Comments from a FORTRAN User

John M. Blatt*

Comments on features of compilers in general, and Fortran in particular

Introduction

The calculation of the binding energy of the nucleus H^3, the triton, can only be done numerically, and even then it is a quite complicated problem, involving a large number of three-dimensional integrations and the location of the minimum of a function of a large number of variables. Because of the limited time during which the actual coding could be carried out (two months in New York), it was decided to use FORTRAN rather than a coding scheme closer to the basic machine language of the IBM 704, such as SAP. Indeed, a quick reading of the FORTRAN manual and some comments from FORTRAN users indicated that FORTRAN would save both time and effort and would be a generally satisfactory scheme to use for this problem. Actual experience with FORTRAN coding on this problem, however, has converted Paul into Saul: if a similar problem should come up again, the author would be very reluctant indeed to use FORTRAN.

It is the purpose of this paper to comment on the features of compilers in general, and FORTRAN in particular, that are required by the advanced coder. It is the author's belief that no satisfactory scheme exists at present, with the possible exception of the English AUTOCODE. The reason for this lack is not that a satisfactory compiler is hard to write. On the contrary, it would be much easier to write than FORTRAN. Rather, it has not been written because the logical and physical requirements of a satisfactory compiler for advanced coders have not been studied to a sufficient extent.

Although this paper contains some negative comments on FORTRAN, they are not meant to disparage the very real achievement represented by FORTRAN. FORTRAN was a pioneer effort, well ahead of other compilers, and is very successful indeed for the uses envisioned by the designers of the routine. The fact that FORTRAN has shortcomings when used for a quite different kind of coding is not surprising and in no sense detracts from the credit due to the designers.

The remarks in this paper are just scattered comments, rather than the results of an extensive study of present and/or proposed compiling schemes. They are offered with the hope that others, more experienced in this field, may perhaps find some of them useful. From what little the author has seen of the proposed ALGOL scheme, it would *not* be satisfactory in actual use by an advanced coder.

Purposes of Compilers

There are basically two different classes of machine users, both of which require compiling routines: (A) users who have no, or very little, experience in coding, and whose problems are short compared to the available machine storage and machine time; and (B) users who are experienced coders with really big problems, in which machine storage and machine time must be considered seriously.

We shall denote compilers for these two classes of users as compilers of type A and B, respectively.

It is not always realized just how seriously the requirements of type A and type B compilers differ from each other. In actual fact, they are completely incompatible. FORTRAN was designed as a type A compiler, and works extremely well when used in that way. As a type B compiler, it is therefore, necessarily, quite unsatisfactory.

Some of the differences in requirements relate to:

(i) machine time for actual computation,
(ii) machine time for compiling,
(iii) storage space taken by the object program, and efficient use of storage space for lists of constants,
(iv) access to machine instructions and/or machine-like instructions,
(v) coupling between different subroutines,
(vi) error detection,
(vii) nature of the Manual for the compiler.

These and other points are discussed below.

* Institute of Mathematical Sciences, New York University, and University of New South Wales, Kensington, N.S.W., Australia. Supported in part by a grant from the U. S. Atomic Energy Commission. FORTRAN was used to code a fairly elaborate calculation in nuclear physics. Based on this experience, these comments are made regarding the requirements of compiling routines for complicated problems, in general, and aspects of FORTRAN, in particular.

First, however, we would like to comment on the frequently made statement that advanced programmers do not require compiling routines anyway and should use either machine language or else a routine such as SAP which is very close to machine language. The author strongly disagrees with this point of view. Machine language, and even SAP, takes much longer to write than a language such as FORTRAN, and many more actual symbols must be written down on paper and transferred to cards. There is no reason to make an advanced programmer do unnecessary and tedious bookkeeping merely because he is an advanced programmer. Furthermore, the more actual symbols are written on paper, the more actual cards are punched and the higher is the probability of purely trivial errors. These can of course be detected and eliminated by code-checking, but this takes time and effort that could well be devoted to better purpose. The machine must always be considered the slave of man, not man the slave of the machine.

Quite apart from the increase in sheer work and number of trivial errors connected with machine language or SAP coding, there is another major point for insisting on compiling routines—translation to a different machine. At the present time, machines are being designed in many places, and new machines are coming out all the time. If a program is written in the source language of a universally recognized compiler, it may be assumed that the actual compiling routine for each new machine will be readily available. Translation of the code for the new machine is then a simple operation—recompilation. Some of the efficiency of the original program may be lost, but enough will remain to make this procedure worthwhile, especially if the new machine is much faster than the previous one. On the other hand, a program coded in machine language, or in SAP, is necessarily tied to one particular machine and becomes obsolete when the machine becomes obsolete. Re-writing an entire major program for a new machine is almost as big a job as writing the original program. The re-writing job is usually not even attempted.

Thus type B compilers are necessary. The rest of this paper is concerned with the requirements which they must fulfil.

Manual for the Compiler

The manual associated with a compiling routine is an integral part of the compiling system as a whole; it is one link in a chain—in the case of FORTRAN, the weakest link. No matter how good the basic compiling routine may be, it becomes unusable, or nearly so, if the manual is bad.

Manuals for type B compilers must be quite different from manuals for type A compilers. The type B user needs to know, in considerable detail, just what the object program is, corresponding to the various types of source statements. In particular, he needs to know the number of storage locations used in the object program, and the computation time. This information should be given systematically, at the same point at which the source language statement in question is explained logically. In the present FORTRAN manual much of this information is missing, and the rest of it is located in scattered places throughout the later sections of the manual.

This requirement of machine detail for type B manuals means that a new type of B manual must be written for each new machine, even though the source language of the compiler is unchanged. Although a type B user will not in general need to know the detailed machine language and other particular features of the machine on which the object program is to be run, the type B user can never be satisfied merely with understanding the source language itself. He is presumably fighting against storage space and machine time limitations, and he must be given adequate information about the object program produced by the compiler.

Quite apart from the nature of the information contained in the manual, there is the question of the *manner* in which this information is presented. All too often, the writing of the manual is done by the person, or group, responsible for writing the compiling routine itself. This is bad policy: People who are good at talking to machines, are often not nearly so good when it comes to talking to people. The requirements are, after all, quite different: In talking to people, redundancy, repetition, and emphasis are essential to secure understanding. None of this is required in talking to a machine; in fact redundancy and repetition are vices, and emphasis is impossible.[1] Thus the best way to produce a compiler manual is to have it written by a user, with the advice and consent of the authors of the compiling routine.[a]

Machine Time for Compiling

The more elaborate and "fancy" the compiling routine, the more machine time is required for the actual compilation. In the case of type A compilers, this is no problem: the compiling routine can do most of the code-checking, and once a routine compiles, it almost always works the way it is intended to work by the type A user. This is by no means true for type B users. Even after the trivial errors have been eliminated from a subroutine, there are any number of non-trivial errors which may be contained in the subroutine, and an even larger number of non-trivial errors which arise out of the interaction between different subroutines. No matter how elaborate the compiling routine is made, it can never test for such errors. Thus re-compilation is a *frequent* occurrence in type B program

[1] In the author's opinion, this is sufficient reason that a language such as FORTRAN or ALGOL can never become a common language among mathematicians. Mathematicians are, usually, people.

[a] Superscripts in parentheses refer to Editor's related comments on p. 506.

development, and the machine time required for compilation becomes a major factor.

During the author's stay in New York, he noticed the frequency with which type B users were making hand-punched corrections on the object program (binary deck) merely to avoid recompiling. *This procedure is the reductio ad absurdum of the whole philosophy of compiling routines!* Instead of having simplified things for the user, the compiling routine (because of the machine time for compiling) actually forces the user to learn the basic machine language, study in detail the object program produced by the compiler, and correct this program by the most primitive and time-consuming method imaginable.

One way out of this difficulty would be to leave the compiler pretty much as it is for the first compilation, but speed up re-compilations. For example, the compiler could punch out, in binary form, the information which FORTRAN now puts on an output tape, and this information could be read in at the time of the re-compilation. Alternatively, FORTRAN could be made to read the previous output tape for any re-compilation (this method would necessitate a lot of tape handling).

These intermediate solutions appear unsatisfactory, however. They are at best make-shift, compared to the radical solution of the "instantaneous compiler". We suggest that one essential requirement for a type-B compiler is that its operation be substantially as fast, or nearly as fast, as the speed of reading in the source program. Such a compiler will be called "instantaneous". Clearly, with an instantaneous compiler, re-compilation is no problem, and no-one need learn to make hand-punched corrections on the object program deck.

This requirement of instantaneous operation takes precedence over all others. No matter how desirable a certain feature of a projected compiling routine may appear, it should be eliminated ruthlessly if it conflicts with instantaneous operation. An example is the optimization of the use of index registers carried out by FORTRAN. This involves a logical tracing of the flow of the whole program, with branchings determined on a statistical basis. It all sounds very desirable, and is in fact desirable for type A users. But it slows down the operation of the compiler to a very appreciable extent, much too much for type B users. For type B users, it would be both simpler and better to allow the user to specify which index is to be represented in the machine by an index register, with perhaps a list of priorities in case the object machine does not have enough index registers.[2]

This radical requirement of instantaneous compilation is by no means visionary. In fact, AUTOCODE used extensively in England is substantially instantaneous. The author has had no experience with AUTOCODE and therefore cannot comment on its suitability in other respects.

[2] The present alternative allowed by FORTRAN II is not to optimize at all. This is unacceptable for type B programs.

It is the author's impression, perhaps incorrect, that this requirement is being overlooked in the design of the ALGOL language. Although the author has never written a compiling routine, he finds it difficult to imagine how an instantaneous compiler could be written for a language as complex as ALGOL. Thus, in the end, ALGOL may turn out to be quite suitable for type A use, and well-nigh useless for type B coding. *The source language for a type B compiler is severely limited by the requirement of instantaneous operation and cannot be decided on the basis of purely mathematical considerations, in splendid isolation from the very real problems of writing an instantaneous compiling routine.*[b]

Machine Time for Running Object Program

Every advanced programmer knows that there are tricks for speeding up a program—tricks which depend on the particular machine and tricks which are common to many different machines. One example is provided by integrations using Simpson's rule. The coefficients, 1, 2, and 4, can be generated in a binary machine by shifting or by altering the exponent of a floating-point number; this is much faster than machine multiplication. With FORTRAN, the user is denied access to the shifting instructions of the basic machine language. Actually, multiplication of a floating-point by a fixed-point number is explicitly forbidden.

A solution to this problem is provided in FORTRAN III: the user is allowed access to the basic machine language and may use direct machine instructions in sections of the source program. In our opinion, this solution is not a forward step but a backward step in the development of compiling routines. One main reason for using compilers is the translation problem, discussed above. By going back to the basic machine language, translation is made impossible, or at least very difficult.

An alternative, and in our opinion preferable, solution is to include, as part of the permissible statements of the source language, statements that are easily translatable to machine operations such as shifting.

To continue with this example, there could be source language statements for "multiply by 2^n" and for "multiply by 10^n". In a binary machine, the first of these would be compiled as a fast operation, the second as an ordinary multiplication. In a decimal machine, the first would be an ordinary multiplication, the second a fast operation. Since practically all machines are either binary or decimal and since the type B user knows which machine he is writing for, this is sufficient for type B compilers and avoids the use of basic machine language as far as shifting operations are concerned.

Another example of the inadequacy of FORTRAN for type B operation is the simple problem of computing a check sum for a list of numbers. In FORTRAN, only two types of addition are possible: floating point addition, and addition of truncated (modulo 2^{15}) fixed point numbers.

Both of these are logically unsuitable for check sums, the first because numbers with small exponents never alter the checksum, the second because a large fraction of the bits in the table are excluded from the check. Furthermore, even if one ignores the logical difficulty associated with the floating-point addition, each such addition takes three and one-half times as long as the ordinary fixed-point addition in the machine. Thus the computation time for the checksum becomes appreciable, merely because of properties of the compiler.

It is highly recommended that any type B compiler include, as part of the source language, the usual arithmetic and logical commands available in most computers today. This means both fixed-point (to full precision!) and floating point arithmetic operations, as well as some of the simpler logical operations such as collation, negation, and the like. Since these are all easily translated, their inclusion does not conflict with the primary requirement of instantaneous operation.

The criterion for inclusion of a certain type of "machine-like" command in the source-language of a type-B compiler is twofold: (i) The proposed command must be useful for saving machine time and/or storage space for the object program,[c] and (ii) It must be easily translatable into machine language for many machines.

Organization of Storage Space

A type A compiling routine can be written with the idea that the storage space of the machine is substantially infinite. No attempt need be made to conserve storage space. FORTRAN, being a type A compiler, is very wasteful of storage space. When used for type B operation, this feature becomes a serious problem. In the author's routine, it was necessary to read in the program in two stages, use 7 out of the 10 tapes, and do a considerable amount of overwriting of lists of numbers in the COMMON store, in order to get the program into the machine at all.

For a routine of this type, not only is FORTRAN wasteful of storage space, but the organization of the COMMON storage is tremendously awkward and productive of errors. In FORTRAN II, constants are either stored within each subroutine, or they are placed in a COMMON list available to all subroutines.

Logically, however, there are at least two, and perhaps three, different functions of COMMON storage:

(i) Temporary storage, used by several subroutines, and overwritten in turn by each new subroutine; this is also known as "working space".

(ii) Storage of tables of numers used by several different subroutines.

Classification (ii) may be divided into two subclassifications:

(iia) Tables which are erased during the operation of the program.

(iib) Permanent tables, which may be computed by a preliminary interlude and are not changed thereafter.

In writing a complex program it is well-nigh impossible to determine in advance the number of items in each of these lists. Since FORTRAN lumps all these lists together into one COMMON list, the COMMON list quickly becomes a patchwork quilt of segments from these three classes. Furthermore, any change in the COMMON list affects *all* subroutines and requires re-compilation of all of them. Since re-compilation with FORTRAN is very time-consuming, the net effect is a most undesirable coupling between subroutines which are logically and mathematically quite distinct.

A possible way out would be to provide, in addition to fixed addresses and addresses relocatable with respect to zero, addresses relocatable with respect to a set of numbers which may be specified by the programmer at the time the object program for the main program is read into the machine. There may well be other, and better, ways to allow the advanced programmer latitude in the organization of storage space. All we can do here is to call attention to the existence of this problem.[d]

Code Checking Facilities

In advanced program development, dynamic code checking is essential. Post mortem printouts of the contents of the fast store are frequently difficult to interpret, and the tracing of errors by this method is at best slow and difficult, at worst impossible. In FORTRAN II, code checking sequences must be incorporated into the source language program, since there is no way to get such information out of the compiled program. Since this undesirable feature of FORTRAN II is corrected in FORTRAN III, no further comment is necessary beyond saying that any type B compiling routine must contain this dynamic code check feature.

Another aspect of code checking is the checking done by the compiler itself during the process of compilation. FORTRAN does a lot of that, and should of course do so for type A use. For type B use, however, much of this code checking is more of a nuisance than a help, and since it slows down the compiling it is downright undesirable. For example, FORTRAN checks that every branch in a computed GO TO statement leads to actual source program statements. In writing the source program, the author has sometimes included extra, unused branches in such computed GO TO statements, with the idea of using them later on for additional branches. FORTRAN then refused to compile the routine until dummy statements were made for these (unused) branches.

The main reason against doing extensive code-checking during the compilation process is that it is logically impossible to find the majority of errors in this way. In type B programming, the most frequent type of error arises from faulty interaction between different subroutines, and this type of error cannot be found by a compiler which necessarily compiles one subroutine at a time. Of course, any code checking which does not conflict with instantaneous operation of the compiler is desirable, and should be incorporated into the compiling

routine. But as soon as the code-checking features slow down the compiling appreciably, they should be eliminated; instantaneous operation is much more important than code-checking during compilation.

Conclusion

We hope that the various, perhaps rather disconnected, points in the preceding sections will suggest ideas and avenues for exploration to people better versed in the field of compiling routines than the author. This paper is in no sense a blueprint for a type B compiler, just some odd comments from a machine user.

In conclusion, the author would like to express his appreciation and gratitude for the extensive and unstinting help given him by the staff of the A. E. C. Computing Center, Institute of Mathematical Sciences, New York University, in particular Drs. Richtmyer, Isaacson, Goldstein, and Mrs. Bernice Weizenhofer. He also admires, and is grateful for, the patience that all of them exhibited in the face of his irascible nature. The author thanks the US Atomic Energy Commission for a grant of money and machine time which alone made this work possible and, last but not least, wants to record his appreciation of the labors of his colleagues, Dr. Graham Derrick and Mr. David Mustard, without whose devoted efforts the code would certainy not have been finished in the allocated time.

Appendix: Minor Comments on FORTRAN

The following aspects of FORTRAN gave rise to minor nuisances in actual coding; they are all easily corrected, and perhaps some of them have already been corrected in FORTRAN III (with which the author is not familiar).

(1) Indices must start at 1, that is, an array of numbers a_n must have as its first member a_1. In practically all advanced coding, it is preferable to start such a list with a_0, and quite frequently occasion arises in which the initial member of the list has a negative subscript.

It is of course quite easy for the coder to get around this by defining a dummy index m related to n in such a way that m starts with 1. However, these trivial things are among the most likely source of errors, and it would be highly desirable to let the *machine* do this dummy indexing. This is in general line with the philosophy that the machine should be the slave of man.

A "compatible" method would be the following kind of DIMENSION statement:

DIMENSION A(5–19, –14 – –8)

This describes a two-dimensional array A(n, m) with n ranging from 5 to 19, m ranging from –14 to –8. Furthermore, we introduce the convention that the initial value of the index may be omitted if it is in fact equal to unity, i.e., the dimension statements,

DIMENSION B(1–100) and DIMENSION B(100)

are equivalent. Since the latter statement is the type used for FORTRAN at the moment, compatibility is assured.

Whenever the initial index value, call it n^1, differs from 1, the machine operates internally with the dummy index $n' = n - n_0 + 1$, which dummy index does start with 1. The programmer, however, is no longer required to do this bookkeeping.

(2) In iterative loops, organized through the DO statement of FORTRAN, the programmer must not enter the loop anywhere in its middle; rather, the loop must be entered at its first instruction. Furthermore, if the DO loop is completed ("the DO is satisfied"), the index variable of the DO is not available for further use; the index variable *is* available if an exit is made from the loop before completion of the loop.

Both these restrictions stem from the special way in which such loops are organized in FORTRAN; time is saved by using increment fields of instructions, rather than complete memory positions, to store the value of the loop index. For the same reason, indices are restricted to values less than 2^{15}.

Although this is perfectly all right for type A coding, when applied to type B programming it is a textbook example of being penny-wise but pound-foolish. In advanced coding, the trick of entering a loop somewhere in its middle, and even entering it at different points depending on what is to be done, is used all the time. The time saved by FORTRAN's special method of organizing DO loops is more than made up by the time, and machine storage space, lost through having to enter each loop at the beginning and by the nuisance of having to restore the loop index explicitly if the loop has run to completion. Furthermore, the restriction that indices must be less than 2^{15} has been extended in FORTRAN to encompass *all* fixed-point quantities, with highly undesirable results (see the discussion of check sums in section 5). Again, by this special method of organizing loops, indices in a loop must necessarily be positive numbers, which is a nuisance for the programmer.

As a general rule for type B compilers, special restrictions and conditions on variables and indices should be avoided like the plague. It takes a long while to accustom oneself to them, and even at best these special regulations are copious sources of coding errors. If a really substantial amount of machine time can be gained by imposing special restrictions, then the source language of the compiler should provide *two* source statements, one fast but with restrictions, the other slower but unrestricted. For example, the statement DO may retain its present meaning and restrictions, but the additional statement LOOP may be provided, which does the same job as DO, but without the special restrictions on DO statements.

It should be noted that considerations of this type are essentially nonmathematical, and it is the author's impression, perhaps incorrectly so, that such nonmathematical considerations are not being given sufficient weight in the design of the ALGOL language.

Editor's Comments on Compilers

The following comments are meant to provide additional information and not to detract from the value of Dr. Blatt's complaints (see Opinions, p. 501) which, while specifying FORTRAN, refer to many others as well. Certainly it is about time that more compiler builders started designing translators for the good programmer.—A.J.P.

(a) Actually many such compiler manuals have been written by users, though they have not been widely distributed, e.g., a FORTRAN manual by Westinghouse, and an IT manual by Texas Instruments. However, to accent the author's complaint, most of these manuals are intended to further isolate the occasional user from the machine. Is it not obvious that in the next few years —if not already—there will be a large educated audience who will be able to use—and probably insist upon—more control over the manipulation of their codes originally composed in an ALGOL-like form.

(b) Actually there has been a large number of compilers built which used the stated principle as the design motivation, e.g., IT, RUNCIBLE, GAT, and CORREGATE for the IBM 650, and MAD for the IBM 701, to name an admittedly partial list. CORREGATE permits modifications to the object code in source language with only these modifications retranslated. An ALGOL translator is being built at Oak Ridge, and in several centers in Europe, to function as type B compilers. Indeed, one at Mainz translates at the paper tape input speed and the object code commences running when the tape has been completely read. Ironically there have been some complaints that ALGOL—as a language—has been heavily organized so as to permit type B translators to be built.

(c) The translator GENIE, being built at the Rice Institute, is an example of a system where certain machine properties can be exploited in codes written in GENIE. In general, they do not appear to be too difficult to translate into actions on other machines than the one for which GENIE was designed.

(d) The author here refers to the assembly problem, many of whose aspects are independent of the form of the source code. Systems such as the authors would like to see are currently being built, e.g., the ACT system designed for the Signal Corps.

LETTERS (continued)

Recommendations of the SHARE ALGOL Committee, *Comm. Assoc. Comp. Mach. 2* (Oct. 1959), 25–26.

4. The absence of the return statement as defined in ALGOL 58 necessitates the use of an artifice such as a labeled dummy statement, in the event that the last written statement in a procedure body is not necessarily the last executed statement. We believe that the committee should offer some justification for its action in this matter.

5. We notice that there is no stop statement in ALGOL 60. Admittedly, "stop" may mean all things to all translators, but there should be some standard method for denoting the termination of a dynamic statement sequence.

6. Section 4.7.6 appears to be incomplete and unnecessarily complex, by virtue of the following reasoning:

(a) If a quantity is non-local to a procedure body, it must be local to some block which includes the procedure body, else the procedure body is not completely defined.

(b) Hence, "a procedure statement written outside the scope of any non-local quantity of the procedure body" is *ipso facto* outside the scope of the procedure body, and is accordingly undefined. (The scope of a procedure body may be defined analoguosly to the scope of a label, where the procedure identifier in the heading and the same identifier in a statement correspond respectively to a particular label and a reference to it.)

Thus, section 4.7.6 seems to be a special case of the principle of scope, and might be emended to read as follows:

"A procedure statement is defined if and only if it occurs within the scope of the procedure body, and the procedure body is completely defined."

7. We regret that no provision for the specification of initial data was made. If ALGOL was designed primarily for the communication of algorithms rather than for machine implementation, then we concede that such a provision is unnecessary.

8. Section 5.4.4 implies that a function designator may occur *only* as the left part of an assignment statement, lest the procedure be activated recursively. Was this the intention of the Committee?

Any comments from the Committee members or from interested bystanders would be welcomed.

H. ISBITZ — W. DOBRUSKY
RUTH ANDERSON — D. ENGLUND
E. BOOK — H. MANELOWIT
H. BRATMAN — SONYA SHAPIRO
System Development Corporation
Santa Monica, California

Note of Amplification

E. F. CODD

In parts 1 and 2 of the paper "Multiprogra Scheduling" (June 1960 issue, pp. 347–350), the ter "space-shared" is used. It seems desirable to clarify th scope of this term, particularly as in one instance the ter was altered to "space-(memory)-shared."

The term "space-shared" applies not only to the interm storage (e.g., core) but also to auxiliary storage and inpu output devices (e.g., drums, disks, tape units, card reader and printers). In the case of tape units, all tape units of given type constitute a single (composite) space-share facility for which the natural unit of space is a single tap unit. A similar remark applies to card readers and printer

It should be emphasized that the scheduling algorith described in Part 3 of the paper handles in one operatic any number of different facilities and is not a scheme fo internal storage alone.