appears to be between 1d. and 2d. per instruction of the translated program. This is to be compared with the similar cost of about ½d. per instruction for the compilation of a single instruction from symbolic machine language. This cost is not great when compared with the other programming costs unless many repetitions of translation are required. As always, the principal feature of programming costs is the programmer himself and nothing, and certainly not FORTRAN, will prevent bad programmers wasting a lot of machine time. My conclusion is that if a programmer cannot save time and effort by using FORTRAN (or any other autocode) then he is probably not worth employing. I assume, of course, that the problems concerned are of the appropriate type for the autocode.

The principal lesson of FORTRAN here is that an autocode such as FORTRAN must be capable of compiling subroutines to machine-language code, to be combined later by a process of loading much cheaper than any translation.