THE FORTAN AUTOMATIC CODING SYSTEM

BY

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

Reprinted from the PROCEEDINGS OF THE WESTERN JOINT COMPUTER CONFERENCE
Los Angeles, California, February 1957

PRINTED IN THE U.S.A.

The FORTRAN Automatic Coding System

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‖

Introduction

THE FORTRAN project was begun in the summer of 1954. Its purpose was to reduce by a large factor the task of preparing scientific problems for IBM's next large computer, the 704. If it were possible for the 704 to code problems for itself and produce as good programs as human coders (but without the errors), it was clear that large benefits could be achieved. For it was known that about two-thirds of the cost of solving most scientific and engineering problems on large computers was that of problem preparation. Furthermore, more than 90 per cent of the elapsed time for a problem was usually devoted to planning, writing, and debugging the program. In many cases the development of a general plan for solving a problem was a small job in comparison to the task of devising and coding machine procedures to carry out the plan. The goal of the FORTRAN project was to enable the programmer to specify a numerical procedure using a concise language like that of mathematics and obtain automatically from this specification an efficient 704 program to carry out the procedure. It was expected that such a system would reduce the coding and debugging task to less than one-fifth of the job it had been.

Two and one-half years and 18 man years have elapsed since the beginning of the project. The FORTRAN system is now complete. It has two components: the FORTRAN language, in which programs are written, and the translator or executive routine for the 704 which effects the translation of FORTRAN language programs into 704 programs. Descriptions of the FORTRAN language and the translator form the principal sections of this paper.

The experience of the FORTRAN group in using the system has confirmed the original expectations concerning reduction of the task of problem preparation and the efficiency of output programs. A brief case history of one job done with a system seldom gives a good measure of its usefulness, particularly when the selection is made by the authors of the system. Nevertheless, here are the facts about a rather simple but sizable job. The programmer attended a one-day course on FORTRAN and spent some more time referring to the manual. He then programmed the job in four hours, using 47 FORTRAN statements. These were compiled by the 704 in six minutes, producing about 1000 instructions. He ran the program and found the output incorrect. He studied the output (no tracing or memory dumps were used) and was able to localize his error in a FORTRAN statement he had written. He rewrote the offending statement, recompiled, and found that the resulting program was correct. He estimated that it might have taken three days to code this job by hand, plus an unknown time to debug it, and that no appreciable increase in speed of execution would have been achieved thereby.

† Internat'l Business Machines Corp., New York, N. Y.
‡ Mass. Inst. Tech., Computation Lab., Cambridge, Mass.
§ Radiation Lab., Univ. of California, Livermore, Calif.
‖ United Aircraft Corp., East Hartford, Conn.

The FORTRAN Language

The FORTRAN language is most easily described by reviewing some examples.

Arithmetic Statements

Example 1: Compute:

$$\text{root} = \frac{-(B/2) + \sqrt{(B/2)^2 - AC}}{A}.$$

FORTRAN Program:

```
ROOT
  = (-(B/2.0) + SQRTF((B/2.0)**2 - A*C))/A.
```

Notice that the desired program is a single FORTRAN statement, an arithmetic formula. Its meaning is: "Evaluate the expression on the right of the = sign and make this the value of the variable on the left." The symbol * denotes multiplication and ** denotes exponentiation (*i.e.*, A**B means A^B). The program which is generated from this statement effects the computation in floating point arithmetic, avoids computing (B/2.0) twice and computes (B/2.0)**2 by a multiplication rather than by an exponentiation routine. [Had (B/2.0)**2.01 appeared instead, an exponentiation routine would necessarily be used, requiring more time than the multiplication.]

The programmer can refer to quantities in both floating point and integer form. Integer quantities are somewhat restricted in their use and serve primarily as subscripts or exponents. Integer constants are written without a decimal point. Example: 2 (integer form) vs 2.0 (floating point form). Integer variables begin with I, J, K, L, M, or N. Any meaningful arithmetic expression may appear on the right-hand side of an arithmetic statement, provided the following restriction is observed: an integer quantity can appear in a floating-point expression only as a subscript or as an exponent or as the argument of certain functions. The functions which the programmer may refer to are limited only by those available on the library tape at the time, such as SQRTF, plus those simple functions which he has defined for the given problem by means of function statements. An example will serve to describe the latter.

Function Statements

Example 2: Define a function of three variables to be used throughout a given problem, as follows:

```
ROOTF(A, B, C)
  = (-(B/2.0) + SQRTF((B/2.0)**2 - A*C))/A.
```

Function statements must precede the rest of the program. They are composed of the desired function name (ending in F) followed by any desired arguments which appear in the arithmetic expression on the right of the = sign. The definition of a function may employ any previously defined functions. Having defined ROOTF as above, the programmer may apply it to any set of arguments in any subsequent arithmetic statements. For example, a later arithmetic statement might be

```
THETA = 1.0 + GAMMA*ROOTF(PI, 3.2*Y
                          + 14.0, 7.63).
```

DO Statements, DIMENSION Statements, and Subscripted Variables

Example 3: Set Q_{max} equal to the largest quantity $P(a_i+b_i)/P(a_i-b_i)$ for some i between 1 and 1000 where $P(x)=c_0+c_1x+c_2x^2+c_3x^3$.

FORTRAN Program:

```
1) POLYF(X)=C0+X*(C1+X*(C2+X*C3)).
2) DIMENSION A(1000), B(1000).
3) QMAX=-1.0 E20.
4) DO 5 I=1, 1000.
5) QMAX=MAXF(QMAX, POLYF(A(I)
          +B(I))/POLYF(A(I)-B(I))).
6) STOP.
```

The program above is complete except for input and output statements which will be described later. The first statement is not executed; it defines the desired polynomial (in factored form for efficient output program). Similarly, the second statement merely informs the executive routine that the vectors A and B each have 1000 elements. Statement 3 assigns a large negative initial value to QMAX, -1.0×10^{20}, using a special concise form for writing floating-point constants. Statement 4 says "DO the following sequence of statements down to and including the statement numbered *5* for successive values of *I* from *1* to *1000*." In this case there is only one statement 5 to be repeated. It is executed 1000 times; the first time reference is made to A(1) and B(1), the second time to A(2) and B(2), etc. After the 1000th execution of statement 5, statement 6—STOP—is finally encountered. In statement 5, the function MAXF appears. MAXF may have two or more arguments and its value, by definition, is the value of its largest argument. Thus on each repetition of statement 5 the old value of QMAX is replaced by itself or by the value of POLYF(A(I)+B(I))/POLYF(A(I)-B(I)), whichever is larger. The value of QMAX after the 1000th repetition is therefore the desired maximum.

Example 4: Multiply the $n\times n$ matrix $a_{ij}(n\leq 20)$ by its transpose, obtaining the product elements on or below the main diagonal by the relation

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k}\cdot a_{j,k} \qquad (\text{for } j \leq i)$$

and the remaining elements by the relation

$$c_{j,i} = c_{i,j}.$$

FORTRAN Program:

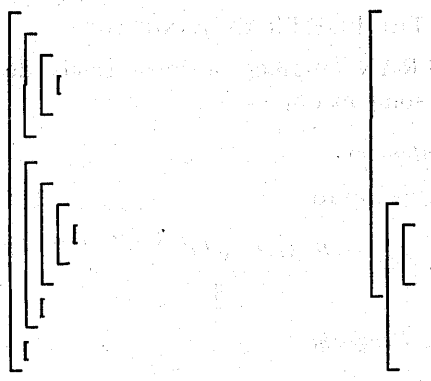
```
      DIMENSION A(20, 20), C(20, 20)
      DO 2 I = 1, N                                  P
      DO 2 J = 1, I                                  Q
      C(I, J) = 0.0
      DO 1 K = 1, N                                  R
1     C(I, J) = C(I, J) + A(I, K) * A(J, K)
2     C(J, I) = C(I, J)
      STOP
```

As in the preceding example, the DIMENSION statement says that there are two matrices of maximum size 20×20 named A and C. For explanatory purposes only, the three boxes around the program show the sequence of statements controlled by each DO statement. The first DO statement says that procedure P, *i.e.*, the following statements down to statement 2 (outer box) is to be carried out for I=1 then for I=2 and so on up to I=N. The first statement of procedure P(DO 2 J=1, I) directs that procedure Q be done for J=1 to J=I. And of course each execution of procedure Q involves N executions of procedure R for K=1, 2, · · · , N.

Consider procedure Q. Each time its last statement is completed the "index" J of its controlling DO statement is increased by 1 and control goes to the first statement of Q, until finally its last statement is reached and J=I. Since this is also the last statement of P and P has not been repeated until I=N, I will be increased and control will then pass to the first statement of P. This statement (DO 2 J=1, I) causes the repetition of Q to begin again. Finally, the last statement of Q and P (statement 2) will be reached with J=I and I=N, meaning that both Q and P have been repeated the required number of times. Control will then go to the next statement, STOP. Each time R is executed a new term is added to a product element. Each time Q is executed a new product element and its mate are obtained. Each time *P* is executed a product row (over to the diagonal) and the corresponding column (down to the diagonal) are obtained.

The last example contains a "nest" of DO statements, meaning that the sequence of statements controlled by one DO statement contains other DO statements. Another example of such a nest is shown in the next column, on the left. Nests of the type shown on the right are not permitted, since they would usually be meaningless.

Although not illustrated in the examples given, the programmer may also employ subscripted variables having three independent subscripts.



READ, PRINT, FORMAT, IF and GO TO Statements

Example 5: For each case, read from cards two vectors, ALPHA and RHO, and the number ARG. ALPHA and RHO each have 25 elements and ALPHA(I) ≤ALPHA(I+1), I=1 to 24. Find the SUM of all the elements of ALPHA from the beginning to the last one which is less than or equal to ARG [assume ALPHA(1)≤ARG<ALPHA(25)]. If this last element is the *N*th, set VALUE=3.14159 * RHO(*N*). Print a line for each case with ARG, SUM, and VALUE.

FORTRAN Program:

```
      DIMENSION ALPHA(25), RHO(25)
1)    FORMAT(5F12.4).
2)    READ 1, ALPHA, RHO, ARG
      SUM=0.0
      DO 3 I=1, 25
      IF (ARG-ALPHA(I)) 4, 3, 3.
3)    SUM=SUM+ALPHA(I)
4)    VALUE=3.14159 * RHO(I-1)
      PRINT 1, ARG, SUM, VALUE
      GO TO 2.
```

The FORMAT statement says that numbers are to be found (or printed) *5* per card (or line), that each number is in fixed point form, that each number occupies a field *12* columns wide and that the decimal point is located *4* digits from the right. The FORMAT statement is not executed; it is referred to by the READ and PRINT statements to describe the desired arrangement of data in the external medium.

The READ statement says "*READ* cards in the card reader which are arranged according to FORMAT statement 1 and assign the successive numbers obtained as values of *ALPHA*(I) I=1, 25 and *RHO*(I) I=1, 25 and *ARG*." Thus "ALPHA, RHO, ARG" is a description of a list of 51 quantities (the size of ALPHA and RHO being obtained from the DIMENSION statement). Reading of cards proceeds until these 51 quantities have been obtained, each card having five numbers, as per the FORMAT description, except the last which has the value of ARG only. Since ARG terminated the list, the remaining four fields on the last card are not read. The PRINT statement is similar to READ except that it specifies a list of only three quantities. Thus

ach execution of PRINT causes a single line to be printed with ARG, SUM, VALUE printed in the first three of the five fields described by FORMAT statement 1.

The IF statement says "*If ARG−ALPHA(I)* is negative go to statement 4, if it is zero go to statement 3, and if it is positive go to 3." Thus the repetition of the two statements controlled by the DO consists normally of computing ARG−ALPHA(I), finding it zero or positive, and going to statement 3 followed by the next repetition. However, when I has been increased to the extent that the first ALPHA exceeding ARG is encountered, control will pass to statement 4. Note that this statement does not belong to the sequence controlled by the DO. In such cases, the repetition specified by the DO is terminated and the value of the index (in this case I) is preserved. Thus if the first ALPHA exceeding ARG were ALPHA (20), then RHO (19) would be obtained in statement 4.

The GO TO statement, of course, passes control to statement 2, which initiates reading the 11 cards for the next case. The process will continue until there are no more cards in the reader. The above program is entirely complete. When punched in cards as shown, and compiled, the translator will produce a ready-to-run 704 program which will perform the job specified.

Other Types of FORTRAN Statements

In the above examples the following types of FORTRAN statements have been exhibited.

Arithmetic statements
Function statements
DO statements
IF statements
GO TO statements
READ statements
PRINT statements
STOP statements
DIMENSION statements
FORMAT statements.

The explanations accompanying each example have attempted to show some of the possible applications and variations of these statements. It is felt that these examples give a representative picture of the FORTRAN language; however, many of its features have had to be omitted. There are 23 other types of statements in the language, many of them completely analogous to some of those described here. They provide facilities for referring to other input-output and auxiliary storage devices (tapes, drums, and card punch), for specifying preset and computed branching of control, for detecting various conditions which may arise such as an attempt to divide by zero, and for providing various information about a program to the translator. A complete description of the language is to be found in *Programmer's Reference Manual, the FORTRAN Automatic Coding System for the IBM 704.*

Preparation of a Program for Translation

The translator accepts statements punched one per card (continuation cards may be used for very long statements). There is a separate key on the keypunching device for each character used in FORTRAN statements and each character is represented in the card by several holes in a single column of the card. Five columns are reserved for a statement number (if present) and 66 are available for the statement. Keypunching a FORTRAN program is therefore a process similar to that of typing the program.

Translation

The deck of cards obtained by keypunching may then be put in the card reader of a 704 equipped with the translator program. When the load button is pressed one gets either 1) a list of input statements which fail to conform to specifications of the FORTRAN language accompanied by remarks which indicate the type of error in each case; 2) a deck of binary cards representing the desired 704 program, 3) a binary tape of the program which can either be preserved or loaded and executed immediately after translation is complete, or 4) a tape containing the output program in symbolic form suitable for alteration and later assembly. (Some of these outputs may be unavailable at the time of publication.)

The FORTRAN Translator

General Organization of the System

The FORTRAN translator consists of six successive sections, as follows.

Section 1: Reads in and classifies statements. For arithmetic formulas, compiles the object (output) instructions. For nonarithmetic statements including input-output, does a partial compilation, and records the remaining information in tables. All instructions compiled in this section are in the COMPAIL file.

Section 2: Compiles the instructions associated with indexing, which result from DO statements and the occurrence of subscripted variables. These instructions are placed in the COMPDO file.

Section 3: Merges the COMPAIL and COMPDO files into a single file, meanwhile completing the compilation of nonarithmetic statements begun in Section 1. The object program is now complete, but assumes an object machine with a large number of index registers.

Section 4: Carries out an analysis of the flow of the object program, to be used by Section 5.

Section 5: Converts the object program to one which involves only the three index registers of the 704.

Section 6: Assembles the object program, producing a relocatable binary program ready for running. Also on demand produces the object program in SHARE symbolic language.

(*Note:* Section 3 is of internal importance only; Section 6 is a fairly conventional assembly program. These sections will be treated only briefly in what follows.)

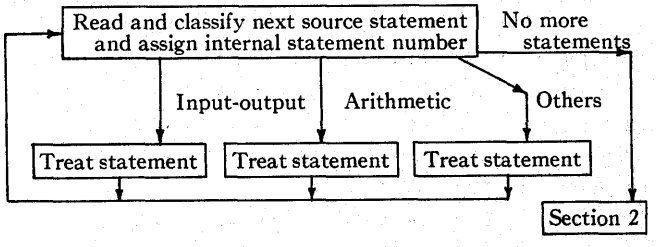
Within the translator, information is passed from section to section in two principal forms: as compiled instructions, and as tables. The compiled instructions (*e.g.*, the COMPAIL and COMPDO files, and later their merged result) exist in a four-word format which contains all the elements of a symbolic 704 instruction; *i.e.*, symbolic location, three-letter operation code, symbolic address with relative absolute part, symbolic tag, and absolute decrement. (Instructions which refer to quantities given symbolic names by the programmer have those same names in their addresses.) This symbolic format is retained until section 6. Throughout, the order of the compiled instructions is maintained by means of the symbolic locations (internal statement numbers), which are assigned in sequential fashion by section 1 as each new statement is encountered.

The tables contain all information which cannot yet be embodied in compiled instructions. For this reason the translator requires only the single scan of the source program performed in section 1.

A final observation should be made about the organization of the system. Basically, it is simple, and most of the complexities which it does possess arise from the effort to cause it to produce object programs which can compete in efficiency with hand-written programs. Some of these complexities will be found within the individual sections; but also, in the system as a whole, the sometimes complicated interplay between compiled instructions and tables is a consequence of the desire to postpone compiling until the analysis necessary to produce high object-program efficiency has been performed.

Section 1 (Beeber, Herrick, Nutt, Sheridan, and Stern)

The over-all flow of section 1 is



For an input-output statement, section 1 compiles the appropriate read or write select (RDS or WRS) instruction, and the necessary copy (CPY) instructions (for binary operations) or transfer instructions to prewritten input-output routines which perform conversion between decimal and binary and govern format (for decimal operations). When the list of the input-output statement is repetitive, table entries are made which will cause section 2 to generate the indexing instructions necessary to make the appropriate loops.

The treatment of statements which are neither input-output nor arithmetic is similar; *i.e.*, those instructions which can be compiled are compiled, and the remaining information is extracted and placed in one or more of the appropriate tables.

In contrast, arithmetic formulas are completely treated in section 1, except for open (built-in) subroutines, which are added in section 3; a complete set of compiled instructions is produced in the COMPAIL file. This compilation involves two principal tasks: 1) the generation of an appropriate sequence of arithmetic instructions to carry out the computation specified by the formula, and 2) the generation of (symbolic) tags for those arithmetic instructions which refer to subscripted variables (variables which denote arrays) which in combination with the indexing instructions to be compiled in section 2 will refer correctly to the individual members of those arrays. Both these tasks are accomplished in the course of a single scan of the formula.

Task 2) can be quickly disposed of. When a subscripted variable is encountered in the scan, its subscript(s) are examined to determine the symbols used in the subscripts, their multiplicative coefficients, and the dimensions of the array. These items of information are placed in tables where they will be available to section 2; also from them is generated a subscript combination name which is used as the symbolic tag of those instructions which refer to the subscripted variable.

The difficulty in carrying out task 1) is one of *level;* there is implicit in every arithmetic formula an order of computation, which arises from the control over ordering assigned by convention to the various symbols (parentheses, +, −, *, /, etc.) which can appear, and this implicit ordering must be made explicit before compilation of the instructions can be done. This explicitness is achieved, during the formula scan, by associating with each operation required by the formula a *level number*, such that if the operations are carried out in the order of increasing level number the correct sequence of arithmetic instructions will be obtained. The sequence of level numbers is obtained by means of a set of rules, which specify for each possible pair formed of operation type and symbol type the increment to be added to or subtracted from the level number of the preceding pair.

In fact, the compilation is not carried out with the raw set of level numbers produced during the scan. After the scan, but before the compilation, the levels are examined for empty sections which can be deleted, for permutations of operations on the same level which will reduce the number of accesses to memory, and for redundant computation (arising from the existence of common subexpressions) which can be eliminated.

An example will serve to show (somewhat inaccurately) some of the principles employed in the level-analysis process. Consider the following arithmetic expression:

$$A + B**C*(E + F).$$

In the level analysis of this expression parentheses are in effect inserted which define the proper order in which the operations are to be performed. If only three implied levels are recognized (corresponding to $+$, $*$ and $**$) the expression obtains the following:

$$+(*(**A))+(*(**B**C)*[+(*(**E))+(*(**F))]).$$

The brackets represent the parentheses appearing in the original expression. (The level-analysis routine actually recognizes an additional level corresponding to functions.) Given the above expression the level-analysis routine proceeds to define a sequence of new dependent variables the first of which represents the value of the entire expression. Each new variable is generated whenever a left parenthesis is encountered and its definition is entered on another line. In the single scan of the expression it is often necessary to begin the definition of one new variable before the definition of another has been completed. The subscripts of the u's in the following sets of definitions indicate the order in which they were defined.

$$u_0 = + u_1 + u_3$$
$$u_1 = * u_2$$
$$u_2 = **A$$
$$u_3 = * u_4 * u_5$$
$$u_4 = **B**C$$
$$u_5 = + u_6 + u_8$$
$$u_6 = * u_7$$
$$u_7 = **E$$
$$u_8 = * u_9$$
$$u_9 = **F.$$

This is the point reached at the end of the formula scan. What follows illustrates the further processing applied to the set of levels. Notice that u_9, for example, is defined as $**F$. Since there are not two or more operands to be combined the $**$ serves only as a level indication and no further purpose is served by having defined u_9. The procedure therefore substitutes F for u_9 wherever u_9 appears and the line $u_9 = **F$ is deleted. Similarly, F is then substituted for u_8 and $u_8 = *F$ is deleted. This elimination of "redundant" u's is carried to completion and results in the following:

$$u_0 = + A + u_3$$
$$u_3 = * u_4 * u_5$$
$$u_4 = **B**C$$
$$u_5 = + E + F.$$

These definitions, read up, describe a legitimate procedure for obtaining the value of the original expression. The number of u's remaining at this point (in this case four) determines the number of intermediate quantities which *may* need to be stored. However, further examination of this case reveals that the result of u_3 is in the accumulator, ready for u_0; therefore the store and load instructions which would usually be compiled between u_3 and u_0 are omitted.

Section 2 (Nelson and Ziller)

Throughout the object program will appear instructions which refer to subscripted variables. Each of these instructions will (until section 5) be tagged with a symbolic index register corresponding to the particular subscript combination of the subscripts of the variable [*e.g.*, (I, K, J) and (K, I, J) are two different subscript combinations]. If the object program is to work correctly, every symbolic index register must be so governed that it will have the appropriate contents at every instant that it is being used. It is the source program, of course, which determines what these appropriate contents must be, primarily through its DO statements, but also through arithmetic formulas (*e.g.* $I = N+1$) which may define the values of variables appearing in subscripts, or input formulas which may read such values in at object time. Moreover, in the case of DO statements, which are designed to produce loops in the object program, it is necessary to provide tests for loop exit. It is these two tasks, the governing of symbolic index registers and the testing of their contents, which section 2 must carry out.

Much of the complexity of what follows arises from the wish to carry out these tasks optimally; *i.e.*, when a variable upon which many subscript combinations depend undergoes a change, to alter only those index registers which really require changing in the light of the problem flow, and to handle exits correctly with a minimum number of tests.

If the following subscripted variable appears in a FORTRAN program

$$A(2*I + 1, 4*J + 3, 6*K + 5),$$

the index quantity which must be in its symbolic index register when this reference to A is made is

$$(c_1 i - 1) + (c_2 j - 1)d_i + (c_3 k - 1)d_i d_j + 1,$$

where c_1, c_2, and c_3 in this case have the values 2, 4, and 6; i, j, and k are the values of I, J, and K at the moment, and d_i and d_j are the I and J dimensions of A. The effect of the addends 1, 3, and 5 is incorporated in the address of the instruction which makes the reference.

In general, the index quantity associated with a subscript combination as given above, once formed, is not recomputed. Rather, every time one of the variables in a subscript combination is incremented under control of a DO, the corresponding quantity is incremented by the appropriate amount. In the example given, if K

is increased by n (under control of a DO), the index quantity is increased by $c_3d_id_jn$, giving the correct new value. The following paragraphs discuss in further detail the ways in which index quantities are computed and modified.

Choosing the Indexing Instructions; Case of Subscripts Controlled by DO's

We distinguish between two classes of subscript; those which are in the range of a DO having that subscript as its index symbol, and those subscripts which are not controlled by DO's.

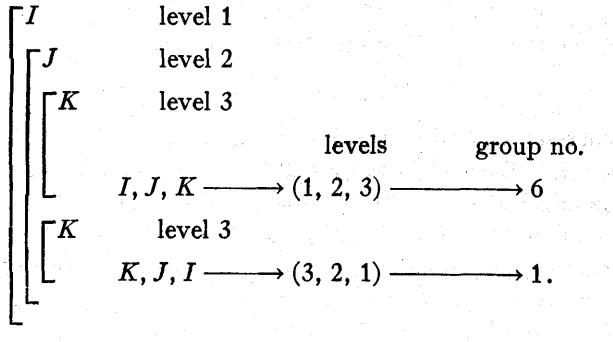
The fundamental idea for subscripts controlled by DO's is that a sequence of indexing instruction groups can be selected to answer the requirements, and that the choice of a particular instruction group depends mainly on the arrangement of the subscripts within the subscript combination and the order of the DO's controlling each subscript.

DO's often exist in *nests*. A nest of DO's consists of all the DO's contained by some one DO which is itself not contained by any other. Within a nest, DO's are assigned level numbers. Wherever the index symbol of a DO appears as a subscript within the range of that DO, the level number of the DO is assigned to the subscript. The relative values of the level numbers in a subscript combination produce a group number which, along with other information, determines which indexing instruction group is to be compiled.

The source language,

```
    DO 10 I = 1, 5
    DO 10 J = 1, 5
    DO  5 K = 1, J
 5  ··· A(I, J, K) ··· (some statement referring to
        A(I, J, K))
    DO 10 K = J, 5
10  ··· A(K, J, I) ···
```

produces the following DO structure and group combinations:



Producing the Decrement Parts of Indexing Instructions

The part of the 704 instruction used to change or test the contents of an index register is called the decrement part of the instruction.

The decrement parts of the FORTRAN indexing instructions are functions of the dimensions of arrays and of the parameters of DO's; that is, of the initial value n_1, the upper bound n_2, and the increment n_3 appearing in the statement DO 1 $i=n_1, n_2, n_3$. The general form of the function is $[(n_2-n_1+n_3)/n_3]n_3g$ where g represents necessary coefficients and dimensions, and $[x]$ denotes the integral part of x.

If all the parameters are constants, the decrement parts are computed during the execution of the FORTRAN executive program. If the parameters are variable symbols, then instructions are compiled in the object program to compute the proper decrement values. For object program efficiency, it is desirable to associate these computing instructions with the outermost DO of a nest, where possible, and not with the inner loops, even though these inner DO's may have variable parameters. Such a variable parameter (*e.g.*, N in "DO 7 $I=1, N$") may be assigned values by the programmer by any of a number of methods; it may be a value brought in by a READ statement, it may be calculated by an *arithmetic* statement, it may take its value from a *transfer* exit from some other DO whose index symbol is the pertinent variable symbol, or it may be under the control of a DO in the nest. A search is made to determine the smallest level number in the nest within which the variable parameter is not assigned a new value. This level number determines the place at which computing instructions can best be compiled.

Case of Subscripts not Controlled by DO's

The second of the two classes of subscript symbols is that of subscript symbols which are not under control of DO's. Such a subscript can be given a value in a number of ways similar to the defining of DO parameters: a value may be read in by a READ statement, it may be calculated by an arithmetic statement, or it may be defined by an exit made from a DO with that index symbol.

For subscript combinations with no subscript under the control of a DO, the basic technique used to introduce the proper values into a symbolic index register is that of determining where such definitions occur, and, at the point of definition, using a subroutine to compute the new index quantity. These subroutines are generated at executive time, if it is determined that they are necessary.

If the index quantity exists in a DO nest at the time of a transfer exit, then no subroutine calculations are necessary since the exit values are precisely the desired values.

Mixed Cases

In cases in which some subscripts in a subscript combination are controlled by DO's, and some are not, instructions are compiled to compute the initial value

of the subscript combination at the beginning of the outside loop. If the non-DO-controlled subscript symbol is then defined inside the loop (that is, after the computing of the load quantity) the procedure of using a subroutine at the point of subscript definition will bring the new value into the index register.

An exception to the use of a subroutine is made when the subscript is defined by a transfer exit from a DO, and that DO is within the range of a DO controlling some other subscript in the subscript combination. In such instances, if the index quantity is used in the inner DO, no calculation is necessary; the exit values are used. If the index quantity is not used, instructions are compiled to simulate this use, so that in either case the transfer exit leaves the correct function value in the index register.

Modification and Optimization

Initializing and computing instructions corresponding to a given DO are placed in the object program at a point corresponding to the lowest possible (outermost) DO level rather than at the point corresponding to the given DO. This technique results in the desired removal of certain instructions from the most frequent innermost loops of the object program. However, it necessitates the consideration of some complex questions when the flow within a nest of DO's is complicated by the occurrence of transfer escapes from DO-type repetition and by other IF and GO TO flow paths. Consider a simple example, a nest having a DO on I containing a DO on J, where the subscript combination (I, J) appears only in the inner loop. If the object program corresponded precisely to the FORTRAN language program, there would be instructions at the entrance point of the inner loop to set the value of J in (I, J) to the initial value specified by the inner DO. Usually, however, it is more efficient to reset the value of J in (I, J) at the end of the inner loop upon leaving it, and the object program is so constructed. In this case it becomes necessary to compile instructions which follow every *transfer* exit from the inner loop into the outer loop (if there are any such exits) which will also reset the value of J in (I, J) to the initial value it should have at the entrance of the inner loop. These instructions, plus the initialization of both I and J in (I, J) at the entrance of the outer loop (on I), insure that J always has its proper initial value at the entrance of the inner loop even though no instructions appear at that point which change J. The situation becomes considerably more complicated if the subscript combination (I, J) also appears in the outer loop. In this case two independent index quantities are created, one corresponding to (I, J) in the inner loop, the other to (I, J) in the outer loop.

Optimizing features play an important role in the modification of the procedures and techniques outlined above. It may be the case that the DO structure and subscript combinations of a nest describe the scanning of a two- or three-dimensional array which is the equivalent of a sequential scan of a vector; *i.e.*, a reference to each of a set of memory locations in descending order. Such an equivalent procedure is discovered, and where the flow of a nest permits, is used in place of more complicated indexing. This substitution is not of an empirical nature, but is instead the logical result of a generalized analysis.

Other optimizing techniques concern, for example, the computing instructions compiled to evaluate the functions (governing index values and decrements) mentioned previously. When some of the parameters are constant, the functions are reduced at executive time, and a frequent result is the compilation of only one instruction, a reference to a variable, to obtain a proper initializing value.

In choosing the symbolic index register in which to test the value of a subscript for exit purposes, those index registers are avoided which would require the compilation of instructions to modify the test instruction decrement.

Section 4 (Haibt) and Section 5 (Best)

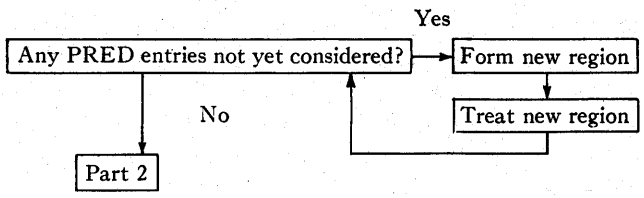
The result of section 3 is a complete program, but one in which tagged instructions are tagged only symbolically, and which assumes that there will be a real index register available for every symbolic one. It is the task of sections 4 and 5 to convert this program to one involving only the three real index registers of the 704. Generally, this requires the setting up, for each symbolic index register, of a storage cell which will act as an *index cell*, and the addition of instructions to load the real index registers from, and store them into, the index cells. This is done in section 5 (tag analysis) on the basis of information about the pattern and frequency of flow provided by section 4 (flow analysis) in such a way that the time spent in loading and storing index registers will be nearly minimum.

The fundamental unit of program is the *basic block;* a basic block is a stretch of program which has a single entry point and a single exit point. The purpose of section 4 is to prepare for section 5 a table of predecessors (PRED table) which enumerates the basic blocks and lists for every basic block each of the basic blocks which can be its immediate predecessor in flow, together with the absolute frequency of each such basic block link. This table is obtained by an actual "execution" of the program in Monte-Carlo fashion, in which the outcome of conditional transfers arising out of IF-type statements and computed GO TO's is determined by a random number generator suitably weighted according to whatever FREQUENCY statements have been provided.

Section 5 is divided into four parts, of which part 1 is the most important. It makes all the major decisions concerning the handling of index registers, but records

them simply as bits in the PRED table and a table of all tagged instructions, the STAG table. Part 2 merely reorganizes those tables; part 3 adds a slight further treatment to basic blocks which are terminated by an assigned GO TO; and finally part 4 compiles the finished program under the direction of the bits in the PRED and STAG tables. Since part 1 does the real work involved in handling the index registers, attention will be confined to this part in the sequel.

The basic flow of part 1 of section 5 is,



Consider a moment partway through the execution of part 1, when a new region has just been treated. The less frequent basic blocks have not yet been encountered; each basic block that has been treated is a member of some region. The existing regions are of two types: transparent, in which there is at least one real index register which has not been used in any of the member basic blocks, and opaque. Bits have been entered in the STAG table, calling where necessary for an LXD (load index register from index cell) instruction preceding, or an SXD (store index register in index cell) instruction following, the tagged instructions of the basic blocks that have been treated. For each basic block that has been treated is recorded the required contents of each of the three real index registers for entrance into the block, and the contents upon exit. In the PRED table, entries that have been considered may contain bits calling for interblock LXD's and SXD's, when the exit and entrance conditions across the link do not match.

Now the PRED table is scanned for the highest-frequency link not yet considered. The new region is formed by working both forward over successors and backward over predecessors from this point, always choosing the most frequent remaining path of control. The marking out of a new region is terminated by encountering 1) a basic block which belongs to an opaque region, 2) a basic block which has no remaining links into it (when working backward) or from it (when working forward), or which belongs to a transparent region with no such links remaining, or 3) a basic block which closes a loop. Thus the new region generally includes both basic blocks not hitherto encountered, and entire regions of basic blocks which have already been treated.

The treatment of hitherto untreated basic blocks in the new region is carried out by simulating the action of the program. Three cells are set aside to represent the object machine index registers. As each new tagged instruction is encountered these cells are examined to see if one of them contains the required tag; if not, the program is searched ahead to determine which of the three index registers is the least undesirable to replace, and a bit is entered in the STAG table calling for an LXD instruction to that index register. When the simulation of a new basic block is finished, the entrance and exit conditions are recorded, and the next item in the new region is considered. If it is a new basic block, the simulation continues; if it is a region, the index register assignment throughout the region is examined to see if a permutation of the index registers would not make it match better, and any remaining mismatch is taken care of by entries in PRED calling for interblock LXD's.

A final concept is that of index register activity. When a symbolic index register is initialized, or when its contents are altered by an indexing instruction, the value of the corresponding index cell falls out of date, and a subsequent LXD will be incorrect without an intervening SXD. This problem is handled by activity bits, which indicate when the index cell is out of date; when an LXD is required the activity bit is interrogated, and if it is on an SXD is called for immediately after the initializing or indexing instruction responsible for the activity, or in the interblock link from the region containing that instruction, depending upon whether the basic block containing that instruction was a new basic block or one in a region already treated.

When the new region has been treated, all of the old regions which belonged to it simply lose their identity; their basic blocks and the hitherto untreated basic blocks become the basic blocks of the new region. Thus at the end of part 1 there is but one single region, and it is the entire program. The high-frequency parts of the program were treated early; the entrance and exit conditions and indeed the whole handling of the index registers reflect primarily the efficiency needs of these high-frequency paths. The loading and unloading of the index registers is therefore as much as possible placed in the low-frequency paths, and the object program time consumed in these operations is thus brought near to a minimum.

Conclusion

The preceding sections of this paper have described the language and the translator program of the FORTRAN system. Following are some comments on the system and its application.

Scope of Applicability

The language of the system is intended to be capable of expressing virtually any numerical procedure. Some problems programmed in FORTRAN language to date include: reactor shielding, matrix inversion, numerical integration, tray-to-tray distillation, microwave propagation, radome design, numerical weather prediction, plotting and root location of a quartic, a procedure for playing the game "nim," helicopter design, and a number

of others. The sizes of these first programs range from about 10 FORTRAN statements to well over 1000, or in terms of machine instructions, from about 100 to 7500.

Conciseness and Convenience

The statement of a program in FORTRAN language rather than in machine language or assembly program language is intended to result in a considerable reduction in the amount of thinking, bookkeeping, writing, and time required. In the problems mentioned in the preceding paragraph, the ratio of the number of output machine instructions to the number of input FORTRAN statements for each problem varied between about 4 and 20. (The number of machine instructions does not include any library subroutines and thus represents approximately the number which would need to be hand coded, since FORTRAN does not normally produce programs appreciably longer than corresponding hand-coded ones.) The ratio tends to be high, of course, for problems with many long arithmetic expressions or with complex loop structure and subscript manipulation. The ratio is a rough measure of the conciseness of the language.

The convenience of using FORTRAN language is necessarily more difficult to measure than its conciseness. However the ratio of coding times, assembly program language vs FORTRAN language, gives some indication of the reduction in thinking and bookkeeping as well as in writing. This time reduction ratio appears to range also from about 4 to 20 although it is difficult to estimate accurately. The largest ratios are usually obtained by those problems with complex loops and subscript manipulation as a result of the planning of indexing and bookkeeping procedures by the translator rather than by the programmer.

Education

It is considerably easier to teach people untrained in the use of computers how to write programs in FORTRAN language than it is to teach them machine language. A FORTRAN manual specifically designed as a teaching tool will be available soon. Despite the unavailability of this manual, a number of successful courses for nonprogrammers, ranging from one to three days, have been completed using only the present reference manual.

Debugging

The structure of FORTRAN statements is such that the translator can detect and indicate many errors which may occur in a FORTRAN-language program. Furthermore, the nature of the language makes it possible to write programs with far fewer errors than are to be expected in machine-language programs.

Of course, it is only necessary to obtain a correct FORTRAN-language program for a problem, therefore all debugging efforts are directed toward this end. Any errors in the translator program or any machine malfunction during the process of translation will be detected and corrected by procedures distinct from the process of debugging a particular FORTRAN program.

In order to produce a program with built-in debugging facilities, it is a simple matter for the programmer to write various PRINT statements, which cause "snapshots" of pertinent information to be taken at appropriate points in his procedure, and insert these in the deck of cards comprising his original FORTRAN program. After compiling this program, running the resulting machine program, and comparing the resulting snapshots with hand-calculated or known values, the programmer can localize the specific area in his FORTRAN program which is causing the difficulty. After making the appropriate corrections in the FORTRAN program he may remove the snapshot cards and recompile the final program or leave them in and recompile if the program is not yet fully checked.

Experience in debugging FORTRAN programs to date has been somewhat clouded by the simultaneous process of debugging the translator program. However, it becomes clear that most errors in FORTRAN programs are detected in the process of translation. So far, those programs having errors undetected by the translator have been corrected with ease by examining the FORTRAN program and the data output of the machine program.

Method of Translation

In general the translation of a FORTRAN program to a machine-language program is characterized by the fact that each piece of the output program has been constructed, instruction by instruction, so as not only to produce an efficient piece locally but also to fit efficiently into its context as a result of many considerations of the structure of its neighboring pieces and of the entire program. With the exception of subroutines (corresponding to various functions and input-output statements appearing in the FORTRAN program), the output program does not contain long precoded instruction sequences with parameters inserted during translation. Such instruction sequences must be designed to do a variety of related tasks and are often not efficient in particular cases to which they are applied. FORTRAN-written programs seldom contain sequences of even three instructions whose operation parts alone could be considered a precoded "skeleton."

There are a number of interesting observations concerning FORTRAN-written programs which may throw some light on the nature of the translation process. Many object programs, for example, contain a large number of instructions which are not attributable to any particular statement in the original FORTRAN program. Even transfers of control will appear which do not correspond to any control statement (*e.g.*, DO, IF, GO TO) in the original program. The instructions arising from an arithmetic expression are optimally

arranged, often in a surprisingly different sequence than the expression would lead one to expect. Depending on its context, the same DO statement may give rise to no instructions or to several complicated groups of instructions located at different points in the program.

While it is felt that the ability of the system to translate algebraic expressions provides an important and necessary convenience, its ability to treat subscripted variables, DO statements, and the various input-output and FORMAT statements often provides even more significant conveniences.

In any case, the major part of the translator program is devoted to handling these last mentioned facilities rather than to translating arithmetic expressions. (The near-optimal treatment of arithmetic expressions is simply not as complex a task as a similar treatment of "housekeeping" operations.) A list of the approximate number of instructions in each of the six sections of the translator will give a crude picture of the effort expended in each area. (Recall that Section 1 completely treats arithmetic statements in addition to performing a number of other tasks.)

Section Number	*Number of Instructions*
1	5500
2	6000
3	2500
4	3000
5	5000
6	2000

The generality and complexity of some of the techniques employed to achieve efficient output programs may often be superfluous in many common applications. However the use of such techniques should enable the FORTRAN system to produce efficient programs for important problems which involve complex and unusual procedures. In any case the intellectual satisfaction of having formulated and solved some difficult problems of translation and the knowledge and experience acquired in the process are themselves almost a sufficient reward for the long effort expended on the FORTRAN project.