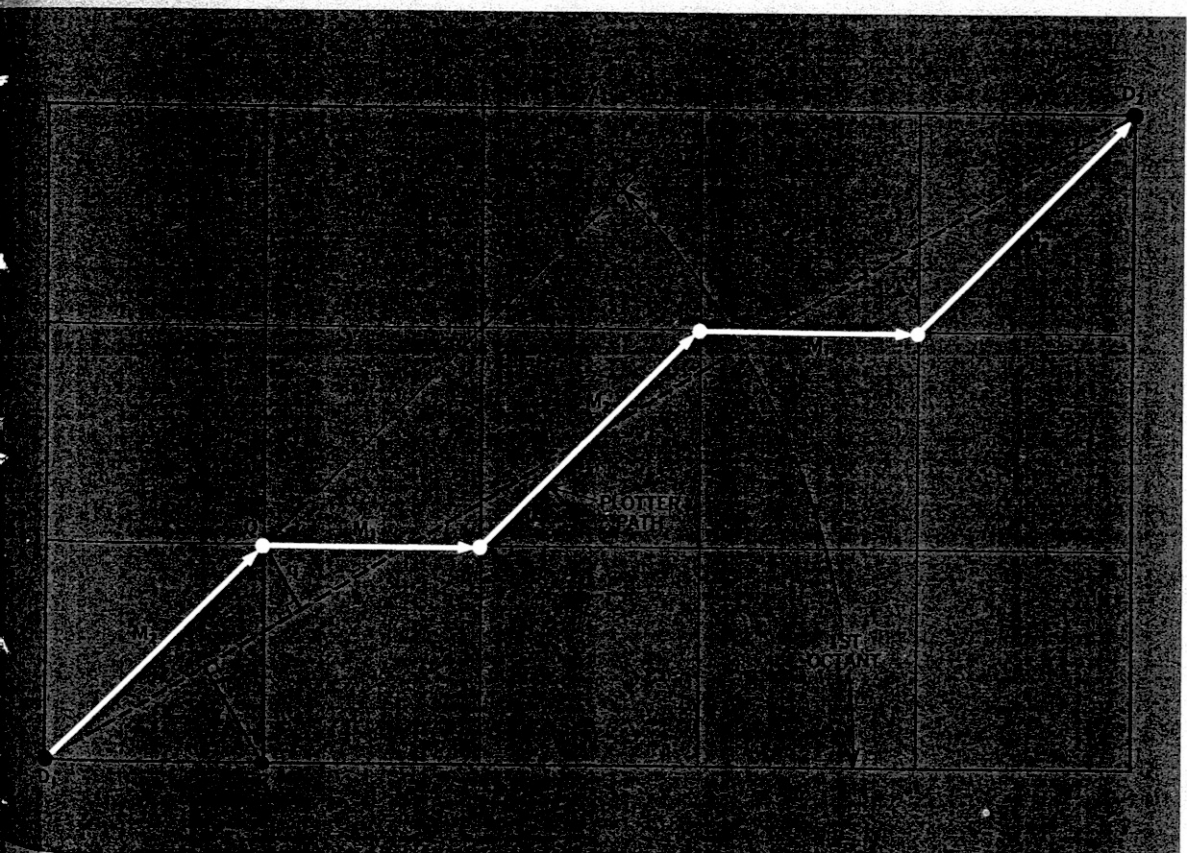
IBM

Volume Four | Number One | 1965



This paper discusses a compiler organization in which phases act sequentially on a source program held in core storage.

A brief description of each phase of the 1401 FORTRAN *compiler is given to illustrate the general scheme.*

Serial compilation and the 1401 FORTRAN compiler

by L. H. Haines

The IBM 1401 FORTRAN compiler[1] was designed as a set of phases that operate sequentially on the source program. The source program having been placed in core storage, the compiler phases enter core one at a time. Each phase overlays its predecessor, operates on the source program and, in turn, is overlaid by the next phase of the sequence. Thus, in contrast to the customary technique of passing the source program against the compiler in core, the compiler is passed against the source program which resides in core. It is assumed that the source program is more concise than the object program, and that an object program of interest can be accommodated in core.

The 1401 FORTRAN compiler was designed for a minimum of 8000 positions of core, with tape usage being optional. The fundamental design problem stems from the core storage limitation. Because the average number of instructions per phase and the number of phases selected are inversely related (at least to a significant degree), the phase organization was employed to circumvent the core limitation. The 1401 FORTRAN compiler has 63 phases, an average of 150 instructions per phase, and a maximum of 300 instructions in any phase.

efficiency of the compilation technique

Experience with the compiler suggests that, even though the basic compilation technique requires repetitive scanning of the source program, the time required for this redundant effort is small. The experience lends some credence to the following argument.

If a phase is decomposed into two serially acting phases, assume that the average execution time is increased by a multiplicative factor r. If each component phase is again decomposed into two phases, the average total execution time of the resulting four phases increases to r^2 times the original. After k decompositions, 2^k phases result, with an average total execution time of r^k times the original. Under this assumption, it follows that an n-phase compiler takes $r^{\log_2 n}$ times longer than a comparable one-phase compiler. Because some phases do not involve scanning, this estimate may tend to be high.

Based on experience with the present compiler, it is conjectured that $r \simeq 1.05$ and that the redundant work occasioned by the use of 63 phases increases compilation time by a factor of $1.05^{\log_2 63} \simeq 1.3$.

However, this technique reduces compilation time elsewhere, so that the net increase can be expected to be less than conjectured. Since the phases act serially, tape searching is unnecessary. Moreover, no external sorts or merges are required. After one pass of the processor, the object program lies in core storage, ready for execution.

Also, the present technique permits an approach to coding that tends to reduce compilation time. Decomposition into an appropriate number of phases often results in more available core space for execution of an individual phase than the minimum needed.

The typical phase requires only on the order of 150 instructions, although twice that number are accommodated by the allocated region in core. Thus, most of the code can be written for efficient execution without regard to economy in the number of instructions employed. Other advantages inherent in this type of coding are that it is usually faster to write and easier to debug.

compilation time

Except for certain extreme cases in which the object program fills almost all of core, the compilation time in seconds t is approximated by a linear function of the number of FORTRAN statements n (the statements assumed to be of average complexity). If the compiler is on tape, reads the source program from cards, and punches a self-loading machine language object program, then $t = (0.96n + 72)$ seconds. As a load-and-go system, with the punching of the object program suppressed, $t = (0.76n + 22)$ seconds. If the system is on cards, an additional 173 seconds are required.

compiler phases

The principal function of each phase of the compiler[2] is indicated below. Secondary functions are subordinated; for example, error checking occurs in almost every phase, but is seldom mentioned.

Phase 00 — Snapshot. Loads a snapshot routine into 350 positions of core storage. This routine lists a specified amount of core storage.

Phase 01 — System Monitor. Brings in the next phase from the system tape or initiates reading of the next phase from cards,

depending on whether the compiler is used as a tape or card system. The monitor and snapshot routines are the only ones that exist in storage throughout compilation. Because the phases act serially, very little is required of this phase which consists of only 20 instructions.

Phase 02 — Loader. Stores the entire source program, statement by statement, with all non-significant blanks eliminated. The source program is stored backwards in order to use the 1401 machine instructions that cause address registers to decrement when processing data. Appended on the right of each statement is a three-position internal sequence number (001 for the first statement, 002 for the second, etc.). The sequenced source program is printed.

Phase 03 — Scanner. Determines the type of each statement and appends a code on the right of each statement. For example, D for DO, S for STOP, I for DIMENSION statements, etc.

Phase 04 — Sort I. Determines if there is enough free storage to expand each statement by three characters.

Phase 05 — Sort II. Statements of the same type are chained together. Each statement expands by three characters—the machine address of the next statement of the same type.

Phase 06 — Sort III. The source program is sorted[3] internally by statement type. The order of sorting is determined by the order in which statements of a given type undergo specific processing by subsequent phases. For example, since DIMENSION statements are processed (Phase 09) before DO statements (Phase 46), the DIMENSION statements are grouped together lower in core than the DO statements.

Phase 07 — Insert Group Mark. This is a housekeeping phase.

Phase 08 — Squoze. The words that helped define the type of each statement are eliminated, shrinking the source program. For example, the word "DIMENSION" in DIMENSION statements is eliminated.

Phase 09 — Dimension I. The DIMENSION statements are scanned, and an array table is generated in free storage. Each table element consists of the name of an array, its dimensions, and sufficient space for additional data to be generated by Phases 11 and 12.

Phase 10 — Equivalence I. Adds simple variables present in EQUIVALENCE statements to the array table. These variables are treated, in effect, as one-element arrays.

Phase 11 — Equivalence II. The array table is altered to show the relationship between arrays. Equated arrays are chained together. Essentially, the procedure makes known to every array whose first element is equivalent to a secondary element of another array the "distance" to the first element of the latter array.

Phase 12 — Dimension II. The object-time addresses which de-

limit each array are computed and inserted in the array table. These addresses are also printed.

Phase 13 — Variables I. The entire source program is scanned for variables. The following changes are made directly within the text of the source program:

- Simple variables are tagged for later processing by Phase 16.
- Subscripted variables with constant subscripts are replaced by the object-time address of the designated array element.
- All other subscripted variables are put into a canonical form which specifies a computation in terms of variables and constants for determining the object-time address of the array element specified.
- Non-subscripted array names appearing in lists are replaced by the object-time address that delimit the array named.
- Non-subscripted array names appearing elsewhere are replaced by the object-time address of the first element of the array.

Phase 14 — Variables II. All free storage (including the array table) is cleared and partitioned into two tables areas—Tables I and II. Parameters needed for the randomizer of Phase 16 are computed.

Phase 15 — Variables III. Does housekeeping for Phase 16.

Phase 16 — Variables IV. The source program is scanned twice for simple variables (already tagged by Phase 13). During the first scan, the compiler looks for variables being defined, i.e., those appearing on the left of an equal sign or in input lists. By means of a randomizer that computes an indirect address (a Table I address at which is located a Table II address), each such variable and its object-time address is stored uniquely and sequentially (one after another) in Table II. The object-time address replaces the variable name in the source program. During the second scan, all other variables are picked up, and the same process is carried out, except that undefined variables are noted and Table II entries are flagged whenever referenced.

Phase 17 — Variables V. Table II is scanned. The absence of a flag indicates an unreferenced variable. The object-time address of each variable is printed.

Phase 18 — Constants I. The entire source program is scanned for constants. Each constant encountered is normalized and tagged. Tables I and II are destroyed.

Phase 19 — Constants II. All free storage is again cleared and partitioned into two table areas— Tables I and II. Parameters needed for the randomizer of Phase 20 are computed.

Phase 20 — Constants III. The entire source program is scanned for normalized constants (tagged by Phase 18). By means of a randomizer (as in Phase 16), each normalized constant is stored uniquely and sequentially (next to one another) in Table II.

Once stored, these constants are at their object-time address and are not disturbed for the remainder of the compilation. The object-time address replaces the normalized constant directly in the text of the source program.

Phase 21 — Subscripts. Under the action of Phases 14 through 20, the canonical form for subscripted variables (see Phase 13) now specifies a computation in terms of object-time addresses. This phase simplifies the computation, leaving only the object-time addresses which serve as parameters for a closed subroutine at object time.

Phase 22 — Statement Numbers I. Statement numbers that appear in the source program are reduced to a 3-character representation. Statement numbers within the body of a statement are moved to the front of the statement.

Phase 23 — Format I. FORMAT statements are checked to ensure that they are referenced by input/output statements.

Phase 24 — Format II. The object-time format strings are developed and stored immediately preceding the constants at the lower end of storage.

Phase 25 — Lists I. Duplicate lists are checked and eliminated to optimize storage at object time.

Phase 26 — Lists II. The object-time list strings are developed and stored immediately to the left of the format strings at the lower end of storage.

Phase 27 — Lists III. Each input/output statement is reduced to the address of the list string (when present), the address of the format string (when present), and the tape unit number (where applicable).

Phase 28 — Statement Numbers II. All free storage is again cleared and partitioned into two tables—Tables I and II. Parameters needed for Phase 29 are computed.

Phase 29 — Statement Numbers III. By means of a randomizer (as in Phase 16), statement number representations (Phase 22) appearing *within* statements are stored uniquely, one after another, in Table II. Each such representation is replaced by the machine address at which it is stored.

Phase 30 — Statement Numbers IV. Statement number representations are matched against Table II entries via the randomizer of Phase 29. The sequence number of the statement (Phase 02) replaces the Table II entry. The Table II address (which now contains the sequence number) replaces both the representation and the sequence number in the source program. Undefined and multiply-defined statement numbers are checked.

Phase 31 — Statement Numbers V. Table II is scanned for unreferenced statement numbers (representations).

Note: Whereas the source program originally was composed of constants, variables, and statement numbers of arbitrary length, it is now highly structured and is composed of 3-character machine addresses. Machine addresses substituted for constants and variables are their object-time addresses. Machine addresses substituted for statement numbers are indirect addresses which currently reference Table II entries—sequence numbers of labeled statements. Eventually (Phase 51) these indirect addresses are replaced by the object-time addresses of the labeled statements, i.e., the addresses of object programs compiled from these statements.

Phase 32 — Input/Output I. The residue of each I/O statement other than BACKSPACE, REWIND, and END FILE is substituted into an object-time mask. The filled-in mask and an identifier are stored in lower core immediately adjacent to the last entry in Table II. The identifier is the sequence number (Phase 02) or, only when the statement originally had a statement number, the machine address of the sequence number stored in Table II (Phase 30).

Phase 33 — Arith I. All arithmetic expressions appearing in the source program are scanned. Switches are set to indicate which function routines must be loaded by Phase 52. Minor changes are made to expressions, and sufficient error testing is done to expedite Phase 34.

Phase 34 — Arith II. By means of a transition matrix,[4] each arithmetic expression is broken down into a sequence of one or two operand sub-expressions involving temporary dummy storage locations.

Phase 35 — Arith III. Initialization for Phase 36.

Phase 36 — Arith IV. Redundant references to temporary dummy storage locations are eliminated by forming maximal strings of operands and operators from each sequence of sub-expressions. Each string specifies the computation in which (1) unary operators act on the entire substring immediately to their left and (2) binary operators combine this substring with the operand on their right.

Phase 37 — Arith V. Exponentiation operators are replaced by substrings involving log and anti-log functions. Implied mode changes are made explicit by inserting (or deleting) fix or float operators in the strings. The Table II addresses (Phase 29) appearing *within* IF statements (involving arithmetic expressions) are substituted into masks of object-time instructions. The filled-in mask replaces the addresses in the source program.

Phase 38 — Arith VI. The arithmetic strings are altered so that temporary storage areas are shared whenever possible. Machine addresses are determined for these areas and substituted for the dummy addresses in the strings. Previous arithmetic and IF statements are now stored, each with its identifier (Phase 32), in lower core immediately to the left of the list and format "statements."

Phase 39 — End file, Rewind, Backspace.

Phase 40 — Computed Go To.

Phase 41 — Go To.

Phase 42 — Stop/Pause.

Phase 43 — Sense Light.

Phase 44 — If (Hardware).

Phases 39 through 44 are essentially the same. In each phase, the residue of statements of the indicated type are substituted into masks of in-line, object-time instructions. The filled-in masks are stored with their identifiers (Phase 32) at the next available locations in lower core.

Phase 45 — Continue. No object-time instructions are generated for these statements. Only the identifiers (Phase 30) are stored in lower core.

Phase 46 — Do. DO statements are analyzed for nesting. Illegal nesting is noted. The residue of each DO statement is substituted into an object-time mask; but in general, the exit address is left blank. The partially filled-in mask and its identifier are stored in lower core. An unconditional branch is generated (uniquely) to follow (via Phase 49) the last statement within the range of the DO.

Note: At this point, the entire source program has been transformed into (essentially) object program procedure. For simplicity, we continue to write "statement" when we mean "procedure compiled from statement."

Phase 47 — Resort I. An area is made available for a table to assist in resorting the statements into their original order.

Phase 48 — Resort II. The resort table is filled with the current location of each statement.

Phase 49 — Resort III. The statements are resorted back into their original order with the identifiers eliminated. The Table II entries (sequence numbers of statements originally labeled with statement numbers) are replaced by the current machine addresses of those statements. Exit addresses are substituted into the procedure generated for DO statements (Phase 46). For each executable statement, the sequence number and the object-time starting address of the generated procedure are printed.

Phase 50 — Resort IV. The statements are shifted to the places they will occupy at object time. The Table II entries are bumped accordingly.

Phase 51 — Replace I. The entire object program procedure is scanned for indirect addresses (see note following Phase 31). Each indirect address is replaced by its direct address—now available in Table II.

Phase 52 — Function/Subroutine Loader. Relocatable function routines and subroutines (which comprise Phase 53) are selectively loaded. A table of starting addresses of these routines is created in free storage.

Phase 53 — Relocatable Package. This phase consists of the relocatable routines loaded by Phase 52.

Phase 54 — Format Loader. The object-time format routine, which is included in this phase, is loaded.

Phase 55 — Replace II. Those instructions in the generated object program that should branch to the relocatable routines are modified (via the table of Phase 52) to show the object-time addresses of these routines.

Phase 56 — Snapshot. A snapshot of the generated program is printed if initially requested and if no source program errors have been detected that would make program execution unrewarding.

Phases 57, 58, 59, 60 — Condensed Deck. When requested and if there are no input errors, these phases punch and list the object program as a self-loading condensed card deck.

Phase 61 — Geaux I. This phase prints the end-of-compilation message, initializes the sense lights, and prepares the branch into the object program coding.

Phase 62 — Geaux II. The arithmetic routine (Phase 63) is read into storage. Communication between this routine and the relocatable routines is established. The object program is executed on option.

Phase 63 — Arithmetic Package. This phase consists of the arithmetic routine loaded by Phase 62.

ACKNOWLEDGMENT

The author wishes to express appreciation to his colleagues, G. Mokotoff, S. Smillie, and D. Macklin for their generous assistance. Parts of this paper were prepared by the author for inclusion in Reference 1.

FOOTNOTES

1. A detailed description of the compiler is given in IBM 1401 FORTRAN *Specifications and Operating Procedure,* Systems Reference Library C24-1455-0, International Business Machines Corporation, (Revised Edition, June 1964).
2. Normally, minor modifications are made to a compiler throughout its life. For operational detail, the current program library documentation for Compiler Program 1401-FO-050 should always be consulted.
3. The decision to sort the source program was quite arbitrary. It was considered more efficient than translating the source program "in place." If translating a less concise language than FORTRAN or translating source programs for a large machine on a small machine, the entire source program would not fit in core. In this case, an in-place translation would be much more efficient.
4. This technique is described by K. Samuelson and F. L. Bauer in "Sequential formula translation," *Communications of the ACM* **3**, No. 2, 76–83 (1960).

Authors

Franklin H. Branin, Jr.

Data Systems Division, Development Laboratory, Poughkeepsie, New York.
Physical chemistry (Ph.D., Princeton University, 1950). Before joining IBM in 1957, for seven years at Radio Corporation of America, The Naval Ordnance Laboratory, Los Alamos Scientific Laboratory, and Shell Development Laboratory. Participated in programs involving computer methods for network analysis, matrix computation, and numerical solutions of differential equations. Currently an advisory programmer for problem-oriented programming.

Jack E. Bresenham

General Products Division, Development Laboratory, San Jose, California.
Electrical and industrial engineering (Ph.D., Stanford University, 1964). Joined IBM in 1960. IBM resident graduate scholar 1962 to 1964. Presently a senior programmer at the computation center in San Jose.

Robert M. Carlitz

Currently physics student at Duke University. Since 1962 has been with IBM Development Laboratory, Poughkeepsie, New York, during academic vacations.

Tien Chi Chen

Data Systems Division, Development Laboratory, Poughkeepsie, New York.
Physics (Ph.D., Duke University, 1957). With IBM since 1956 in numerical analysis, molecular quantum theory, optimal programming and machine design areas. Currently directing a problem-oriented programming group.

M. Gerson Ginzberg

Data Processing Division, Special Systems, Ossining, New York.
Mechanical and electrical engineering, applied mechanics (M.S., University of Cincinnati, 1958). Engaged in stress analysis and propeller aerodynamics prior to joining IBM in 1959. Extensive work in programming design for airline reservation systems. Currently senior industry analyst in market requirements.

Leonard H. Haines

Corporate Headquarters, Systems Research and Development, Cambridge, Massachusetts.
Mathematics (M.A., University of California, 1960). Joined IBM in 1957. IBM resident graduate scholar 1963–64. Designed the 1401 FORTRAN compiler. Currently engaged in programming research.

Loren V. Hall

Data Systems Division, Development Laboratory, Poughkeepsie, New York.
Mathematics (M.S., New York University, 1963). Problem-oriented programming since joining IBM in 1963. Engaged in programming of matrix, number theoretic, and statistical problems.