BIT 6 (1966), 1–23

THE COBOL COMPILER FOR THE SIEMENS 3003

PER BRINCH HANSEN and ROGER HOUSE

Abstract.

This paper describes the design of a fast Cobol Compiler with extensive error detection. It is implemented as a 10 pass compiler on the Siemens 3003 computer with a core store of 8000 words, using one systems tape and two working tapes. The structure of the object program produced by the compiler is discussed with respect to storage allocation, administration of files, and addressing of data items. In the description of the compiler particular emphasis is placed on the error detection phase, where the source program is analysed with respect to syntax, data descriptions and operand types.

Contents.

1. Introduction.

The Siemens Cobol Compiler was developed by the compiler group at Regnecentralen, Copenhagen, headed by Peter Naur and Jørn Jensen. The compiler processes full elective COBOL 61, except for a few minor

omissions mainly dictated by the characteristics of the machine and by the existing conventions for data formats on tape. It is implemented as a 10 pass compiler on the Siemens 3003 computer with a core store of 8000 words, using one systems tape and two working tapes. In this configuration it processes a Cobol source deck at the rate of 250 cards per minute, generating final machine code.

The contract for the development of a Cobol compiler was given to Regnecentralen as a result of the demonstration of the highly successful GIER ALGOL compiler at the IFIP Conference in Munich in August, 1962. After an initial period spent in getting acquainted with the language, the design and programming of the system was initiated in March, 1963. Effective testing of individual passes started in May, 1964, and the final system, amounting to 39000 instructions, was delivered in July, 1965, after a total effort of 15 man-years.

The major problem of implementation turned out to be the numerous definition problems created by the vagueness of the official Cobol report (ref. 1). The basic translation scheme was largely taken over from the GIER ALGOL compiler as described by Peter Naur (ref. 2). The present paper describes in detail the design of the Siemens Cobol compiler. The novel features, as compared to the GIER ALGOL compiler, are: the analysis of the complex data structure, the handling of the Copy features, and the administration of data files at run time.

2. The Systems Configuration.

The Siemens 3003 is a large-scale, tape-oriented computer with a minimum core store of 8000 words. Each word consists of 24 bits interpreted as a one-address instruction, a binary integer, or 4 alphanumeric characters. Words and characters may be indirectly addressed. Index registers are not available. Parallel binary operations take 30 to 70 microseconds, serial character operations from 200 to 500 microseconds.

The minimum configuration of peripheral units used by the translator consists of a typewriter, a line printer (750 lines per minute), three magnetic tape units (46000 characters per second) and a card reader (650 cards per minute). The typewriter is used for messages to the operator and instructions to the monitor. The source program is input from the card reader and listed on the line printer together with possible error messages to the programmer. One tape contains the translator, segmented into 10 passes, and two working tapes are used to store the partially translated program. The final object program is normally generated on one of the working tapes, but it may also be punched on cards or paper tape.

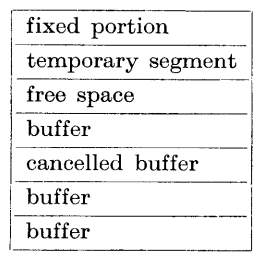
3. The Object Program.

This section describes the structure of the object program produced by the compiler. The main problems are the storage allocation, the administration of files, and the addressing of items.

Program structure. The translated program is stored on magnetic tape, punched cards, or paper tape, as relocatable binary code. It is divided into segments in accordance with the specification in the Cobol Procedure Division. The first segment is the so-called fixed portion, which is held permanently in the core store after the initial loading. The following segments are only transferred to the core store when the execution of the program calls for them.

The fixed portion consists of tables describing data files, storage locations for items in the Working Storage and Constant Sections, and procedure code. The procedure code is composed of fixed subroutines and some generated sequences of instructions. The fixed routines, called the Running System, administer the peripheral units and perform complex data operations (subscription, multiplication, editing, etc.). These routines may be regarded as part of the systems library, i.e. they are only included as required by the particular source program. For the simple operations (perform, move, add) the compiler generates short sequences of instructions referring directly to the operands.

Storage allocation. The following figure shows the utilization of the core store at run time.



The fixed portion resides in the lower end of core. The temporary segments and the buffer areas compete for the rest of the core store. To make the best use of the available store it must be possible to cancel segments and buffers. For segments this is achieved by letting each temporary segment transferred to core overwrite the previous segment. New buffer areas are created at the other end of the core whenever a data file is opened. The buffers are chained together by a list of file table addresses defining the order in which the files have been opened. When a file is closed its buffer areas will be cancelled. This is done simply by deleting the file table address from the chain list. A request for a new

buffer is made in the following way: a look-up in the relevant file table defines the area needed. This is compared with the size of the free space between the program segments and the nearest buffer. If the space is adequate, the new buffer is immediately created on top of the last one. Otherwise the entire core store must be examined for gaps (previously cancelled areas) which can be closed by a compression of the active buffers. This compression is performed on the basis of the chain list and the buffer pointers kept in the file tables. The compression routine is also used to make room for new temporary segments.

File administration. The system processes four kinds of data files: magnetic tape, card input, card output, and line printer. At run time all requests for block transfers are handled by tables defining the type and the current status of each file. The first part of these file tables has a format independent of the type of peripheral unit. The table parameters define the current input-output status, the size and location of buffer areas, the base address of the current data record, as well as the addresses of subroutines that interpret the verbs Open, Close, Read, and Write for the given unit type. Apart from these general parameters, each file table holds some information special for the given type of file. For tape files these are the values of label record items and the addresses of Declarative Use procedures.

Addressing of variables. Working Storage items and Constant Section items are stored in the fixed portion. Literals from the Procedure Division are stored inside the segments where they occur. These items can be addressed absolutely, because all segments are transferred to the same place in core. Items from data files are processed directly in the buffer areas, thus avoiding a movement of every record inside a batch to a fixed working area. These items are addressed relative to the base of the current record. The record base addresses kept in the file tables are updated every time records are read or written. The drawback of this method is that item addresses must be computed at run time. The amount of address calculation is limited, however, by the ability of the compiler to remember an item address over a sequence of generated instructions (until a procedure name or an input-output statement appears).

4. Multipass Translation.

The philosophy behind the multipass translation scheme has been discussed by Peter Naur (ref. 2). The limited size of the core store forces upon us the use of magnetic tapes as auxiliary stores for the translator and the program being translated. The fundamental design problem is

to minimize the traffic between the tapes and the core store. The solution is to divide the translator into a number of segments on the systems tape, each of which is small enough to be held entirely in the core store. These translator segments, called "passes", are taken into core one by one. Each pass performs a single, sequential scan of the Cobol program and produces as output an intermediate program version, that is stored on one of the working tapes, forming the input to the next pass. The intermediate output thus goes back and forth between the two working tapes until an object program has been produced. By using a strictly sequential processing scheme, random references to the systems tape and the working tapes are avoided.

The success of the multipass system becomes evident when it is compared to a onepass system, where the entire translator and the source program reside in core after the initial loading. The basic compilation time for a program is composed of the time taken to load the compiler and the subsequent processing time. The loading time depends mainly on the total size of the compiler and is thus independent of the number of passes. The same is true of the processing time, depending only on the overall logic of the compiler. In a multipass system the compilation time is increased by the time taken to transfer the intermediate program versions between the working tapes. This increase is proportional to the number of passes.

For small programs the loading time is dominant, so that the multipass system is just as fast as a onepass system. For bigger programs the compilation time is increased by a factor f. By ignoring the loading time we get the following conservative estimate: f is less than (1 + passes × transfer time / process time). The average processing time per Cobol word is about 20 msec. With 10 passes and a transfer time of 0.04 msec per word per pass we find $f = 1.02$. The extra time cost of the multipass system as compared to a onepass system is thus negligible. The low value of the transfer time results from using a simultaneous, two-buffer system in the communication with the working tapes. With a one-buffer system, f would increase to 1.11.

5. The Intermediate Languages.

The first translator pass transforms the source program, word by word, to an internal representation of "bytes" and "strings". A byte is a binary integer (equal to a computer word) used to represent a Cobol keyword or a programmer-introduced noun. Keywords are represented by positive integers of fixed value, called directing bytes, because they direct

the individual passes to specific work actions. Nouns are represented by negative values preceded by directing bytes to indicate whether they are data nouns or procedure nouns.

A character string is a sequence of 6-bit units (terminated by a special end mark) used to represent literals, level numbers in data descriptions, and sequence numbers of source cards. Strings are headed by directing bytes indicating their type. Literals are also supplied with bytes defining their size, point location, and sign.

The bytes are normally used by the individual passes as entries in switch tables or data tables. This results in a very clear and fast logic. The uniform structure of the byte string greatly simplifies the packing and unpacking of input-output blocks to be performed by the general pass administration.

6. The General Pass Administration.

The complete translation process is monitored by a translator segment called pass 0, which remains in core after the initial loading. Pass 0 consists of routines that are needed by all the passes, mainly to handle peripheral units. It administers the transfer of control from one translator pass to the next, as well as the intermediate input-output of bytes and strings. In addition, pass 0 controls the operation of the typewriter and the line printer. Finally it contains the Help system used for the debugging of the translator.

The intermediate program text is stored on the working tapes, and the administration of block transfers is performed by pass 0. Inside a pass the next input byte is accessed simply by incrementing a buffer pointer by 1. Each block is terminated by a special end-byte which transfers control to a routine that makes the next input block available. The output of a byte is done by calling an output routine with the relevant byte as a parameter.

The transfer to a new pass consists of rewinding the working tapes to the beginning of the intermediate strings and loading the next segment from the systems tape.

The final object program produced by a translation is placed at the end of one of the working tapes, overwriting the previous end label; it is terminated by a new end label. During the translation the intermediate byte string is output after the end label. In this way it is possible to perform batch compilations without destroying previously compiled object programs.

Handling of peripheral units. The simultaneous operation of the central processor and the peripheral units is utilized extensively during trans-

lation. The two working tapes are mounted on different data channels, allowing the input and output of each pass to run simultaneously. This results in an overall gain of about 10 percent in translation speed. In pass 1 this is augmented by simultaneous reading and listing of the source program.

In order to obtain a tolerable performance of the magnetic tapes, information was included in each block about its size and block number. The tape routines check this information after each block transfer. If a whole block or part of it is skipped because of hardware trouble, a special procedure is initiated to recover the last position on the tape and repeat the transfer. This recovery procedure permits any movement of the tape before the translator is started for a new attempt, including rewinding and moving the tape reel to another unit on the same channel. Our experience shows that these procedures are vital to the effective operation of the tapes.

7. The Translation Process.

The translation consists of an analysis phase, aiming at a complete error detection in a single compilation, and a generation phase, where the final machine code is generated.

The analysis phase may be summarized as follows: The syntactical analysis (passes 1–2) checks the formal structure of the program. The data structure analysis (passes 3–4) checks the existence and uniqueness of items referred to at run time. The data description analysis (pass 5) checks the consistency of the data descriptions. The operand type analysis (pass 6) checks the compatibility of the description of an item and its uses at run time.

The generation phase consists of the selection of appropriate instruction sequences (pass 7), the assignment of final addresses (passes 8–9), and the actual generation of the final machine code (pass 10).

8. The Analysis of Syntax.

The primary aim of this analysis is to provide later passes with a program string of correct syntax, even when the source program contains illegal constructions.

Analysis of single words. Pass 1 analyzes the source program at the character level in order to delimit single words and literals and reduce them from the character form to an internal representation of bytes and strings. Apart from some analysis of context which is necessary in order to recognize the use of integers as level numbers and procedure names, each word is processed independently of its surroundings.

The character analysis is controlled by means of a transition matrix according to a method of Peter Naur (ref. 2 and 3). In each cycle a single word is scanned, character by character, up to the next blank character. The logic is controlled by a single integer-valued state. The current input symbol (a character) and the state value are used to look up a transition matrix. The selected element of the matrix defines a new state value and the address of a work action to be performed.

The character analysis acts as an input routine between the source program and a word matching routine which converts reserved keywords and programmer-introduced nouns to unique byte values. The word matching is implemented as a variant of the Williams Name Reducer (ref. 4) using a refined search technique through linked lists. When a word has been delimited, an index is calculated from the internal representation of its first and last characters. This index is used to reference a switch of 126 entries. Each entry points to a list of Cobol words (some lists may be empty). A search is now performed in the selected list, comparing its elements to the input word. The matching element holds the address of a work action and an output byte value. For non-matched input words, a new byte value is created, and the word itself is placed at the end of the list. The principle of switching on a function of the first and the last characters ensures an even distribution of all words among the 126 lists. In this way the average number of comparisons per word is minimized. Switching on the first character alone would make some lists much longer than the rest (for example, reserved words beginning with *A* and *S*).

Analysis of word structures. From the point of view of pass 2 the source program is composed of "clauses" with a fixed structure of keywords and operands. After removal of optional keywords (performed by pass 1), these Cobol formats are all headed by unique keywords, the format-firsts, which serve to identify them. Examples of format-firsts are FILE-CONTROL, PICTURE, and WRITE. Pass 1 recognizes these format-firsts and precedes them by a unique byte, that is used by pass 2 to delimit the structures. The syntax is checked by pass 2 at three levels of context:

Divisions. The sequence of Cobol formats is checked by means of a transition matrix. This table defines the global structure of divisions, sections, paragraphs, and sentences.

Clauses. The interior keyword structure of clauses and verbs is checked by comparing the input string with the contents of fixed threaded lists.

Expressions. The structure of conditions and formulas and of qualified and subscripted names is analyzed by means of another transition matrix.

The description of Cobol clauses by threaded lists may be illustrated by the verb, WRITE, which has the following format in the source program:

WRITE ⟨dataname⟩ [*FROM* ⟨dataname⟩]

$$\left[\begin{Bmatrix}AFTER\\BEFORE\end{Bmatrix}\ \text{ADVANCING}\ \begin{Bmatrix}\langle\text{dataname}\rangle\\\langle\text{integer}\rangle\end{Bmatrix}\ \text{LINES}\right]$$

Pass 1 removes the optional words, ADVANCING and LINES, inserts a directing byte, FORMAT-FIRST, in front of the keyword WRITE, and precedes the datanouns and integers by directing bytes, DATANOUN and NUMERIC-LITERAL. The legal input to pass 2 may therefore be described by the following table:

entry:	link:	identifier:	alternative:
1	2	WRITE	no
2	3	DATANOUN	no
3	7	FROM	yes
4	8	AFTER	yes
5	8	BEFORE	yes
6	–	FORMATFIRST	no
7	4	DATANOUN	no
8	6	DATANOUN	yes
9	6	NUMLIT	no

This format table has an entry for each byte that may appear inside the clause. An entry holds 5 parameters: (1) An identifier defining the value of a legal input byte. (2) A link to be used as the next entry if the present input byte matches the identifier. (3) A Boolean indicating whether the format allows an alternative byte value at the given point. (Alternative identifiers are always stored in consecutive entries). (4) The address of a work action to be performed on a matching input byte. (5) The address of an error action to be performed in a non-matched entry with no alternatives.

For clarity only the first three parameters are shown in the above example.

In the output from pass 2, redundant clauses have been deleted, and ambiguous keywords are replaced by unique bytes. Clauses with illegal syntax are erased completely.

9. The Analysis of Data Structures.

The main task of this analysis is to ensure that all names in the source program refer to one, and only one, defined data item or program point.

This analysis is divided into two passes. Pass 3 collects the data and procedure names where they are defined in the program and builds a table describing their tree structure. This table is provided with internal links enabling pass 4 to replace each qualified reference to an item by a single, unique byte representation.

The following example shows to the left a data structure described inside a Cobol Data Division and to the right the corresponding name table built by pass 3:

FD	*A*			entry:	item	tree link	program link	name link
	01	*B*		1	*A*	0	2	0
		02	*C*	2	*B*	1	3	0
		02	*D*	3	C_1	2	4	7
	01	*E*		4	*D*	2	5	0
		02	*C*	5	*E*	1	7	0
		02	*F*	6	*F*	5	–	0
				7	C_2	5	6	0

The byte values given to single nouns (*A*, *B*, *C*, etc.) in pass 1 are consecutive integers (1, 2, 3, etc.). They act as direct entries in the name table. In the above example the noun "*C*" occurs at two points in the name tree. In the input to pass 3 both occurrences have the same byte value (3). Pass 3 associates the direct entry (3) with the first occurrence of "*C*", and generates a separate entry (7) for its second occurrence. This duplicate entry is placed at the end of the table and connected to the first entry by an internal link (the name link). The name table has an entry for each point in the name tree. The internal links are:

The tree link. This points to the most recent item with a lower level number, i.e. to the parent of the present item.

The name link. This connects multiple occurrences of the same noun.

The program link. This defines the order in which all nouns are introduced in the source program (including nouns introduced by the Copy clause). This link is used to process the clauses Copy, Redefined, and Renames.

Pass 3 uses a stack (or push-down list) to establish the tree link in the name table. Each element in the stack holds an entry in the name table together with the corresponding level number. For each entry processed an unstacking is performed until the level number in the top of the stack is less than the one in the present entry. The tree link for the present entry will then be in the top of the stack.

When the name table, defining all items, is complete, pass 4 will scan

the program to analyze all references to these items. Take the following reference: *C* IN *B*, represented by the byte values (3,2). Pass 4 must establish that this qualification sequence exists once, and only once, in the name table. After having looked up the table entry for the first noun (*C*), pass 4 will follow the tree link searching for the following noun (*B*). The next step is to use the name link of the first noun (*C*) to find the entry for its second occurrence (C_2) and repeat the search along its tree link. By performing this process for all duplications of the original entry the uniqueness of the reference is determined.

The output of a qualified data name consists of two bytes: one defining its unique entry in the name table, the other one defining the entry for the corresponding file name. Referring to the previous example, the qualified names, *C* IN *B* and *C* IN *E*, will appear as byte values (3,2) and (3,5) in the input to pass 4. In the output they are transformed to (3,1) and (7,1), respectively. This representation of data names corresponds to the addressing at run time, where items are represented by the address of a file table and a relative address inside the current record.

The output of a procedure name is a unique symbolic address and a segment number. The look-up of procedure names differs somewhat from that of data names, because of the rule that paragraph names need only be unique inside the current section.

The above method used to analyze data names, though similar to the one reported by M. E. Conway (ref. 5), was developed independently by our group.

10. The Analysis of Data Descriptions.

The purpose of this phase is to collect descriptions of all files and data items in a form suitable for the final code generation. This information, collected by pass 5, is stored in the name table created by pass 3. An important part of this analysis is to control the consistency of the data descriptions. This is done in two steps.

Pass 5A scans the Environment and Data Divisions of the source program. During this scan all explicitly stated information about items is collected, and all copy processes are completed. In this phase each item is processed as a unit, independent of all other items. Consider the following data description:

02 A CLASS AN POINT LEFT 1, SIZE 5, PICTURE 9.9

Here pass 5A will report the local conflict between the explicit class clause (alphanumeric) and the point location clause (implying a numeric

item). Likewise the conflict between the size clause and the size derived from the picture will be detected.

The method used for checking data descriptions of single items has been described by Paul Lindgreen (ref. 6). The current class of legal description clauses is defined by a single machine word, where each bit is associated with a specific clause, such as "alphanumeric", "size", etc. This Boolean vector is initialized to "all clauses allowed". The occurrence of a clause will cause a test of a single bit followed by an updating of the vector. The updating is performed by logical addition of a constant vector determined by the clause. As an example: the clause "alphanumeric" will forbid subsequent class clauses as well as the point location clause.

Pass 5B performs a rescan of the complete description table of data items in their order of introduction (as defined by the program link in the table). The purpose of this scan is to verify that descriptions of group items are compatible at all levels. As an example:

```
01 A NUMERIC SIZE 5 .
02 B PICTURE $999 .
02 C PICTURE  999 .
```

In this case there is a conflict between the group size (5) and the sum of elementary sizes (7). There is also a conflict between the group class (numeric) and one of the subordinate classes (*B* is alphanumeric).

Apart from checking that an explicit group description is consistent, pass 5 B will also ensure that it is complete, i.e. that size is specified at all elementary levels. If size and class are not specified at a group level, this information will be derived from the elementary levels.

Pass 5 B uses a stack to check the data descriptions of group items. A sequence of subordinate items (with ascending or identical level numbers) will be placed in the stack, one after another, until an item with a lower level number signals the end of the last group item. At this point an unstacking is performed. During this process an accumulated size and class is formed for items on the same level. When a level ends, the accumulated information is tested against the size and class specified for the group item in the top of the stack. The unstacking, accumulation and checking continues until the input item and the stack item have identical level numbers.

After pass 5, the entries in the name table will hold the following information:

Type information. This is an integer used for the type checking in pass 6. Primarily it indicates whether the operand is a file, a record, a

group, or an elementary item, and whether it is alphanumeric or numeric.

Address information. Pass 5 generates symbolic addresses for all run time file tables, and calculates the relative addresses of all data items inside a record.

Structure information. Data items are described by their size and a possible point location. Information about files is kept in a separate table, the extra table, which defines the type of peripheral unit, the recording mode, and the area needed for buffers.

11. The Analysis of Operand Types.

Pass 6 completes the formal analysis of the source program by checking that the uses of operands in the Procedure Division are compatible with their descriptions in the name table. As a final preparation for the generation of code, all names in the program are replaced by complete data descriptions, and all procedure statements are converted to Reverse Polish form.

Verbs. In a statement like the following: ADD A, B GIVING C, the byte values for A, B, and C are used to look up the name table to ensure that A and B are described as elementary numeric items, whereas C must be an elementary numeric or edited item not belonging to the Constant Section. The conversion to Polish form (A, B, $+$, C, $=$) is straightforward, because there is no parenthesis structure.

Expressions. Formulas and conditions are converted to Reverse Polish form by the well-known algorithm described by Dijkstra (ref. 7). This method is based on the use of an operator stack and a priority system for the operators. Simultaneously with this conversion, the checking of operands is performed by a pseudo-evaluation of the expression. This process imitates the run time evaluation, working with the descriptions of the operands instead of with values. The evaluation, which is based on input in the Polish form, assumes the existence of an operand type stack. Each time an operand occurs, a description of its type is stacked. When a binary operator is met, it will be checked for compatibility with the two operand types in the top of the stack. Following this the two operand types are replaced in the stack by a single resulting type. As an example consider the following condition composed of an alphanumeric and numeric relation:

IF A = 'B' AND $C+5 = D$ THEN ...

The following figure shows to the left the Polish string developed for this expression. The corresponding development of the operand type stack during the pseudoevaluation is shown to the right:

Polish string:	Type stack
A	alphanum
'*B*'	alphanum, alphanum
=	Boolean
C	Boolean, numeric
5	Boolean, numeric, integer
+	Boolean, numeric
D	Boolean, numeric, numeric
=	Boolean, Boolean
AND	Boolean
THEN	-

The evaluation proceeds as follows: The first operator (=) tests whether the two (alphanumeric) types in the top of the stack are compatible. The resulting type of this alphanumeric relation is "Boolean". Following this a numeric variable (*C*) is added to an integer literal (5). The resulting sum is of type "numeric". Evaluation of the complete numeric relation ($C+5=D$) gives a Boolean result. Finally the combination of two Booleans, by AND, is in itself a Boolean. The stack ends up by holding the type of the complete expression. The operator, THEN, checks that this type is Boolean.

In the checking of operand types, each operand is described by a bit pattern in one machine word. Each bit in this type vector is associated with a given class of operators (e.g. arithmetic, numeric assign, relation, etc.). With this representation the check whether an operand is correct involves only the test of a single bit. This method of type checking is due to Peter Naur (ref. 8).

Formulas. Pass 6 supplies all arithmetic expressions with the following collective information: (1) An indication of whether ROUNDING and SIZE ERROR tests are specified. (2) The maximum size and point location for the entire expression.

Conditions. In abbreviated relations like the following: IF A GREATER *B* AND LESS *C* ..., the missing terms are copied from the nearest complete relation. In order to generate optimal code for compound conditions, all logical operators (AND, OR, NOT) are replaced by conditional jumps which ensure that the minimum number of relations are evaluated at run time. This may be illustrated by the above example, where it is apparent that the relation following AND need not be evaluated if the preceding relation turns out to be false. In this case the compiler will replace the AND operator by a special byte, FALSE-GOTO instructing pass 7 to generate a conditional jump instruction. As de-

scribed by Per Brinch Hansen (ref. 9), this transformation can be completely integrated with the algorithms for the type checking and the conversion to Polish form.

12. The Generation of Machine Code.

The final object program is produced in 4 steps. Pass 7 simulates the run to select appropriate sequences of machine instructions. Pass 8 scans the program to define all symbolic addresses created by previous passes. Pass 9 distributes the final addresses to the places in the program where they are used. Pass 10 produces the final program, as directed by the output from passes 7–9.

Code selection. The final program contains two kinds of code: (1) calls of routines in the running system performing complex operations (e.g. read a record), and (2) code consisting of operations which are built into the machine and which address the operands directly (e.g. addition, simple move).

Pass 7 specifies the code to be generated by a simulation of the run, similar in principle to a method described by Jørn Jensen (ref. 10). This method uses an operand stack to operate on the descriptions of the operands. The stack keeps track of where the operands exist at run time, and which working areas are used for intermediate results. The major aim is to minimize the use of intermediate working areas (by changing the order of reference to operands), and to generate simple code for simple operations (e.g. an addition involving an assignment to one of the operands is performed directly in the receiving area).

Pass 7 delivers 4 kinds of output: (1) Address bytes instructing pass 8 to define internal program points. (2) Operand bytes consisting of symbolic base addresses (of file tables or the working storage area) and relative addresses. These are replaced by absolute addresses in pass 9, and in pass 10 they are inserted as address parts of the generated instructions. (3) Instruction bytes which direct pass 10 to generate specific sequences of instructions. (4) A bit vector defining the routines to be included from the running system.

Address definition. In earlier passes symbolic addresses are created whenever a reference is needed from one point in the program to another. Examples are procedure addresses created in pass 4, file table addresses created in pass 5, and conditional jumps from pass 6. These symbolic addresses are consecutive integers. They act as direct entries in a table of final addresses to be built by pass 8.

The final addresses are defined in the following way. When pass 8

begins, an address pointer is set to zero. The directing bytes from pass 7 instruct pass 8 to increment the address pointer in accordance with the number of instructions to be generated in pass 10. Thus, for example, when the bytes DEFINE SYMBADDR 5, are encountered, the address pointer will contain the final address of that particular point in the program. The current value of the address pointer is then placed in word 5 of the address table.

Pass 8 also creates the run time file tables on the basis of the information in the extra table delivered by pass 5.

Address distribution. Pass 9 completes the addressing by distributing the final addresses in the address table. When a directing byte followed by a symbolic address is encountered, the symbolic address is used as an entry in the address table to get the final address. As an example: the input bytes GOTO SYMBADDR 5 might be output as GOTO FINALADDR 2093.

Final generation. The final program is put together by pass 10. The entire running system is stored as one segment of machine code on the systems tape inside pass 10. Only the necessary routines are included in the object program. This selection and relocation of running system routines creates an addressing problem when the included routines refer to each other, or are referred to by the generated code. To handle this, the address parts in the included routines are modified by a special routine on the basis of a table which contains the original base addresses and the sizes of all running system routines.

13. The Handling of Errors.

When an error has been detected the compiler will modify the offending spot in the source program in such a way that later passes will accept it as formally correct. This approach allows the translation to proceed through all the checking phases without going astray or producing an avalanche of error messages. Code generation, however, is not attempted for programs with formal errors.

Syntactical errors. Clauses with erroneous keyword structure are completely removed from the source program. Because of the strictly sequential processing this cannot be done during a single scan. The solution adopted is to let pass 2 number all format-firsts consecutively. When an error is detected inside a format the current format number is placed in a list which is left in core in order that pass 3 may recognize and erase the erroneous structure.

Semantical errors. A reference to an item that is undefined or ambigu-

ous (pass 4), an inconsistent data description (pass 5), or an operand that is used in the wrong context (pass 6) is handled by replacing the actual operand by a pseudo-operand of "undeclared type". This type of operand is accepted in all contexts by later passes.

Error messages are displayed on the line printer following the listing of the source program. Each message is preceded by a sequence number identifying the erroneous source card.

14. Copy Processing.

Copy from Library. The Cobol Library is stored on the systems tape between pass 1 and pass 2 in the form of Cobol card images. The inclusion of library code, performed by pass 1, thus merely involves a change of input from cards to tape. An exception is the INCLUDE clause, where parts of the library text must be replaced during the inclusion. Likewise, if a data description is copied from the library, the level numbers must be adjusted to their surroundings. This is done by pass 3.

Standard Label Records. Pass 2 creates a data description of label records.

Renaming of files. This option is interpreted as a shorthand method for rewriting a complete file description. When a file is introduced by renaming in pass 3, the name of it is stored temporarily together with the name of the file being copied. When all data names in the File Section have been processed, entries for the renaming-files are created in the name table. Only the file name itself gets an entry, connected by the tree link to the file being copied, whereas no entries are created for the subordinate items (this is merely a space saving device). The renaming is completed in pass 7, where file descriptions are copied inside the extra table.

Copy of data items. COPY of group items is handled in pass 3 by internal copying of the entries in the name table. This solution is very attractive compared to a literal copying of the source text, since such a process would require two additional passes. Copy clauses pointing backwards in the source program are processed when they occur in the input. During this process the level numbers and the table links are adjusted. Copy clauses pointing forwards are stored temporarily and processed at the end of the Data Division.

Pass 3 only copies information about the tree structure, but not the corresponding data descriptions. The copy clauses are therefore transmitted to pass 5, which performs similar copying inside the description table.

Copy of data values. The original value clauses result in code which at

run time is performed first. The copying of values can therefore be handled by subsequent code which moves the already inserted values to the copy items.

15. Other Tasks.

Literals. Pass 1 outputs literals as character strings preceded by bytes defining their size, point location, and sign. In pass 7 literals used as operands are adjusted to the size of the surrounding operands by extending them with leading and trailing zeroes (or blanks). Figurative constants are adjusted by a duplication of their basic value. Pass 7 then reserves storage areas for the literals and generates code to initialize their values.

Condition names. The value clauses for a condition variable are treated as a declaration of a Boolean procedure to be generated by pass 7. The reference to a condition name at run time is a call of this procedure to test whether the variable has one of the values associated with the condition name.

Picture. Picture characters are converted to a unique string of bytes in pass 1. In pass 5 a data description (size, class, and point location) is extracted from the picture, and a description of possible editing operations is generated for pass 7. Pass 7 generates the run time parameters for the editing routine. These involve descriptions of the source and the receiving items, a string containing the fixed insertion characters, and the identification of a floating string to replace leading zeroes.

Renames. The renames clause allows the programmer to refer to a sequence of elementary items inside a record, without respect to the previously defined group structure. Pass 4 will look up a unique representation of the first and the last elementary items embraced by the renames reference. This is done by means of the program link. Pass 5 completes the renames process: the relative address of the renames item is set equal to the relative address of the first elementary item, and the size is calculated as the total size of all the elementary items.

Redefines. This clause allows the programmer to go back to a previous point in the data tree and redefine the structure from that point. It allows him to reference the same storage area with different names. A redefines clause causes pass 5 to set the current relative address inside a record back to its value at the redefined point.

16. Summary of the Passes.

Pass 1. Analysis of single words:
Listing of source program. Inclusion of library code. Conver-

sion of source program to bytes and strings. Deletion of optional symbols.

Pass 2. Analysis of word structures:
Syntactical check of clauses. Deletion of redundant clauses. Differentiation of ambiguous keywords. Recording of incorrect clauses.

Pass 3. Collection of data structures:
Construction of data name table. Start of Copy Processing. Delimiting of segments in Procedure Division. Deletion of incorrect clauses.

Pass 4. Distribution of unique names:
Qualified names are checked for existence and uniqueness and replaced by single byte values. Subscripts are supplied with names of the lower array levels. Move Corresponding statements are split into single moves.

Pass 5. Collection of data descriptions:
Construction of data description table. Completion of Copy processing. Descriptions are checked for consistency. Relative addressing of data items.

Pass 6. Distribution of data descriptions:
Operand names are replaced by their descriptions. Conversion of statements to Reverse Polish form.
Check of operand types.

Pass 7. Selection of code:
Selection of running system routines to be included and code pieces to be generated. Assignment of working locations.

Pass 8. Definition of final addresses:
Construction of address table. Generation of file tables.

Pass 9. Distribution of final addresses.

Pass 10. Generation of object program:
Inclusion of running system routines. Generation of code pieces. Segmentation of program.

17. The Debugging System.

The division of the translator into passes makes it possible to design, program, and debug each pass as an autonomous program. This is an important point when the task of writing a compiler must be distributed among a group of people.

In order to achieve a systematic and thorough testing of the compiler, the following scheme was used. The passes were checked individually,

but in their natural sequence (pass 1 was debugged first, then pass 2, etc.). The test input to each pass consisted of Cobol source programs specially written to involve all the logic of the pass. The sequential testing scheme ensures that these source programs will be transformed into correct input bytes by the previously debugged passes. The debugging of the current pass is then reduced to verifying that the output bytes produced by it are correct.

The main task of the debugging system, Help, is to print out the values of output bytes and strings from the passes. The setting of the debug state is done by preceding the Cobol source program by a special test card, indicating from which passes the test output is wanted. A test option is also provided for the listing of internal tables built-up by the passes. It should be stressed that the Help system does not affect a normal translation either by space or longer run time. The Help system is loaded in the upper end of core and is overwritten unless a test card indicates that it should be protected. The test mode is set by replacing a number of dummy instructions inside pass 0 by jumps to the Help system.

About 280 Cobol programs of an average size of 60 source cards each were needed to check the entire system. The total computer time used for assembling and debugging of the translator was approximately 600 hours.

18. Evaluation of the Compiler.

The following is an evaluation of the size and performance of the translator itself and the object programs produced by it. In describing the performance of the object programs we will be more concerned with advising the user how to make the best use of Cobol as it is, rather than in debating the merits of the language.

The size of the translator is as follows:

Phase:	Program size: (words)		Fixed tables: (words)
Pass 0	1510		110
Pass 1	3200		1660
Pass 2	1840		3300
Pass 3	2080		190
Pass 4	1470		190
Pass 5	4050		300
Pass 6	3000		370
Pass 7	3180		450
Pass 8	1690		760
Pass 9	330		200
Pass 10	3950		1270
Running system	4900		0
The total system	30200	+	8800

The rather large size of Siemens COBOL (39000 words of 24 bits) as compared to GIER ALGOL (5800 word of 42 bits) is due to the complicated structure of the COBOL 61 language. Instead of having a few basic constituents that may be used in many contexts (like in ALGOL 60) COBOL 61 consists of a large number of unrelated clauses, each of which requires a special piece of code in each pass.

The speed of the compiler has been measured during translation of a series of realistic Cobol programs. This indicates a basic loading time of the compiler itself of 45 seconds plus an average translation time of 0.25 seconds per source card (or 15 milliseconds per generated instruction). As an example, a source program consisting of 750 cards was translated into an object program of 12460 instructions in 3.8 minutes.

To be compatible with other programming systems running on the Siemens 3003 the compiler was assembled as a relocatable program. In the absence of index registers the necessary address modification is performed by a loading routine. This accounts for about two thirds of the basic 45 seconds.

The reading and listing of the source program (pass 1) takes about 30 percent of the total translating time. Relatively slow are also the syntactical analysis (pass 2 = 10 percent), the run time simulation (pass 7 = 9 percent) and the final generation (pass 10 = 15 percent). The remaining passes are quite fast (4–7 percent each).

A comparison with the performance of other Cobol compilers has been made on basis of the Report on Bureau of Ships Cobol Evaluation Program (ref. 11). The participant manufacturers in this experiment were Remington Rand, RCA, IBM, General Electric, and National Cash Register. The fastest translation reported was obtained by the COBOL 60 translator developed for the GE-225. This machine is very similiar to the Siemens 3003 with respect to storage capacity and internal speed. Using a card reader, a line printer, and 6 magnetic tapes a Cobol program of 328 cards was translated into 4300 instructions in 16 minutes. Our own tests indicate that the corresponding translation time on the Siemens 3003 would amount to 2 minutes only.

To enable the user to estimate the run time of object programs in advance, and to point out certain very time consuming operations, a selected set of Cobol statements have been timed on the Siemens 3003. The execution times, listed in the appendix, point to the following bottlenecks in the running system:

Arithmetic. Exponentiation, multiplication, and division, being performed by subroutines, are 10–50 times slower than the built-in addition and subtraction.

Data scanning. Examine and editing are necessarily quite slow because the source item must be analyzed character by character.

Address computations. Operations involving a conversion of decimal integers to binary addresses (go to ... depending, reference to subscripted variables) require about 1 millisecond per conversion. This will probably turn out to be the major bottleneck in realistic programs. A considerable improvement would result if binary integers were introduced as variables in the Working-Storage Section (defined as data names with level number 77 and the description, USAGE COMPUTATIONAL).

File handling. The running system has been tested extensively for typical conversion runs, such as cards-to-cards, cards-to-printer, cards-to-tape, tape-to-printer, and tape-to-tape. In such applications the peripheral units were operating at full speed.

Acknowledgements.

The basic overall design of Siemens Cobol is due to Peter Naur and Jørn Jensen. The design, programming, and testing of the individual passes was done by Sven Eriksen, Jørn Jensen, Peter Kraft, Paul Lindgreen, Ole Riis, Peter Villemoes, and the present authors. We wish especially to record our debt to Peter Villemoes for his coordination of the group during the initial phases of the design, and to Mrs. Berta Kiær for her invaluable assistance with the documentation of the system. Finally, Dr. Herbert Donner, Mr. Helmut Hoseit, and Miss Karin-Herta von Lucius of Siemens und Halske in Munich should be given credit for their patience and aid to the group during the debugging.

REFERENCES

1. *COBOL-1961,* Report to Conference on Data Systems Languages, U. S. Department of Defense, 1961.
2. Naur, P., *The Design of the GIER ALGOL Compiler,* Annual Review of Automatic Programming 4, Pergamon Press, London, 1964. BIT vol. 3, no. 2–3, 1963.
3. Naur, P., *State Analysis of Linear Texts,* (unpublished), June 1965.
4. Williams, F. A., *Handling Identifiers as Internal Symbols in Language Processors,* Comm. ACM 2, June 1959.
5. Conway, M. E., *Design of a Separable Transition-Diagram Compiler,* Comm. ACM 6, July 1963.
6. Lindgreen, P., *Collection and Mutual Control, of RECORD-Description in COBOL by means of an Implicit State-Technique,* Proc. NordSAM 64, Stockholm, August 1964.
7. Dijkstra, E. W., *ALGOL 60 Translation,* Annual Review of Automatic Programming 3, Pergamon Press, London, 1963.
8. Naur, P., *Checking of Operand Types in ALGOL Compilers,* Proc. NordSAM 64, Stockholm, August 1964. BIT vol. 5, no. 3, 1965.

9. Brinch Hansen, P., *An Optimal Compilation of Boolean Expressions in COBOL 61*, Proc. NordSAM 64, Stockholm, August 1964.
10. Jensen, J., *Generation of Machine Code in ALGOL Compilers*, Proc. NordSAM 64, Stockholm, August 1964. BIT vol. 5, no. 4, 1965.
11. Siegel, M., and Smith, A. E., *Interim Report on Bureau of Ships COBOL Evaluation Program*, Comm. ACM 5, May 1962.

Appendix: Selected Operation Times.

Statement:	Test example:	Milliseconds:
go to (simple)	GO TO *P*.	0.02
go to (depending)	GO TO *P*, *Q*, *R* DEPENDING ON *I*.	1.13
alter	ALTER *P* TO PROCEED TO *Q*.	0.07
perform (empty loop)	PERFORM *P*.	0.22
relation	IF *B* GREATER *C* THEN.	1.14
addition	ADD *B* TO *C*.	0.5
division	DIVIDE *B* INTO *C*	5.7
multiplication	COMPUTE *A* = *B* * *C*.	17.2
exponentiation	COMPUTE *A* = *G* ** *I* (*I* = 2)	14.2
	(*I* = 3)	26.0
	(*I* = 4)	29.8
move (simple)	MOVE *B* TO *C*.	0.29
move (editing)	MOVE *X* TO *Y*.	3.8
examine	EXAMINE *B* TALLYING ALL 9.	6.3
subscripted variable	COMPUTE *B* = *D* (*I*).	1.25
	COMPUTE *B* = *E* (*I*, *J*).	2.20
	COMPUTE *B* = *F* (*I*, *J*, *K*).	3.12

Test operands:	Picture:
A	9(16)
B, *C*, *D*, *E*, *F*	9(8)
G	9(4)
I, *J*, *K*	9
X	9(6)*V*9(2)
Y	$$$$$,$$9.99–

A/S REGNECENTRALEN
COPENHAGEN
DENMARK