16.01.001.043
February 26, 1962

IBM CONFIDENTIAL*
TECHNICAL REPORT

TABLES, FLOW CHARTS, AND PROGRAM LOGIC

M. S. Montalbano

ABSTRACT

Data processing problems to be solved by computers must now be described twice: first during the identification and definition of the problem, and then during actual programming. The heterogeneous activities which precede programming are commonly called "systems analysis." In its present state, the art of systems analysis is characterized chiefly by being neither systematic nor analytic. Tabular techniques can help make it both. The basic groupings of information in a program table are functionally similar to stages in the orderly acquisition of information about business systems. This structural similarity, added to the analytical and logical power which a table possesses, provides a promising basis for a truly "problem-oriented" language -- one useful in talking about and describing problems from the outset, rather than in converting one kind of detailed problem description to another.

International Business Machines Corporation
Advanced Systems Development Division Laboratory
San Jose, California

CONTENTS

TABLES, FLOW CHARTS, AND PROGRAM LOGIC

I. INTRODUCTION

The objective of this paper is to describe how program tables can be used to develop, display, implement and record the logical structure of digital computer procedures.

Data processing problems to be solved by computers must now be described twice: first during the identification and definition of the problem, and then during actual programming. The heterogeneous activities which precede programming are commonly called "systems analysis." In its present state, the art of systems analysis is characterized chiefly by being neither systematic nor analytic. Tabular techniques can help make it both. The basic groupings of information in a program table are functionally similar to stages in the orderly acquisition of information about business systems. This structural similarity, added to the analytical and logical power which a table possesses, provides a promising basis for a truly "problem-oriented" language--one useful in talking about and describing problems from the outset, rather than in converting one kind of detailed problem description to another.

The kind of tables which form the basis of tabular programming languages sort the information they display into four groups, which are customarily described as follows:

The "condition stub," which names logical variables.

The "condition entry," which lists permissible combinations of values of these logical variables.

The "action stub," which names action variables.

The "action entry," which lists sequences of values of these action variables.

Each set of logical-variable values in the condition entry is associated with a set of action-variable values in the action entry. Such an association is called a "rule." A rule is thus of the form: "If A and B and C and ... are true, then take consecutive actions P and Q and R and ..." In other words, a rule is an "if...then..." statement in which the "if" is followed by a conjunction of values for a prescribed set of logical variables, and the "then" is followed by a conjunction of values for a prescribed set of action variables.

We shall be concerned primarily with the condition entry portion of program tables and, in particular, with how the condition entry idea can most effectively be used to analyze, describe, program and document computer procedures with complicated branching structures.

II. A PRELIMINARY EXAMPLE

A concrete (though hypothetical) example will perhaps elucidate the abstractions of the preceding section and help set the stage for the abstractions of the next one. Consider the billing procedure of a wholesaler with three product lines, several classes of customer, and a discount and payment structure which depends upon class of customer, product line and dollar amount of invoice. These variables are as follows:

Product Lines:
1. Engines
2. Pumps
3. Fans

Classes of Customers:

1. Retail
2. Government Agencies
3. Engine Agents
4. Pump Agents
5. Pump Distributors
6. Fan Distributors

Dollar Ranges:
1. Less than $10.00
2. $10.00 to $49.99
3. $50.00 to $99.99
4. $100.00 or more

The information we have listed in the example thus far is the raw material for all decisions about discount and terms. It is also,

except for minor differences in arrangement, a completed condition stub for a tabular program.

This point is important. The act of specifying the grounds on which a program's decisions are to be based is, functionally, the same as the act of filling out a condition stub in a tabular programming language. To this extent, at least, the development of a tabular program is parallel to the system analysis phase of computer program development. Both are concerned with specifying the logical variables on which decisions are to be based and the values which these logical variables can assume.

Before we examine this example further, note the ready-made code by which we can now refer to the varying combinations determining the wholesaler's billing decisions. A three-digit number, whose positions each represent a value for one of the three kinds of variable listed (product, customer and dollar range, in turn), can now completely describe any set of factors: The code number 334, for example, designates an order from an engine agent for a fan costing $100.00 or more.

What is the next step? In analyzing a system, one next determines which <u>significant</u> combinations of logical-variable values occur. Our example shows three product lines, six classes of

customer and four dollar-amount ranges. Thus, the total number of possible product-customer-amount combinations is 72--the product of three, six and four. Generally, however, not all possibilities will occur. If, for example, no engine stocked costs less than $50.00, no combinations which include both engine and either code value 1 or 2 in the dollar range would ever occur in actual practice. All such combinations could either be omitted from consideration in the computer program, or included only to check clerical consistency.

Of the combinations which do occur, some may not be significant. Retail purchasers, for example, may all be billed identically whatever they order and however much it costs. The product-line and dollar-amount tests are thus not significant in this case, since, although different logical combinations do occur, they do not affect the action to be taken.

This requires a further extension of our coding scheme. In the case of tests which are not significant, X replaces one of the digits in the code. For example, X1X will indicate that retail purchasers have only one rule applied to them whatever they order and however much it costs.

One further convention completes the code for our present purposes. A bar ($\bar{}$) over a digit indicates that it is the only one

not admissible in the position it occupies. Government agencies, for example, may get discounts only on purchases totalling $100.00 or more, irrespective of product class. The corresponding coding would thus be X24 when the discount applied and X2$\bar{4}$ when it did not.

In this second stage of our system analysis--the stage of spelling out the combinations of logical values which actually occur and are significant in a given situation--we are, in substance, filling out the condition entry portion of a table. The information is the same in both cases; it is merely the manner of presentation which may be different. Thus, we again find a correspondence between a functional section of a table and a functional stage in the analysis of a system for computer solution or processing. And, again, the correspondence is important. The closer we can make the structure of our programming language correspond to the structure of a system analysis, the closer we can come to constructing an effective problem-oriented language for business problems.

It should now be possible to interpret Table 1, which displays all the relevant facts about our hypothetical case.

This table is divided into four quadrants by intersecting vertical and horizontal double lines. The northwest quadrant is the condition stub. The northeast is the condition entry. The

procedure of Fig. 3 might be called the "delayed rule" method. In the delayed-rule method, our objective is to delay as long as possible the tests which isolate rules for us.

Let us first consider Fig. 1. At the top of the page we have the condition entry portion of our original table, represented now as a set of ten Rule Identifiers, labelled, as they were in Table 2, with the Roman numerals I-X. To its right is a seven-by-five array which displays a row-by-row count of digit occurrences in the condition entry. We call this array the Row Count Matrix. The entries in this matrix tell us, for the row in which they appear, how many times a 1 occurs in the condition entry, how many times a 2 occurs, etc.

Looking at the first row of the Row Count Matrix, we learn that 1 occurs five times in the first row of the condition entry, and so does 2. The third row is similarly seen to be made up of two 1's, two 2's, five 3's and one 4.

In the procedure illustrated in Fig. 1, we ask those questions which will determine a rule for us as quickly as possible. We do this by looking for the smallest number in the Row Count Matrix and asking the question associated with this number. In Fig. 1, there are three equally small numbers in the Row Count Matrix--all 1's.

TABLE 1
BILLING PROCEDURE FOR SAMPLE WHOLESALE PROBLEM

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1. Engine 3. Fan 2. Pump	X	X	X	1	1	$\overline{1}$	2	$\overline{2}$	2	2	$\overline{2}$	3	$\overline{3}$	Else
1. Retail 3. Eng. Agt. 5. Pump Dist. 2. Gov't. 4. Pump Agt. 6. Fan Dist.	1	2	2	3	3	3	4	4	5	5	5	6	6	
1. Less than $10.00 3. $50.00 to $99.99 2. $10.00 to $49.99 4. $100.00 or more	X	4	$\overline{4}$	3	4	X	X	X	1	2	X	X	X	
Discount	0	15%	0	33%	40%	10%	25%	10%	30%	33%	15%	25%	10%	Error
Consignment	No	No	No	Yes	Yes	No	Yes	No	No	No	No	No	No	
Terms	C.O.D.	Net 30	Net 30	Net 30	Net 30	Net 30	Net 30	Net 30	30-60-90-	30-60-90-	Net 30	Net 30	Net 30	

southwest is the action stub; the southeast the action entry. The columns to the right of the vertical double line describe the rules. These are "if...then..." statements in which the "if" portion is described above the horizontal double line and the "then" portion is described below it.

To help further our understanding, let us interpret a few of the rules.

Rule No. 1 says: "If order is from a retail purchaser, then allow no discount, do not ship on consignment, ship C.O.D."

Rule No. 2 says: "If order is from a government agency and totals $100.00 or more, then allow 15% discount, do not ship on consignment, terms are net 30 days."

Rule No. 6 says: "If order is from an engine agent, but is not for an engine, allow 10% discount, do not ship on consignment, terms are net 30 days."

Rule No. 14 is a catch-all If no one of the previous rules applies, a coding error has been made. The code 132, for example, would be an error, since this company does not stock engines which cost less than $50.00.

III. EFFICIENCY, COMPLETENESS, AND CONSISTENCY

The remainder of this paper will focus largely on the portion of a table in which logical relationships are displayed--the condition entry. For business problems characterized by a complex logical structure, the condition entry can provide an analytical, logical and descriptive tool useful in system analysis, programming, debugging and modification.

Some of the benefits we can derive from an effective exploitation of the condition entry are:

1. Programming. The ability to compile sets of branching instructions which occupy minimum space in computer memory and which require a minimum average number of executions.

2. Analysis. The ability to make easy, comprehensive checks on the completeness and consistency of sets of logical alternatives.

3. Debugging. The ability to maintain identifiers which will display in short compass the prior "branch history" of a program without expensive breakpoint or statement-by-statement monitoring.

4. Modification. The ability to modify sets of branching instructions quickly, accurately and with a full realization of all the implications of such a modification.

For convenience, let us call the codes we identified in the previous section Rule Identifiers. The condition entry of Table 1 is made up of Rule Identifiers: X1X, X24, X2$\bar{4}$, 133...$\bar{3}$6X, Else. (This last is in a special category which we discuss below.)

(In the terminology of symbolic logic, the Rule Identifiers would be called the disjuncts of a logical statement in disjunctive normal form. The case most frequently considered, permitting only the values 0 and 1 for each logical variable, is the true-false propositional calculus or Boolean algebra. We have generalized this to permit more than two values for logical variables and to permit sums of disjuncts to be included by use of the not and don't-care conventions.)

Let us consider how we might exploit the logical ability of the condition entry--regarded as a set of Rule Identifiers--to achieve some of the benefits we listed above.

A. PROGRAMMING

Table 2 shows the condition stub and condition entry for another hypothetical example--a program which requires identification of ten pipe products. An example with highly redundant information was chosen intentionally to illustrate how we can eliminate

TABLE 2
CONDITION STUB AND CONDITION ENTRY FOR SAMPLE PIPE PROBLEM

	I	II	III	IV	V	VI	VII	VIII	IX	X
1. Black 2. Galvanized	1	1	1	1	1	2	2	2	2	2
1. Single Length 2. Double Length	1	1	1	2	2	1	1	1	1	2
1. Plain End 3. Threaded and Coupled 2. Threaded Only 4. Threaded One End	1	2	3	3	3	1	2	3	3	4
1. Light Wall 3. Heavy Wall 2. Standard Wall	1	2	2	2	3	2	2	2	2	2
1. Unoiled 2. Oiled	1	2	2	2	2	1	1	1	1	1
1. Uniform 3. Random 2. Semi-random	1	2	2	3	3	1	2	2	3	3
1. 1-inch 3. 2-inch 5. 4-inch 2. 1 1/2-inch 4. 2 1/2-inch	3	4	4	5	5	1	1	2	2	3

redundancy by means of Rule Identifiers.

The two programming objectives we wish to achieve are:

1. Minimum number of branching instructions in memory.

2. Minimum average number of executed branching instructions.

For simplicity, we assume binary branching, though the arguments given would be equally valid for other types.

Since we wish to differentiate among ten products, the minimum number of binary branching instructions we can get by with will be nine.

If the ten products occur with equal frequency, the theoretical minimum average number of branching instructions we could execute would be $\log_2 10 = 3.32$.

How close can we come to these two objectives?

Figures 1 and 3 illustrate two different methods of converting the condition entry of Table 2 to a set of branching instructions. Figures 2 and 4 display the resulting flow charts. Both flow charts have nine branchpoints (the minimum number); but one will require an average of 5.4 executed branch steps, the other will require 3.4.

The procedure we follow in Fig. 1 might be called the "quick rule" method. In the quick-rule method, our objective is to make as soon as possible those tests which will isolate a rule for us. The

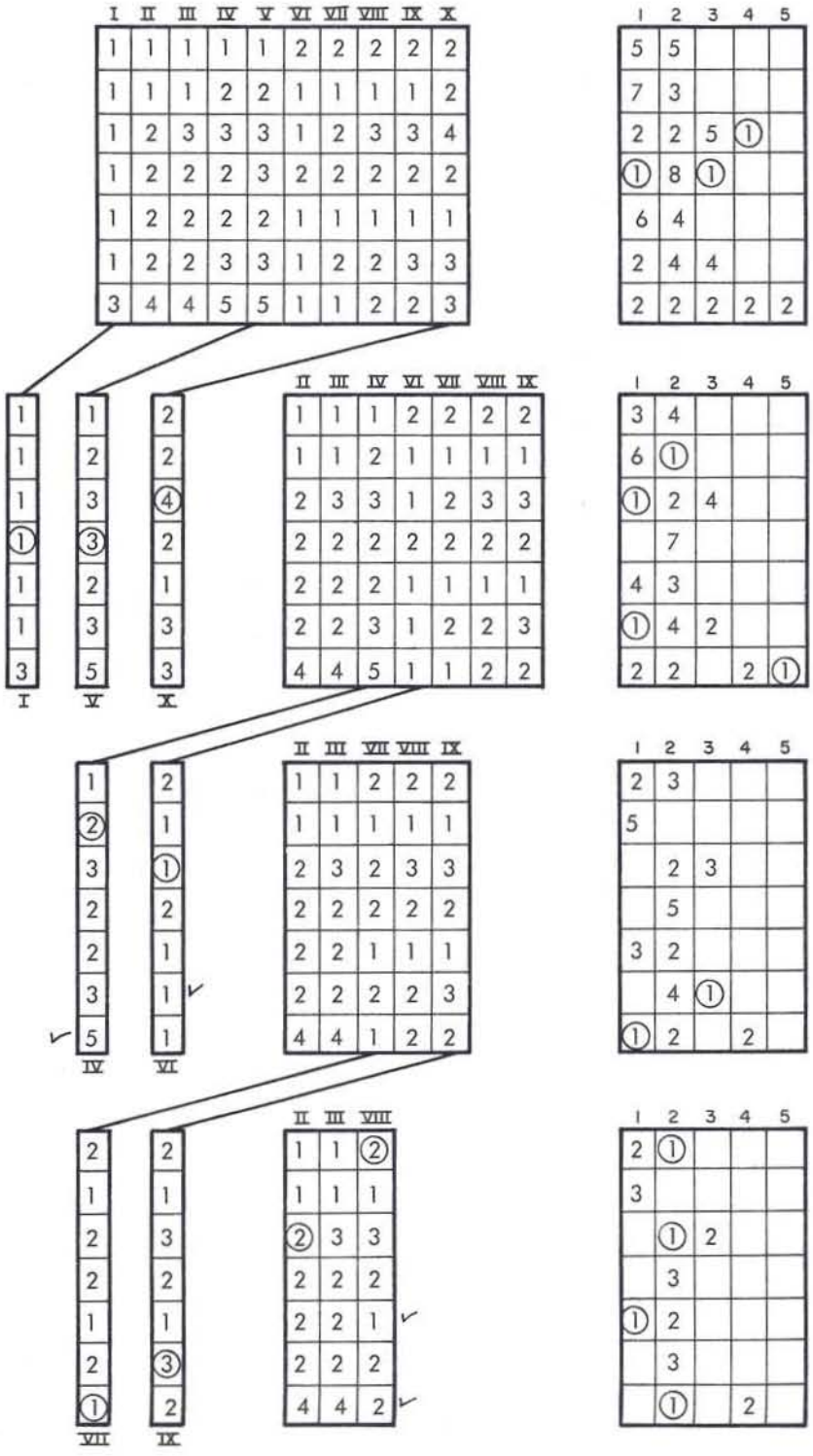

Fig. 1. Quick-Rule Method of Deriving Flow Charts from Tables

These 1's tell us that there is one occurrence of a 4 in Row 3; one occurrence of a 3 in Row 4, and one occurrence of a 1 in Row 4. The corresponding questions are: "Is this product threaded on one end?"; "Is this product heavy-wall?"; "Is this product light-wall?" If the answer to one of these questions is "yes," the pipe is correspondingly identified as product X, V or I. The first three branch-points of Fig. 2 ask these questions.

These products are now eliminated from further consideration. The condition entry is thus reduced to seven columns. The Row Count Matrix for this reduced condition entry shows four 1's. In this case, however, we are not as fortunate as we were the first time. The 1's occur in pairs, so only two rules can be isolated at this stage; Rules IV and VI. We can select Rule IV on the basis either of a 2 in the second position or a 5 in the seventh position. Similarly, Rule VI can be picked out on the basis either of a 1 in the third position or a 1 in the sixth position. The circles in selected Rules IV and VI show which tests we actually make in the flow chart of Fig. 2; the checkmarks show the alternative tests we could have made. The remaining steps follow in the same manner. The complete flow chart (Fig. 2) is the end result of the process.

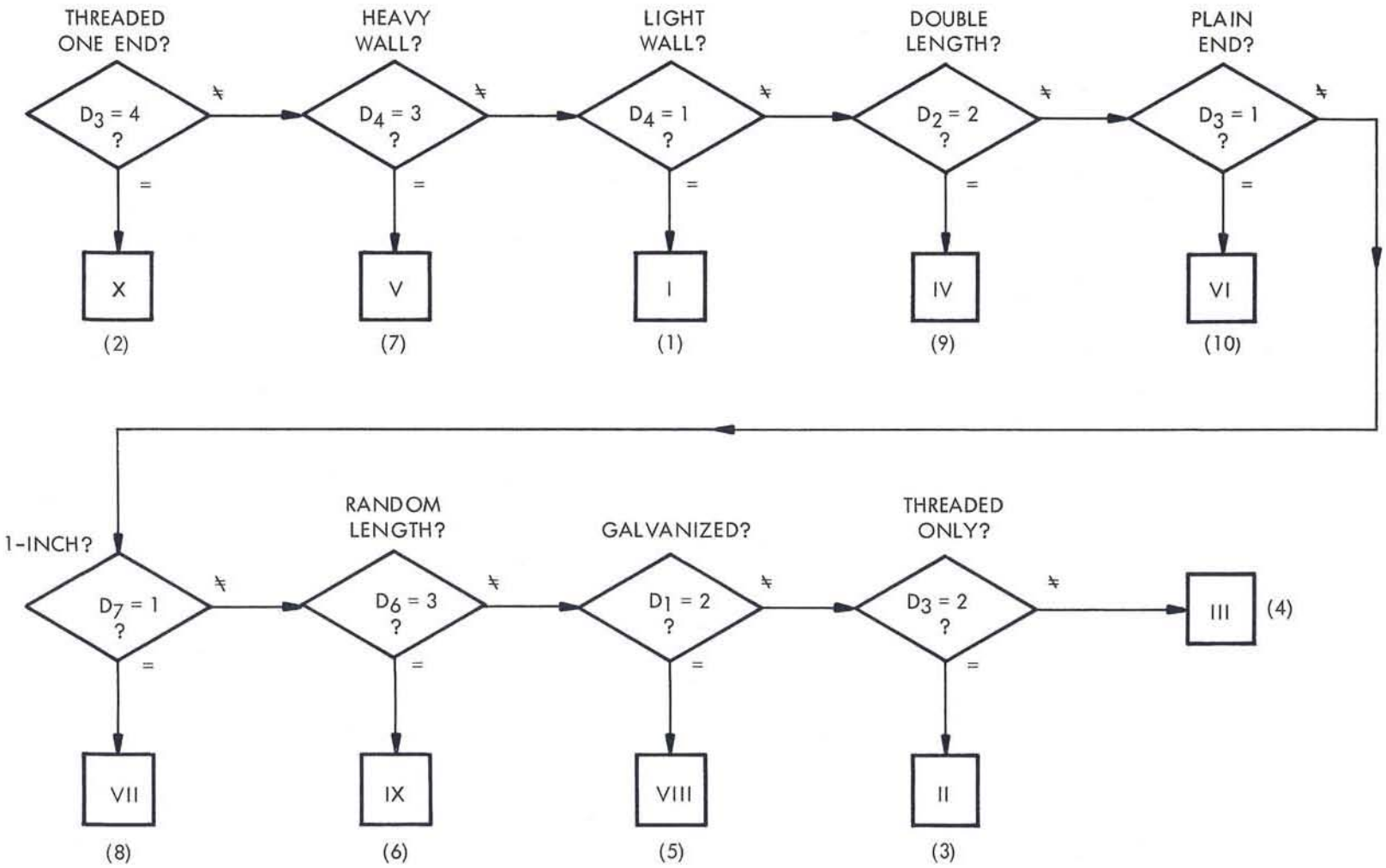

Fig. 2. Flow Chart Derived from Table 2 by Quick-Rule Method: Average Decisions Per Rule - 5.4 (with Uniform Rule Distribution)

As we have previously noted, this flow chart is efficient with respect to storage but not efficient with respect to average execution time. Let us consider Fig. 3 to see how we can schedule our tests so as to minimize average execution time.

In Fig. 3 (in which we omit the Row Count Matrix and the untested rows in the condition entry), we schedule our tests so as to delay rule identification as long as we can. To do this, we employ a procedure which might be described as: "Ask those questions first which will make the two differentiated groups of Rule Identifiers as similar in size as possible." This procedure is illustrated in Figs. 3 and 4. If the rules are of equal frequency, the flow chart of Fig. 4 will result in an average number of 3.4 branch-instruction executions per product. Like its predecessor, the flow chart of Fig. 4 also requires a minimum memory space. (The numbers in brackets and parentheses which are shown in Fig. 4 will be discussed later.)

If the rules were not of equal frequency, but their relative frequencies were known, the "minimum-average-path" principle we have just described would require only minor modification. Each rule would have its relative frequency associated with it as a "weight." Instead of a Row Count Matrix, one would have a Row Weight Count

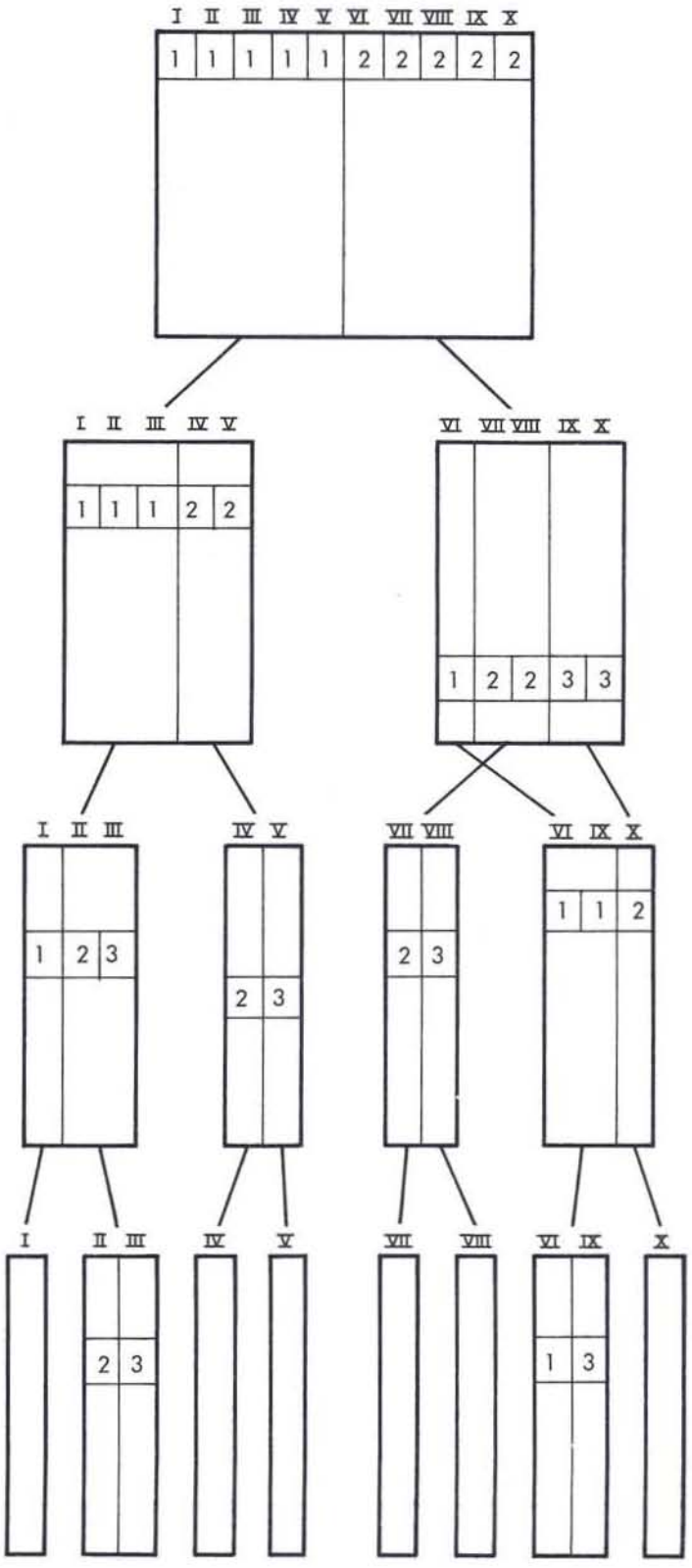

Fig. 3. Delayed-Rule (Minimum-Average-Path) Method of Deriving Flow Charts from Tables

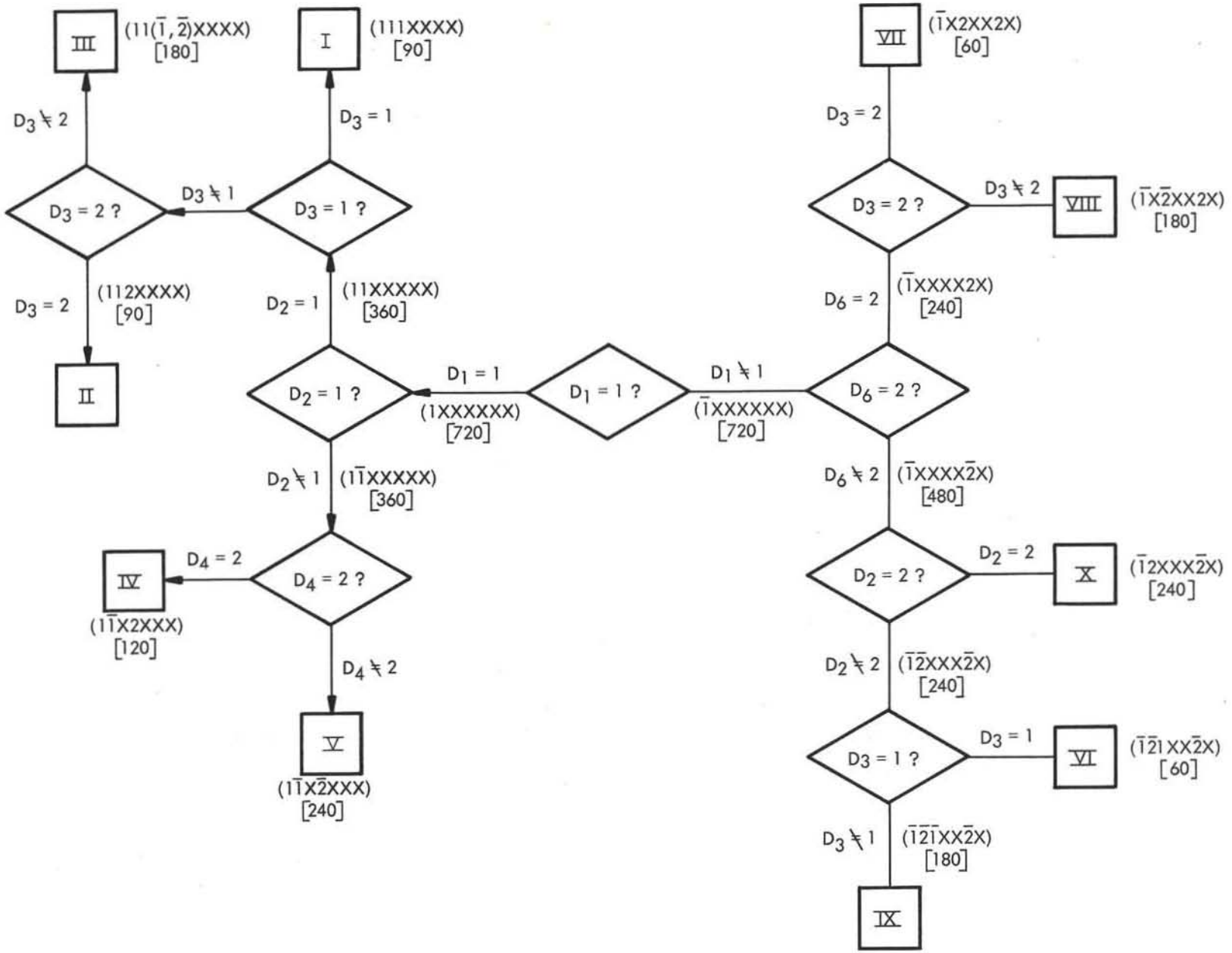

Fig. 4. Flow Chart Derived from Table 2 by Delayed-Rule Method: Average Decisions Per Rule - 3.4

Matrix. The objective would then become to divide the condition entry into groups of as nearly equal weight as possible.

(The procedure we have described is equivalent to the Shannon-Fano coding procedure in information theory.)

B. ANALYSIS

Descriptions of complicated sets of interacting decisions are liable to be inconsistent or incomplete. This is particularly true of descriptions made up of the kind of statements which are recorded during the course of a system analysis--statements which are most often dredged from a busy man while he is simultaneously trying to keep abreast of the procedure he is describing and plumbing his unconscious for relationships he <u>feels</u> rather than <u>knows</u>.

The Rule Identifiers provide a ready means to check sets of such statements for both completeness and consistency. This kind of checking can be done:

First, by the system analyst to establish his own understanding;

Second, by the programmer to check the system analysis;

Third, by the compiler to check the program.

These are brave claims. Can they be justified?

A sketchy attempt at justification is given in the remainder of this section. A more thorough discussion would require a paper in its own right. In dealing with the completeness and consistency of programming statements, we are dealing with a problem which is a central and major source of programming difficulty. The cause of this difficulty is "combinatorial complexity." This same combinatorial complexity plagues any discussion of the difficulty itself. As a result, such a discussion must chart a hazardous course between tedium and obscurity. The most common landfall for such a course is one shoal or the other--or both.

Let us start our voyage by dealing with a point of difficulty which we avoided in the previous section. This is the occurrence of a "don't-care" indication in the condition entry. How does such an indication affect the "table-to-flow-chart" procedure discussed above?

Consider Rule 2 in our wholesaling example (Table 1). The Rule Identifier for this rule is X24. What does the X--the "don't-care" indicator--signify in this case? It signifies that any permissible digit in the first position will lead to Rule 2; in other words, 124, 224, and 324 are equivalent rules--as long as our order is from a government agency and is for a total amount of $100.00 or more,

the 15% discount will apply, whether the article purchased is an engine, a pump, or a fan. Thus, the effect of a "don't-care" indicator is to consolidate several columns into one--the number of columns depending upon the number of alternatives possible for the logical variable to which the "don't-care" applies. If two "don't-care" indications occur in the same column, the number of columns consolidated into one is $\underline{m} \times \underline{n}$, where $\underline{m}$ and $\underline{n}$ are the number of alternative values for the first and second logical variables respectively. The extension to three or more "don't-care" entries is done similarly.

Table 3, and Figs. 5 and 6 may make this clearer by illustrating the relationship between compound rules--those in which "don't-care" entries occur--and simple rules--those in which each variable is specified exactly.

Table 3 is an uncoded table or, rather, the uncoded condition stub and condition entry of a table. The "don't-care" condition is indicated by the absence of an entry in any cell where the test is not significant. Note that one of the rules in this table is a catch-all rule called "Else"--this is the rule that applies when none of the others does.

Figure 5 shows just the condition entry portion of the same table, first as a coded set of compound rules in which "don't-care"

TABLE 3
CONDITION STUB AND CONDITION ENTRY OF SAMPLE TABLE CONTAINING "DON'T CARE" ENTRIES

	1	2	3	4	5	6	7	8
A eq. 2.5	Y	Y	Y	N	N	N		Else
B vs. 19	<	<	=	=	=	=	>	
C				=P	=P	=Q		
D is pos.	Y	N		Y	N			

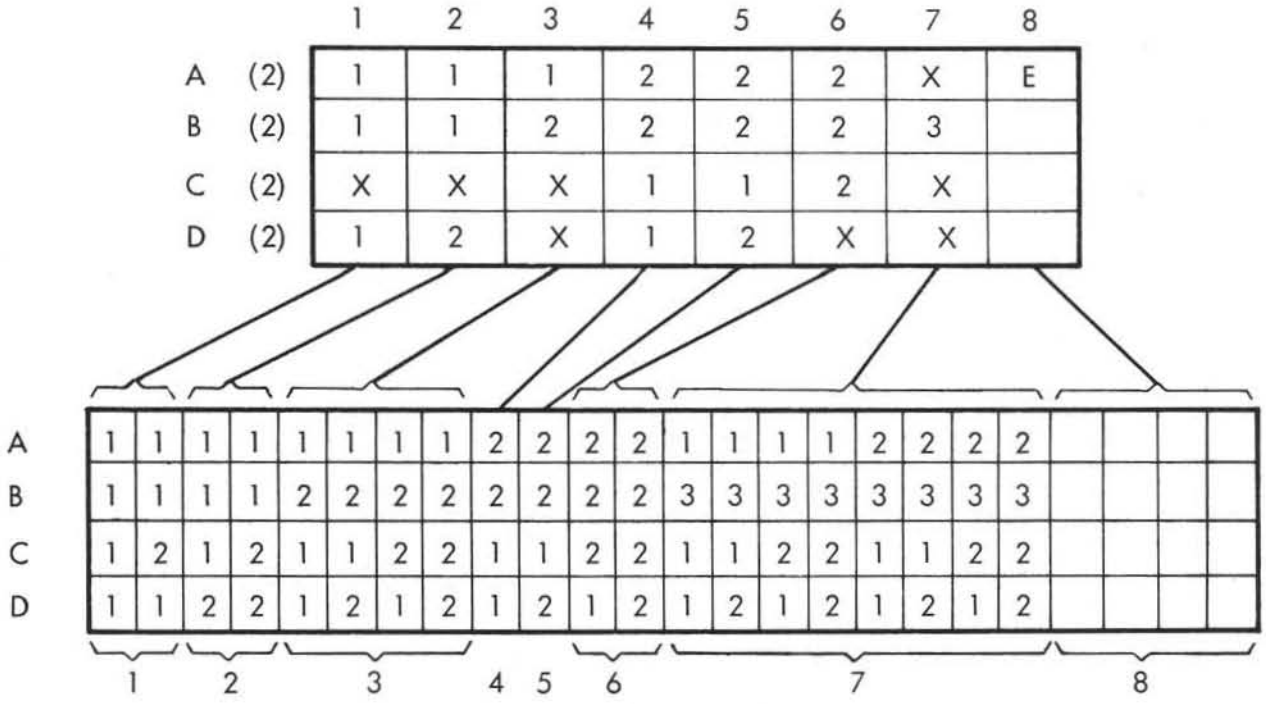

Fig. 5. Coded Condition Entry Portion of Table 3, Showing Relationship Between Compound and Simple Rules

is indicated by the presence of an X in a cell and, second, as the equivalent set of simple rules into which the compound-rule table can be analyzed.

In the simple-rule table, we have allowed four columns for Rule 8--the Else rule. How do we know that Rule 8 breaks down into four columns?

The total number of simple rules possible is the same as the total number of Rule Identifiers we can write. In this case, our Rule Identifiers are of the form a b c d, where a can be either 1 or 2; b can be either 1, 2, or 3; c can be either 1 or 2; and d can be either 1 or 2. The total number of different Rule Identifiers we can write is thus 2 x 3 x 2 x 2 = 24. We get 20 of these 24 when we decompose compound rules 1-7 into simple rules. The remaining four must therefore make up the simple rules combined into Rule 8.

Figure 6 illustrates some of the further analysis possible. Part I of Fig. 6 is merely a copy of the compound-rule table of Fig. 5 written in more compact form, with a number under each compound rule which tells how many simple rules it represents.

Part II is a Row Count Matrix for Rules 1-7. Note that the count is not the actual number of occurrences of X, 1, 2, or 3 in Part I, but the weighted number of occurrences--each occurrence of

a digit adds the column weight (the number at the foot of the column in Part I) to the Row Count.

Part III shows the Row Count Matrix for Rules 1-7 after the X's have been converted into the 1's and 2's, or 1's, 2's, and 3's they represent. In the first row of Part II, for example, the count in the X-column is 8. Since A, the variable in the first row, takes on only the values 1 and 2, half of the simple rules covered by the X have 1's in this digit position and half have 2's. We thus add 4 to the 1-count and 4 to the 2-count to get the Part III entries in that row: twelve 1's and eight 2's.

(Part III could, of course, have been obtained directly from the simple-rule table of Fig. 5.)

Part IV of Fig. 6 displays the Row Count Matrix for Rule 8, the Else rule. This is obtainable directly from the Part III Row Count Matrix. In the digit positions corresponding to two-valued variables, the 24 simple rules will divide into twelve 1's and twelve 2's. In the three-variable position, the simple rules will divide into eight 1's, eight 2's, and eight 3's.

An inspection of Part III readily shows which digit values are missing in each row. These missing digit values must occur in the Else column. This information is enough to enable us to make out the

	1	2	3	4	5	6	7	8
A	1	1	1	2	2	2	X	E
B	1	1	2	2	2	2	3	
C	X	X	X	1	1	2	X	
D	1	2	X	1	2	X	X	
	(2)	(2)	(4)	(1)	(1)	(2)	(8)	[4]

PART I

(1 - 7) X	1	2	3
8	8	4	
0	4	8	8
16	2	2	
14	3	3	

PART II

(1 - 7) 1	2	3
12	8	
4	8	8
10	10	
10	10	

PART III

(8) (Else)		
	4	
4		
2	2	
2	2	

PART IV

Fig. 6. Analysis of "Else" Rule in Table 3

Else Row Count Matrix. (We could also go on to write out all the simple rules which make up the Else column, if we chose.)

At the end of a procedure such as this--one which can be carried out easily by the system analyst, the programmer, or the computer--we have carefully checked the complete set of alternatives implicit in the set of logical variables we specified in the condition stub. For every possible combination, we have specified which action we should take.

We can check consistency as well as completeness--as part of the same procedure. Suppose the original Table 3 had contained a rule R with these entries

Y
=
=P
N

The Rule Identifier for this is 1212. Rule R is thus seen to be inconsistent with Rule 3, since the Rule Identifier of Rule 3 is 12XX--which includes 1212 as one of the simple rules of which it is made up. In other words, the statement that one action is to be taken whenever A = 1 and B = 2 is inconsistent with the statement that a different action

Rules 1 through 8 occur with equal frequency, we have an average of 3.25 branch instruction executions per rule.

C. DEBUGGING

Figures 4 and 7 display digital codes which are associated with the lines connecting branchpoint symbols. These codes record the branch history of the line segment next to which they appear. A code like $\bar{1}\bar{2}XXX\bar{2}X$ (Fig. 4), for example, tells us that variables one, two and six have been tested prior to the point in the flow chart at which this number appears, and that the tests have determined that the current value of the first variable is not 1, the second variable is not 2, and the sixth variable is not 2.

These codes are, in effect, incomplete Rule Identifiers when they occur between successive branchpoints, and complete Rule Identifiers when they occur between branchpoints and action boxes. If a computer program were written so as to maintain the current Rule Identifier (complete or incomplete) in a standard location available to a debugging program, the information provided by such a Rule Identifier would be a powerful and economical debugging aid. It would also provide assistance in systematic program checkout and diagnostics, since an automatic means to cycle through all permissible

Rule Identifiers could readily be devised for each program so constructed.

(As we have previously noted, the numbers in square brackets which accompany the Rule Identifiers merely record how many simple rules are contained in them. The large numbers associated with the rules of Fig. 4 measure the large amount of redundancy in the system.)

D. MODIFICATION

The point about modification can readily be made. Consider the pipe program described by Table 2. Suppose we wished to discontinue manufacturing light-wall pipe and wished to substitute a new product: galvanized, double-length, threaded one end, heavy-wall, oiled, semi-random, 4-inch pipe. All we need do is strike one column from the table and add another--an easy transition compared to the frantic redrawing which this same problem would require of a flow chart.

IV. THE USE OF TABLES

The preceding discussion, in which tables and their corresponding flow charts have been prepared side-by-side, serves to indicate the relative merits of the two forms of display of logical structure. The table is superior to the flow chart in displaying computer-independent information; the flow chart is superior in displaying computer-dependent information. If the problems we discussed above were to be programmed for machines which did branching by some other method than binary choice, the flow charts would be different but the tables would be unchanged. In this sense, tables are problem-oriented; flow charts are computer-oriented.

There is another aspect in which tables are problem-oriented. We have been considering primarily the condition entry portion of the table, since this is the point of greatest difference with past practice. But the division of the table into its four sections is, in itself, a useful aid in problem description. The condition stub is a list of all the questions and permissible answers pertinent to a particular problem. The condition entry is a list of all permissible combinations of answers. The action stub is a list of all the actions pertinent to a particular problem. The action entry is a list of the permissible sequences of actions. The rules serve to associate a

specific set of answers with a particular sequence of actions.

The advantages in problem description seem to be very great. In practice, how does one codify the kind of information about which data-processing programs are written? Would not stages characterized by the following four kinds of question describe many of the common elements in what are, admittedly, varied, individual, and complicated interactions?

1. "How do you know when to or how to do such-and-such?"

2. "Do this condition and that condition ever occur together?"

3. "What steps might you have to take in doing such-and-such?"

4. "When this condition and this condition and this condition, etc., occur together, which steps do you actually take?"

These questions, stated generally to avoid restricting their applicability unduly, are representative of the different kinds of question which are important in a procedural study. The four sections of a table correspond, functionally, to these four kinds of question.

V. CONCLUSION

There are, of course, many problems meriting further investigation: tables in their present form can become unwieldy when problem segments are prefaced by one or two simple decisions rather than six or seven complicated ones; it would sometimes be convenient to have rules in a table refer to other rules in the same table; Rule Identifiers in which the variable values are connected by "or" rather than "and" would sometimes be a convenience, and so on. Such further investigation would be desirable, since it would enhance the already considerable merit of tables as a means to implement program logic.

REFERENCES

1. Grad, Burton, "Tabular Form in Decision Logic," Datamation, July, 1961.

2. Kavanagh, Thomas F., "TABSOL--A Fundamental Concept for Systems Oriented Languages," Proceedings of the 1960 Eastern Joint Computer Conference.

3. Evans, Orren Y., "Advanced Analysis Method for Integrated Electronic Data Processing," IBM General Information Manual, #F20-8047.