Design of a Separable Transition-Diagram Compiler*

MELVIN E. CONWAY
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) COBOL [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed COBOL compiler which can be built by two people with an assembler in less than a year.

The compiler design presented here has the following properties.

1. It processes full elective COBOL except for automatic segmentation and its byproducts, such as those properties of the ALTER verb which are affected by segmentation. The verbs DEFINE, ENTER, USE and INCLUDE are accessible to the design but were not included in the prototype coded at the Case Computing Center.

2. It can be implemented as a true one-pass compiler (with load-time fixup of forward references to procedure names) on a machine with 10,000 to 16,000 words of high-speed storage. In this configuration it processes a source deck as fast as current one-pass algebraic compilers.

3. It can be segmented into many possible configurations, depending on the source computer's storage size, such that (a) once a segment leaves high-speed storage it will not be recalled; (b) only two working tapes are required, and no tape sorting is needed. One such configuration requires five segments for a machine with 8000 six-bit characters of core storage.

Of course any compiler can be made *one-pass* if the high-speed storage of the source computer is plentiful enough; therefore, what this exposition has to offer is a collection of space-saving techniques whose benefits are real enough to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to COBOL compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete COBOL syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing modules which communicate with each other according to the following restrictions: (1) the only communication between modules is in the form of discrete items of information; (2) the flow of each of these items is along fixed, one-way paths; (3) the entire program can be laid out so that the input is at the left extreme, the output is at the right extreme, and everywhere in between all information items flowing between modules have a component of motion to the right.

Under these conditions each module may be made into a *coroutine*; that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.[1] There is no bound placed by this definition on the number of inputs and outputs a coroutine may have.

The coroutine notion can greatly simplify the conception of a program when its modules do not communicate with each other synchronously. Consider, as an example, a program which reads cards and writes the string of characters it finds, column 1 of card 1 to column 80 of card 1, then column 1 of card 2, and so on, with the following wrinkle: every time there are adjacent asterisks they will be paired off from the left and each "**" will be replaced by the single character "↑". This operation is done with the exponentiation operator of FORTRAN and COBOL. The flowchart of such a program, viewed as a subroutine pro-

* The work described here was performed at Case Institute of Technology in 1961 and was supported in part by Univac Division of Sperry Rand Corporation.

[1] To the best of the author's knowledge the coroutine idea was concurrently developed by him and Joel Erdwinn, now of Computer Sciences Corporation.

viding characters to a main routine which calls it, is shown in Figure 1.

Notice that the simple requirement of compressing asterisks requires the introduction of a switch which

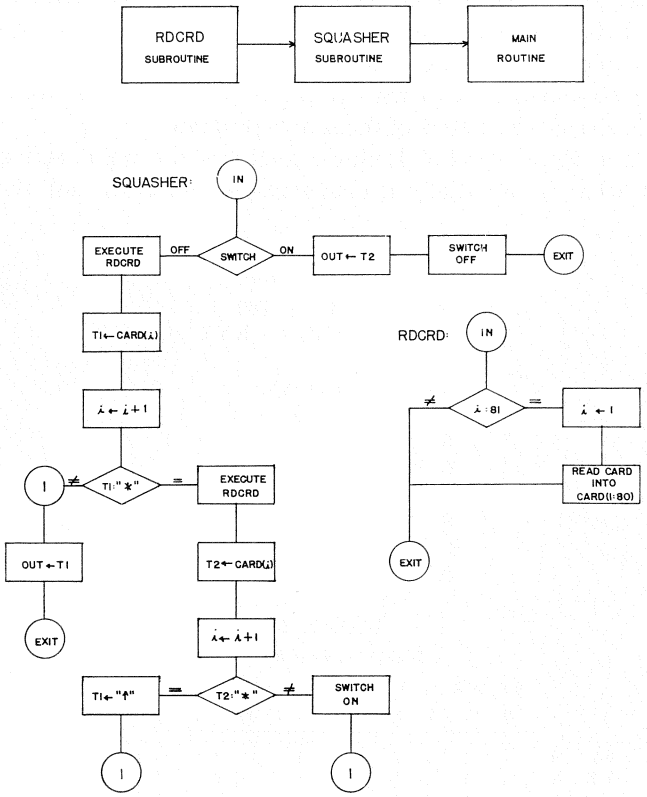


FIG. 1. Asterisk squasher subroutine with environment. SWITCH is initialized OFF; i is initialized to 81; OUT is output area of SQUASHER.

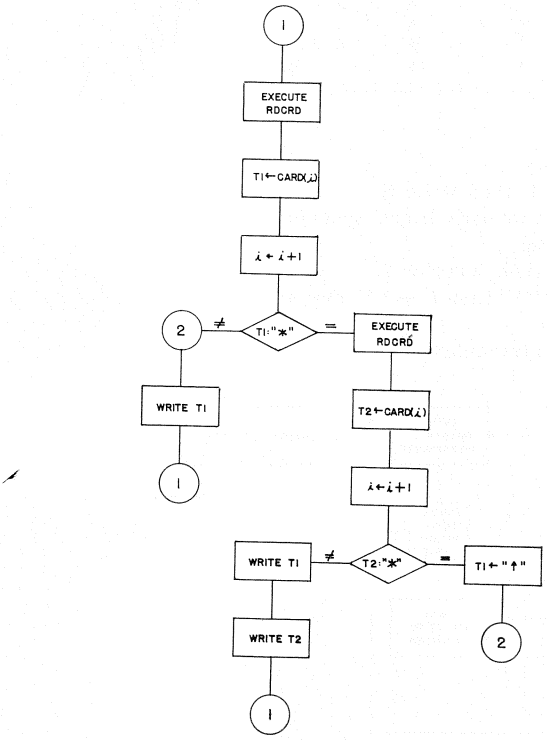


FIG. 2. Asterisk squasher as coroutine. First entrance is to 1.

bifurcates the subroutine and selects the part to be used at each call depending on the situation at the last call. The reason for the switch is that *each call of the subroutine must result in output of exactly one character*. A programmer well versed at inventing counterexamples could dream up a requirement for a subroutine which would necessitate an absurd arrangement of switches in order to preserve the one-output-for-each-call relationship required of the subroutine.

The coroutine approach to the same problem accomplishes the switching job implicitly by use of the subroutine calling sequence. When coroutines A and B are connected so that A sends items to B, B runs for a while until it encounters a read command, which means it needs something from A. Then control is transferred to A until it wants to "write," whereupon control is returned to B *at the point where it left off*. Figure 2 shows the asterisk squasher when both it and the using program are coroutines.

Figure 3 illustrates the essence of separability. Instead of having modules A and B communicating as coroutines with a coroutine linkage between write statements in A and read statements in B such that control is passed back and forth once each time an item is transferred, it is possible *without changing anything in A or B except the read and write linkages* to have A write all its items on a tape, to rewind the tape and then to have B read all the items from the tape. In this sense, then, the pair of programs A and B can operate as a one-pass or a two-pass processor with only trivial modification.

As background, the coroutine linkage on the Burroughs 220 is described here. The 220 is a sequential, single-address machine with the sequence counter called the P-register; during the execution of an instruction it contains the location of the next instruction to be fetched. Unless the current instruction causes the machine to branch, the P-register will contain one plus the location of the current instruction. The UNCONDITIONAL BRANCH instruction BUN A works by placing its *address A* into the P-register. The STORE P instruction STP B places the contents of *P plus one* into the address part of the contents of location B. The standard subroutine call is

```
STP   EXIT
BUN   ENTRANCE
```

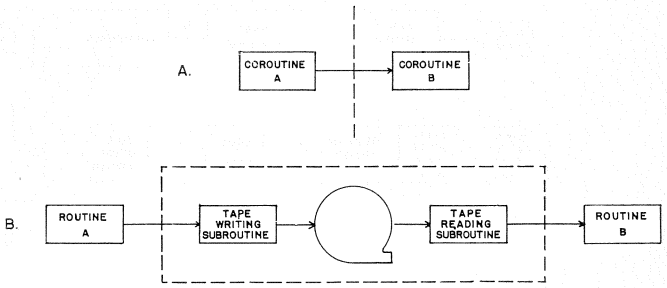


FIG. 3. Illustration of a property of separable programs.
A. A and B, linked as coroutines, communicate directly.
B. A writes its entire output before B reads anything.

where EXIT contains a BUN instruction whose address will be altered by the STP instruction in the call. A pair of subroutines becomes a pair of coroutines by adding to each an isolated BUN instruction which we can call its router, and by changing the addresses of the STP, BUN calls as follows: when coroutine *A* calls coroutine *B* the call is

```
STP   AROUTER
BUN   BROUTER.
```

Thus, the router is actually a generalization of the switch of Figure 1. Getting a system of coroutines started is a matter of properly initializing the routers.

Figure 3 shows that the coroutines of a separable program may be executed alternately or serially. When true parallel processors are available the fact that the coroutines of a separable program may be executed *simultaneously* becomes even more significant.

COBOL Compiler Organization

Figure 4 presents the coroutine structure of the COBOL compiler designed at Case. The program is separable under the condition that the two pairs of modules which share tables are considered to be a single coroutine.

The reader is asked to understand that the present treatment is concerned more with exposition than completeness. A more thorough treatment would not ignore COPY, pictures, and literals, for example. Let it suffice to say that these features can be accommodated without any significant design changes over what is presented here.

In Figure 4 solid arrows are communication paths between coroutines; dashed arrows show access to tables. When the dashed arrow points to a table the latter is being built; when the dashed arrow points away the table is supplying values to the using coroutine. The specific operations performed by the coroutines will be discussed in the following four sections.

Lexical Analysis

The input, on line B of Figure 4, to the Diagrammer consists of (one-word) items denoting either names or COBOL basic symbols. The class of basic symbols, over 300 elements in size, consists of all characters of the source alphabet (other than numerics, alphabetics, hyphen in names, and space) together with all the COBOL reserved words and the paragraph symbol ¶. (This internal special symbol is inserted by the card scanner whenever card column 8 is not blank. Such a device converts many format recognition problems to syntax analysis problems.) The lexical analysis process embodied in the Basic Symbol Reducer and the Name Reducer converts the source program into a sequence of integer-coded one-word items in one-to-one correspondence (with the exception of ¶) with the words and special characters of the source program.

The Basic Symbol Reducer analyzes the input string by what is essentially a character-pair analysis, but the

TABLE 1

Class	*Character*
0	012 ··· 789
1	ABC ··· XYZ
2	–
3	b (space)
4	= * /
5	,
6	.
7	″
8	¶ (
9	)
X	+

TABLE 2

Right Character R

Left Character L		0-9 0	A-Z 1	– 2	b 3	=*/ 4	, 5	. 6	″ 7	¶(8	) 9	+ X
0-9	0	1	1	1	3		3	1			3	
A-Z	1	1	1	1	3		3	1			3	
–	2	1	1	1	6							
b	3	2	2	2	5	5		2	4	5		2
=*/	4				6	7						
,	5				6							
.	6	1			1							
″	7				5		5	5				
¶(	8	6	6		6							
)	9				6		6	6				
+	X	1			6							

Functions:
1. S ← SL
2. S ← empty
3. Function 1, then write S
4. Enter non-numeric literal scanner
5. Do nothing
6. Write L, then function 2
7. If LR = "**" then $R \leftarrow$ "↑" else error

Blank: error
S is a string accumulator

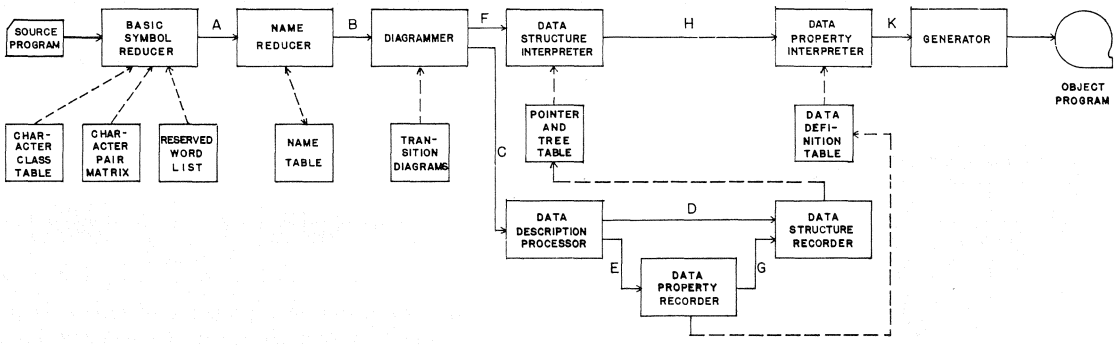


FIG. 4. COBOL Compiler Organization

character pair matrix (for the IBM "H" set) need not be 49 × 49 but only 11 × 11 since there are only eleven operationally different types of characters. These types are listed in Table 1, the Character Class Table. Furthermore, only three bits are required for each entry of the Character Pair Matrix, which occupies eleven words on the 220. Table 2 shows the matrix with the set of actions taken for each character pair.

The Reserved Word List and Name Table were built for the 220 according to a method of F. A. Williams [3]. When the source machine has an adequate memory both tables can be combined into one, initialized to the set of reserved words. On line A of Figure 4 all the reduction has taken place except that names are in a fixed 30-character-plus-identification format.

In light of the organization of the compiler into lexical analysis, syntactic analysis and synthesis (including data storage allocation in the Data Division and code generation in the Procedure Division), with the three in series there is a clear-cut strategy for getting both high compiling speed and low space consumption from the one-pass version of the design: make the lexical analysis as fast as possible even at the expense of some space, and make the rest as compact as possible even at the expense of speed. This is explained by the simple fact that most of the time is spent in lexical analysis. In our experience with the prototype we found that the input speed difference between lexical analysis alone and lexical plus syntactical analysis

condition: **IF** formula { IS [**NOT**] **GREATER** THAN | IS [**NOT**] **LESS** THAN | IS [**NOT**] **EQUAL** TO | **EQUALS** | **EXCEEDS** } formula

FIG. 5. COBOL-like definition of *condition*

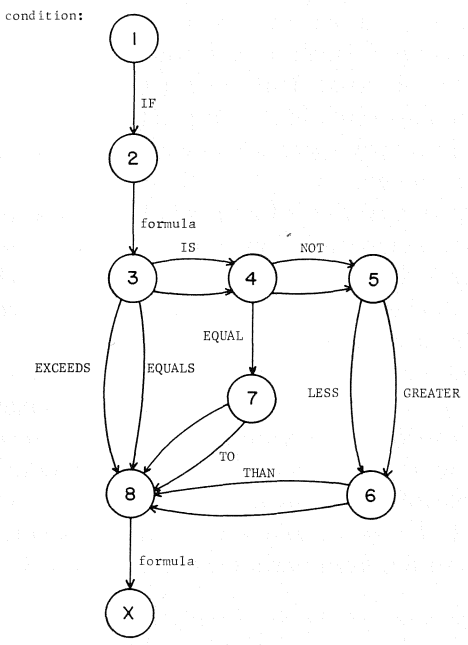


FIG. 6. Transition diagram definition of *condition*

was about ten percent. It appears that nothing short of pure sabotage can be done to the syntactical analysis and synthesis portions to slow the whole compiler down to less than 75 percent of the speed of the lexical analysis routine alone; hence our name "Seventy-five Percent Rule" for this strategy.

Syntactical Analysis

An abbreviated definition of *condition* is given for this exposition in Figure 5. Keep in mind that the syntactical analyzer (Diagrammer) sees single symbols entering it for things we call IF, IS, NOT, *data-name*, and so on. Call these symbols which are input to the diagrammer *input symbols*. Observe, then, that any sequence of input symbols properly called a *condition* must correspond to one of five paths through the *condition* definition, starting at the left IF and ending at the right *formula*, each path corresponding to a particular choice of relation. If the Diagrammer is thought of as having a window which displays each input symbol as it comes from the lexical analyzer, the definition of a syntactic type like *condition* is a rule for predicting, for each input symbol in the window, what the legal set of successors of that symbol is. A *transition diagram* is a formalization of this notion of what a definition is. Figure 6 shows the diagram equivalent to the definition of Figure 5.

A transition diagram is a network of nodes and directed paths with two distinguished types of nodes: an entrance node (usually drawn at the top) has at least one path leading from it, and an exit node (labeled "X") has at least one path leading to it and no paths leading from it. Every transition diagram defines a syntactic type which is not an input symbol, and every such syntactic type has one transition diagram which defines it. A transition diagram has exactly one entrance node and at least one exit node.

Each path is said to be blank (as, for example, one path leading from node 3 to 4 in Figure 6) or to have a symbol on it. The symbol is either an input symbol, or else it is a syntactic type defined by a transition diagram. (We use capitalized words on the paths for reserved words and lowercase words for names and syntactic types.) No two paths leading from a node may be blank or may have the same symbol on them. No transition diagram may have a sequence of blank paths leading from the entrance to an exit node. The set of blank paths may contain no loop.

There will be one transition diagram labeled (that is, defining) *COBOL program*. The Diagrammer starts at its entrance node. The object is to get to an exit node; to have done so implies that a COBOL program has traveled past the window of the Diagrammer. Similarly, getting from the entrance to an exit of any transition diagram means that the corresponding syntactic type has been traversed.

The rules of the Diagrammer for leaving a node are as follows.

STEP 1. Examine all paths leaving the node which have input symbols on them. If there is a match with the symbol in the window, read the next input symbol into the window and traverse the path. Now go to Step 5.

STEP 2. If there was no match in Step 1, try each remaining nonblank path leaving the node. Each path will correspond to some transition diagram. The path may be traversed if and only if it is possible to get from the entrance to an exit of that diagram. This is attempted by pushing down in a last-in-first-out stack called the *linkage stack* the current node number and then going to Step 1 for the entrance node of the particular transition diagram being tried.

STEP 3. If there was no match in Step 2 and there is a blank path leading from the node, follow it and go to Step 5.

STEP 4. If Step 3 was unsatisfied the Diagrammer is at a dead end. If the linkage stack is nonempty this condition is a failure to traverse, in Step 2, a particular path corresponding to the transition diagram in which the dead end occurs. Pop up the linkage stack, reposition the window to the symbol present when the diagram was entered, and try another path in Step 2. If the linkage stack is empty a syntactical error exists in the input string.

STEP 5. There are two cases.

a. The path just traversed does not end at an exit node. Go to Step 1 for this new node.

b. Otherwise, pop up the linkage stack, return to the node whose number was at the top of the stack, and traverse the path corresponding to the diagram just exited. Now go back to Step 5.

The above procedure constitutes the entirety of that part of the Diagrammer which checks syntax.

Recall the condition that no two paths leading from the same node may have the same symbol on them. The following question arises: What if two paths leading from the same node have syntactic types on them, and the two transition diagrams defining these types have paths leading from the entrance nodes which have the same symbol on them? If this happens, then with certain pathological languages the interpretation of a given input string might depend on the order in which the paths leading from a node are tried. Indeed, the same problem can occur if this nonuniqueness exists at a deeper level than the first. Because it is desirable not to have to worry about this problem, let us consider it now in more detail.

Two conditions on a system of transition diagrams are presented and their effects on regularizing languages are discussed. The first condition, called the "No-Loop Condition," says that no transition diagram will make a reference to itself (i.e. it will not have a path with the syntactic type which it defines) without having first read an input symbol after it was entered. For, if after entering a transition diagram defining syntactic type t no input symbol has been read when an attempt is made to traverse a path with t on it, a periodic (infinite loop) condition exists. And only then does an infinite loop exist, since if the input string is finite a loop which reads-input symbols will terminate.

The second condition, called the "No-Backup Condition," defines away any need to specify an order in which the nonblank paths leading from a node should be tried. Every path has associated with it a set of input symbols, called its set of initial input symbols, defined as follows. When the path has a syntactic type on it an input symbol is an initial input symbol if and only if when it is in the input window and the transition diagram defining the syntactic type on the path in question is entered, that symbol will be read before either a dead end occurs or the diagram is exited. The class of initial input symbols of a blank path is empty, and the class of initial input symbols of a path with an input symbol on it is the one-element class containing that symbol. The No-Backup Condition says that the No-Loop Condition holds and that for every node in the system of transition diagrams the sets of initial input symbols of all the paths leading from that node are disjoint. For, if the classes of initial input symbols for all the nonblank paths leading from a node are disjoint, then the classes of input strings which enable the respective paths to be traversed will be disjoint, and therefore the order in which these paths are tried will be irrelevant. The No-Backup Condition is clearly stronger than it has to be in order to obtain independence of order of testing, but this condition confers another property on a system of transition diagrams which is to be sought: such a system of diagrams will never require backing up the input string during scanning. All nonerror dead ends encountered within any transition diagram will be encountered before any input symbol is read; thus the response to a dead end in Step 4 (when the linkage stack is nonempty) is simply to pop up the stack and try another path in Step 2, without repositioning the window. In fact the tests of Steps 1 and 2 may be freely intermixed.

The No-Backup Condition makes error indication more specific because an error dead end can be immediately discovered without first emptying the linkage stack by the fact that at least one symbol has been read since entering the transition diagram harboring the dead end.

In a sense, the No-Backup Condition is a device for legislating out of existence the ambiguity problem [4] in languages defined by transition diagrams. The crucial point here is that the syntax of COBOL-61 and ALGOL-60 may be defined by transition diagrams which satisfy the No-Backup Condition. A one-pass compiler for either of these languages which is constructed according to the Seventy-five Percent Rule and which uses No-Backup transition diagrams will be competetive in both compiling speed and memory space with a compiler of any other contemporary design.

The catch in all this is that a set of No-Backup diagrams for a given language is constructed by a process which is neither straightforward nor easy to describe. The COBOL *condition*, for example, may begin with a left parenthesis which can surround an entire condition or just the left *formula* of a relational test. To expose the subtleties of the construction of No-Backup diagrams, we change the definition of *condition* in Figure 5 to retain only the essentials. Further, we define *formula*. These are both done in Figure 7. Examples of conditions are $(A = B)$, $A = B$, $((A + B) = C)$, $(A + B) = C$, and so on.[2] An equivalent set of No-Loop transition diagrams are given in Figure 8.

[2] The presence of the IF, strictly speaking an error in the COBOL manual as well as Figure 5, has been eliminated in Figure 7.

This is not a set of No-Backup definitions, however, because encountering an initial left parenthesis gives no indication whether it surrounds the entire condition or just the left formula. Consider $(A + B) = C$. The sequence of transitions will be as follows.

1 → 2	read (
1	enter *condition*
6	enter *formula*
9	enter *term*
12 → X	read A
9 → 10	*primary* traversed
10 → X	+ is not *
6 → 7	*term* traversed
7 → 8	read +
9	enter *term*
12 → X	read B
9 → 10	*primary* traversed
10 → X	) is not *
8 → 7	*term* traversed
7 → X	) is not +
1 → 4	*formula* traversed

---

condition: { (condition) | formula = formula }

formula: { term | formula + term }

term: { primary | term * primary }

primary: { data-name | (formula) }

FIG. 7. Another definition of *condition*

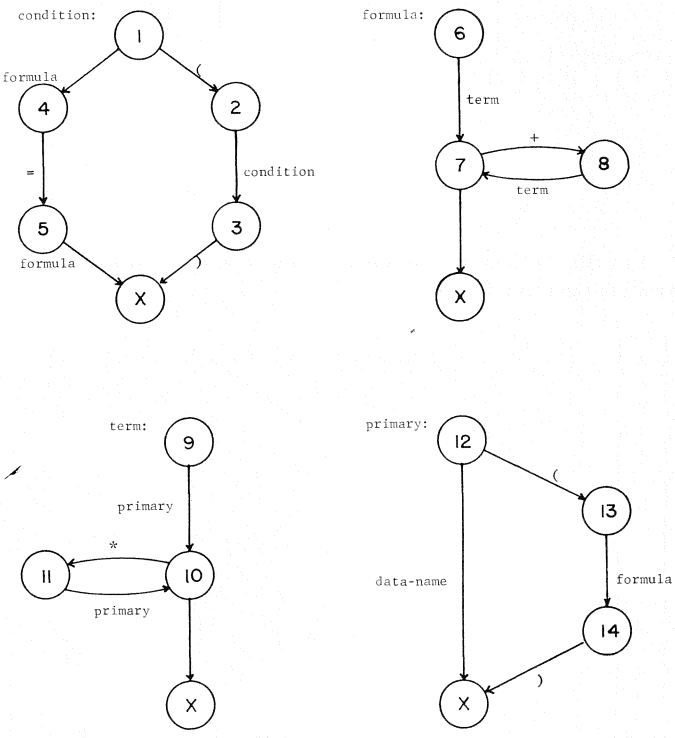


FIG. 8. No-Loop definitions equivalent to Fig. 7

At this point the Diagrammer reaches a dead end. The condition can only be scanned directly if the input is backed up and the first transition is made 1 → 4, not 1 → 2.

The difficulty is that *formula* can also begin with "("; to scan without backup means that to go down a path with a "(" on it should not involve a committment that the string being scanned is a condition and not a formula, or vice versa. Thus a solution is to introduce a *multiple-exit* transition diagram so that all syntactic types beginning with the same input character are defined by that diagram; the different syntactic types result in different exits, labeled XI or X2. Figure 9 defines *condition* this way.

There is little else to the Diagrammer except the generation of Polish. This is accomplished by little subroutines called *actions* which are activated when certain paths are traversed. Figure 10 gives the actions required to create trailing-operator Polish from formulas. Remember that an action on a path with a syntactic type is executed *after* the corresponding transition diagram is exited. The actions for translation of ALGOL into the intermediate language suggested by Grau [5] would be as simple as Figure 10 suggests.

In a computer representation each path is represented by an item occupying one word or less, if possible. All

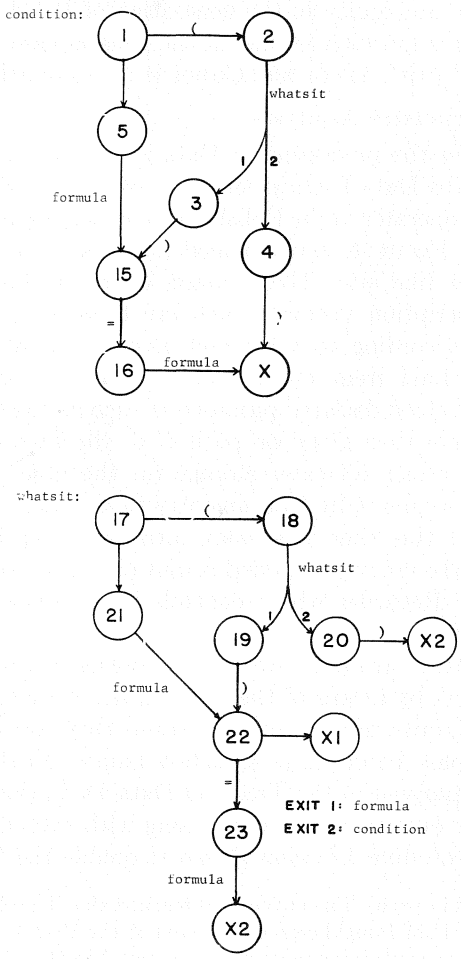


FIG. 9. No-Backup diagram to replace the *condition* diagram of Fig. 8.

paths from a given node are grouped together and a node number, instead of being an element of a sequential set of integers, is the address of the first path word of that node. A given path word contains a final node number, an action number, a bit to distinguish between input symbols and syntactic types, an input symbol or syntactic type number (in the latter case, the number of the entrance node), and a bit to distinguish the last path from a node. (If this path is blank its word can be eliminated simply by beginning the node to which it would connect in its place.) Exit nodes are represented by special one-word items. Clearly, if the syntax is to be coded in the programming language using the advantages of symbolic addressing (and this is most desirable), then an unusual programming system is required for building a transition-diagram compiler. Macros may be useable in this connection but would be cumbersome if they are processed slowly. The Case assembler was given a definable operation with specifiable operand fields to implement symbolic coding of the syntax.

The No-Backup Condition removes ambiguity in the task of syntax recognition. There is no reassurance that the introduction of multiple-exit diagrams confers the same blessings on the translation task. What if, for example, the actions associated with the several syntactic types of a multiple-exit diagram were very different? There would still be an ambiguity in the generation of Polish. The answer to this question is that in the general case the problem is real; with ALGOL and COBOL it does not exist.

Data Structure Analysis[3]

Consider the processing of Data Division item descriptions by the Data Description Processor. Each clause generates an operator in the Polish string which is passed along line C of Figure 4, accompanied perhaps by a numerical parameter like size. These clause operators are used to build description vectors which are then completed and checked according to a method given in another article [6]. The final item description vector and other non-Boolean-valued declared parameters such as size and point location are then given on path E to the Data Property Recorder which allocates storage for the object-program item and sequentially adds an entry to the Data Definition Table. At this time the index into the Data Definition Table of the entry just added is sent on line G to the Data Structure Recorder which controls its insertion into the Tree Table.

The Data Structure Recorder handles qualification of data names by means of the Tree Table. We now consider the techniques used therein, because they are crucial to the one-pass handling of qualified names. First, observe that the purpose of the Tree and Data Definition tables is to supply enough information along with each data name in the Procedure Division Polish to enable the Generator to generate complete code. The Polish written during the Procedure Division follows path F to the Data Structure Interpreter where sufficiently qualified names (which are actually sequences of integer indices into the name table) pick up new single integer values which are one-to-one with declared object-program data items, not with one-word names, as in the Williams algorithm [3]. This final integer value for each item is actually the index which came down path G when the Data Definition Table entry for that item was created. That is, this new representation of the data item name can be loaded into an index register to access directly from the Data Definition Table all the important information about the item. The procedural Polish moves along path H to the Data Property Interpreter, where this access operation is performed and the new name of each data item is replaced by the set of proper-

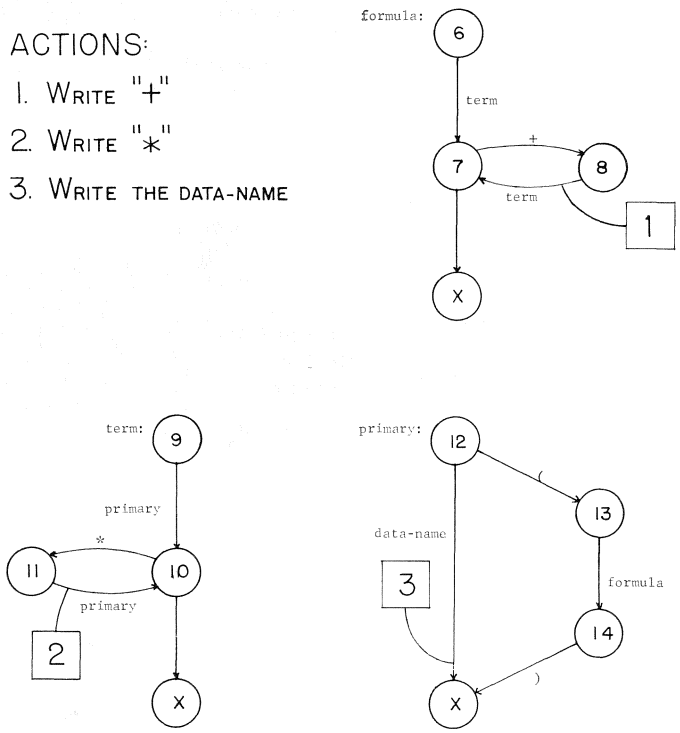


FIG. 10. Actions whose numbers are in square boxes generate trailing-operator Polish.

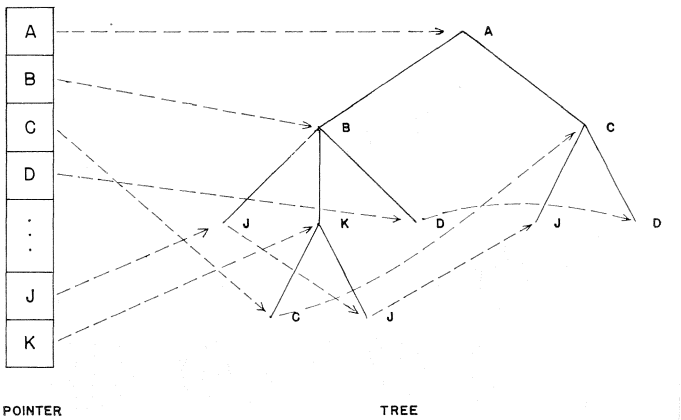


FIG 11. Structure of data and data-names in example

[3] Added in proof. The author has learned that Harold W. Lawson, Jr. of IBM Poughkeepsie delivered at the 1962 ACM Conference a paper containing most of the material in this section: The Use of Chain List Matrices for the Analysis of COBOL Data Structures.

ties of this item. The procedural Polish then enters the Generator along path K. The Generator can create complete data-handling code because accompanying each operand in the Polish is a complete description of the corresponding object-program item.

Consider the following hypothetical data structure:

```
FD  A
    01  B
        03  J
        03  K
            04  C
            04  J
        03  D
    01  C
        02  J
        02  D
```

Assume further that the internal representations given by the Williams Name Reducer are as follows:

```
A:   31
B:   32
C:   33
D:   34
J:   40
K:   41.
```

The structural relationships are shown by the solid lines of Figure 11. This structure can be represented by the following tree table:

Loc		*Name*	*Up*
001	A	31	000
002	B	32	001
003	J	40	002
004	K	41	002
005	C	33	004
006	J	40	004
007	D	34	002
008	C	33	001
009	J	40	008
010	D	34	008

LOC is not in the table, but gives the relative location of each word of the table. The NAME field contains the internal representation of the data name and the UP field gives the LOC address of the immediate parent in the data structure.

Notice that level numbers do not enter the table; they are used only to provide structural information. Specifically, if two adjacent item descriptions in the Data Division have equal level numbers, they have the same parent, namely the most recent item with a lower level number; if the second item description has a higher level number it is a descendant of the first; if the second item description has a lower level number then there is a preceding item description with the same level number as the second, such that all intervening level numbers are higher; these two have the same parent. The UP entry is constructed from the level number with the aid of a push-down list. The details of this construction will be given later.

The Data Structure Interpreter will accept sequences of name representations as the representations of qualified names, for example, (33, 41, 31, 0) for "C IN K IN A". This will be called a qualification sequence. It is assumed that no name has 0 as an internal representation.

Given the correct entrance point into the table (LOC = 005), it is a simple matter to establish that (33, 41, 31, 0) is in the table; just start at the entrance point and match the names in the qualification sequence against the names in the NAME column, using the UP column to specify the next word to check. Thus, 33 matches against word 005, 41 matches against word 004, 31 does not match against 002 but it does against 001. The matching process is successful unless a table word with the UP entry = 000 is encountered before the qualification sequence is exhausted. The process is equivalent to crawling up the data structure tree matching elements of the qualification sequence with names at nodes of the tree.

Entrance can be made into the table by a list of pointers. This list contains in location k the address of the first word of the tree table containing an occurrence of the name representation whose value is k.

Loc	*Pointer*
⋮	
31	001
32	002
33	005
34	007
⋮	
40	003
41	004

Hence, entrance into the tree table is gained by looking at word k of the pointer table, where k is the first element of the qualification sequence.

Now it remains only to take care of multiple occurrences of names. Consider "J IN C": (40, 33, 0). This would be found by jumping up the tree table from 009 to 008, except that there is no entrance to word 009. This entry is accomplished by linking up all equal names with a NEXT column in the tree table corresponding to the dashed lines of Figure 11. The final form of the Tree Table follows.

Loc	*Name*	*Up*	*Next*	*Link*
001	31	000	000	
002	32	001	000	
003	40	002	006	
004	41	002	000	
005	33	004	008	
006	40	004	009	
007	34	002	010	
008	33	001	000	
009	40	008	000	
010	34	008	000	

The LINK column points to the entry in the Data Definition Table corresponding to the data item represented by each word in the Tree Table; this is the index which is provided by the Data Property Recorder along path G and which is the unique internal representation of declared

object-program data items. It should now be clear that path D carries ordered pairs (level number, name) for constructing the Tree Table entries. Path D could be eliminated by sending the ordered pairs through the Data Property Recorder.

Figure 12 describes the program used by the Data Structure Recorder for building the Tree Table from these ordered pairs. If the level FD is given the value 01 then one must be added to all level numbers before they enter this program. Figure 13 shows the program used by the Data Structure Interpreter for matching qualification sequences. Notice that existence and uniqueness checks are made.

A useful characteristic of the Tree Table method for representing data structures is the ease with which the immediate descendants of an item can be found in the table. Thus, the CORRESPONDING modifier is easily accomodated.

In most realizations of the design presented here the Data Definition Table will be the largest of all. In source machines with small memories the Data Property Interpreter, which is a trivial program, can be made a separate memory load, sharing storage only with the Data Definition Table. Happily, this table can be written directly on tape by the Data Property Recorder; no access to it is required except by the Data Property Interpreter.

As different as the syntactic structures of ALGOL and COBOL may appear to be, essentially the same compiler may be used for both source languages. Qualification in COBOL has its analogy in ALGOL as follows: if every block is given an internally-generated name then the identifiers local to that block are qualified by the block name. The Tree Table thus provides a method for distinguishing multiple uses of the same identifier.

Code Generation

Because the Polish intermediate language is simply a minimal representation of the information in the source language, the form of the intermediate language is more naturally related to the source language than to the object language. This naturalness is evident in the simplicity of attaching actions to transition diagram paths once the embodiment of the source language definition in transition diagrams has been decided.

Whether a similar naturalness exists for the translation from intermediate language to object code is heavily dependent on the nature of the object machine. A measure of such naturalness might be given as follows. If the sequence of operators (as distinguished from operands) in the inter-

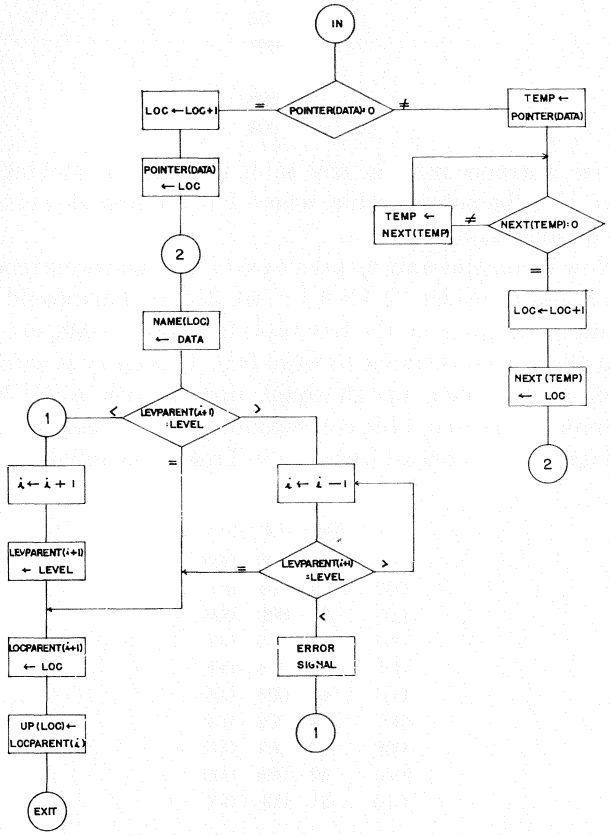


FIG. 12. The Tree and Pointer table builder. Input is (LEVEL, DATA). LOC gives location of new Tree Table word. The counters i, LEVPARENT (1) and the arrays POINTER, NEXT, and UP are initialized to zero. The error violates rule at bottom of page VI-19 of COBOL manual.

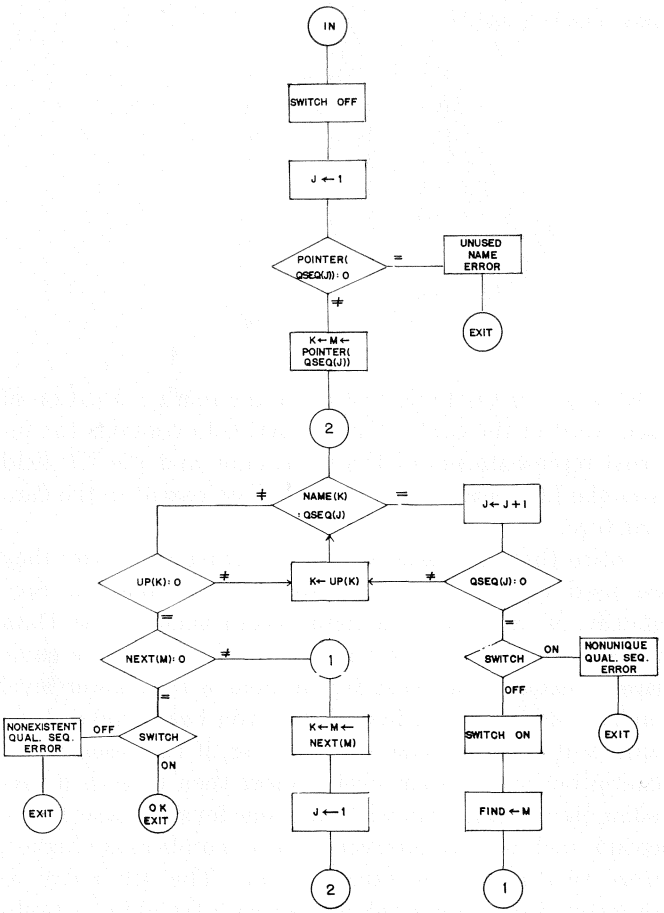


FIG. 13. Qualification sequence analyzer. Sequence is in QSEQ (1:...). LOC of found Tree Table word is in FIND.

mediate string strongly determines the sequence of operators (as distinguished from addresses or names) in the object string, that is, if the form of the object code is pretty well independent of the content of the Data Definition Table, then the generation is natural. Unnatural generation is characterized by large amounts of testing of information in the Data Definition Table before the form of the response to an operator in the intermediate code can be determined. Depending on the source language, this measure of naturalness is one component of the characterization of an object computer as being "commercial" or "scientific."

For example, the translation of an ALGOL intermediate language like that of Grau [5] to Burroughs B5000 code is natural to the point of being trivial by comparison with other machines. The translation of COBOL intermediate language to the code of a word-oriented machine like the Burroughs 220 is painfully unnatural; the response to the MOVE-operator in the Case prototype compiler occupied 27 pages of flowchart.

The extension to code generation of techniques like those discussed above in connection with syntactical analysis and Polish generation has been considered [7] for natural generation processes. The construction of *general* methods for code generation which are as neat as those which exist for source-to-intermediate translation is an important unsolved problem.

The experience of this author in construction of COBOL generators is limited to the 220. The following observations may apply to other word-oriented computers. Much trouble in generating addresses, shift counts and partial field specifications can be saved if care is taken in choosing the representation of fields in the Data Definition Table. The 220 hardware representation of the ten nonsign digits within a word, counting from the left, is 1, 2, 3, $\cdots$, 8, 9, 0. This was rejected in favor of the 7070 representation: 0, 1, 2, $\cdots$, 7, 8, 9. Using the latter representation all digits of a 10,000-word memory are addressed monotonely by a five-digit number. A field A is represented by three numbers: N_A is the four-digit memory address of the word containing the leftmost digit of A; L_A is a single digit giving the position of the leftmost digit of A within N_A; R_A is the digit address (in the sense of the five-digit number, above) of the rightmost digit of A, relative to digit 0 of N_A. Thus, $R_A - L_A + 1$ is the length of the field, the integer part of $(R_A \div 10) + 1$ is the number of words containing A, and $R_A \pmod{10}$ is the position of the rightmost digit of A. The moral of this story is that the Generator should be able to do additive arithmetic in a radix equal to the number of bytes per word for each byte size of the object machine, so that no division need be performed. Once this representation was chosen, the unnatural 220 generators reduced to sequences of arithmetic tests on the Data Definition Table entries interspersed with some minor arithmetic operations on these entries. Because of the space problem and the reassurance given by the Seventy-five Percent Rule, a natural choice of language in which to express the generators was a simple three-address interpretive code. Such a choice leaves something to be desired in the way of elegance.

Finally, we discuss the assignment of addresses to branch instructions. No one-pass compiler can generate complete code, since in response to a GO TO statement which jumps forward the compiler cannot possibly know on the first pass what address to put into the branch instruction. Since, with the subset of COBOL being considered here, forward branching is the only reason for a second compiler pass, it is usually economical to perform the remaining operations at load time by fixing up the branch addresses in the object memory.

To be more specific, assume that three statements, GO TO INDECISION, occur in the source program before the procedure name INDECISION occurs as a paragraph name. Furthermore, assume that the three GO TO's generate unconditional branches (BUNs) at locations 0528, 0742 and 0856; also the name INDECISION is finally defined to have the value 1234. Clearly, the desired coding is

```
0528:   BUN  1234
          ⋮
0742:   BUN  1234
          ⋮
0856:   BUN  1234.
```

This is what actually gets generated:

```
0528:   BUN  0000
          ⋮
0742:   BUN  0528
          ⋮
0856:   BUN  0742
          ⋮
xxxx:   FIXUP(0856, 1234).
```

The FIXUP(0856, 1234) is not loaded into memory but is an instruction to the loader to fix up the address of the word in location 0856 to be "1234". Before making the change, the loader checks for zero in the address; if the address is zero loading is resumed after the fixup terminates; otherwise the nonzero address specifies the next location to be fixed up.

In the compiler this technique requires storage for an address plus a bit for each unique procedure name. The bit records whether the value of the procedure name has been defined yet by the occurrence of the name in a paragraph or section heading. The address storage cell, initialized to zero, holds the value of the name (if it has been defined) or the location of the most recent forward branch to that name; this cell provides the address of every branch instruction generated.

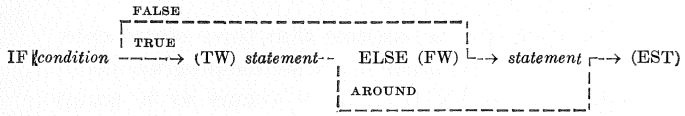
The object location of each instruction generated is determined by a location counter, LC, which records the location of the instruction currently being generated and is incremented immediately after the generation of each instruction.

As Figure 14 shows, the Polish operators controlling sequencing in conditional sentences are written at the places given below in parentheses.

IF *condition* (TW) *statement* ELSE (FW) *statement* (EST)
IF *condition* (TW) *statement* (FW) (EST). (ESN)

Notice that the second form never appears without ending in a period. The operator names TW, TN, FW, FN, EST, and ESN are mnemonics for True is now, True is next, False is now, False is next, End statement, and End sentence, respectively. Where instead of *statement* the phrase NEXT SENTENCE occurs, the operators TW and FW are replaced by TN and FN.

The three basic control paths within a conditional statement are shown as dotted arrows below. They are given the names TRUE, FALSE, and AROUND.



A fourth path, called NEXT, handles NEXT SENTENCE and will be treated later. When conditional statements are nested, the TRUE, FALSE, and AROUND paths exhibit nested last-in-first-out (LIFO) behavior. As might be expected, then, three LIFO stacks called TRUE, FALSE'

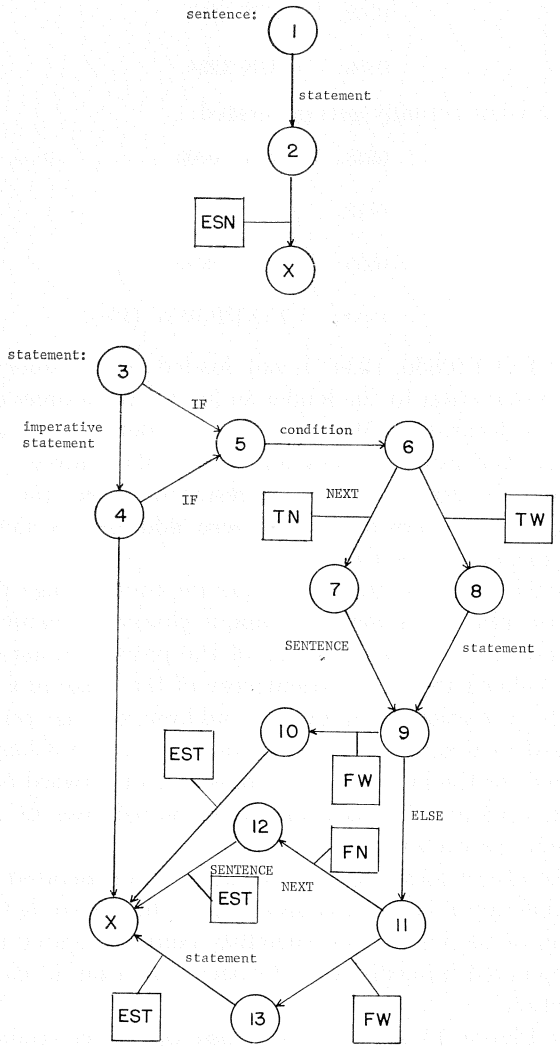


FIG. 14. Production of sequence-controlling operators. Square boxes contain names of symbols written by actions.

and AROUND are used by the Generator to create these paths. The following table defines the beginning and end of each path.

Path	*Beginning*	*End*
TRUE	test operator	TW or TN
AROUND	FW or FN	EST
FALSE	test operator	FW or FN

The test operator in the Polish is the last thing resulting from a test in a condition and generates code for a forked conditional branch with *two* branch addresses, a TRUE and a FALSE address.

Before specifying the generator actions in response to the several sequential operators let us define a few terms. There are the three stacks, the location counter LC, and two cells TEMP and NEXT. The top element of the TRUE (FALSE, AROUND) stack stores the location of the most recent true (false, around) branch instruction. (The AROUND stack serves double duty and also stores the locations of beginnings of test coding for use in processing of AND and OR operators.) The NEXT cell contains the location of the most recently generated branch to the beginning of the next sentence.

Several standard procedures occur in the Generator. GENERATE(x) writes the coding or fixup x. (FIXUP(L, A) is as above, and when read by the loader causes location L to have "A" put into its address, and so on.) ON(A, S) puts the contents of A onto the LIFO stack S. OFF(S) has as its value the top of the stack S, which is removed in the process. For example, ON(OFF(S), T) transfers the top element from S to T.

Table 3 defines the Generator responses to the sequential

TABLE 3. GENERATOR RESPONSES TO SEQUENTIAL AND BOOLEAN OPERATORS

Operator	*Response*
TW	OFF(AROUND); GENERATE(FIXUP(OFF(TRUE), LC))
TN	OFF(AROUND); TEMP ←OFF(TRUE); GENERATE(FIXUP(TEMP, NEXT)); NEXT ← TEMP
FW	ON(LC, AROUND); GENERATE(BUN 0); GENERATE(FIXUP(OFF(FALSE), LC))
FN	ON(LC, AROUND); GENERATE(BUN 0); TEMP ← OFF(FALSE); GENERATE(FIXUP(TEMP, NEXT)); NEXT ← TEMP
EST	GENERATE(FIXUP(OFF(AROUND), LC))
ESN	GENERATE(FIXUP(NEXT, LC)); NEXT ← 0
NOT	TEMP ← OFF(FALSE); ON(OFF(TRUE), FALSE); ON(TEMP, TRUE)
OR	OFF(AROUND); TEMP ← OFF(TRUE); GENERATE(FIXUP(TEMP, OFF(TRUE))); ON(TEMP, TRUE); TEMP ← OFF(FALSE); GENERATE(FIXUP(OFF(FALSE), OFF(AROUND))); ON(TEMP, FALSE); ON(LC, AROUND)
AND	OFF(AROUND); TEMP ← OFF(FALSE); GENERATE(FIXUP(TEMP, OFF(FALSE))); ON(TEMP, FALSE); TEMP ← OFF(TRUE); GENERATE(FIXUP(OFF(TRUE), OFF(AROUND))); ON(TEMP, TRUE); ON(LC, AROUND)

and Boolean operators. Note that AND, OR, and NOT generate no coding.

Depending on the code structure of the object machine, more than two branch addresses can be generated per test. Let [TEMP] denote the contents of TEMP, and let z be a variable whose values are the stack names TRUE and FALSE. Each time a z-branch must be generated the following occurs:

```
TEMP ← OFF(z);  ON(LC, z);
GENERATE( BRANCH [TEMP] ).
```

Before any code generation for a test, ON(0, z). At the end of the code generation for each test, ON(LC, AROUND).

This brief treatment of conditions may be clarified for the studious reader by an example. For the sake of discussion assume the following coding to be generated in response to the test operators LSS, EQL, and GTR:

A,B,LSS: LDA A, SUB B, BNA true address,
BUN false address;
A,B,EQL: LDA A, SUB B, BZA true address,
BUN false address;
A,B,GTR: LDA B, SUB A, BNA true address,
BUN false address.

The COBOL sentence to be considered is the following:

IF $X > Y$ IF $A = B$ OR $X = Y$ MOVE C TO D
ELSE NEXT SENTENCE
ELSE IF $C < D$ NEXT SENTENCE ELSE
MOVE E TO F.

The Diagrammer produces the following Polish:

X, Y, GTR, TW, A, B, EQL, X, Y, EQL, OR,
TW, C, D, MOVE, FN, EST,
FW, C, D, LSS, TN, FW, E, F, MOVE,
EST, EST, ESN.

The Generator's response is given in Table 5. After loading, the code appears in memory as shown in Table 4.

Except for the handling of the "≡" operator, which requires one bit of storage at object time, the techniques given here have direct applicability to the translation of ALGOL; in fact, they are equivalent to those given by Huskey and Wattenburg [8], with some modification for reducing storage requirements in the Generator when generation is unnatural. See also Arden, Galler, and Graham [9] for optimization techniques which might be useful for some object computers when Generator space is available.

On Producing Compilers

In the past few years there has been an expenditure of energy toward both writing and speaking about compilers which will generate copies of themselves. When a claim of superiority for such compilers is made it usually says that a compiler which can reproduce itself greatly simplifies the conversion to a new source or object language. Usually the arguments given in support of this claim take little or no account of the set of available methods which the proposed technique would supplant.

No compiler-writing technique will eliminate the requirement to analyze the task which the compiler to be created must perform, although it can provide a convenient language with which to carry out the analysis. The chief purpose of a compiler-writing technique is to reduce the labor which follows analysis and which is necessary for the production of the actual compiler. There are other ways to create a cheap compiler than simply to use a compiler as a programming aid. This article attempts to suggest one such way.

If a fast compiler is desired more can be said. The front end of any fast, one pass compiler will be written with an assembler; that's a corollary of the Seventy-five Percent Rule and some common sense about efficiency of compiler-generated code. Furthermore, the really fast compilers will have only one pass; that's the result of an analysis of how much extra work must be done by a multi-pass compiler. Notice that a corollary of these two statements is that really fast compilers can be written only for source languages which permit one-pass compilation. This proposition ought to be taken into account by language designers.

Our experience in the development of the prototype suggests that one analyst-programmer, with one or two understanding individuals around to talk to occasionally, can produce a COBOL compiler (*sans* library and object-program I-O control system) in a year or less, if he is provided with an assembler which permits incorporating all the special formats he will need into the assembly language.

Acknowledgments. Joseph Speroni of the Case Computing Center worked closely with the author during the six months of the project described here. His contributions were indispensable to the creation of the prototype program and a significant part of the design. Before the specific COBOL effort was begun, the author worked with

TABLE 4. APPEARANCE OF GENERATOR OUTPUT AFTER LOADING

```
1000  LDA Y
1001  SUB X
1002  BNA 1004
1003  BUN 1016
1004  LDA A
1005  SUB B
1006  BZA 1012
1007  BUN 1008
1008  LDA X
1009  SUB Y
1010  BZA 1012
1011  BUN 1023
1012  LDA C
1013  STA D
1014  BUN 1015
1015  BUN 1023
1016  LDA C
1017  SUB D
1018  BNA 1023
1019  BUN 1021
1020  BUN 1023
1021  LDA E
1022  STA F
1023  next sentence
```

Gilbert Steil, now of Mitre Corporation, in an investigation of the application of transition diagrams to the recognition of ALGOL; this work led to the No-Backup Condition and the desirability for the sake of efficiency of separating lexical and syntactical analysis. Both Mr. Speroni and Mr. Steil assisted in the proofreading of this article.

REFERENCES

1. Irons, E. T. A syntax directed compiler for ALGOL 60. *Comm. ACM 4* (Jan. 1961), 51.
2. COBOL—1961—Revised specifications for a common business oriented language. U. S. Government Printing Office, Washington, D. C., 1961, O-598941. Certain terminology and definitions are taken from this document without reference to any subsequent amendments thereto.
3. Williams, F. A. Handling identifiers as internal symbols in language processors. *Comm. ACM 2* (June 1959), 21.
4. Cantor, D. G. On the ambiguity problem of Backus systems. *J. ACM 9* (1962), 477.
5. Grau, A. A. A translator-oriented symbolic language programming language. *J. ACM 9* (1962), 480.
6. Conway, M. E. and Speroni, J. Arithmetizing declarations: an application to COBOL, *Comm. ACM 6* (Jan. 1963), 24.
7. Warshall, S. A syntax directed generator. Proc. EJCC, 1961, 295.
8. Huskey, H. D. and Wattenburg, W. H. Compiling techniques for Boolean expressions and conditional statements in ALGOL 60. *Comm. ACM 4* (Jan. 1961), 70.
9. Arden, B. W., Galler, B. A., and Graham, R. M. An algorithm for translating Boolean expressions. *J. ACM 9* (1962), 222.

TABLE 5. GENERATOR RESPONSES TO INPUT IN EXAMPLE

Polish Output		*Internal states after response*
Initial state		LC = 1000, NEXT = 0
X		
Y		
GTR	1000: LDA *Y*	LC = 1001, FALSE = 0, TRUE = 0
	1001: SUB *X*	LC = 1002, TEMP = 0, TRUE = 1002
	1002: BNA 0	LC = 1003, TEMP = 0, FALSE = 1003
	1003: BUN 0	LC = 1004, AROUND = 1004
TW	FIXUP(1002, 1004)	AROUND empty, TRUE empty
A		
B		
EQL	1004: LDA *A*	LC = 1005, FALSE = 1003/0, TRUE = 0
	1005: SUB *B*	LC = 1006, TEMP = 0, TRUE = 1006
	1006: BZA 0	LC = 1007, TEMP = 0, FALSE = 1003/1007
	1007: BUN 0	LC = 1008, AROUND = 1008
X		
Y		
EQL	1008: LDA *X*	LC = 1009, FALSE = 1003/1007/0, TRUE = 1006/0
	1009: SUB *Y*	LC = 1010, TEMP = 0, TRUE = 1006/1010
	1010: BZA 0	LC = 1011, TEMP = 0, FALSE = 1003/1007/1011
	1011: BUN 0	LC = 1012, AROUND = 1008/1012
OR		AROUND = 1008, TEMP = 1010, TRUE = 1006
	FIXUP(1010, 1006)	TRUE = 1010, TEMP = 1011, FALSE = 1003/1007
	FIXUP(1007, 1008)	FALSE = 1003/1011, AROUND = 1012
TW	FIXUP(1010, 1012)	AROUND empty, TRUE empty
C		
D		
MOVE	1012: LDA *C*	LC = 1013
	1013: STA *D*	LC = 1014
FN	1014: BUN 0	AROUND = 1014, LC = 1015, TEMP = 1011, FALSE = 1003
	FIXUP(1011, 0)	NEXT = 1011
EST	FIXUP(1014, 1015)	AROUND empty
FW	1015: BUN 0	AROUND = 1015, LC = 1016
	FIXUP(1003, 1016)	FALSE empty
C		
D		
LSS	1016: LDA *C*	FALSE = 0, TRUE = 0, LC = 1017
	1017: SUB *D*	LC = 1018, TEMP = 0, TRUE = 1018
	1018: BNA 0	LC = 1019, TEMP = 0, FALSE = 1019
	1019: BUN 0	LC = 1020, AROUND = 1015/1020
TN	FIXUP(1018, 1011)	AROUND = 1015, TEMP = 1018, TRUE empty, NEXT = 1018
FW	1020: BUN 0	AROUND = 1015/1020, LC = 1021
	FIXUP(1019, 1021)	FALSE empty
E		
F		
MOVE	1021: LDA *E*	LC = 1022
	1022: STA *F*	LC = 1023
EST	FIXUP(1020, 1023)	AROUND = 1015
EST	FIXUP(1015, 1023)	AROUND empty
ESN	FIXUP(1018, 1023)	NEXT = 0