The general considerations underlying the design of the system's COBOL *compiler are discussed.*

A brief outline of the operation and structure of the compiler is included.

Finally, attention is focused on certain techniques which are incorporated within the compiler.

Design of an integrated programming and operating system

Part V: The system's COBOL compiler

by R. T. Dorrance

This paper describes the general considerations governing the design of the system's 7090/94 COBOL compiler (IBCBC),[1,2] gives a brief outline of the structure of the compiler and explains some of the techniques that were incorporated.

IBCBC operates under control of the IBJOB monitor which is in turn under the control of the system's basic monitor, IBSYS. Input to the compiler is a COBOL source program. Output from the compiler consists of an augmented replica of the source program, a list of messages describing errors detected during compilation, and an intermediate tape of generated symbolic instructions. IBMAP, a separate system component also under control of IBJOB, assembles the generated symbolic instructions into a form acceptable to IBLDR. Compiler and object program input/output functions are performed by IOCS.

general design considerations

In order to achieve simplicity in structure and speed in operation, the compiler was designed to retain all of the source program's symbolic names and their associated attributes in storage. Thus, random references to the attributes can be made, and the necessity of bringing such information in and out of memory is avoided.

IBCBC is broken into eleven phases whereas its predecessor, the 7090 COMMERCIAL TRANSLATOR compiler, consisted of but four. As soon as a function is completed, the memory space utilized by the function is reassigned. This approach (used with discretion

to avoid the necessity of extra passes over the information streams) has had a marked affect on the compiler's speed by providing the memory space necessary to apply certain techniques described later in the paper.

Excessive use of input/output devices can lead to slow compilation. A common attack on the problem involves retention of information in storage in the hope that overflow spilling can be avoided. IBCBC utilizes conditional spilling for most of its major information streams with the result that time benefits are realized for all but relatively large source programs.

Some of the instruction generators are able to represent given functions by in-line instructions or by subroutine linkages. The former improves execution speed while the latter conserves space. The compiler can make the choice of generation mode (subject to overriding control by the user) according to the nature of the function. For example, space is conserved for on-line printing conversions while speed savings are obtained for like conversions in arithmetic expressions.

Guessing is attempted for minor source program errors in format but is avoided for errors in syntax. Many compiler instructions and much user misunderstanding are saved by the simplicity of the standard response of the compiler: deletion of the offending clause, issuance of an error message, and resumption of scanning at the next recognizable clause.

the translation process

As mentioned above, the compiler consists of eleven program phases. The first of these remains in storage throughout the compilation process. The other ten phases are loaded into storage successively with each new phase replacing all or most of its predecessor. Loading of the eleven phases occurs once for each compilation of a source program.

Phase 1 contains a supervisory routine, a communication region, and a set of general purpose subroutines which are used by more than one phase of the compiler. The supervisory routine controls the phase to phase and compiler to monitor transitions. The communication region permits preservation of information over more than one phase. The general purpose subroutines perform many functions; e.g., control of the numerous compiler tables, isolation and classification of source program data, conversion of values, and various dictionary operations.

Phase 2 generates object program initialization instructions and scans the source program's *Identification Division.*[3] Like all other phases, this phase may generate requests for the issuance of specific diagnostic messages.

Phase 3 scans the source program's *Environment Division* and makes dictionary entries for object program input/output files and specially defined hardware device names.

Phase 4 scans the source program's *Data Division.* Principal functions achieved are the definition of data item names and formats, the preservation of more file attributes, and the preparation of text for Phase 5.

Phase 5 uses dictionary and input text information to calculate the lengths and relative locations of data items, to generate data storage reservation, and to generate length calculation subroutines for variable length arrays.

Phase 6 scans the source program's *Procedure Division*. Two of its principal functions are the definition of procedure names and the preparation of a condensed internal text representing the content of properly written source program statements.

Phase 7 expands, converts, and permutes the text of the preceding phases in order to render it suitable for input to the instruction generators in Phase 10.

Phase 8 summarizes the file information and produces IOCS control cards.

Phase 9 uses dictionary and table information to generate address calculation subroutines for subscripted expressions. Utilized dictionary and table information is destroyed at the conclusion of this phase to make room for the bulk of the subsequent phase.

Phase 10 contains most of the instruction generators. Input to the generators is the statement text prepared by Phase 7. Generator output is an encoded text converted immediately by a general purpose subroutine to the IBMAP symbolic instruction form.

Phase 11 processes accumulated message requests and produces replete diagnostic messages cross-referenced to source program card numbers. Upon conclusion of Phase 11, control is returned to the IBJOB monitor for transition to IBMAP.

The remainder of the paper is devoted to a description of certain techniques that have been incorporated within the compiler.

scanning method

Although thorough syntactical scanning of source programs is potentially an expensive method in both space and time, the following approach proved to be both practical and rapid.

The IBCBC scanning method may be considered as a sequence of questions posed by the compiler and computed responses—wherein the action taken and the choice of the next question depend upon the response. For example, in scanning an identified clause, proper syntax can be established by use of an appropriate series of such questions.

IBCBC scanning is governed by a set of syntax vectors, each of which occupies three machine words. The first word contains a specific question (relative to a particular COBOL word) or a categorical question (relative to class membership, such as whether an entity is a literal, arithmetic operator, or data-name). The second word contains both a location of an executable stream of instructions and a location of the subsequent syntax vector. The second word is chosen if the response is "true." Similarly, the third word has two location references and is used if the response is "false."

A *group interpreter* routine controls matching of questions with responses and determines the resultant routing. Scanning progresses alternatively in or out of an interpretive mode since each execution of a stream of action instructions is concluded by a

return to the group interpreter for processing of the next syntax vector. Significantly, the group interpreter has two entry points. The first is used to request classification of the next source program information group before another question is posed. The second dictates retention of the current information group for further interrogation.

The group interpreter is supported by four other routines. The *unit level scan* obtains and classifies the next source program unit (such as an alphanumeric name). The *group level scan* calls upon the unit level scan until able to classify the next information group (such as a data name with associated qualifying names). The *subscript scan* assists the group level scan in the examination of subscript expressions. The *dictionary routine* enters name definitions in the *symbolic dictionary*, recognizes references to defined names, and prepares the internal text equivalent of source program information groups.

table handling

A technique aimed at preventing capacity overflow of any of the compiler's numerous tables is accomplished by means of the Coalesced Indirect Table Reference Unification Scheme (CITRUS).

CITRUS permits dynamic definition of the boundaries of a general table region. Individual tables are assigned to the general table region and "float" about in the region in compliance with the requirements for space.

Each table is governed by a table control block which contains the current location of the table origin and the relative position of the most recent table entry. With the use of CITRUS, a table may be opened or closed at any time. The reserved area for an individual table may be reduced to encompass only the actual table data. Automatic reduction of the reserved area and/or relocation of tables may occur to make room for another table growing beyond its reserved boundary. It is important to note that such growth is permitted.

Statistics gathered on CITRUS performance indicate that the time spent in movement of table data is normally negligible.

dictionary methods

The compiler constructs two dictionaries in storage. The *symbolic dictionary* contains defined source program names, and the *attribute dictionary* contains properties associated with the names.

Name definitions appear in the symbolic dictionary in the same order as in the source program. This fact simplifies evaluation of qualified name references since a qualification hierarchy is determined by source program order. Association of a name reference with a previously defined name may seem to imply a linear search of the symbolic dictionary. Actually, such association is achieved by applying a transformation to the name to obtain a relative position in an intermediate table. Entries in the intermediate table point to appropriate symbolic dictionary entries. Difficulties arising from the fact that identical values may result from the transformation of different names are resolved by provision for association of a set of pointers,

with positive identification achieved by name comparison.

Each symbolic or attribute dictionary entry occupies one machine word. Overlong entries are stored in an overflow table, and appropriate pointers become their dictionary representation. The use of one word per entry permits very rapid processing.

locating and remembering

Two methods of referring to object program logical records are used. One, called the *locate mode*, permits references to logical records which are variably located within input/output buffer areas. This mode tends to conserve storage and is fast when the frequency of reference is low. The other, called the *transmit mode*, requires that successive logical records be moved between a variably located input/output area and a fixed location work area. This mode is attractive when the frequency of references is high or, in case of an output file, when a significant amount of constant information is to be included with each logical record.

The compiler normally generates instructions for the locate mode but allows the transmit mode, using the COBOL statements READ . . . . . INTO and WRITE . . . . . FROM. The latter mode poses no special problems since each data field appears in a fixed, predetermined location. In the locate mode, however, the relative position of a field within a logical record is known, but the starting address of the logical record must be computed. The resolution of this problem involves assignment of an erasable storage word, called a *base locator*, for each input/output file. The base locator is set each time a logical record is read or written.

Stated simply, an index register is loaded from a base locator for each reference to a field whose location is relative to the particular base address. In reality, however, the loading is significantly reduced by the compiler's ability to remember a loaded value over a limited sequence of generated instructions. Further reduction in loading requirements results from the fact that loading of different bases is rotated over the set of available index registers.

error and warning messages

The compiler diagnostic messages are designed to be comprehensive and clear.

The messages are comprehensive in the sense that they are issued for any detected error or questionable practice. The intent is to avoid the need for many compilations by determining all evident difficulties. Since this philosophy can lead to voluminous message output, an effort was made to prevent the generation of messages containing redundant information.

Clarity results from the use of a standard message form which consists of a severity code (warning, error, or disaster), a source program card number, a specific statement of the difficulty, and indication of the compiler action taken. Parametric substitution is used to tailor the message to the particular fault. For example, a message referring to a variable's improper format may include the variable's name, its actual format, and the format assumed by the compiler. Clarity is also obtained by minimal use of abbreviations and ambiguous terminology, and by the use of an initial heading to define the message format.

The basic text for the messages is concentrated in the final compiler phase rather than being scattered throughout the various phases. As a result, all of the messages are processed and printed last. The common alternative technique of printing each message immediately following the representation of the associated statement was rejected for several reasons. First, such a scheme requires distribution of full message text through all phases of a compiler. Second, the rejected scheme makes necessary the time-consuming merging of error messages with the output representation of the source program (in order that the messages for errors detected on passes other than the first appear in proper order). On the other hand, the scheme adopted requires the presence of only the skeletal forms necessary for issuing message requests. Furthermore, the implemented scheme provides storage for elaborate message construction from the message skeletons and substitution parameters, permits the use of the same message by different phases, and centralizes the message function for easy maintenance and review.

ACKNOWLEDGMENT

Credit for the concepts presented in this article belong to the members of both the 7090/94 COBOL and the 7090/94 COMMERCIAL TRANSLATOR projects.

CITED REFERENCES AND FOOTNOTES

1. For a description of the COBOL language, see IBM 7090/7094 Programming Systems: IBJOB Processor: Part 5: COBOL Compiler (IBCBC), Systems Reference Library J28-6260, International Business Machines Corporation, 1962.
2. For a description of the organization of IBCBC and its relationship to other IBJOB components, see IBM 7090/7094 Programming Systems: IBJOB Processor, Systems Reference Library C28-6275, International Business Machines Corporation, 1963.
3. "Identification Division", "Environment Division", "Data Division" and "Procedure Division" are the formal names of the parts of a COBOL source program. The reference in Footnote 1 discusses the makeup and purpose of each part.