*This paper describes the system's* 7090/94 FORTRAN *compiler. Comment is made on the design problem and objectives.*

*The general structure and operation of the compiler are examined.*

*Indexing procedures for array reference and iteration control within the object programs produced by the compiler are detailed.*

# Design of an integrated programming and operating system

## Part IV: The system's FORTRAN compiler

**by R. Larner**

This paper is devoted to the 7090/94 version of the system's FORTRAN compiler, IBFTC, which translates FORTRAN IV language programs into MAP assembly language programs.

In addition to previous FORTRAN, FORTRAN IV includes language for double precision and complex arithmetic. Although consequently a more complicated language, this did not in itself necessitate significant design modifications. On the other hand, the compiler's IBSYS/IBJOB environment had a marked influence on its design since (1) certain translation and compilation functions are performed, respectively, by the system's assembler and loader, and (2) IOCS is available during compiler and object program operation. Thus, design of the compiler was substantially simplified. This, in turn, permitted more attention to other design problems, in particular:

- Generation of optimal object program code (in the sense of speed of execution) especially for the execution of iterative computations.
- Preservation of modularity within the compiler so that subsequent extension of its language or functional improvements in its parts could be readily accommodated.
- Attainment of higher translation speeds.

In this paper, we will (1) describe the over-all structure and

operation of the compiler and (2) discuss in some detail the nature of the object program code produced for array reference and iteration control. Discussion (2) is included since the speed of execution of an object program derived from a source program written in FORTRAN is very often governed by the method employed for array reference and iteration control. This matter has received a good deal of attention in the present compiler design.

The next section of the paper describes the general structure and operation of the compiler and the final section details the form and placement of the indexing instructions within an object program.

## Structure and operation of the compiler

The compiler produces three sets of MAP object program instructions:

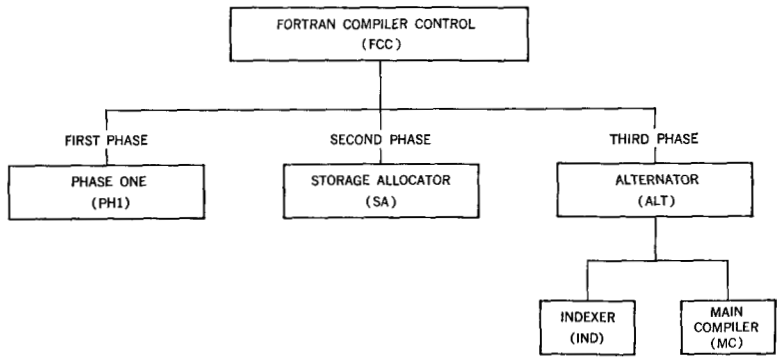
*Set 1.* Storage allocating pseudo-operations for data and variables.
*Set 2.* Logical and arithmetic instruction sequences corresponding to the executable source program statements.
*Set 3.* Indexing instructions for array reference and iteration control.

For general reasons of efficiency, the translation is performed in three phases. The first phase scans and internally encodes the entire source program, producing an intermediate file which is later used in compilation of Set 2 object program code. During the second phase, Set 1 is compiled. Sets 2 and 3 are produced during the third phase.

compiler structure

Basically, the compiler consists of a small control program, the FORTRAN Compiler Control program (FCC), and programs for the three phases mentioned above. The latter programs are designated, respectively, as Phase One (PHI), the Storage Allocator (SA), and the Alternator (ALT). The Alternator is actually the control program for the final phase and its function is to alternate control between two principal subroutines, the Indexer (IND) and the Main Compiler (MC).[1] The hierarchy of control among these programs is shown in Figure 1, with the controlling programs

Figure 1 IBFTC subprogram organization



above and the controlled below. To effect one translation, FCC calls the three phases in succession.

IBFTC is a collection of closed subroutines. In addition to those already mentioned, there are four other important subroutines:

- Diagnostic Routine. A utility subroutine used for issuing diagnostic messages and assigning severity levels to them.
- Table X Routine. An I/O routine tailored to accept one Table X statement at a time as output from Phase One, and to issue one Table X statement at a time to Phase Two.
- Table Routine. A routine which dynamically assigns fixed length blocks of core storage to a variety of tables, chains the the blocks together, and performs routine information retrieval functions.
- MAP Code Generator. An interpretive string concatenation subroutine used to fabricate the IBMAP input cards.

Figure 2 illustrates the usage of the latter programs; again the controlling programs are shown above the controlled.

To the monitor, FCC is a closed subroutine representing the entire compiler and for that reason is sometimes referred to as IBFTC.

first phase of compilation

Phase One makes one pass over the source program, reducing it statement by statement to tabular form. This process results in the formation of the following tables:

- *Table X.* This table is a file containing an encoded representation of each executable statement appearing in the source program. Each executable source program statement is represented by one or more Table X statements. Each of the Table X statements contains, as its first word, an internal formula number and a number identifying the statement type. There is a unique format for the text after the identifier for each type. For non-arithmetic text, the entries in Table X are, for the most part, straightforward tabulations of the source statements. For arithmetic statements, an entry consists solely of an ordered Lambda string, which is a

**Figure 2 IBFTC subroutine usage**

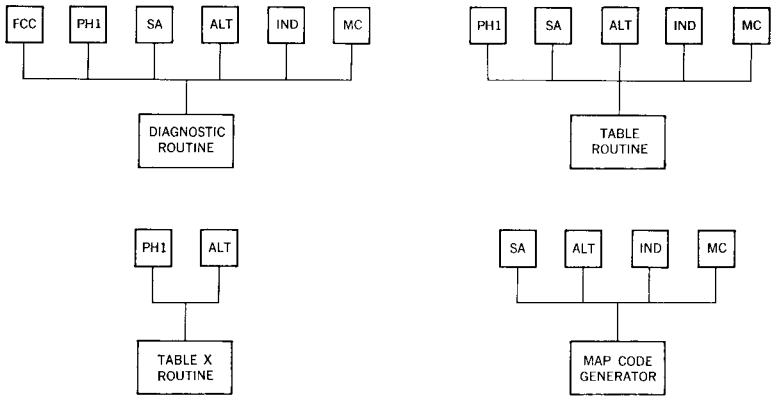


Table 1 Example of a Lambda string

| *Level no.* | *Operator* | *Operand* |
|---|---|---|
| 2 | * | C |
| 2 | * | D |
| 1 | + | 2 |
| 1 | + | B |
| 0 | = | A |

string of words describing the arithmetic (or logical) expression, ordered correctly for computation, but independent of machine language. The Lambda string requires one word per entry. Each entry is called a *Lambda triple*, where "triple" refers to a *level number*, an *operand*, and an *operator*. The level number identifies a subexpression. The level numbers may be operands of triples in other subexpressions. To illustrate, the Lambda string for the expression A = B + C*D is shown in Table 1. For additional detail on Lambda strings, the reader is referred to the literature.[2]

- *Name Tables.* The Name Table is a dictionary of all source program variables, function names, and subroutine names. Each entry contains two words: the first contains the alphanumeric representation of the name and the second contains a bit string describing the name in terms of type and usage.

- *TR Tables.* A family of tables which are constructed, used, and abandoned by the compiler as it performs the translation process. A single table is made up of a set of blocks that are chained together in a master pool of erasable blocks. The blocks of a particular table are not necessarily sequential and there is no minimum or maximum number set aside for any particular one. Storage allocation for the TR tables is controlled by the Table Routine. During Phase One, TR tables are formed to store EQUIVALENCE, COMMON, DIMENSION, FORMAT, and DATA statement information.

Internally, Phase One may be divided into a:

- Control program.
- Classification routine.
- Dictionary scan.
- Family of statement processors.
- Arithmetic master scan.
- Arithmetic level analysis subroutine.
- A Lambda string reordering and optimization subroutine.

In addition there is a common set of utility subroutines, the most important of which is a group of character collecting routines. These routines collect symbols and punctuation from the source cards in any of several delimiting modes. The Phase One control program collects a statement from the monitor and calls the classification routine to classify the statement as either arithmetic or non-arithmetic. If non-arithmetic, it calls the dictionary scan to identify the statement, then calls the correct statement processor. If arithmetic, it calls the arithmetic master scan. The arithmetic master scan may also be used by other statement processors, such as those used to process statements involving CALL and IF. All statement processors return control to the Phase One control program when finished.

The arithmetic master scan collects, in order across the statement, certain characters including operators and operands. This collection of characters is called an *N-word* and contains the

nucleus of what will become a Lambda triple. Every time an N-word is collected, the level analysis subroutine is called. This subroutine, as a result of its many calls by the arithmetic master scan, generates a Lambda string for the arithmetic expression. At the end of the expression, the arithmetic master scan calls the reordering and optimization routine which reorders the Lambda string for computation, putting it in proper form for inclusion in the text of a Table X statement.

**second phase of compilation**

During the second phase of compilation, the Storage Allocator processes the Name Table and those TR tables which catalog EQUIVALENCE, COMMON, DIMENSION, FORMAT, and DATA source statements. As its first task, the Storage Allocator compiles FORMAT statements. The TR space occupied by the FORMAT table is freed immediately afterward.

COMMON and EQUIVALENCE variables are processed next. A string of variables with separating increments is set up for each COMMON block. For each of these strings, matching strings may be produced from the EQUIVALENCE table information. To lay out all of the strings, several temporary TR tables must be employed. Considerable checking is done to insure proper parity of double word variables. For each string, a set of BSS and EQU pseudo-operations is compiled, allocating the storage for the variables. Here, good use is made of the multi-location counter feature of the MAP language, i.e., each COMMON block label is defined in the object program as a control section which, in turn, is assigned a unique symbolic location counter of the same name. Thus, COMMON statements are simply compiled into a USE[3] pseudo-operation followed by a sequence of BSS pseudo-operations. Next, a similar matching of non-COMMON variables against EQUIVALENCE strings is performed, and the storage allocation pseudo-operations again are compiled.

A scan is then made over the Name Table to compile the storage allocation for variables not appearing in COMMON or EQUIVALENCE statements. Lastly, the DATA statement tables are processed, i.e., location counters are reset to their appropriate variable definitions and the data is compiled. This technique allows "scatter" loading of data into COMMON blocks from several BLOCK DATA programs.

**third phase of compilation**

Normally[4], during the third and last phase of compilation, the executable part of the object program is compiled. One sequential pass is made over Table X to compile the executable code. During this phase, compilation is controlled by the Alternator, which controls two subprograms, the Indexer and the Main Compiler.

The Alternator controls the statement-by-statement processing of Table X. By performing a concurrent scan of TR tables which catalog branches and DO loops, the Alternator can determine when its position in Table X corresponds to the beginning of a *basic block*[5] or DO nest. Everytime it is at one of these positions, it calls the Indexer, which immediately compiles all of the indexing

instructions for that basic block or DO nest. Then, the Main Compiler is called once for each Table X statement within range of the DO. Each time the Indexer is called, it generates certain TR tables. These are used by the Main Compiler for making array references within the range just processed by the Indexer, and by the Alternator for collating the indexing instructions with the Main Compiler instructions.

After processing all Table X statements, the Alternator compiles the prologue for the initialization of program argument references, and compiles storage allocation pseudo-operations for temporary storage as required by the compiled statements. When this is finished an END card is compiled and the object program file is complete.

The Main Compiler (MC), when called, translates a given Table X entry into object program language (MAP). A pointer to the Table X entry is passed to MC for use as an argument. A minimal control program within MC inspects the Table X identifier word, compiles a location symbol for it, and calls a statement processor according to the statement type. Corresponding to each type of Table X statement, there is a unique statement processor. For those statement processors dealing with Lambda strings, there is a common routine which translates Lambda strings into object code. The Lambda routine, together with its subroutines, constitutes the bulk of the Main Compiler.

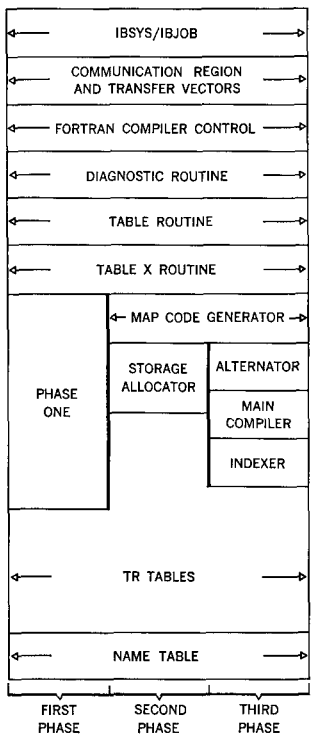
The order in which one carries out the "and" and "or" operations of a complex logical expression may affect the time required for its evaluation. IBFTC translates logical expressions by means of an "anchor point" technique. For example, the evaluation of L1 .AND. L2 .AND. L3 is carried out by evaluating L1 first. If it is false, then the evaluation of L2 and L3 is skipped. Non-zero logical variables are considered true, and zero variables are considered false. Hence, the 7090 instructions NZT (non-zero test) and ZET (zero test) are employed, always followed by transfers to some "true" or "false" point in the expression. The equivalent of this technique is employed for relational expressions, after the right side of the relational operator has been subtracted from the left. DeMorgan's theorem[6] is not applied to the Lambda string in Table X by Phase One. It is applied by the Lambda routine when determining the proper test and branch for a particular logical or relational operation.

The Main Compiler builds up several TR tables during the course of a compilation for processing by the Alternator after the Table X pass. These tables catalog temporary storage requirements of compiled arithmetic expressions and points at which references to program arguments must be initialized in the prologue.

The Indexer, by scanning appropriate TR tables, generates a program's indexing instructions. This function will be indicated in some detail by the discussion of indexing in the object program which is given in the last section of the paper.

Figure 3 shows IBFTC memory utilization (not to scale). The

**Figure 3 IBFTC memory utilization**



upper memory boundary is a parameter furnished by the Basic Monitor (IBSYS). The boundary between the TR tables and the Name Table is not fixed until all Name Table entries have been made, thereby allowing maximum utilization of erasable core storage for TR tables. All other boundaries are functions of assembly parameters. As indicated at the bottom of the figure, the three core maps corresponding to the three phases of compilation are shown.

table handling

The procedure adopted for handling general core storage tables (the TR tables) has made it possible to avoid all but one I/O scratch file during the translation. As previously mentioned, there is no fixed maximum or minimum space set aside for any one table. This approach has proved successful in avoiding table overflow and has greatly simplified the development of the compiler.

The TR table area is shown in Figure 3. Available space (depending on which phase is operating) is segmented by the Table Routine into a set of blocks of fixed length. When one of the phases requires space, the Table Routine is called to furnish a block. When more space is required, an identical call is made. The blocks are chained together forwards and backwards by the Table Routine using the first word of each block. The blocks for a given table are not necessarily contiguous. Request calls may also be made to the Table Routine to locate the first entry of a given table, or to release all blocks of a given table, or to release leading or trailing blocks from a given block.

Table 2 Array placement in core

| *Location* | *Element* |
|---|---|
| A | A(1, 1, 1) |
| A+1 | A(2, 1, 1) |
| A+2 | A(3, 1, 1) |
| A+3 | A(1, 2, 1) |
| A+4 | A(2, 2, 1) |
| . | . |
| . | . |
| . | . |
| A+23 | A(3, 4, 2) |

Sequential scanning is of course slower with TR tables than with sequentially stored tables; however, most of the TR tables are not scanned but, rather, are referred to indirectly through pointers in Table X and in other TR tables.

## Object program indexing and iteration control

Generally, the speed of execution of FORTRAN-compiled object programs will be governed by the coding for the indexing associated with programmed array references and iterations. We now examine the nature of the object code produced by IBFTC for indexing procedures.

FORTRAN arrays are stored in core columnwise in ascending memory sequence. For instance, a 3 by 4 by 2 array A would be stored as shown in Table 2. In this example the location of the general element, A(I,J,K), is A+12(K−1)+3(J−1)+(I−1).

In general, if A had dimension (M,N,P) then the location of the element A(I,J,K) would be A+MN(K−1)+M(J−1)+(I−1).

Distinct procedures are employed to produce code corresponding to parts of the source program which are, respectively, within and not within the range of DO statements.

non-DO case

The part of the source program outside the range of all DO statements is divided into a set of *basic blocks*, defined as contiguous stretches of program into which there are no transfers. Basic

blocks are treated by the translator as units for purposes of optimization. At the beginning of a block, array element references within it are considered undefined; these indices must all be computed and/or set either at the beginning of the basic block or within it.

The base address of the array, adjusted by the contribution of any constant terms in the complete address expression, is inserted in the address field of an instruction which references a subscripted variable. This address is modified by an index register containing, in two's complement form, the appropriate increment (a function of subscript variables and coefficients).

As an example, assume there is a 5 by 10 array A and a source reference A(I,3*J−2). The complete address of A(I,3*J−2) is A+(3J−3)5+I−1. The address field of the instruction is set equal to A−16 and the index register is set equal to the two's complement of 15J+I. A source reference A(2,2) will simply produce an object program reference A+6. Note that the last dimension of an array, 10 in the examples given, has no effect in either the address or index.

For optimization within a basic block, the same index registers are used for all quantities that are expressed by the same formula and assignment priority is given to those formulas which occur most frequently. The index quantity is computed and loaded at the earliest point in the basic block at which all the subscript variables have attained their last value.

According to a control card option, up to seven index registers may be used by the object program. Index register 4 is reserved for immediate usage only; that is, for calling sequence usage, address computation, and for index quantities for which there is no other available register. The latter are termed *spill tags*. At the beginning of a basic block, the full set of index registers is available and assignments are made, according to frequency priority, in the order 1, 2, 3, 5, 6, 7. If there are more formulas than available registers, the excess become spill tags, which are loaded into index register 4 before every usage. It has been found that spill tags occur very rarely, even when only registers 1, 2, and 4 are available to the program.

Table 3 displays an example of a basic block composed of six equations. It is assumed that only three index registers (1, 2, and 4) will be available for the resulting object program. Equations 1, 3, 5, and 6 involve reference to array elements. Equations 2 and 4 (not explicity given) refer to setting the indices J and K appropriately to accomplish the particular computation required by the program. To follow the example, it will be useful to note that the D array has 2 rows. Examination of the object code reveals that index register 1 is used in addressing: A(K) appearing in Equations 1 and 3; B(K) appearing in Equation 5; and A(K−1) and A(K+1) which appear in Equation 6. Index register 2 is used in addressing: D(I,I−1) appearing in Equation 1; and D(I,I−1), D(I,I), and B(3*I−3) which appear in Equation 5. Index registers

Table 3 Compilation of a basic block

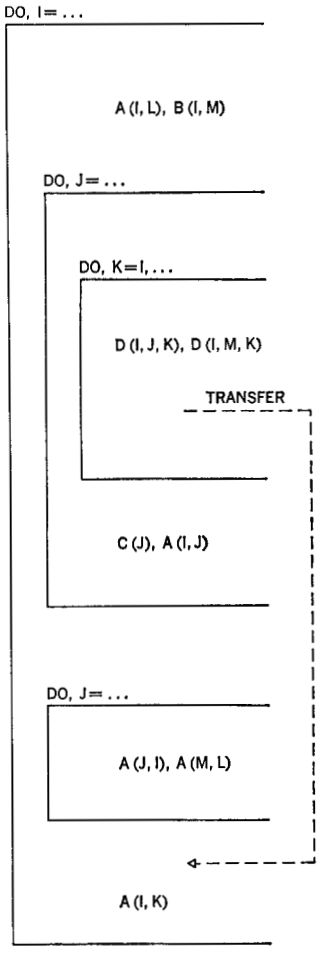
| *Source Program* | *Object Program* | |
|---|---|---|
| (1) A(K)=B(J)+D(I, I−1) | LAC | K, 1 |
| | LDQ | I |
| | VLM | =3B17, , 18 |
| | PAC | , 2 |
| | LAC | J, 4 |
| | CLA | B−1, 4 |
| | FAD | D−7, 2 |
| | STO | A−1, 1 |
| (2) J=··· | · | · |
| | · | · |
| | · | · |
| (3) A(J)=A(K) | CLA | A−1, 1 |
| | LAC | J, 4 |
| | STO | A−1, 4 |
| (4) K=··· | · | · |
| | · | · |
| | · | · |
| (5) D(I, I−1) =D(I, I)+B(3*I−3)−B(K) | LAC | K, 1 |
| | CLA | D−1, 2 |
| | FAD | B−4, 2 |
| | FSB | B−1, 1 |
| | STO | D−7, 2 |
| (6) A(K−1)=A(K+1)+A(L) | LAC | L, 4 |
| | CLA | A−1, 4 |
| | FAD | A, 1 |
| | STO | A−2, 1 |

1 and 2 are loaded only once. Index register 4 is used for the spill tags associated with B(J) in Equation 1, A(J) in Equation 3 and A(L) in Equation 6 so that it is necessary to load the register three times. Note that the spill tags have been associated with items less frequently used in the program.

**DO case**

The procedure for indexing with DOs present is quite different from the non-DO case. A complete DO nest is treated as an optimization unit, whereas the basic block was used in the non-DO case. An index register quantity in a DO nest is always an integral multiple of the index of the immediately controlling DO. The address field of an instruction contains the value of any remaining terms of an array reference. As a result, the address field may require a modification within the body of the object program. The objective is to minimize the number of instructions within the innermost DO loops. Since the address field value is not a function of the immediately controlling DO index, it may usually be initialized outside the loop, often outside bounding loops as well. The procedure reduces the number of index register quantities required to be active within the DO. As an example, assume we have a 5 by 10 array A, a 5 by 2 by 10 array Z and a singly dimensioned array Q. Within the immediate range of a DO with index J, references to A(J,I), A(J,P), Z(J,3*L,2), A(J+2,P−3) all require the same index register quantity J. Further, A(I,J), Z(L,J,L), Q(5*J) all require the same index register quantity 5J.

Figure 4 Use of index registers a DO nest




Instructions to perform address modifications are generated and placed outside the nest, or at a position as near to the outside as possible, depending on points of redefinition of its factors. This "push out" is affected by transfers in the nest, fixed point variables on the left of arithmetic statements, input lists, CALL arguments, etc. Similarly, the computation of initial load values, test values and increment values is pushed out as far as possible.

Consider the DO nest example shown in Figure 4 which has array element symbols shown at various points to indicate that, at the particular point, the program references the array. In the object program, the address modification instructions will be placed as indicated in Table 4. C(J) requires no address modification. A(M,L) requires no index register. The initial load value for the index quantity for D(I,J,K) and D(I,M,K), being variable and a function of I, is computed outside the J loop. No other load values require object program computation. All initial loads are performed just outside the controlling DO loop.

Table 4 Placement of array reference instructions

| *Array reference* | *Object program placement* |
|---|---|
| A(I, L) | outside the nest |
| B(I, M) | outside the nest |
| D(I, J, K) | outside the K loop |
| D(I, M, K) | outside the first J loop |
| A(I, J) | outside the first J loop |
| A(J, I) | outside the second J loop |
| A(M, L) | outside the nest |
| A(I, K) | immediately following the transfer destination shown |

In this nest, 5 index registers will suffice for the complete nest with no saves or reloads required. If only 2 registers are available, then they will suffice for the I and J loops, but will be saved and reloaded at the peripheries of the K loop. No spill tags are required in the example as the complete set of available registers may be used in the immediate range of every DO. As in the non-DO case, spill tags always require index register 4 and are reloaded before every usage.

ACKNOWLEDGMENT

The author gratefully acknowledges the assistance in early planning and subsequent development of IBFTC by Mr. Maurice Ackroyd (the Storage Allocator), by Mr. David Stemple (the Indexer), by Mr. Harold Stern (Arithmetic level analysis) and by Mr. Richard Stuchinski (Phase One organization).

FOOTNOTES

1. In other literature, this principal set of programs operative during the third phase of compilation (the Alternator, Indexer, and Main Compiler) have been referred to collectively (and incongruously) by the formal name *Phase Two.*
2. This notational technique is essentially that developed by Peter B. Sheridan and reported in his paper, "The Arithmetic Translator—Compiler of the IBM FORTRAN Automatic Coding System" appearing in Communications of the ACM, February, 1959.
3. "USE" is a MAP pseudo-operation which specifies a location counter to be used for relative location assignment of successive instructions. The MAP language allows multiple location counters to be specified.
4. In case of a BLOCK DATA compilation, the third phase is not executed. The purpose of BLOCK DATA compilation in IBFTC is to allow the insertion of literals in COMMON areas.
5. A basic block is a contiguous set of program instructions which is outside the range of any DO statement and into which there are no transfers.
6. The application of DeMorgan's theorem to logical expressions allows, for example, the expression ".NOT. (U .AND. V)" to be replaced by "(.NOT. U) .OR. (.NOT. V)." Such a distribution of .NOT.'s yield expressions which are most easily computed (require the fewest machine instructions).